阿里云 开发者社区 | Apache Flink

实时即未来

Apache Flink 年度最佳实践

一线大厂 生产级应用

独家实践 经验分享

Alibaba Group
阿里巴巴集团

扫一扫二维码图案，关注我吧

开发者社区

阿里云实时计算

实时计算交流钉钉群

Flink 社区微信公众号

目录

仅 1 年 GitHub Star 数翻倍，Apache Flink 做了什么?

作者：王峰（莫问） 阿里巴巴大数据实时计算负责人，资深技术专家

阿里妹导读：Apache Flink 是公认的新一代开源大数据计算引擎，其流水线运行系统既可以执行批处理程序也可以执行流处理程序。目前，Flink 已成为 Apache 基金会和 GitHub 社区最为活跃的项目之一。在 Flink Forward Asia 2019 上，阿里巴巴资深技术专家，实时计算负责人王峰（莫问）总结了 2019 年 Flink 在中国的发展和演进，阿里对 Flink 社区的贡献以及未来 Flink 的最新发展方向。

GitHub 地址：https://github.com/apache/flink

Flink：最活跃 Apache 项目之一

首先，简单总结一下 Flink 社区的发展情况。自 2014 年 Flink 贡献给开源社区之后，其发展非常迅速。目前，Flink 可以称之为 Apache 基金会中最为活跃的项目之一，在 GitHub 上其访问量在 Apache 项目中位居前三。从 Star 数量上看，仅仅是 2019 年一年的时间，Flink 在 GitHub 上的 Star 数量就翻了一倍，Contributor 数量也呈现出持续增长的态势。通过相关数据可以看出，越来越多的企业和开发者正在不断地加入 Flink 社区，并为 Flink 的发展贡献力量。其中，中国开发者也做出了巨大的贡献。

欢迎一起在 GitHub 上点个 Star

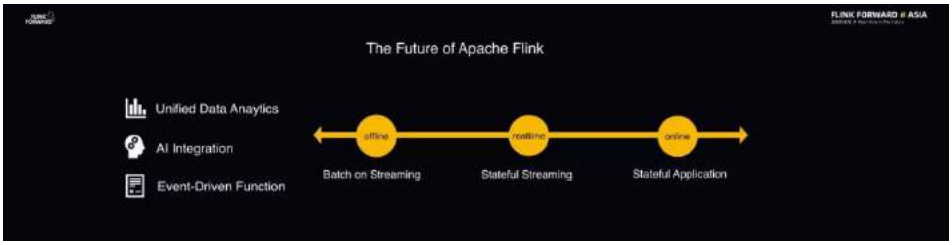
Apache Flink 在中国的应用

随着 Flink 社区的快速发展，其技术也逐渐走向成熟。在 2019 年，国内已经有大量的本土互联网公司开始采用 Apache Flink 作为主流的实时计算解决方案。同时，在全球范围内，优步、网飞、微软和亚马逊等国际互联网公司也逐渐开始使用 Apache Flink。

Apache Flink 的未来

如今，Flink 的主要应用场景基本上还是数据分析，尤其是实时数据分析。Flink 本质上是一款流式数据处理引擎，覆盖的场景主要是实时数据分析、实时风控、实时 ETL 处理等。未来，社区希望 Flink 演化成为统一的数据引擎。

- 在离线数据处理方面，希望 Flink 能够在流数据处理的基础之上进一步实现批与流的统一，提供统一的数据处理和分析的解决方案。
- 另一方面，朝着在线数据分析处理的方向演进，即利用 Flink 的核心优势、Event-Driven Function 的能力以及 Flink 自带的状态管理等特性实现在线的函数计算。



近年来，AI 场景发展得如火如荼并且计算的规模也越来越大。因此，Flink 社区也希望能够主动拥抱 AI 场景，在 Flink 机器学习方面支持 AI 场景，甚至和 AI 原生的深度学习引擎比如 Flink + TensorFlow、Flink + PyTorch 等实现协同，提供大数据 +AI 的全链路解决方案。

统一的数据分析解决方案

下图为 Apache Flink 批流一体的发展路线图。在 1.9 版本之前，Flink 的批和流还属于两条 Code Path，DataSet 和 DataStream 是两条独立的 API，具有两套不同的运行时环境，尚未实现批流一体的高度融合。所以在 2019 年发布的 Flink 1.9 版本和即将发布的 1.10 版本中，社区投入了大量精力去做 Flink 批流一体架构的整合。经过一年的努力，在 Flink 1.10 版本中已经实现了 Flink Task 的运行时环境、

执行引擎层以及 SQL 和 Table 层面的批和流的高度统一。但是目前而言，Flink 在架构上还没有完全实现批流全部统一。未来，社区希望将 DataSet 和 DataStream 两套 API 做到批流高度融合。

统一 Flink SQL

SQL 是在大数据处理中当之无愧的“王道”语言，同时也是最通用、最主流的语言。在 Flink 1.9 版本中发布了一部分统一的 SQL 功能，而未来在 1.10 版本中也会发布更多的新功能，比如采用了批流统一的 Query 处理器、支持完整的 DDL 功能。此外，Flink 还通过了 TPC-H 和 TPC-DS 的测试集验证，已达到生产级可用状态。Flink 1.10 版本还增强了对于 Python 的支持，目前 Flink SQL 能够非常方便地使用 Python UDF。除此之外，Flink 也积极地拥抱了 Hive 生态，使得 Flink SQL 能够兼容 Hive，这样用户能够以极低的成本尝试 Flink 的新技术。

统一 SQL 架构

下面将从技术层面分享 Flink Unified SQL 的架构是如何实现批流的融合，进而实现统一处理的。对于用户的一条 SQL 而言，无论是批处理还是流处理，可能读取数据的模式是相同的，只不过输出结果可能是一次性输出或者持续性输出。在 Flink 中，可以对于用户输入的 SQL 采用统一的处理器进行解析、编译、优化等动作，最终产生一个 Flink Job 提交到 Flink 集群中运行。

在查询处理的过程中，新版本的 Flink 增加了非常多的优化技术，比如执行计划策略的优化、执行算子的优化、二进制数据结构的优化、代码自动生成的优化以及 JVM 的优化等，使得 SQL 编译出来的 Job 执行效率更高。在 Runtime 方面，也对 Flink 执行引擎做了重构，对核心底层功能进行抽象，抽象出了可插拔的调度策略以及 Shuffle Service，这样一来 Runtime 非常灵活，能够自由适配流和批的 Job 模式，甚至能够实现同一 Job 中流算子和批算子的自由转换。

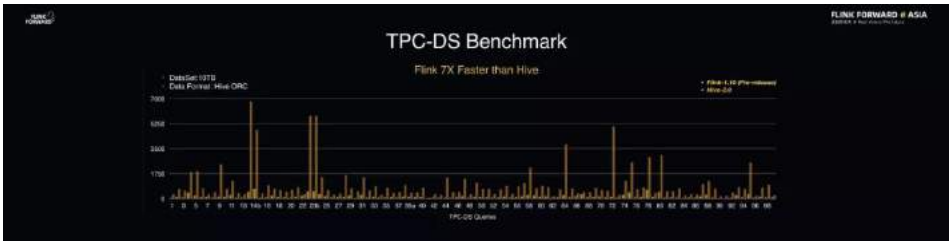
Flink 与 Hive 生态系统集成

让大家能够真正将 Flink SQL 用起来，不仅仅需要考虑优秀的内核技术或者完善的功能，也需要考虑到用户的迁移成本。最理想的情况就是让大家既能够享受到 Flink SQL 的新技术成果，同时又不用去修改已有的系统或者数据以及元数据等。因此，Flink SQL 在 2019 年的重大成果之一就是更好地对接了 Hive 生态。

在 Flink 1.10 版本中，批流一体的 SQL 将直接无缝对接 Hive 的 metastore，可以与 Hive 直接共享元数据，Flink Connector 能够直接读取 Hive 的分区表数据，并且不会产生任何影响。同时，Flink 还兼容 Hive 的 UDF，可以直接运行在 Hive 集群环境中，不需要定义额外的集群。整体的效果使得用户仅花费极低的成本就能够在 Hive SQL 和 Flink SQL 之间非常自由地实现切换。Flink SQL 的另外一个先天优势是可以支持流数据，也就是同一套业务逻辑在处理 Hive 数据的同时，也可以对接到 Kafka 等消息队列来处理实时数据。

TPC-DS Benchmark 测试效果

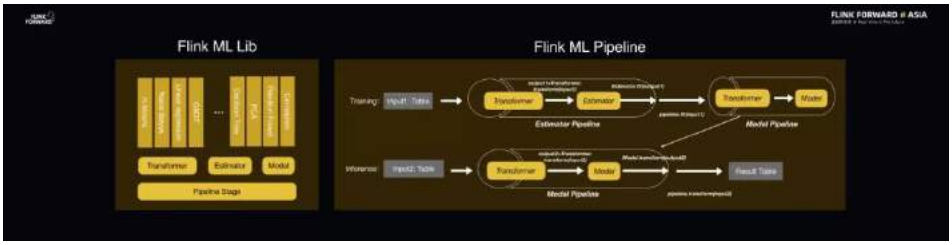
下图为 Flink 在 TPC-DS 的 Benchmark 测试的性能表现。这里的数据集规模为 10TB，数据格式为 Hive ORC，对比版本中，Hive 使用的是 3.0 版本，Flink 使用的 1.10 Pre-Release 版本。



结果表明，Flink 不仅能够跑通 99 个 TPC-DS 的查询，同时其性能还能够达到 Hive 的 7 倍。通过 Benchmark 就可以看到 Flink SQL 无论是在功能完善性、性能还是其他各个方面都已经达到了业界的高标准，达到了生产级可用。

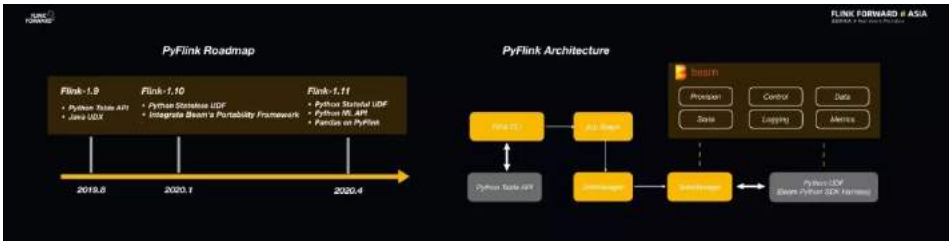
Flink 拥抱 AI

2019 年，整个技术圈里最火的当属 AI 了。而 Flink 除了做数据处理之外，还希望能够更好地拥抱 AI 场景。2019 年，Flink 在 AI 方面首先铺垫了机器学习基础设施，这部分所做的第一件事情就是实现了 Flink ML Lib 的基础 API，称之为 ML Pipeline。



ML Pipeline 的核心是机器学习的流程，其中的核心概念包含 Transformer、Estimator、Model 等。Flink 机器学习算法的开发人员可以使用这套 API 去开发不同的 Transformer、Estimator、Model，去实现各种经典的机器学习算法，非常方便。基于 ML Pipeline 这套 API 还能够自由组合组件来构建机器学习的训练流程和预测流程。

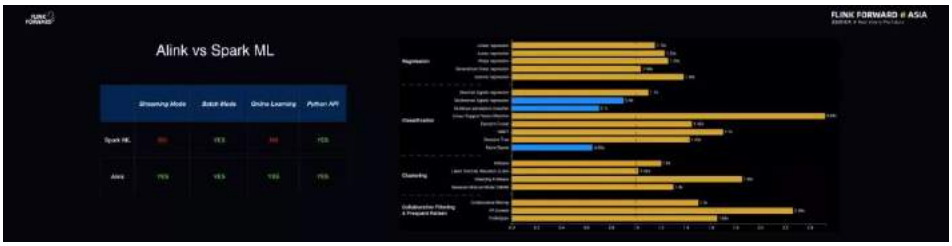
对于 AI 算法的开发人员而言，他们最喜欢的往往并不是 SQL 而是 Python。因此，Flink 对于 Python 的支持也尤为重要。在 2019 年，Flink 社区也投入了大量的资源来完善 Flink 的 Python 生态，诞生了 PyFlink 项目。并且在 Flink 1.9 版本中实现了 Python 对于 Table API 的支持。但这是不够的，在 Flink 1.10 版本中还重点支持了 Python UDF 特性。为了实现这一目标一般有两种技术选择，一种是从无到有地实现从 Java 到 Python 的通信，另一种是直接使用成熟的框架。很幸运的是 Beam 社区在 Python 支持上非常强大，因此 Flink 社区与 Beam 社区之间开展了良好的合作，Flink 使用了 Beam 的 Python 资源，比如 SDK、Framework 以及数据通信格式等。在未来，Flink 会进一步完善对于 Python API 和 UDF 的支持，在 ML Pipeline 上更多地支持 Python，同时也希望引入更多成熟的 Python 库。



Alibaba Alink

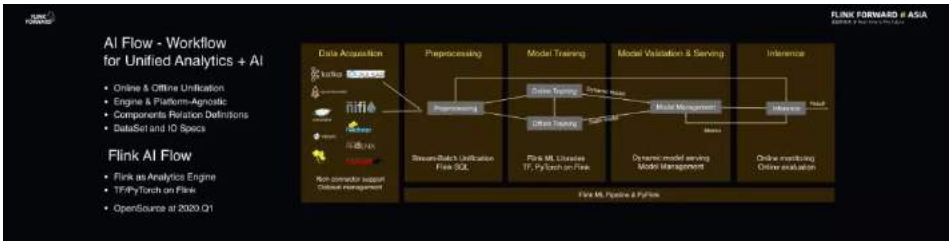
众所周知，阿里巴巴在 2018 年重磅推出了 Blink，也就是阿里内部的 Flink 版本。而 Alink 则是阿里巴巴内部的基于 Flink 的机器学习算法库，由阿里云机器学习 PAI 团队开发。Alink 是一套分布式、批流一体的机器学习算法库，它既非常好地利用了 Flink 批流一体的计算能力以及在机器学习基础设施上的一些优势，还结合了阿里巴巴的业务场景。目前，Alink 的上百个机器学习算法也正在向 Flink 社区贡献，希望能够成为新一代的 Flink ML。为了尽快让大家享受到 Alink 的技术红利，阿里巴巴也决定同时开源 Alink 项目。

将 Alink 与主流的机器学习算法库进行对比，可以发现其最大的优势就是不仅能够支持批式训练的机器学习场景，也能够支持在线的机器学习场景。Alink 在离线的机器学习场景下与主流的 Spark ML 做了对比，在功能集合上所有算法基本一致，此外还做了性能对比，Alink 和 Spark ML 在离线训练场景下的性能基本在一个水平线上，旗鼓相当。但是 Alink 的优势在于一些算法能够以流式方法进行计算，更好地实现在线机器学习。



AI Flow

另外，AI 部分的新项目——AI Flow 也值得关注。AI Flow 是大数据及 AI 的处理流程平台，在 AI Flow 中定义不同数据之间的关系以及元数据格式等就能够非常方便地搭建一套大数据及 AI 处理的流程。整个 Workflow 并不绑定某一引擎或者平台，但是用户可以借助 Flink 批流一体的能力去搭建自己的大数据及 AI 解决方案。目前，AI Flow 项目正在准备中，预计将于明年的第一季度以与 Alink 相同的模式进行开源。



云原生 (Cloud Native)

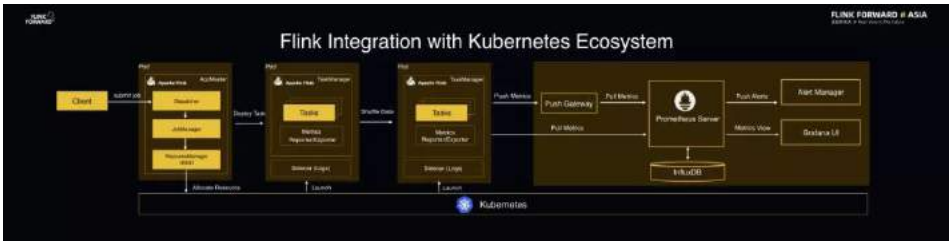
Flink 与 Kubernetes 生态系统集成

Flink 1.10 版本将会发布 Flink 与 Kubernetes 生态系统的集成功能，使得 Flink 能够原生地运行在 Kubernetes 管理平台之上。之所以要将 Flink 放在 Kubernetes 之上，是因为这样做有以下几点优势：

- 第一，Kubernetes 能够在多租户场景下为 Flink 带来更好的体验。
- 第二，目前各大公司都在逐步采用 Kubernetes 做 IT 设施的管理，如果 Flink

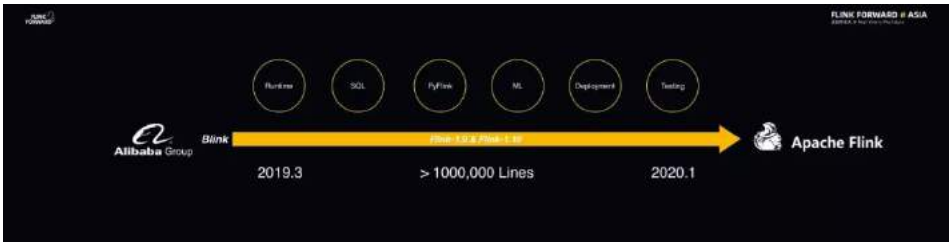
能够运行在 Kubernetes 之上，对于用户而言就能够实现更大规模的资源共享和统一管理，降低成本的同时能够提高效率。

- 第三，Kubernetes 云原生生态发展非常迅速，如果 Flink 能够与 Kubernetes 生态实现很好的整合，就能够让 Flink 享受到 Kubernetes 生态的技术红利，使得 Flink 能够在生产环境下提供运维保障。



阿里巴巴 Blink 贡献给 Apache Flink 社区

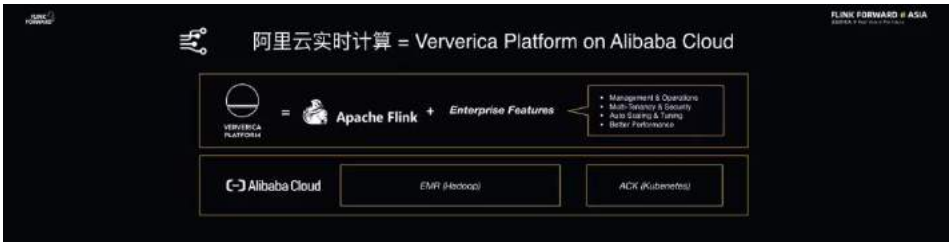
2019 年 3 月，Blink 正式开源。与此同时，阿里巴巴也希望将 Blink 的能力贡献回 Flink，共建一套 Flink 社区。而 Flink 通过 1.9 和即将发布的 1.10 两个大版本的迭代基本完成了这项工作。在这 10 个月的工作中，阿里巴巴向 Flink 社区贡献了超过一百万行代码，将 Blink 中积累的大量架构优化工作都推回给了 Flink 社区，不仅包括 Runtime、SQL、PyFlink，还包括新的 ML 等。



阿里云实时计算 -Ververica Platform on Alibaba Cloud

在将 Blink 逐步贡献到 Flink 之后，阿里巴巴决定在 2020 年将两套内核逐渐合并为一套内核，将 Blink 内核合并到 Flink 内核中，全面支持开源社区的发展。未来，

阿里云的产品和内部服务都会基于开源的 Flink 内核来实现。此外，阿里巴巴的技术团队和 Flink 创始团队 一起合作，联合打造了 Flink 企业版：Ververica Platform。这套全新的企业版将会支持阿里巴巴内部业务和云上业务。阿里巴巴也将投入更多力量到开源 Flink 的发展和社区的建设当中，也希望和广大业界同仁一起助力 Flink 中文社区的发展。



Lyft 基于 Apache Flink 的大规模准实时数据分析平台

作者：徐赢　高立

（徐赢　Lyft Technical Lead，实时数据平台

高立　Lyft Technical Lead，计算数据平台）

摘要：如何基于 Flink 搭建大规模准实时数据分析平台？在 Flink Forward Asia 2019 上，来自 Lyft 公司实时数据平台的徐赢博士和计算数据平台的高立博士分享了 Lyft 基于 Apache Flink 的大规模准实时数据分析平台。

本次分享主要分为四个方面：

1. Lyft 的流数据与场景
2. 准实时数据分析平台和架构
3. 平台性能及容错深入分析
4. 总结与未来展望

一、Lyft 的流数据与场景

关于 Lyft

Lyft 是位于北美的一个共享交通平台，和大家所熟知的 Uber 和国内的滴滴类似，Lyft 也为民众提供共享出行的服务。Lyft 的宗旨是提供世界最好的交通方案来改善人们的生活。

Lyft 的流数据场景

Lyft 的流数据可以大致分为三类，秒级别、分钟级别和不高于 5 分钟级别。分钟级别流数据中，自适应定价系统、欺诈和异常检测系统是最常用的，此外还有 Lyft 最新研发的机器学习特征工程。不高于 5 分钟级别的场景则包括准实时数据交互查询相关的系统。

Lyft 数据分析平台架构

如下图所示的是 Lyft 之前的数据分析平台架构。Lyft 的大部分流数据都是来自于事件，而事件产生的来源主要有两种，分别是手机 APP 和后端服务，比如乘客、司机、支付以及保险等服务都会产生各种各样的事件，而这些事件都需要实时响应。

在分析平台这部分，事件会流向 AWS 的 Kinesis 上面，这里的 Kinesis 与 Apache Kafka 非常类似，是一种 AWS 上专有的 PubSub 服务，而这些数据流都会量化成文件，这些文件则都会存储在 AWS 的 S3 上面，并且很多批处理任务都会弹出一些数据子集。在分析系统方面，Lyft 使用的是开源社区中比较活跃的 presto 查询引擎。Lyft 数据分析平台的用户主要有四种，即数据工程师、数据分析师以及机

器学习专家和深度学习专家，他们往往都是通过分析引擎实现与数据的交互。

既往平台的问题

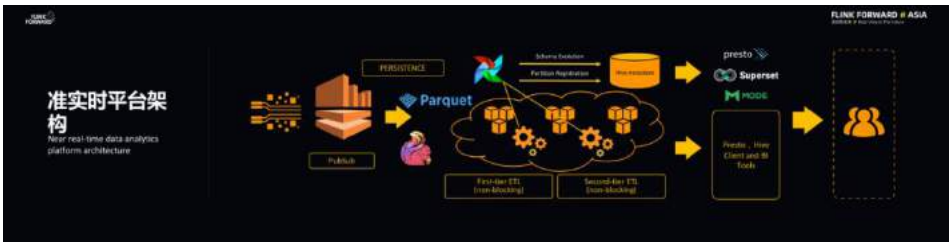
Lyft 之所以要基于 Apache Flink 实现大规模准实时数据分析平台，是因为以往的平台存在一些问题。比如较高的延迟，导入数据无法满足准实时查询的要求；并且基于 Kinesis Client Library 的流式数据导入性能不足；导入数据存在太多小文件导致下游操作性能不足；数据 ETL 大多是高延迟多日多步的架构；此外，以往的平台对于嵌套数据提供的支持也不足。

二、准实时数据分析平台和架构

准实时平台架构

在新的准实时平台架构中，Lyft 采用 Flink 实现流数据持久化。Lyft 使用云端存储，而使用 Flink 直接向云端写一种叫做 Parquet 的数据格式，Parquet 是一种列数据存储格式，能够有效地支持交互式数据查询。Lyft 在 Parquet 原始数据上架构实时数仓，实时数仓的结构被存储在 Hive 的 Table 里面，Hive Table 的 metadata 存储在 Hive metastore 里面。

平台会对于原始数据做多级的非阻塞 ETL 加工，每一级都是非阻塞的 (nonblocking)，主要是压缩和去重的操作，从而得到更高质量的数据。平台主要使用 Apache Airflow 对于 ETL 操作进行调度。所有的 Parquet 格式的原始数据都可以被 presto 查询，交互式查询的结果将能够以 BI 模型的方式显示给用户。



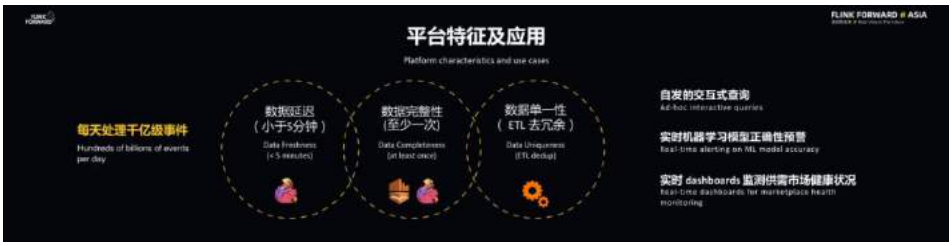
平台设计

Lyft 基于 Apache Flink 实现的大规模准实时数据分析平台具有几个特点：

- 首先，平台借助 Flink 实现高速有效的流数据接入，使得云上集群规模缩减为原来的十分之一，因此大大降低了运维成本。
- 其次，Parquet 格式的数据支持交互式查询，当用户仅对于某几个列数据感兴趣时可以通过分区和选择列的方式过滤不必要的数据，从而提升查询的性能。
- 再次，基于 AWS 的云端存储，平台的数据无需特殊存储形式。
- 之后，多级 ETL 进程能够确保更好的性能和数据质量。
- 最后，还能够兼顾性能容错及可演进性。

平台特征及应用

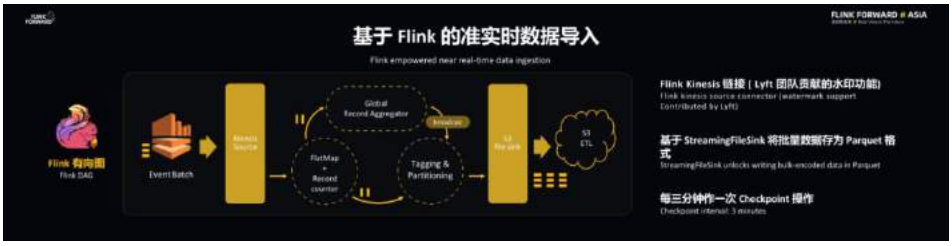
Lyft 准实时数据分析平台需要每天处理千亿级事件，能够做到数据延迟小于 5 分钟，而链路中使用的组件确保了数据完整性，同时基于 ETL 去冗余操作实现了数据单一性保证。



数据科学家和数据工程师在建模时会需要进行自发的交互式查询，此外，平台也会提供实时机器学习模型正确性预警，以及实时数据面板来监控供需市场健康状况。

基于 Flink 的准实时数据导入

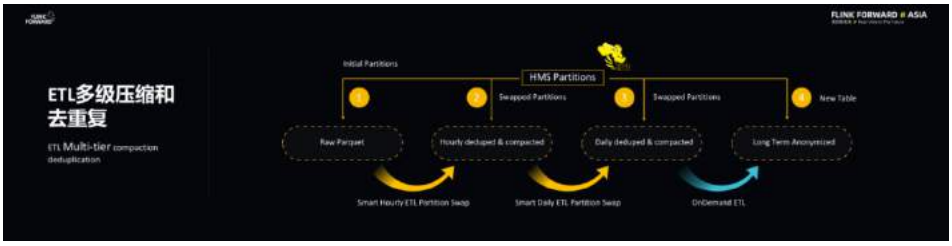
下图可以看到当事件到达 Kinesis 之后就会被存储成为 EventBatch。通过 Flink-Kinesis 连接器可以将事件提取出来并送到 FlatMap 和 Record Counter 上面，FlatMap 将事件打撒并送到下游的 Global Record Aggregator 和 Tagging Partitioning 上面，每当做 CheckPoint 时会关闭文件并做一个持久化操作，针对于 StreamingFileSink 的特征，平台设置了每三分钟做一次 CheckPoint 操作，这样可以保证当事件进入 Kinesis 连接器之后在三分钟之内就能够持久化。



以上的方式会造成太多数量的小文件问题，因为数据链路支持成千上万种文件，因此使用了 Subtasks 记录本地事件权重，并通过全局记录聚合器来计算事件全局权重并广播到下游去。而 Operator 接收到事件权重之后将会将事件分配给 Sink。

ETL 多级压缩和去重

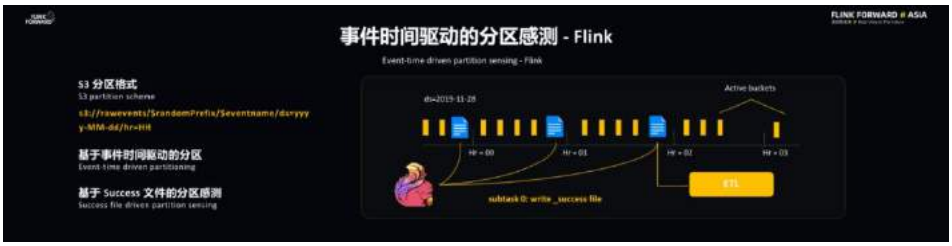
上述的数据链路也会做 ETL 多级压缩和去重工作，主要是 Parquet 原始数据会经过每小时的智能压缩去重的 ETL 工作，产生更大的 Parquet File。同理，对于小时级别压缩去重不够的文件，每天还会再进行一次压缩去重。对于新产生的数据会有一个原子性的分区交换，也就是说当产生新的数据之后，ETL Job 会让 Hive metastore 里的表分区指向新的数据和分区。这里的过程使用了启发性算法来分析哪些事件必须要经过压缩和去重以及压缩去重的时间间隔级别。此外，为了满足隐私和合规的要求，一些 ETL 数据会被保存数以年计的时间。



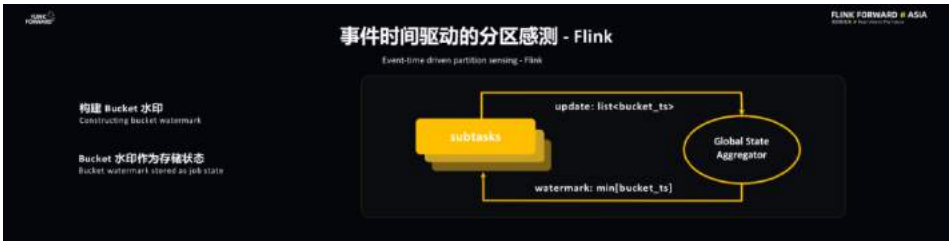
三、平台性能及容错深入分析

事件时间驱动的分区感测

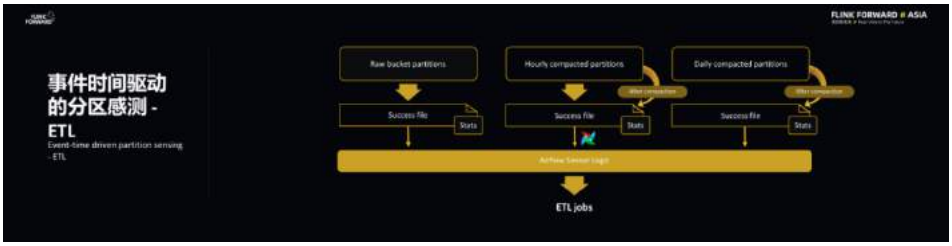
Flink 和 ETL 是通过事件时间驱动的分区感测实现同步的。S3 采用的是比较常见的分区格式，最后的分区是由时间戳决定的，时间戳则是基于 EventTime 的，这样的好处在于能够带来 Flink 和 ETL 共同的时间源，这样有助于同步操作。此外，基于事件时间能够使得一些回填操作和主操作实现类似的结果。Flink 处理完每个小时的事件后会向事件分区写入一个 Success 文件，这代表该小时的事件已经处理完毕，ETL 可以对于该小时的文件进行操作了。



Flink 本身的水印并不能直接用到 Lyft 的应用场景当中，主要是因为当 Flink 处理完时间戳并不意味着它已经被持久化到存储当中，此时就需要引入分区水印的概念，这样一来每个 Sink Source 就能够知道当前写入的分区，并且维护一个分区 ID，并且通过 Global State Aggregator 聚合每个分区的信息。每个 Subtasks 能够知道全局的信息，并将水印定义为分区时间戳中最小的一个。



ETL 主要有两个特点，分别是及时性和去重，而 ETL 的主要功能在于去重和压缩，最重要的是在非阻塞的情况下就进行去重。前面也提到 Smart ETL，所谓 Smart 就是智能感知，需要两个相应的信息来引导 Global State Aggregator，分别是分区完整性标识 SuccessFile，在每个分区还有几个相应的 States 统计信息能够告诉下游的 ETL 怎样去重和压缩以及操作的频率和范围。



Schema 演进的挑战

ETL 除了去重和压缩的挑战之外，还经常会遇到 Schema 的演化挑战。Schema 演化的挑战分为三个方面，即不同引擎的数据类型、嵌套结构的演变、数据类型演变对去重逻辑的影响。

S3 深入分析

Lyft 的数据存储系统其实可以认为是数据湖，对于 S3 而言，Lyft 也有一些性能的优化考量。S3 本身内部也是有分区的，为了使其具有并行的读写性能，添加了 S3 的熵数前缀，在分区里面也增加了标记文件，这两种做法能够极大地降低 S3 的 IO 性能的影响。标识符对于能否触发 ETL 操作会产生影响，与此同时也是对于 presto 的集成，能够让 presto 决定什么情况下能够扫描多少个文件。

Parquet 优化方案

Lyft 的准实时数据分析平台在 Parquet 方面做了很多优化，比如文件数据值大小范围统计信息、文件系统统计信息、基于主键数据值的排序加快 presto 的查询速度以及二级索引的生成。

基于数据回填的平台容错机制

如下两个图所示的是 Lyft 准实时数据分析平台的基于数据回填的平台容错机制。对于 Flink 而言，因为平台的要求是达到准实时，而 Flink 的 Job 出现失效的时候可能会超过一定的时间，当 Job 重新开始之后就会形成两个数据流，主数据流总是从最新的数据开始往下执行，附加数据流则可以回溯到之前中断的位置进行执行直到中断结束的位置。这样的好处是既能保证主数据流的准实时特性，同时通过回填数据流保证数据的完整性。

对于 ETL 而言，基于数据回填的平台容错机制则表现在 Airflow 的幂等调度系统、原子压缩和 HMS 交换操作、分区自建自修复体系和 Schema 整合。

四、总结与未来展望

体验与经验教训

利用 Flink 能够准实时注入 Parquet 数据，使得交互式查询体验为可能。同时，Flink 在 Lyft 中的应用很多地方也需要提高，虽然 Flink 在大多数情况的延时都能够得到保证，但是重启和部署的时候仍然可能造成分钟级别的延时，这会对于 SLO 产生一定影响。

此外，Lyft 目前做的一件事情就是改善部署系统使其能够支持 Kubernetes，并且使得其能够接近 0 宕机时间的效果。因为 Lyft 准实时数据分析平台在云端运行，因此在将数据上传到 S3 的时候会产生一些随机的网络情况，造成 Sink Subtasks 的停滞，进而造成整个 Flink Job 的停滞。而通过引入一些 Time Out 机制来检测 Sink Subtasks 的停滞，使得整个 Flink Job 能够顺利运行下去。

ETL 分区感应能够降低成本和延迟，成功文件则能够表示什么时候处理完成。此外，S3 文件布局对性能提升的影响还是非常大的，目前而言引入熵数还属于经验总结，后续 Lyft 也会对于这些进行总结分析并且公开。因为使用 Parquet 数据，因此对于 Schema 的兼容性要求就非常高，如果引入了不兼容事件则会使得下游的 ETL 瘫痪，因此 Lyft 已经做到的就是在数据链路上游对于 Schema 的兼容性进行检查，检测并拒绝用户提交不兼容的 Schema。

未来展望

Lyft 对于准实时数据分析平台也有一些设想。

- **首先**，Lyft 希望将 Flink 部署在 Kubernetes 集群环境下运行，使得 Kubernetes 能够管理这些 Flink Job，同时也能够充分利用 Kubernetes 集群的高可扩展性。
- **其次**，Lyft 也希望实现通用的流数据导入框架，准实时数据分析平台不仅仅支持事件，也能够支持数据库以及服务日志等数据。
- **再次**，Lyft 希望平台能够实现 ETL 智能压缩以及事件驱动 ETL，使得回填等事件能够自动触发相应的 ETL 过程，实现和以前的数据的合并，同时将延时数据导入来对于 ETL 过程进行更新。
- **最后**，Lyft 还希望准实时数据分析平台能够实现存储过程的改进以及查询优化，借助 Parquet 的统计数据来改善 presto 的查询性能，借助表格管理相关的开源软件对存储管理进行性能改善，同时实现更多的功能。

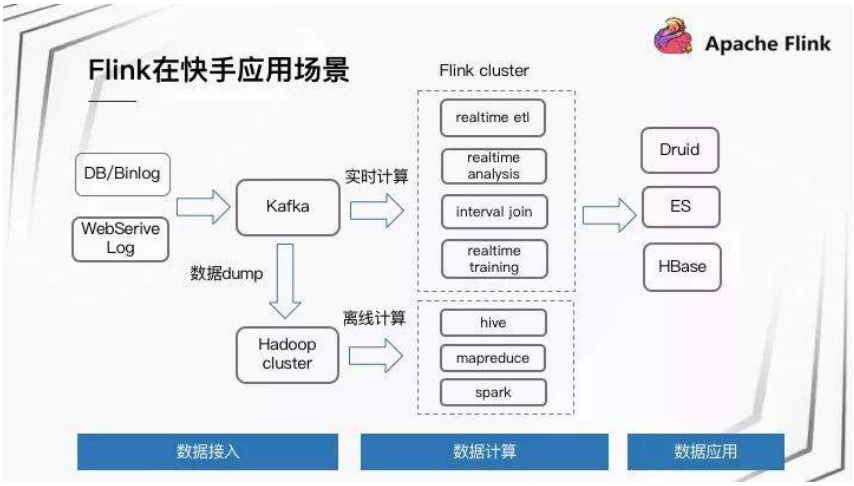
日均处理万亿数据！ Apache Flink 在快手的应用实践与技术演进之路

作者：董亭亭（快手实时计算引擎团队负责人）

作为短视频分享跟直播的平台，快手有诸多业务场景应用了 Flink，包括短视频、直播的质量监控、用户增长分析、实时数据处理、直播 CDN 调度等。本文将从 Flink 在快手的应用场景以及目前规模、Flink 在落地过程的技术演进过程、未来计划这三个方面详细介绍 Flink 在快手的应用与实践。

一、Apache Flink 在快手的应用场景与规模

1. Flink 在快手应用场景



快手计算链路是从 DB/Binlog 以及 WebService Log 实时入到 Kafka 中，然后接入 Flink 做实时计算，其中包括实时 ETL、实时分析、Interval Join 以及实时训练，最后的结果存到 Druid、ES 或者 HBase 里面，后面接入一些数据应用产品；同时这一份 Kafka 数据实时 Dump 一份到 Hadoop 集群，然后接入离线计算。

Flink 在快手应用的类别主要分为三大类：

- 80% 统计监控：实时统计，包括各项数据的指标，监控项报警，用于辅助业务进行实时分析和监控；
- 15% 数据处理：对数据的清洗、拆分、Join 等逻辑处理，例如大 Topic 的数据拆分、清洗；
- 5% 数据处理：实时业务处理，针对特定业务逻辑的实时处理，例如实时调度。

Flink 在快手应用的典型场景包括：

- 快手是分享短视频跟直播的平台，快手短视频、直播的质量监控是通过 Flink 进行实时统计，比如直播观众端、主播端的播放量、卡顿率、开播失败率等跟直播质量相关的多种监控指标；
- 用户增长分析，实时统计各投放渠道拉新情况，根据效果实时调整各渠道的投放量；
- 实时数据处理，广告展现流、点击流实时 Join，客户端日志的拆分等；
- 直播 CDN 调度，实时监控各 CDN 厂商质量，通过 Flink 实时训练调整各个 CDN 厂商流量配比。

2. Flink 集群规模

快手目前集群规模有 1500 台左右，作业数量大约是 500 左右，日处理条目数总共有 1.7 万亿，峰值处理条目数大约是 3.7 千万。集群部署都是 On Yarn 模式，分为离线集群和实时集群两类集群，其中离线集群混合部署，机器通过标签进行物理隔离，实时集群是 Flink 专用集群，针对隔离性、稳定性要求极高的业务部署。

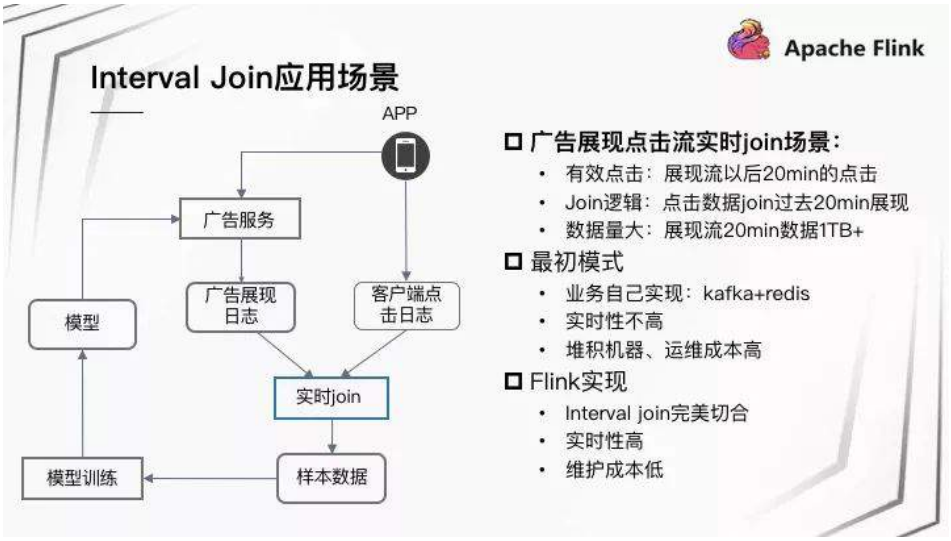
二、快手 Flink 技术演进

快手 Flink 技术演进主要分为三部分：

- 基于特定场景优化，包括 Interval Join 场景优化；
- 稳定性改进，包括数据源控速、JobManager 稳定性、作业频繁失败；
- 平台建设。

1. 场景优化

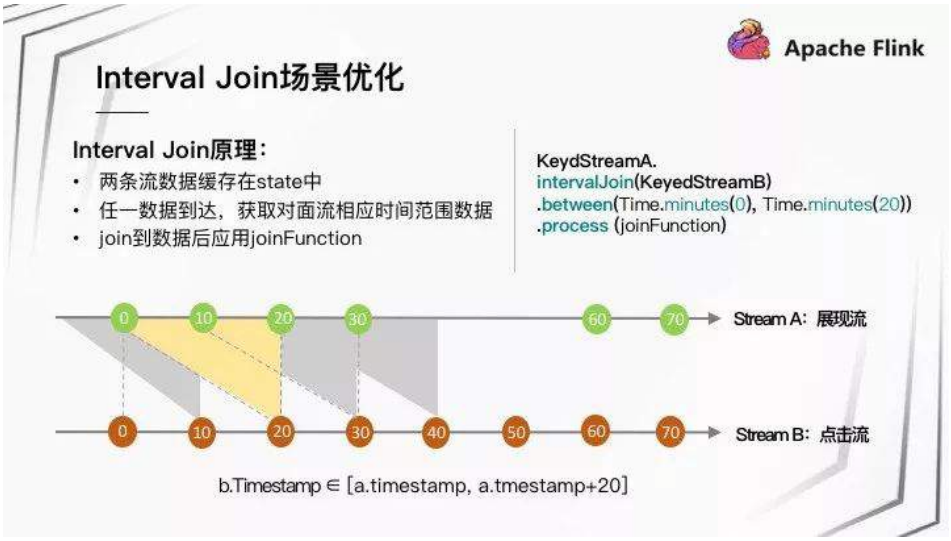
1.1 Interval Join 应用场景



Interval Join 在快手的一个应用场景是广告展现点击流实时 Join 场景：打开快手 App 可能会收到广告服务推荐的广告视频，用户有时会点击展现的广告视频。这样在后端形成两份数据流，一份是广告展现日志，一份是客户端点击日志。这两份数据需进行实时 Join，将 Join 结果作为样本数据用于模型训练，训练出的模型会被推送到线上的广告服务。

该场景下展现以后 20 分钟的点击被认为是有效点击，实时 Join 逻辑则是点击数据 Join 过去 20 分钟展现。其中，展现流的数据量相对比较大，20 分钟数据在 1 TB 以上。最初实时 Join 过程是业务自己实现，通过 Redis 缓存广告展现日志，Kafka 延迟消费客户端点击日志实现 Join 逻辑，该方式缺点是实时性不高，并且随着业务增长需要堆积更多机器，运维成本非常高。基于 Flink 使用 Interval Join 完美契合此场景，并且实时性高，能够实时输出 Join 后的结果数据，对业务来说维护成本非常低，只需要维护一个 Flink 作业即可。

1.2 Interval Join 场景优化



1.2.1 Interval Join 原理

Flink 实现 Interval join 的原理：两条流数据缓存在内部 State 中，任意一数据到达，获取对面流相应时间范围数据，执行 joinFunction 进行 Join。随着时间的推进，State 中两条流相应时间范围的数据会被清理。

在前面提到的广告应用场景 Join 过去 20 分钟数据，假设两个流的数据完全有序到达，Stream A 作为展现流缓存过去 20 分钟数据，Stream B 作为点击流每来一条数据到对面 Join 过去 20 分钟数据即可。

Flink 实现 Interval Join：

```
KeyedStreamA.intervalJoin(KeyedStreamB)
        .between(Time.minutes(0),Time.minutes(20))
        .process(joinFunction)
```

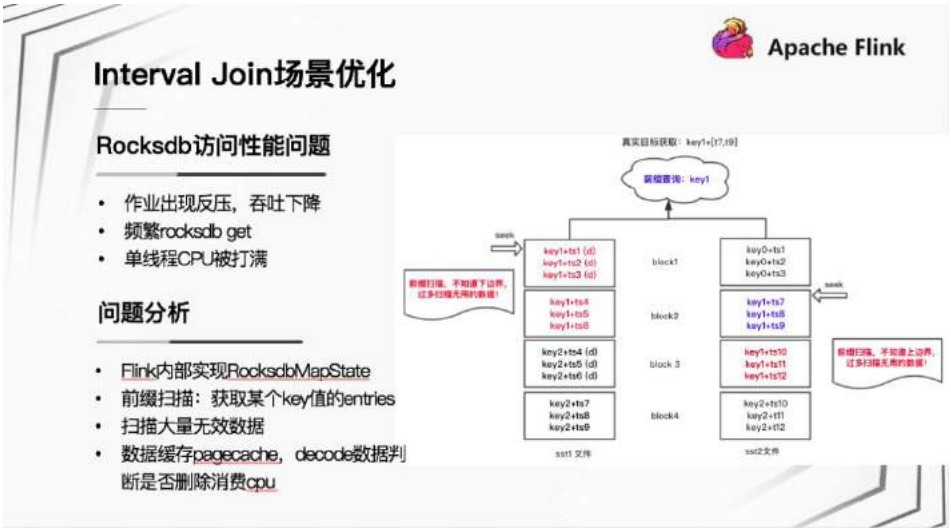
1.2.2 状态存储策略选择

关于状态存储策略选择，生产环境状态存储 Backend 有两种方式：

- FsStateBackend：State 存储在内存，Checkpoint 时持久化到 HDFS；
- RocksDBStateBackend：State 存储在 RocksDB 实例，可增量 Checkpoint，适合超大 State。在广告场景下展现流 20 分钟数据有 1 TB 以上，从节省内存等方面综合考虑，快手最终选择的是 RocksDBStateBackend。

在 Interval join 场景下，RocksDB 状态存储方式是将两个流的数据存在两个 Column Family 里，RowKey 根据 keyGroupId+joinKey+ts 方式组织。

1.2.3 RocksDB 访问性能问题



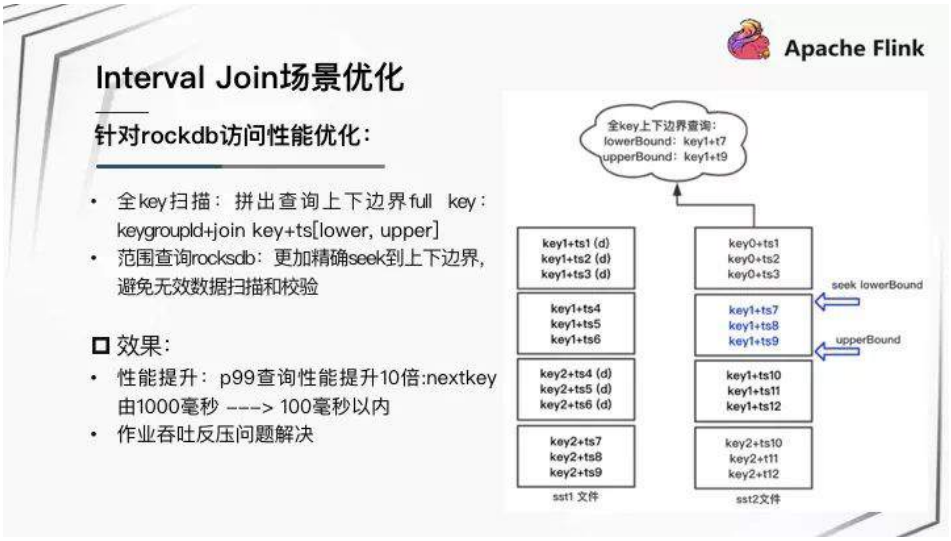
Flink 作业上线遇到的第一个问题是 RocksDB 访问性能问题，表现为：

- 作业在运行一段时间之后出现反压，吞吐下降。
- 通过 Jstack 发现程序逻辑频繁处于 RocksDB get 请求处。
- 通过 Top 发现存在单线程 CPU 持续被打满。

进一步对问题分析，发现：该场景下，Flink 内部基于 RocksDB State 状态存储时，获取某个 Join key 值某段范围的数据，是通过前缀扫描的方式获取某个 Join key 前缀的 entries 集合，然后再判断哪些数据在相应的时间范围内。前缀扫描的方式会导致扫描大量的无效数据，扫描的数据大多缓存在 PageCache 中，在 Decode 数据判断数据是否为 Delete 时，消耗大量 CPU。

以上图场景为例，蓝色部分为目标数据，红色部分为上下边界之外的数据，前缀扫描时会过多扫描红色部分无用数据，在对该大量无效数据做处理时，将单线程 CPU 消耗尽。

1.2.4 针对 RocksDB 访问性能优化

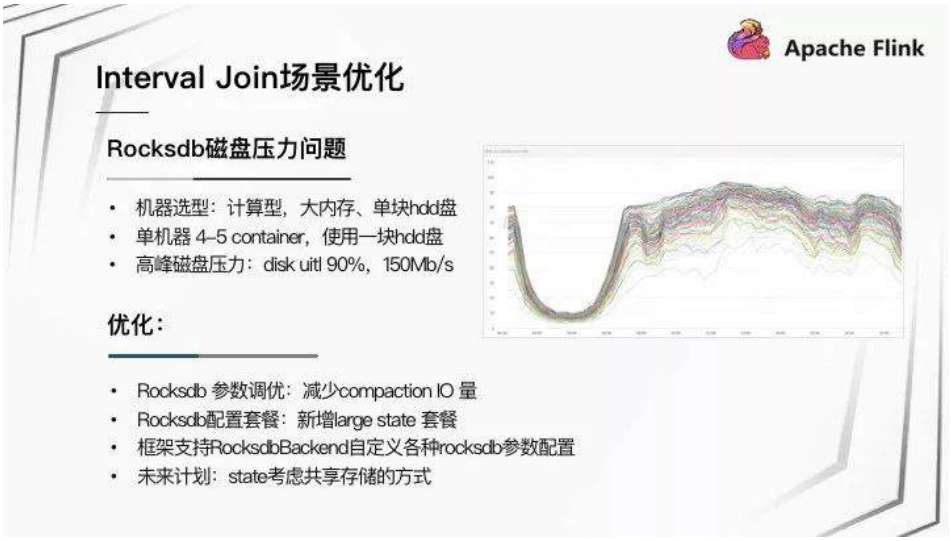


快手在 Interval join 该场景下对 RocksDB 的访问方式做了以下优化：

- 在 Interval join 场景下，是可以精确的确定需访问的数据边界范围。所以用全 Key 范围扫描代替前缀扫描，精确拼出查询上下边界 Full Key 即 keyGroupId+joinKey+ts[lower,upper]。
- 范围查询 RocksDB ，可以更加精确 Seek 到上下边界，避免无效数据扫描和校验。

优化后的效果：P99 查询时延性能提升 10 倍，即 nextKey 获取 RocksDB 一条数据，P99 时延由 1000 毫秒到 100 毫秒以内。作业吞吐反压问题进而得到解决。

1.2.5 RocksDB 磁盘压力问题



Flink 作业上线遇到的第二个问题是随着业务的增长，RocksDB 所在磁盘压力即将达到上限，高峰时磁盘 util 达到 90%，写吞吐在 150 MB/s。详细分析发现，该问题是由以下几个原因叠加导致：

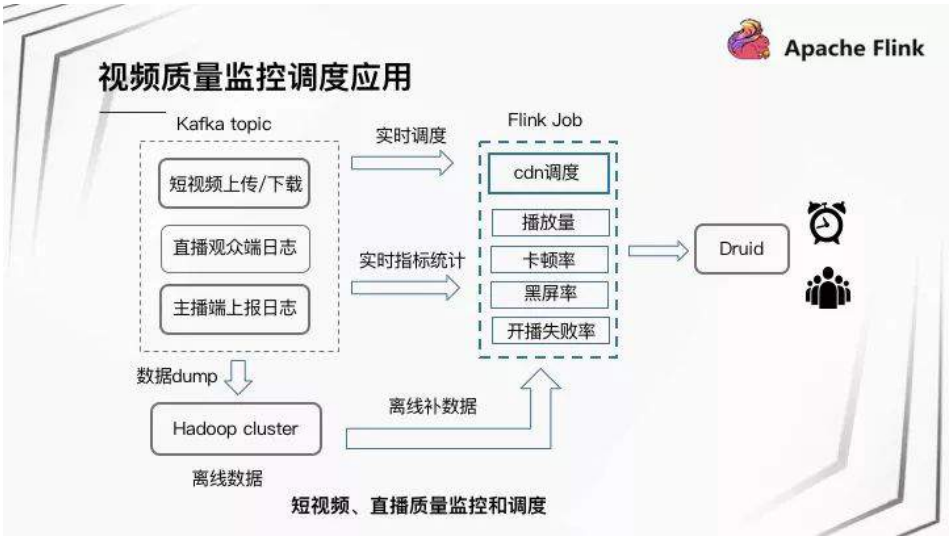
- Flink 机器选型为计算型，大内存、单块 HDD 盘，在集群规模不是很大的情况下，单个机器会有 4-5 个该作业 Container，同时使用一块 HDD 盘。
- RocksDB 后台会频繁进行 Compaction 有写放大情况，同时 Checkpoint 也在写磁盘。

针对 RocksDB 磁盘压力，快手内部做了以下优化：

- 针对 RocksDB 参数进行调优，目的是减少 Compaction IO 量。优化后 IO 总量有一半左右的下降。
- 为更加方便的调整 RocksDB 参数，在 Flink 框架层新增 Large State RocksDB 配置套餐。同时支持 RocksDBStateBackend 自定义配置各种 RocksDB 参数。

- 未来计划，考虑将 State 用共享存储的方式存储，进一步做到减少 IO 总量，并且快速 Checkpoint 和恢复。

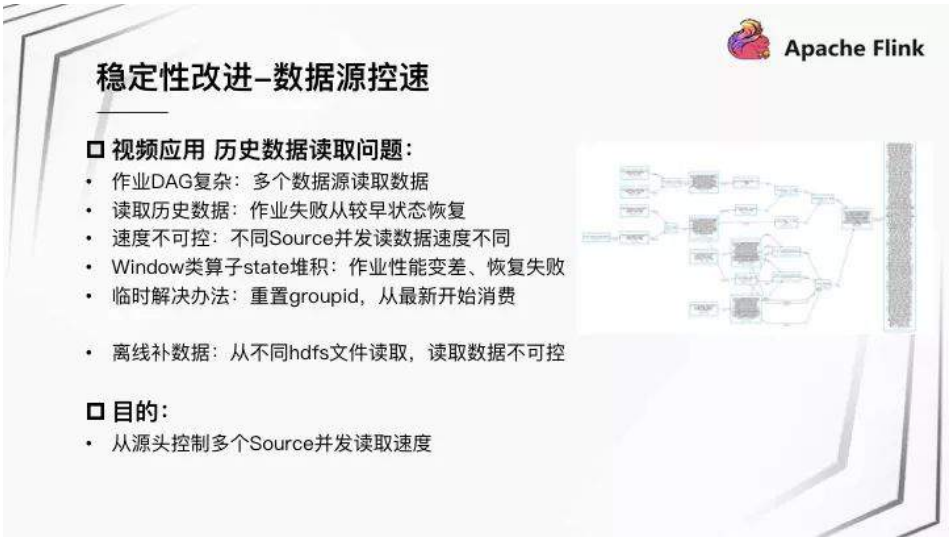
2. 稳定性改进



首先介绍下视频质量监控调度应用背景，有多个 Kafka Topic 存储短视频、直播相关质量日志，包括短视频上传 / 下载、直播观众端日志，主播端上报日志等。Flink Job 读取相应 Topic 数据实时统计各类指标，包括播放量、卡顿率、黑屏率以及开播失败率等。指标数据会存到 Druid 提供后续相应的报警监控以及多维度的指标分析。同时还有一条流是进行直播 CDN 调度，也是通过 Flink Job 实时训练、调整各 CDN 厂商的流量配比。

以上 Kafka Topic 数据会同时落一份到 Hadoop 集群，用于离线补数据。实时计算跟离线补数据的过程共用同一份 Flink 代码，针对不同的数据源，分别读取 Kafka 数据或 HDFS 数据。

2.1 数据源控速



视频应用场景下遇到的问题是：作业 DAG 比较复杂，同时从多个 Topic 读取数据。一旦作业异常，作业失败从较早状态恢复，需要读取部分历史数据。此时，不同 Source 并发读取数据速度不可控，会导致 Window 类算子 State 堆积、作业性能变差，最终导致作业恢复失败。另外，离线补数据，从不同 HDFS 文件读数据同样会遇到读取数据不可控问题。在此之前，实时场景下临时解决办法是重置 GroupID 丢弃历史数据，使得从最新位置开始消费。

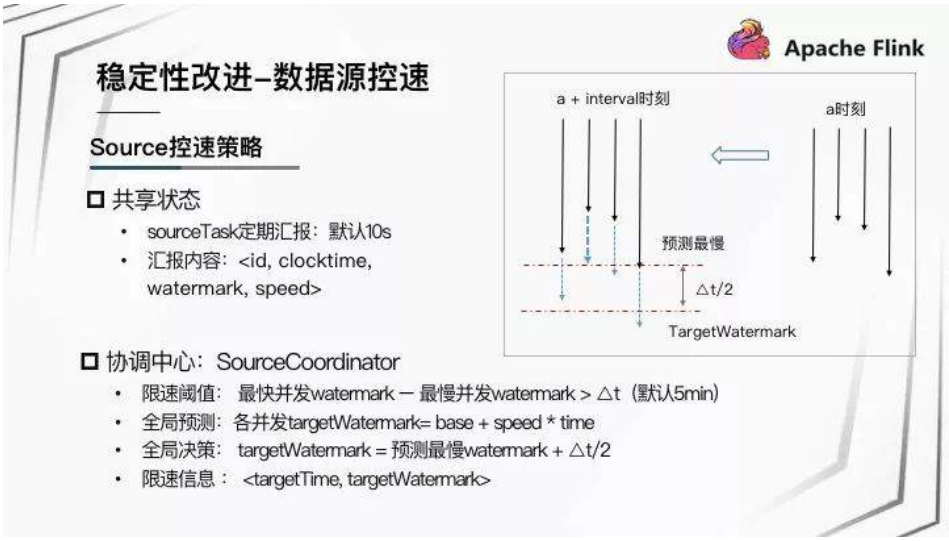
针对该问题我们希望从源头控制多个 Source 并发读取速度，所以设计了从 Source 源控速的策略。

Source 控速策略

Source 控速策略是：

- SourceTask 共享速度状态 <id,ctime,watermark,speed> 提供给 JobManager。
- JobManager 引入 SourceCoordinator，该 Coordinator 拥有全局速度视角，制定相应的策略，并将限速策略下发给 SourceTask。
- SourceTask 根据 JobManager 下发的速度调节信息执行相应控速逻辑。
- 一个小细节是 DAG 图有子图的话，不同子图 Source 源之间互相不影响。

Source 控速策略详细细节



SourceTask 共享状态

- SourceTask 定期汇报状态给 JobManager，默认 10 s 间隔。
- 汇报内容为 <id,clocktime,watermark,speed>。

协调中心 SourceCoordinator

- 限速阈值：最快并发 Watermark − 最慢并发 Watermark > Δt(默认 5 分钟)。只要在达到限速阈值情况下，才进行限速策略制定。

- 全局预测：各并发 targetWatermark=base+speed*time；Coordinator 先进行全局预测，预测各并发接下来时间间隔能运行到的 Watermark 位置。
- 全局决策：targetWatermark = 预测最慢 Watermark+Δt/2；Coordinator 根据全局预测结果，取预测最慢并发的 Watermark 值再浮动一个范围作为下个周期全局限速决策的目标值。
- 限速信息下发：<targetTime,targetWatermark>。将全局决策的信息下发给所有的 Source task，限速信息包括下一个目标的时间和目标的 Watermark 位置。

以上图为例，A 时刻，4 个并发分别到达如图所示位置，为 A+interval 的时刻做预测，图中蓝色虚线为预测各并发能够到达的位置，选择最慢的并发的 Watermark 位置，浮动范围值为 Watermark + Δt/2 的时间，图中鲜红色虚线部分为限速的目标 Watermark，以此作为全局决策发给下游 Task。

SourceTask 限速控制

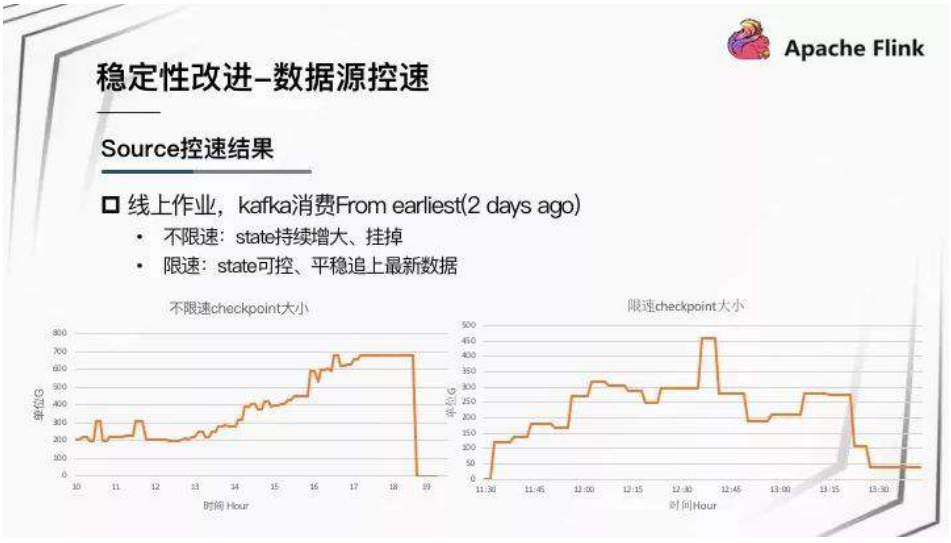
- SourceTask 获取到限速信息 <targetTime,targetWatermark> 后，进行限速控制。

- 以 KafkaSource 为例，KafkaFetcher 获取数据时，根据限速信息 Check 当前进度，确定是否需要限速等待。

该方案中，还有一些其他考虑，例如：

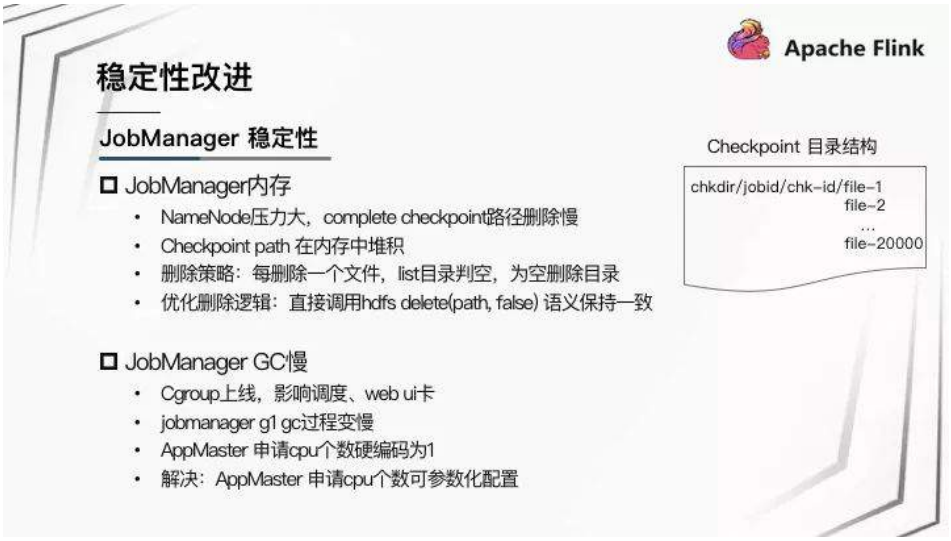
- 时间属性：只针对 EventTime 情况下进行限速执行。
- 开关控制：支持作业开关控制是否开启 Source 限速策略。
- DAG 子图 Source 源之间互相不影响。
- 是否会影响 CheckPoint Barrier 下发。
- 数据源发送速度不恒定，Watermark 突变情况。

Source 控速结果



拿线上作业，使用 Kafka 从最早位置（2 days ago）开始消费。如上图，不限速情况下 State 持续增大，最终作业挂掉。使用限速策略后，最开始 State 有缓慢上升，但是 State 大小可控，最终能平稳追上最新数据，并 State 持续在 40 G 左右。

2.2 JobManager 稳定性

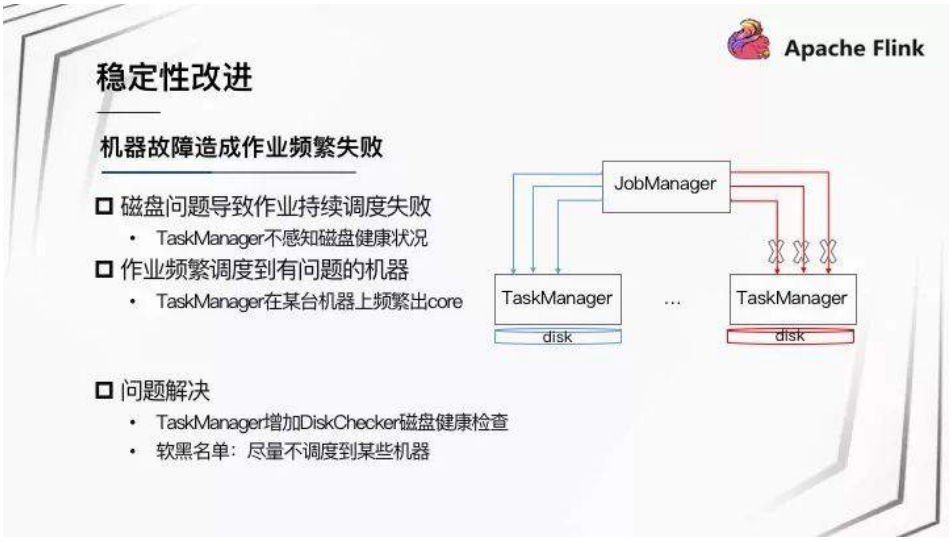


关于 JobManager 稳定性，遇到了两类 Case，表现均为：JobManager 在大并发作业场景 WebUI 卡顿明显，作业调度会超时。进一步分析了两种场景下的问题原因。

场景一，JobManager 内存压力大问题。JobManager 需要控制删除已完成的 Checkpoint 在 HDFS 上的路径。在 NameNode 压力大时，Completed CheckPoint 路径删除慢，导致 CheckPoint Path 在内存中堆积。原来删除某一次 Checkpoint 路径策略为：每删除目录下一个文件，需 List 该目录判断是否为空，如为空将目录删除。在大的 Checkpoint 路径下，List 目录操作为代价较大的操作。针对该逻辑进行优化，删除文件时直接调用 HDFS delete(path,false) 操作，语义保持一致，并且开销小。

场景二，该 Case 发生在 Yarn Cgroup 功能上线之后，JobManager G1 GC 过程变慢导致阻塞应用线程。AppMaster 申请 CPU 个数硬编码为 1，在上线 Cgroup 之后可用的 CPU 资源受到限制。解决该问题的方法为，支持 AppMaster 申请 CPU 个数参数化配置。

2.3 作业频繁失败



机器故障造成作业频繁失败，具体的场景也有两种：

场景一：磁盘问题导致作业持续调度失败。磁盘出问题导致一些 Buffer 文件找不到。又因为 TaskManager 不感知磁盘健康状况，会频繁调度作业到该 TaskManager，作业频繁失败。

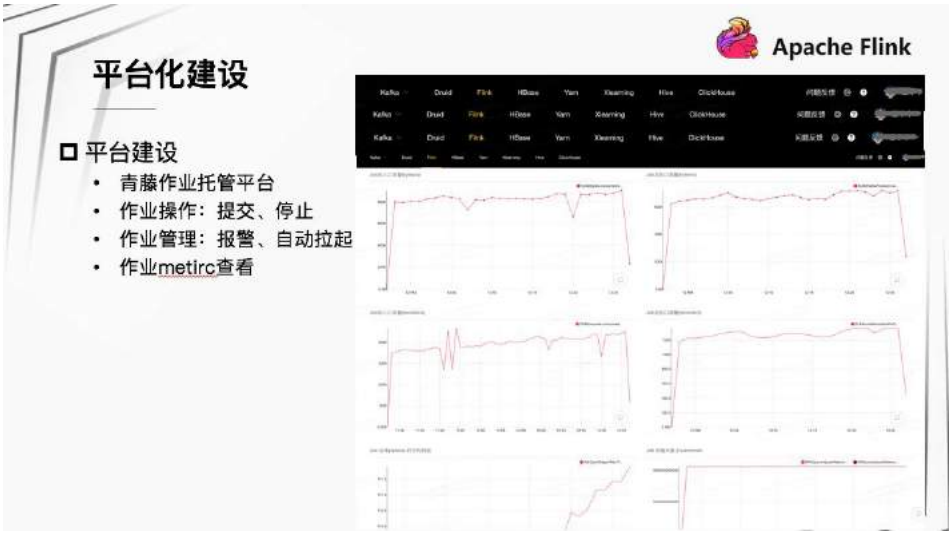
场景二：某台机器有问题导致 TaskManager 在某台机器上频繁出 Core，陆续分配新的 TaskManager 到这台机器上，导致作业频繁失败。

针对机器故障问题解决方法：

- 针对磁盘问题，TaskManager 增加 DiskChecker 磁盘健康检查，发现磁盘有问题 TaskManager 自动退出；
- 针对有些机器频繁出现 TaskManager 出现问题，根据一定的策略将有问题机器加到黑名单中，然后通过软黑名单机制，告知 Yarn 尽量不要调度 Container 到该机器。

3. 平台化建设

3.1 平台建设



快手的平台化建设主要体现在青藤作业托管平台。通过该平台可进行作业操作、作业管理以及作业详情查看等。作业操作包括提交、停止作业。作业管理包括管理作业存活、性能报警，自动拉起配置等；详情查看，包括查看作业的各类 Metric 等。

上图为青藤作业托管平台的一些操作界面。

3.2 问题定位流程优化

我们也经常需要给业务分析作业性能问题，帮助业务 debug 一些问题，过程相对繁琐。所以该部分我们也做了很多工作，尽量提供更多的信息给业务，方便业务自主分析定位问题。

首先，我们将所有 Metric 入 Druid，通过 Superset 可从各个维度分析作业各项指标。

第二，针对 Flink 的 WebUI 做了一些完善，支持 Web 实时打印 jstack，Web DAG 为各 Vertex 增加序号，Subtask 信息中增加各并发 SubtaskId。

第三，丰富异常信息提示，针对机器宕机等特定场景信息进行明确提示。

第四，新增各种 Metric。

三、未来计划

快手的未来规划主要分为两个部分：

第一，目前在建设的 Flink SQL 相关工作。因为 SQL 能够减少用户开发的成本，包括我们现在也在对接实时数仓的需求，所以 Flink SQL 是我们未来计划的重要部分之一。

第二，我们希望进行一些资源上的优化。目前业务在提作业时存在需求资源及并发预估不准确的情况，可能会过多申请资源导致资源浪费。另外如何提升整体集群资源的利用率问题，也是接下来需要探索的问题。

bilibili 实时平台的架构与实践

作者：郑志升（bilibili 大数据实时平台负责人）

摘要：本次分享基于 bilibili 实时计算的痛点，展开对 bilibili Saber 实时计算平台架构与实践的介绍。Saber 是一套基于 Flink 引擎的实时计算平台，对外提供了统一的元数据管理、血缘、权限管理以及作业运维等功能支持。同时上层基于自研的 Saber-BSQL 层，极大简化了 Instance 流的构建，解决了 Streaming Join Streaming（流 式 SJoin），Streaming Join Table（维 表 DJoin），Real-time Feature（实时特征）等三大数据预处理问题。

本次分享主要围绕以下四个方面：

一、实时计算的痛点

二、Saber 的平台演进

三、结合 AI 的案例实践

四、未来的发展与思考

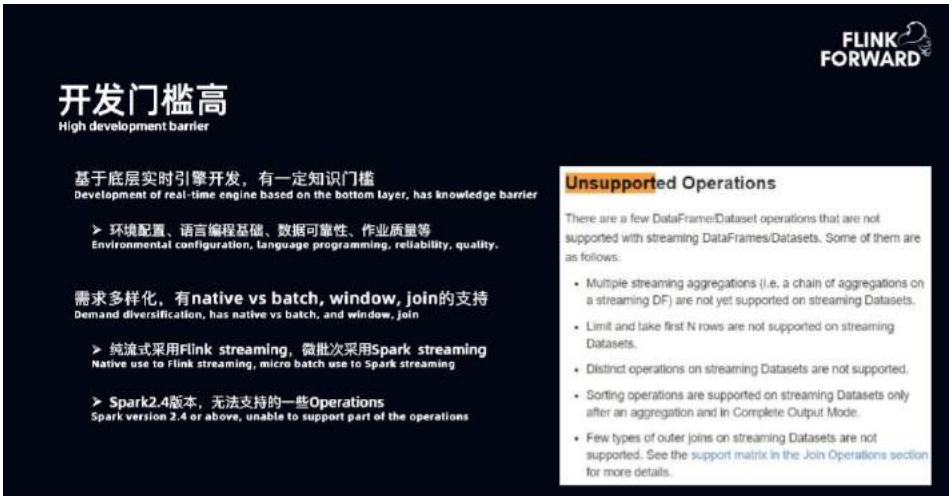
一、背景——实时计算的痛点

1. 痛点

各个业务部门进行业务研发时都有实时计算的需求。早期，在没有平台体系做支撑时开发工作难度较大，由于不同业务部门的语言种类和体系不同，导致管理和维护非常困难。其次，bilibili 有很多关于用户增长、渠道投放的分析等 BI 分析任务。而且还需要对实时数仓的实时数据进行清洗。此外，bilibili 作为一个内容导向的视频网站，AI 推荐场景下的实时计算需求也比较强烈。

2. 痛点共性

- **开发门槛高：**基于底层实时引擎做开发，需要关注的东西较多。包括环境配置、语言基础，而编码过程中还需要考虑数据的可靠性、代码的质量等。其次，市场实时引擎种类多样，用户选择有一定困难。



- **运维成本高：**运维成本主要体现在两方面。首先是作业稳定性差。早期团队有 Spark 集群、YARN 集群，导致作业稳定性差，容错等方面难以管理。其次，缺乏统一的监控告警体系，业务团队需要重复工作，如计算延时、断流、波动、故障切换等。

- **AI 实时工程难：**bilibili 客户端首页推荐页面依靠 AI 体系的支撑，早期在 AI 机器学习方面遇到非常多问题。机器学习是一套算法与工程交叉的体系。工程注重的是效率与代码复用，而算法更注重特征提取以及模型产出。实际上 AI 团队要承担很多工程的工作，在一定程度上十分约束实验的展开。另外，AI 团队语言体系和框架体系差异较大，所以工程是基建体系，需要提高基建才能加快 AI 的流程，降低算法人员的工程投入。

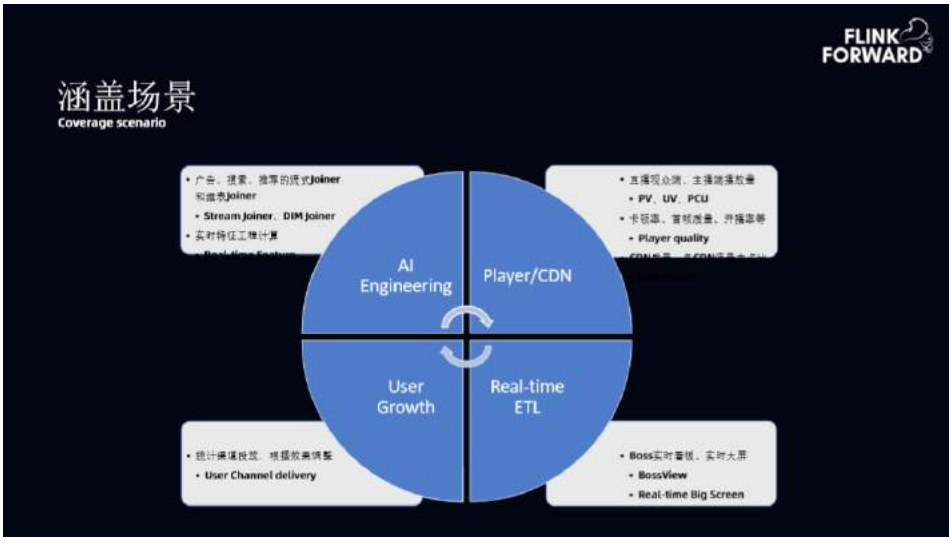
3. 基于 Apache Flink 的流式计算平台

为解决上述问题，bilibili 希望根据以下三点要求构建基于 Apache Flink 的流式计算平台。

- 第一，需要提供 SQL 化编程。bilibili 对 SQL 进行了扩展，称为 BSQL。BSQL 扩展了 Flink 底层 SQL 的上层，既 SQL 语法层。
- 第二点是 DAG 拖拽编程，一方面用户可以通过画板来构建自己的 Pipeline，另一方面用户也可以使用原生 Jar 方式进行编码。
- 第三点是作业的一体化托管运维。

涵盖场景：bilibili 流式计算平台主要涵盖四个方面的场景。

- AI 工程方向，解决了广告、搜索、推荐的流式 Joiner 和维表 Joiner；
- 实时计算的特征支持，支持 Player 以及 CDN 的质量监控。包括直播、PCU、卡顿率、CDN 质量等；
- 用户增长。即如何借助实时计算进行渠道分析、调整渠道投放效果；
- 实时 ETL，包括 Boss 实时播报、实时大屏、看板等。

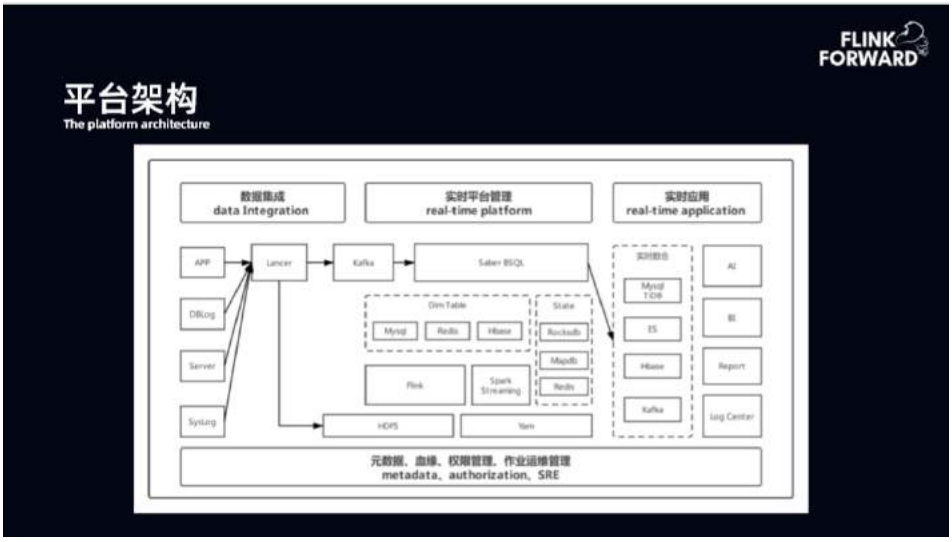


二、Saber 的平台演进

1. 平台架构

实时平台由实时传输和实时计算两部分组成，平台底层统一管理元数据、血缘、权限以及作业运维等。实时传输主要负责将数据传入到大数据体系中。实时计算基于 BSQL 提供各种应用场景支持。

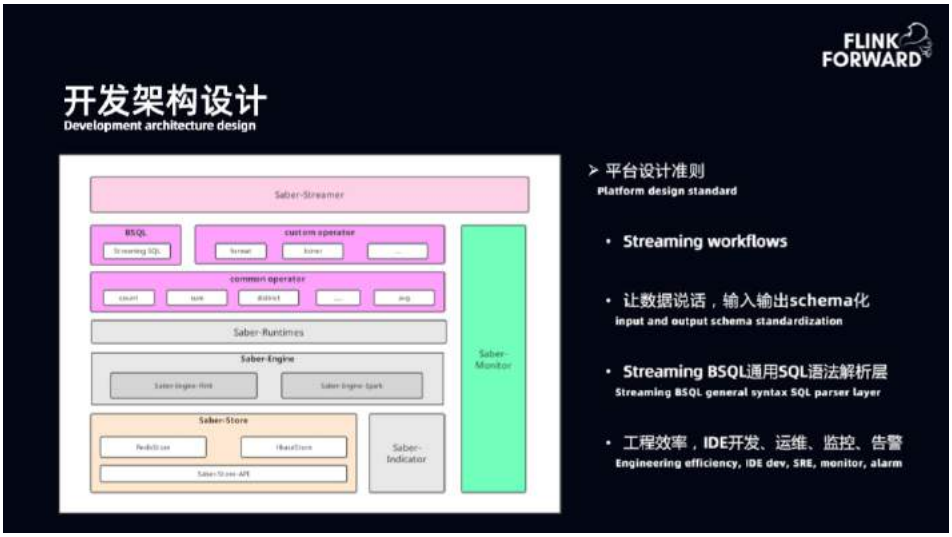
如下图所示，实时传输有 APP 日志、数据库 Binlog、服务端日志或系统日志。bilibili 内部的 Lancer 系统解决数据落地到 Kafka 或 HDFS。计算体系主要围绕 Saber 构建一套 BSQL，底层基于 YARN 进行调度管理。上层核心基于 Flink 构建运行池。再向上一层满足多种维表场景，包括 MySQL、Redis、HBase。状态（State）部分在 RocksDB 基础上，还扩展了 MapDB、Redis。Flink 需要 IO 密集是很麻烦的问题，因为 Flink 的资源调度体系内有内存和 CPU，但 IO 单位未做统一管理。当某一个作业对 IO 有强烈的需求时，需要分配很多以 CPU 或内存为单位的资源，且未必能够很好的满足 IO 的扩展。所以本质上 bilibili 现阶段是将对 IO 密集的资源的 State 转移 Redis 上做缓解。数据经过 BSQL 计算完成之后传输到实时数仓，如 Kafka、HBase、ES 或 MySQLTiDB。最终到 AI 或 BI、报表以及日志中心。



2. 开发架构设计

（1）**开发架构图：**如下图左侧所示。最上层是 Saber-Streamer，主要进行作业提交以及 API 管理。下一层是 BSQL 层，主要进行 SQL 的扩展和解析，包括自定义算子和个性算子。再下层是运行时态，下面是引擎层。运行时态主要管理引擎层作业的上下层。bilibili 早期使用的引擎是 Spark Streaming，后期扩展了 Flink，在开发架构中预留了一部分引擎层的扩展。最下层是状态存储层，右侧为指标监控模块。

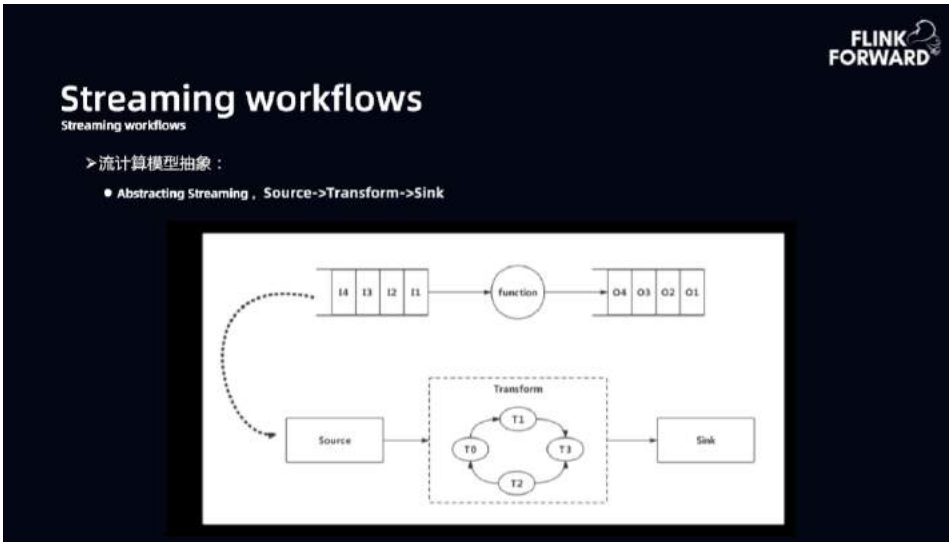
（2）**平台设计准则：**Saber 平台系统设计时团队关注其边界以及规范和准则，有以下四个关键点。第一是对 Streaming workflows 进行抽象。第二是数据规范性，保证 schema 完整。第三是通用的 BSQL 解析层。第四是工程效率。



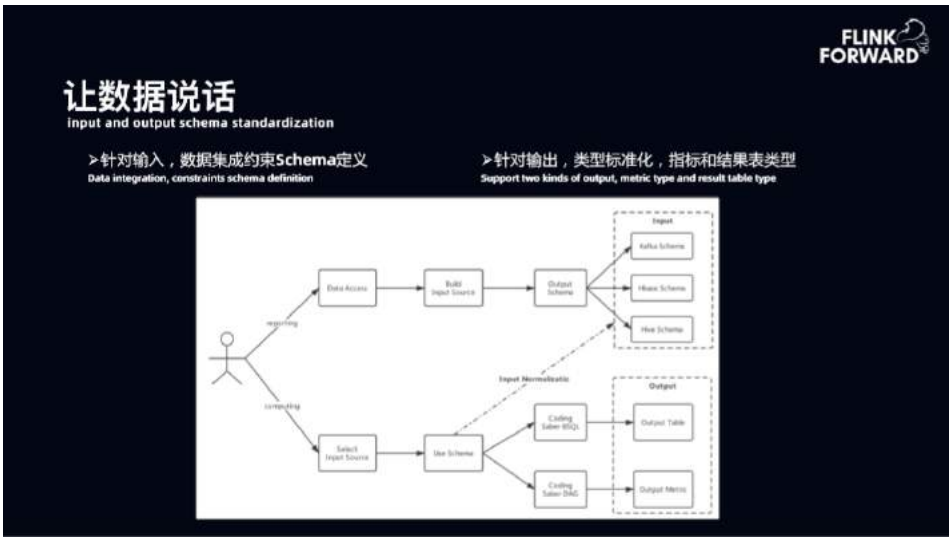
- **Streaming workflows：**下图为流计算模型抽象。大数据计算引擎的本质是数据输入经过一个 function 得到输出，所以 function 本质是一个能够做 DAG 转换的 Transform。Saber 平台期望的流计算抽象形态是提供相应的 Source，计算过程中是一个 Transform 的 DAG，最后有一个 Sink 的输出。

在上述抽象过程中规范语义化标准。即最后输入、输出给定规范标准，底层通

过 Json 表达方式提交作业。在没有界面的情况下，也可以直接通过 Json 方式拉起作业。

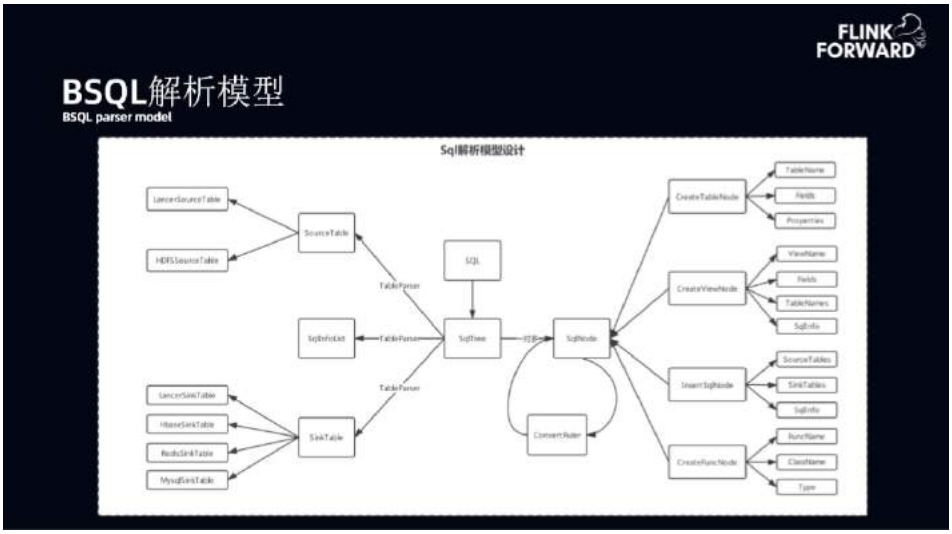


- **让数据说话：**数据抽象化。计算过程中的数据源于数据集成的上报。数据集成的上报有一套统一的平台入口。用户首先需要在平台上构建一个输入的数据源。用户选择了一个对应的数据源，平台可以将其分发到 Kafka、HBase、Hive 等，并且在分发过程中要求用户定义 Schema。所以在数据集成过程中，可以轻松地管理输入语言的 Schema。计算过程中，用户选择 Input Source，比如选择一个 HBase 的表或 Kafka 的表，此时 Schema 已是强约束的。用户通过平台提供的 BSQL 或者 DAG 的方式进行结果表或者指标的输出。

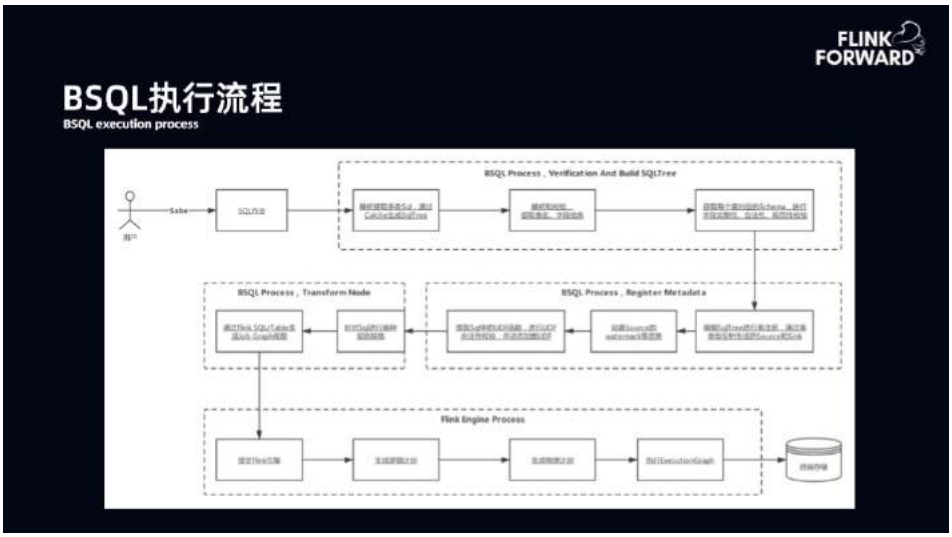


- **BSQL 通用设计：**BSQL 是遵照 Streaming workflows 设计的思想，核心工作围绕 Source、Transform 以及 Sink。Transform 主要依托 Flink SQL，所以 BSQL 更多是在 Source 和 Sink 上进行分装，支持 DDL 的分装。此处 DDL 参照阿里云对外资料进行了扩展。另外，BSQL 针对计算过程进行了优化，如针对算子计算的数据倾斜问题采取分桶 +hash 策略进行打扫。针对 distinct 类 count，非精准计算采用 Redis 的 HyperLogLog。

- **BSQL 解析模型：**BSQL 解析模型拓扑展开如下图。当用户提交了一个 SQL，目标是将 SQL 转化成树。之后可以获取 SqlNode 节点。SqlNode 节点中有很多元数据信息。在 SqlNode 树的情况下实现 Table 解析器，将不同的 SqlNode 节点转化成 Flink 相应的 Streamer 进行映射。

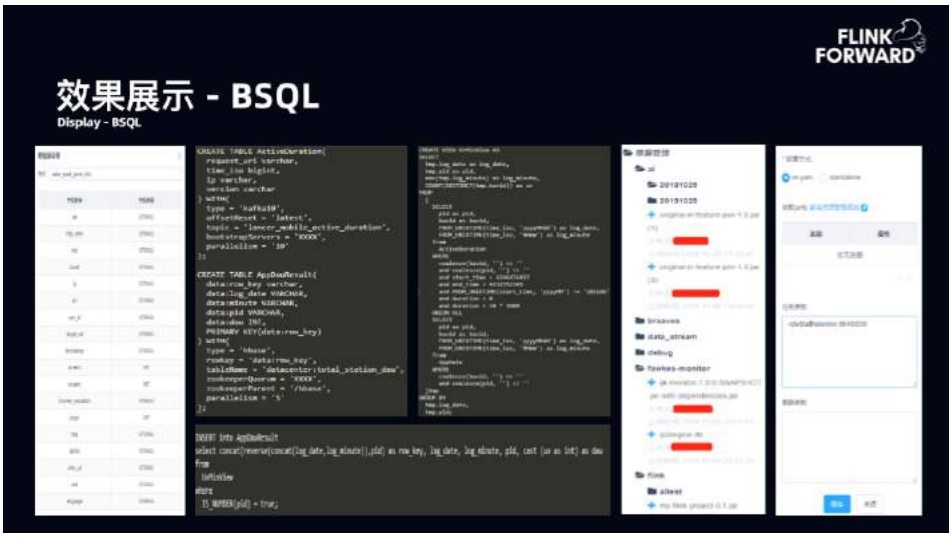


- **BSQL 执行流程：**用户提交 SQL，BSQL 首先进行验证并构建 SQL 树。验证与构建主要是提取表名、字段信息，从元数据库中提取 schema 验证 SQL 的规范性、完整性和合法性。验证完成后，将输入表和结果表注册到 Flink 的运行时态，其中还包括 UDF 和 watermark 信息的完善。另外，平台对 SQL 有一些扩展。第三块是扩展的核心工作，将 SQL 树中扩展的子树转换为新的节点，然后将 SQL 的 DAG 提交到 Flink 上运行。



- **效果展示 –DAG：**如下图所示，DAG 产品展示，包括并行度的设计、日志、监控指标告警输出。

- **效果展示 –BSQL：**用户根据选择的表的输入源的 schema 编写相应的 SQL。最后选择相应 UDF 就可以提交到相应集群。



- **效果展示 – 作业调试：**如下图所示为平台支持的作业调试。如果只有 SQL 开发却没有作业调试环节，是令用户痛苦的。故平台支持通过文件上传的方式以及线上采样的方式进行作业调试 SQL。

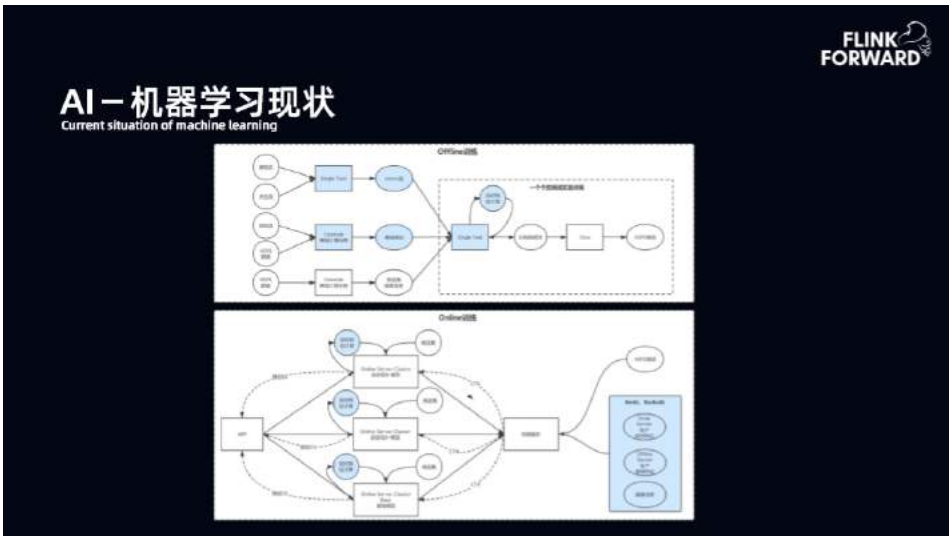
- **效果展示 – 作业运维：**平台提供给用户一些监控指标、用户可自定义扩展的指标以及 bilibili 实现的一些特殊 SQL 的自定义指标。下图所示为部分队列的运

行情况。

三、结合 AI 的案例实践

1. AI– 机器学习现状

AI 体系中有 Offline 和 Online 过程。Online（线上训练）根据流量做 A/B 实验，根据不同实验的效果做推荐。同时每个实验需要有相应的模型 push 到线上。AI 的痛点集中在 Offline（离线训练）。Offline 则通过流式方式进行训练。下图是 Offline 流式训练早期情况。用户需要构建流和流的实时 join，从而产出实时 label 流。而流和维表及特征信息的 join 来产出实时 instance 流，但早期相关的工程服务存在着单点问题，服务质量、稳定性带来的维护成本也很高，致使 AI 在早期 Pipeline 的构建下投入非常大。



2. 弊端与痛点

- **数据时效性：**数据时效性无法得到保证。很多数据是通过离线方式进行计算，但很多特征的时效性要求非常高。
- **工程质量：**单点工程不利于服务扩展以及稳定性保障。
- **工程效率：**每一个实验都有较高门槛，需要做 Label 生产，Features 计算以及 Instance 拼接。在不同业务线，不同场景的推荐背后，算法同学做工程工作。他们掌握的语言不同，导致工程上语言非常乱。另外，流、批不一致，模型的训练在实时环境与离线批次环境的工程差异很大，其背后的逻辑相似，导致人员投入翻倍增长。

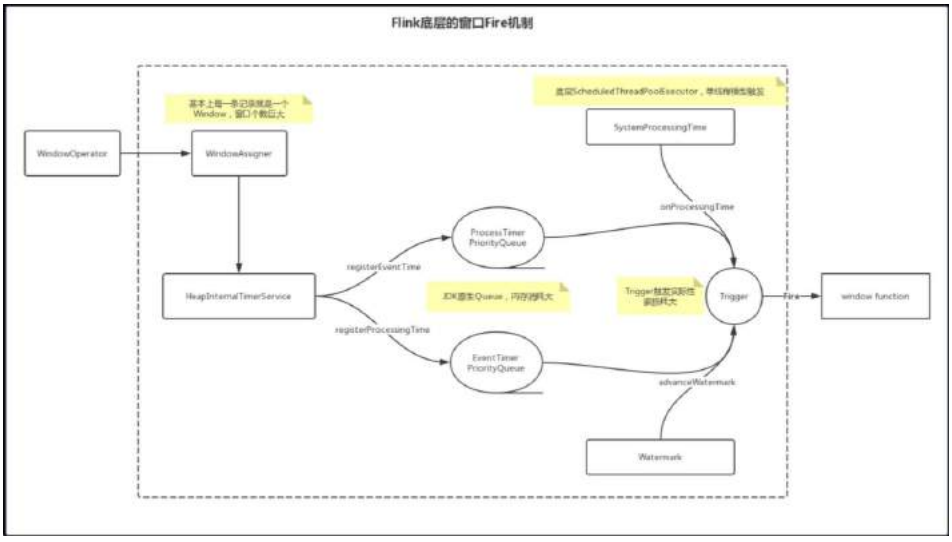
3. 模型训练的工程化

构建一套基于 Saber-BSQL、Flink 引擎的数据计算 Pipeline，极大简化 Instance 流的构建。其核心需要解决以下三个问题：Streaming Join Streaming（流式 SJoin），Streaming Join Table（维表 DJoin），Real-time Feature（实时特征）。

- **SJoin- 工程背景：**流量规模大，如 bilibili 首页推荐的流量，AI 的展现点击 Join，来自全站的点击量和展现。此外，不仅有双流 Join，还有三流及以上的 Join，如广告展现流、点击流、搜索查询流等。第三，不同 Join 对 ETL 的清洗不同。如果不能通过 SQL 的方式进行表达，则需要为用户提供通用的扩展，解决不同业务对 Join 之前的定制化 ETL 清洗。第四，非典型 A Left Join B On Time-based Window 模型。主流 A 在窗口时间内 Join 成功后，需要等待窗口时间结束再吐出数据，延长了主流 A 在窗口的停留时间。此场景较为关键，bilibili 内部不仅广告、AI、搜索，包括直播都需要类似的场景。因为 AI 机器学习需要正负样本均匀以保证训练效果，所以第四点问题属于强需求。
- **SJoin- 工程规模：**基于线上实时推荐 Joiner。原始 feed 流与 click 流，QPS 高峰分别在 15w 和 2w，Join 输出 QPS 高峰达到 10w，字节量高峰为 200M/s。keyState 状态查询量维持在高峰值 60w，包括 read、write、exist 等状态。一小时 window 下，Timer 的 key 量 15w*3600=54 亿条，RocksDBState 量达到 200M*3600=700G。实际过程中，采用原生 Flink 在该规模下会遇到较多的性能问题，如在早期 Flink1.3.* 版本，其稳定性会较差。

- **SJoin- 技术痛点：**下图是 Flink 使用 WindowOperator 时的内部拓扑图。用户打开窗口，每一条记录都是一个 Window 窗口。第一个问题是窗口分配量巨大，QPS 与窗口分配量基本持恒。第二个问题是 Timer Service 每一个记录都打开了一个窗口，在早期原生 Flink 中是一个内存队列，内存队列部分也存在许多问题。底层队列早期是单线程机制，数据 Cache 在内存中，存在许多问题。

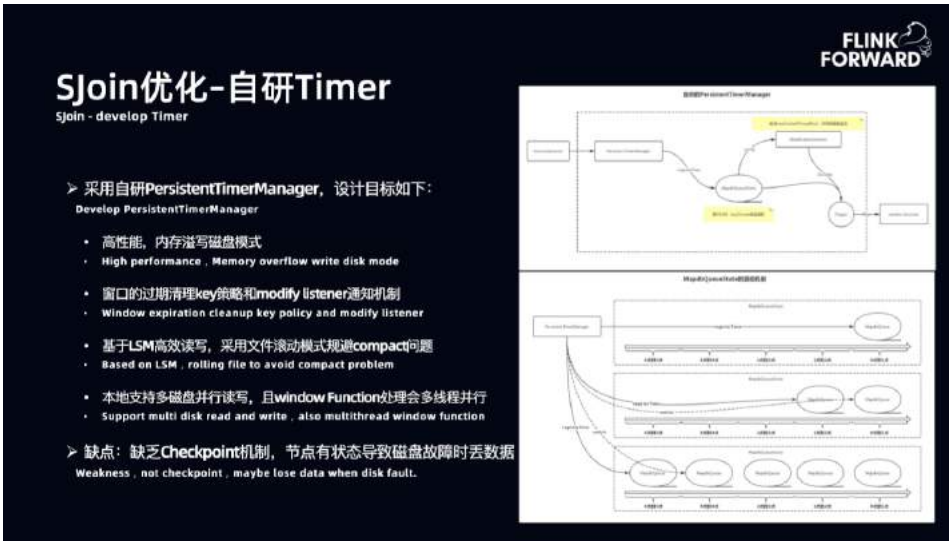
简单总结其技术痛点，首先是 Timer 性能较差，且内存消耗大。第二，Value RocksDB State 在 compact 时会导致流量抖动。类似 HBase，多 level 的 compact 会造成性能抖动和写放大。第三，重启流量过大时，由于 Timer 早期只有内存队列，Window 和 Keystate 恢复周期不可控。从磁盘加载大量数据耗时长，服务 recovery 时间久。



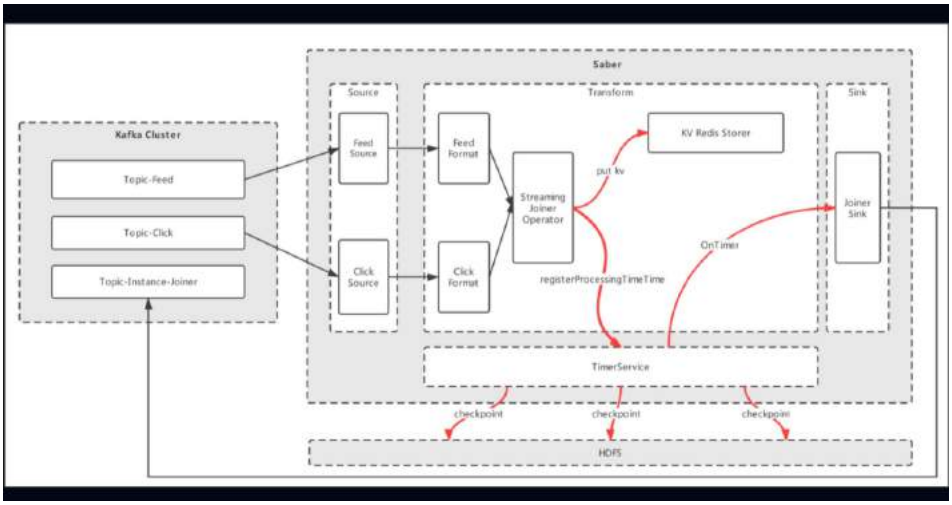
- **SJoin- 优化思路：**首先是 Timer 优化升级。早期社区没有更好的解决方案时，bilibili 尝试自研 PersistentTimerManager，后期升级 Flink，采用基于 RocksDB 的 Timer。第二，启用 Redis 作为 ValueState，提高 State 稳定性。第三，扩展 SQL 语法，以支持非典型 ALeft Join B On Time-based

Window 场景下的 SQL 语义。

- **SJoin 优化 – 自研 Timer：**实现将内存数据达到 Max 之后溢写到磁盘。底层用 MapDB 做磁盘溢写。磁盘溢写原理是 LSM 模型，同样存在数据抖动问题。由于窗口是 1 小时，相当于数据以 1 小时为单位进行 State 管理。如下图右侧所示，当 0 点到 1 点的 1 小时，由于记录在 1 小时后才会吐出，数据进来只有写的动作。在 1 点到 2 点，数据会写入到新的 State，0 点到 1 点的 State 已经到达窗口时间，进行数据吐出。自研 Timer 很好地解决了数据的读写问题和抖动问题。但是由于自研 Timer 缺乏 CheckPoint 机制，如果节点上的磁盘出现故障，会导致 State 数据丢失。



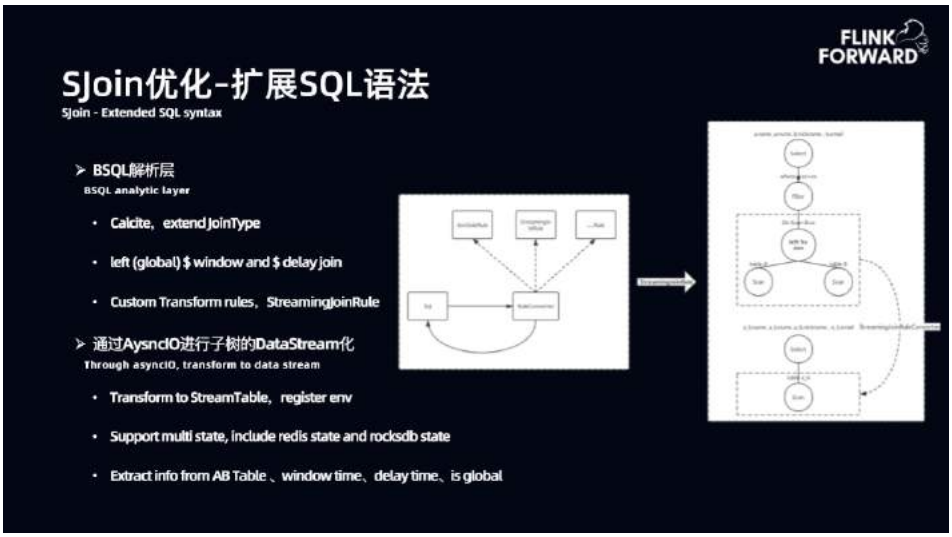
- **SJoin 优化 –RocksDBTimer：**升级 Flink 版本，引入基于 RocksDB 的 Timer。升级后架构如下图所示。数据从 Kafka 获取 Topic-Feed 和 Topic-Click，首先对其进行一层清洗，然后进入自定义的 Joiner Operator 算子。算子做两件事，将主流数据吐到 Redis 中，由 Redis 做 State，同时将需要开窗口的 Key 存储注册到 Timer Service 中。接下来利用 Timer Service 原生的 CheckPoint 开启增量 CheckPoint 过程。当 OnTimer 到达时间后，就可以吐出数据。非常此方案契合 SJoin 在高吞吐作业下的要求。



- **SJoin 优化 – 引入 KVStore：** Flink 原生 State 无法满足要求，在对 Value、IO 要求高时抖动严重，RocksDBState 实际使用中会出现抖动问题。对此，bilibili 尝试过多种改进方案。开 1 小时窗口，数据量约 700G，双流 1 小时窗口总流量达到 TB 级别。采用分布式 KVStore 存储，后续进行压缩后数据量约 700G。

- **SJoin 优化 – 扩展 SQL 语法：**扩展 SQL 的功能诉求是展现流等待 1 小时窗口，当点击流到达时，不立即吐出 Join 完成的数据，而等待窗口结束后再吐出。故扩展了 SQL 语法，虽然目前未达到通用，但是能满足诸多部门的 AI 需求。语法支持 Select * from A left（global）$time window and $time delay join B on A.xx=B.xx where A.xx=xx。给用户带来了很大收益。

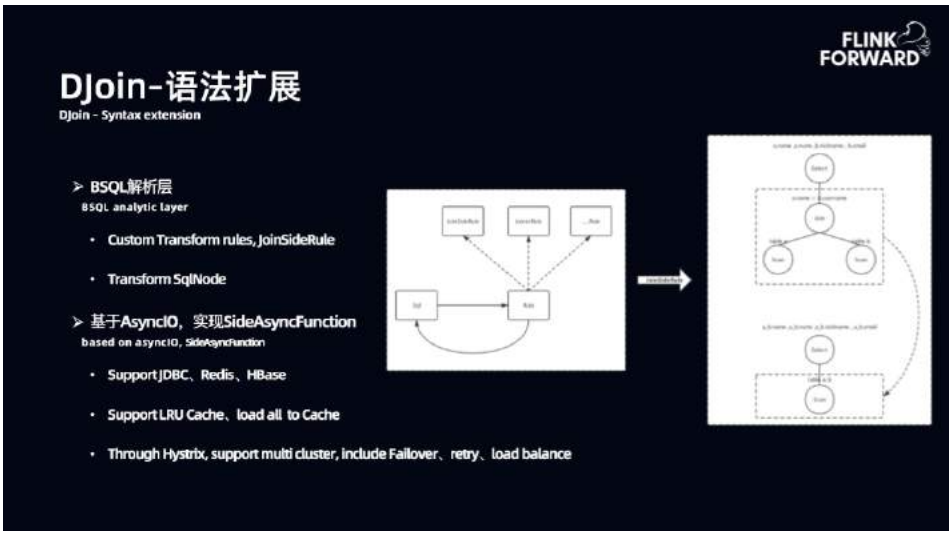
进行 SQL 语义扩展主要有两个关键点。SQL 语义的定义顶层通过 Calcite 扩展 JoinType。首先将 SQL 展开成 SQL 树。SQL 树的一个节点为 left（global）$time window and $time delay join。抽取出该子树，自定义逻辑转换规则。在此定义了 StreamingJoinRute，将该子树转换为新的节点。通过 Flink 提供的异步 IO 能力，将异步子树转换为 Streaming Table，并将其注册到 Flink 环境中。通过以上过程支持 SQL 表达。



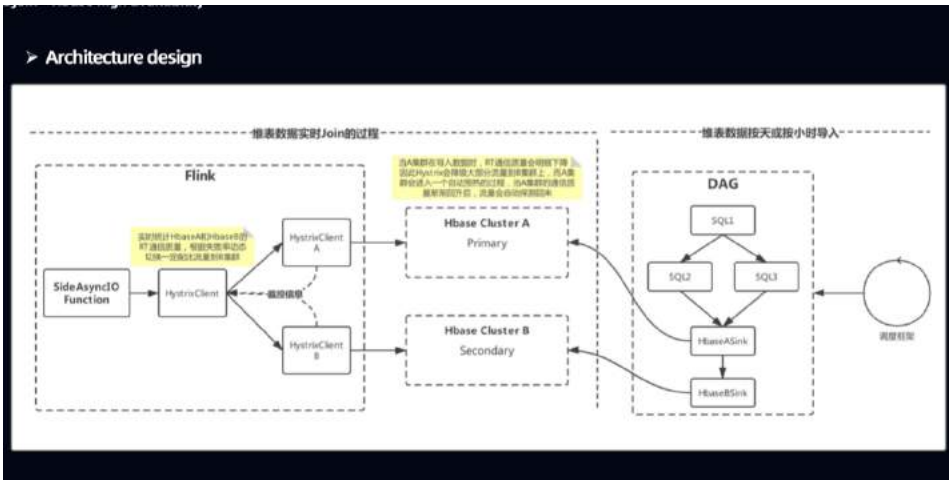
- **DJoin– 工程背景：** bilibili 对于维表数据要求不同。比如一些维表数据很大，以 T 为单位，此时如果用 Redis 存储会造成浪费。而有一些维表数据很小，如实时特征。同时，维表数据更新粒度不同，可以按天更新、按小时更新、按分钟更新等。另外，维表性能要求很高。因为 AI 场景会进行很多实验，例如某一个特征比较好，就会开很多模型、调整不同参数进行实验。单作业下实验组越多，QPS 越高，RT 要求越高。不同维表存储介质有差异，对稳定性有显著影响。调研中有两种场景。当量比较小，可以使用 Redis 存储，稳定性较好。当量很大，使用 Redis 成本高，但 HBase CP 架构无法保证稳定性。

- **DJoin- 工程优化：**需要针对维表 Join 的 SQL 进行语法支持。包括 Cache 优化，当用户写多条 SQL 的维表 Join 时，需要提取多条 SQL 维表的 Key，并通过请求合并查询维表，以提高 IO，以及流量均衡优化等。第二，KV 存储分场景支持，比如 JDBC、KV。KV 场景中，对百 G 级别使用 Redis 实时更新实时查询。T 级别使用 HBase 多集群，比如通过两套 HBase，Failover+LoadBalance 模式保证 99 线 RT 小于 20ms，以提高稳定性。

- **DJoin- 语法扩展：**DJoin 语法扩展与 SJoin 语法扩展类似，对 SQL 树子树进行转化，通过 AsyncIO 进行扩展，实现维表。

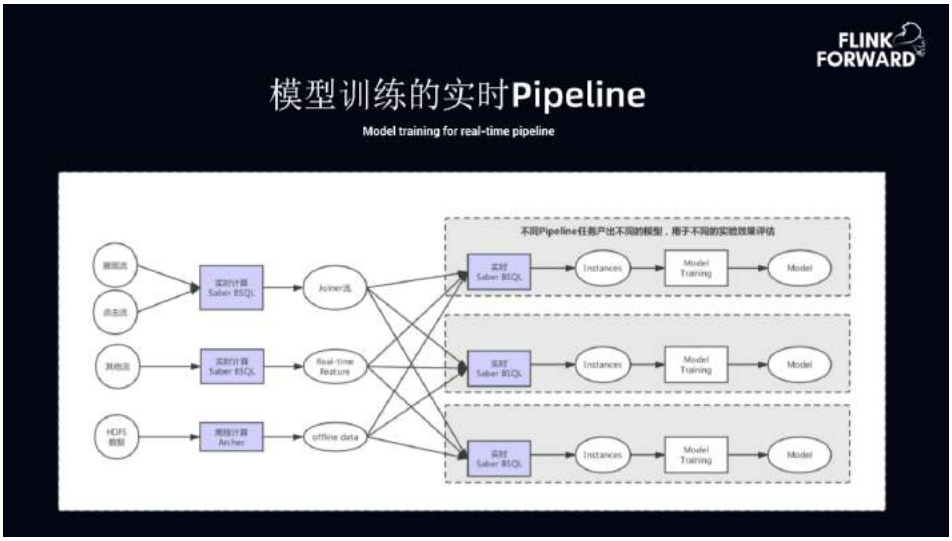


- **DJoin-HBase 高可用：**维表数据达到 T 级别时使用 HBase 进行数据存储。HBase 高可用性采用双 HBase 集群，Failover AB 模式。这时需要考虑两个问题。第一是数据更新机制。数据更新可以是按小时或按天，采用 HFile BulkLoad 模式，串行 +Interval 间隔导入，导入后同步数据预热，以此保证两套 HBase 集群的稳定性。第二是数据查询机制。引入 Hystrix 实现服务熔断、降级回退策略。当 A 集群可用性下降时，根据 AB 的 RT 质量，动态切换一定数据到 B 集群，以保证数据流量均衡。下图为 HBase 双集群架构。右侧是离线，以天为单位，通过调度框架拉起一个 DAG 进行计算。DAG 的输出经过两层串行的 HBase 的 Sink，串行可以保证数据先写完 A 再写 B。运行时态中通过 Flink、AsyncIO 方式，通过两层 HystrixClient。第一层 HystrixClient 主要对第二层 HystrixClient HBase 的 RT 通信质量进行收集，根据 RT 通信质量将流量动态分发到两套 HBase 集群中。在 A 集群稳定性很好时，流量都在 A 集群跑。当 A 集群出现抖动，会根据失败率动态切换一定配比流量到 B 集群。



4. 模型训练的实时 Pipeline

整个体系解决了 AI 模型训练预生成数据给模型的 Pipeline。展现和点击通过 BSQL 方案实现 Joiner。实时特征数据通过 BSQL 进行计算，离线数据通过离线调度解决。维表的 Join 会通过 BSQL 构成 Pipeline，从而给机器学习团队 Instances 流，训练模型，产出模型。



四、未来的发展与思考

1. Saber- 基础功能完善

越来越多人使用平台时，基础运维是最为关键的。Saber 平台将会完善 SQL IDE 开发，如提供更丰富的版本管理、上下线、任务调试、资源管理、基础操作等。同时将丰富化作业运维。包括 SLA、上线审批、优先级、各类系统监控指标、用户自定义指标告警、作业 OP 操作等。

2. Saber- 应用能力提升

Saber 应用能力将会向 AI 方向不断演进。例如模型训练的工程化方面，将引入实验维度概念，通过实验拉起 SQL Pipeline。同时将为做模型训练的同学统一流、批 SQL 复用。并且进行模型实验效果、评估、预警等。实时特征的工程化方面，将会支持多特征复合计算，涵盖特征计算、存储、查询等多个场景。

美团点评基于 Apache Flink 的实时数仓平台实践

作者：鲁昊（美团点评高级技术专家）

摘要：数据仓库的建设是“数据智能”必不可少的一环，也是大规模数据应用中必然面临的挑战，而 Flink 实时数仓在数据链路中扮演着极为重要的角色。本文中，美团点评高级技术专家鲁昊为大家分享了美团点评基于 Apache Flink 的实时数仓平台实践。

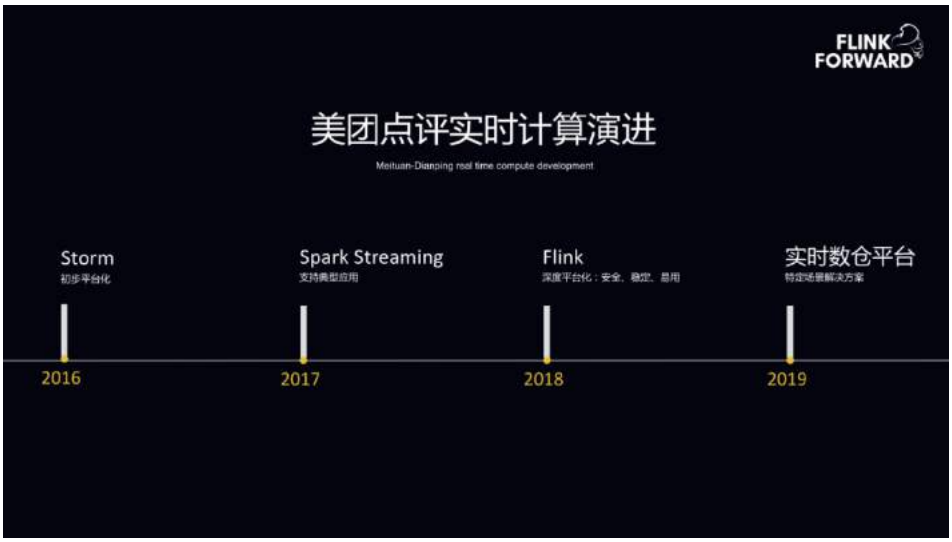
主要内容为以下三个方面：

1. 实时计算演进与业务实践
2. 基于 Flink 的实时数仓平台
3. 未来发展与思考

一、美团点评实时计算演进

美团点评实时计算演进历程

在 2016 年，美团点评就已经基于 Storm 实时计算引擎实现了初步的平台化。2017 年初，我们引入了 Spark Streaming 用于特定场景的支持，主要是在数据同步场景方面的尝试。在 2017 年底，美团点评实时计算平台引入了 Flink。相比于 Storm 和 Spark Streaming，Flink 在很多方面都具有优势。这个阶段我们进行了深度的平台化，主要关注点是安全、稳定和易用。从 19 年开始，我们致力于建设包括实时数仓、机器学习等特定场景的解决方案来为业务提供更好的支持。



实时计算平台

目前，美团点评的实时计算平台日活跃作业数量为万级，高峰时作业处理的消息量达到每秒 1.5 亿条，而机器规模也已经达到了几千台，并且有几千位用户正在使用实时计算服务。

实时计算平台架构

如下图所示的是美团点评实时计算平台的架构。

- 最底层是**收集层**，这一层负责收集用户的实时数据，包括 Binlog、后端服务日志以及 IoT 数据，经过日志收集团队和 DB 收集团队的处理，数据将会被收集到 Kafka 中。这些数据不只是参与实时计算，也会参与离线计算。
- 收集层之上是**存储层**，这一层除了使用 Kafka 做消息通道之外，还会基于 HDFS 做状态数据存储以及基于 HBase 做维度数据的存储。
- 存储层之上是**引擎层**，包括 Storm 和 Flink。实时计算平台会在引擎层为用户提供一些框架的封装以及公共包和组件的支持。
- 在引擎层之上就是**平台层**了，平台层从数据、任务和资源三个视角去管理。
- 架构的最上层是**应用层**，包括了实时数仓、机器学习、数据同步以及事件驱动应用等。

本次分享主要介绍实时数仓方面的建设情况。

从功能角度来看，美团点评的实时计算平台主要包括作业和资源管理两个方面的

功能。其中，作业部分包括作业配置、作业发布以及作业状态三个方面的功能。

- 在**作业配置**方面，则包括作业设置、运行时设置以及拓扑结构设置；
- 在**作业发布**方面，则包括版本管理、编译 / 发布 / 回滚等；
- **作业状态**则包括运行时状态、自定义指标和报警以及命令 / 运行时日志等。

在资源管理方面，则为用户提供了多租户资源隔离以及资源交付和部署的能力。

业务数仓实践

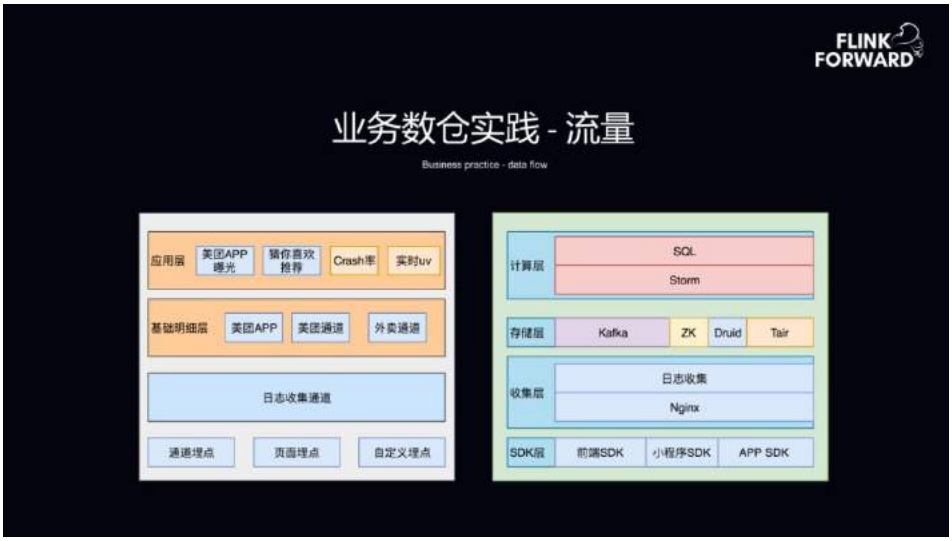
- **流量**

前面提到，现在的美团点评实时计算平台更多地会关注在**安全、易用和稳定**方面，而应用上很大的一个场景就是业务数仓。接下来会为大家分享几个业务数仓的例子。

第一个例子是流量，流量数仓是流量类业务的基础服务，从业务通道而言，会有不同通道的埋点和不同页面的埋点数据，通过日志收集通道会进行基础明细层的拆分，按照业务维度划分不同的业务通道，如美团通道、外卖通道等。

基于业务通道还会进行一次更加细粒度的拆分，比如曝光日志、猜你喜欢、推荐等。以上这些包括两种使用方式，一种是以流的方式提供下游其他业务方使用，另外一方面就是做一些流量方面的实时分析。

下图中右边是流量数仓的架构图，自下向上分为四层，分别是 SDK 层，包括了前端、小程序以及 APP 的埋点；其上是收集层，埋点日志落地到 Nginx，通过日志收集通道收到 Kafka 中。在计算层，流量团队基于 Storm 能力实现了上层的 SQL 封装，并实现了 SQL 动态更新的特性，在 SQL 变更时不必重启作业。

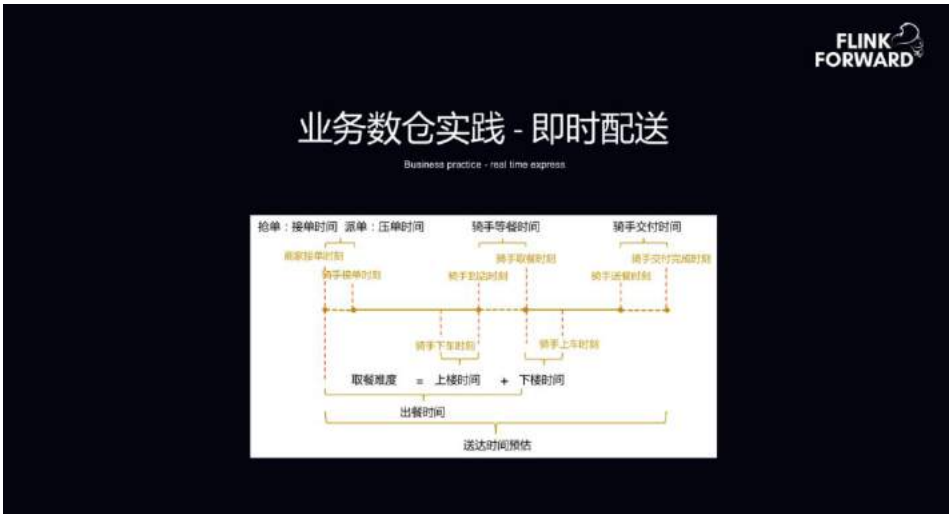


- **广告实时效果**

这里再举一个基于流量数仓的例子 – 广告实时效果验证。下图中左侧是广告实时效果的对比图。广告的打点一般分为请求（PV）打点、SPV（Server PV）打点、CPV（Client PV）曝光打点和 CPV 点击打点，在所有打点中都会包含一个流量的 requestID 和命中的实验路径。根据 requestID 和命中的实验路径可以将所有的日志进行 join，得到一个 request 中需要的所有数据，然后将数据存入 Durid 中进行分析，支持实际 CTR、预估 CTR 等效果验证。

- **即时配送**

这里列举的另外一个业务数仓实践的例子是即时配送。实时数据在即时配送的运营策略上发挥了重要作用。以送达时间预估为例，交付时间衡量的是骑手送餐的交付难度，整个履约时间分为了多个时间段，配送数仓会基于 Storm 做特征数据的清洗、提取，供算法团队进行训练并得到时间预估的结果。这个过程涉及到商家、骑手以及用户的多方参与，数据的特征会非常多，数据量也会非常大。



- **总结**

业务实时数仓大致分为三类场景：流量类、业务类和特征类，这三种场景各有不同。

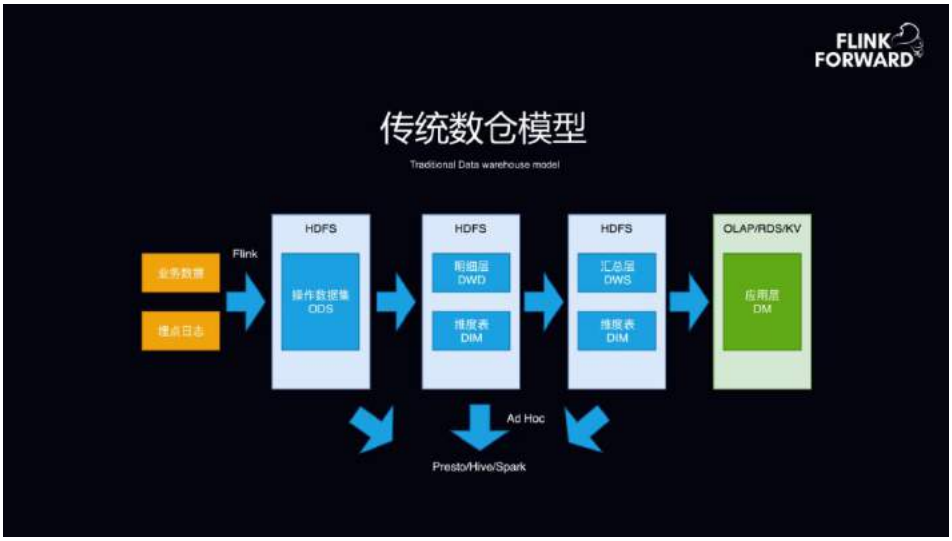
- 在**数据模型**上，流量类是扁平化的宽表，业务数仓更多是基于范式的建模，特征数据是 KV 存储。
- 从**数据来源**区分，流量数仓的数据来源一般是日志数据；业务数仓的数据来源是业务 binlog 数据；特征数仓的数据来源则多种多样。
- 从**数据量**而言，流量和特征数仓都是海量数据，每天百亿级以上，而业务数仓的数据量一般每天百万到千万级。
- 从**数据更新频率**而言，流量数据极少更新，则业务和特征数据更新较多。流量数据一般关注时序和趋势，业务数据和特征数据关注状态变更。
- 在**数据准确性**上，流量数据要求较低，而业务数据和特征数据要求较高。
- 在**模型调整频率**上，业务数据调整频率较高，流量数据和特征数据调整频率较低。

二、基于 Flink 的实时数仓平台

上面为大家介绍了实时数仓的业务场景，接下来为大家介绍实时数仓的演进过程和美团点评的实时数仓平台建设思路。

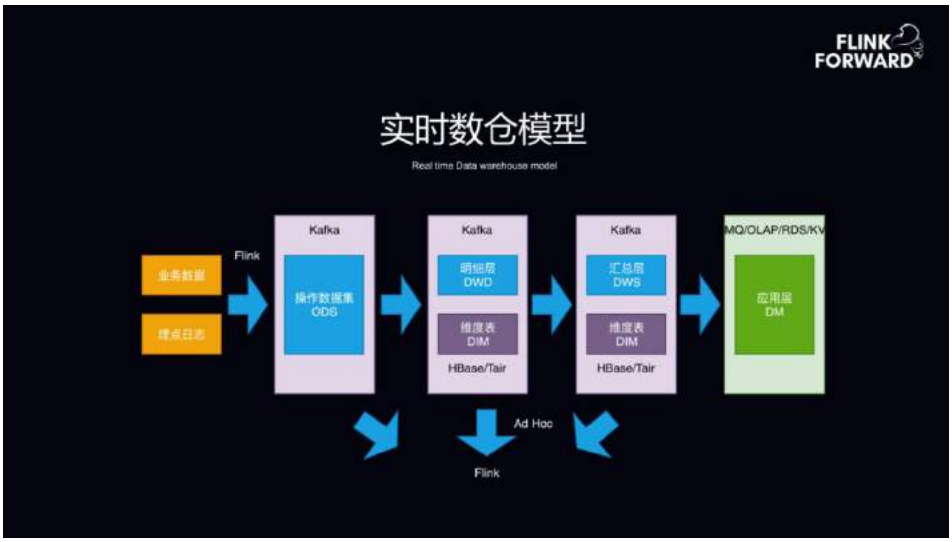
传统数仓模型

为了更有效地组织和管理数据，数仓建设往往会进行数据分层，一般自下而上分为四层：ODS（操作数据层）、DWD（数据明细层）、DWS（汇总层）和应用层。即时查询主要通过 Presto、Hive 和 Spark 实现。



实时数仓模型

实时数仓的分层方式一般也遵守传统数据仓库模型，也分为了 ODS 操作数据集、DWD 明细层和 DWS 汇总层以及应用层。但实时数仓模型的处理的方式却和传统数仓有所差别，如明细层和汇总层的数据一般会放在 Kafka 上，维度数据一般考虑到性能问题则会放在 HBase 或者 Tair 等 KV 存储上，即席查询则可以使用 Flink 完成。



准实时数仓模型

在以上两种数仓模型之外，我们发现业务方在实践过程中还有一种准实时数仓模型，其特点是不完全基于流去做，而是将明细层数据导入到 OLAP 存储中，基于 OLAP 的计算能力去做汇总并进行进一步的加工。

实时数仓和传统数仓的对比

实时数仓和传统数仓的对比主要可以从四个方面考虑：

- 第一个是**分层方式**，离线数仓为了考虑到效率问题，一般会采取空间换时间的方式，层级划分会比较多；则实时数仓考虑到实时性问题，一般分层会比较少，另外也减少了中间流程出错的可能性。
- 第二个是**事实数据存储**方面，离线数仓会基于 HDFS，实时数仓则会基于消息队列（如 Kafka）。
- 第三个是**维度数据存储**，实时数仓会将数据放在 KV 存储上面。
- 第四个是**数据加工过程**，离线数仓一般以 Hive、Spark 等批处理为主，而实时数仓则是基于实时计算引擎如 Storm、Flink 等，以流处理为主。

实时数仓建设方案对比

下图中对于实时数仓的两种建设方式，即准实时数仓和实时数仓两种方式进行了对比。它们的实现方式分别是基于 OLAP 引擎和流计算引擎，实时度则分别是分钟和秒级。

- 在**调度开销**方面，准实时数仓是批处理过程，因此仍然需要调度系统支持，虽然调度开销比离线数仓少一些，但是依然存在，而实时数仓却没有调度开销。
- 在**业务灵活性**方面，因为准实时数仓基于 OLAP 引擎实现，灵活性优于基于流计算的方式。
- 在**对数据晚到的容忍度**方面，因为准实时数仓可以基于一个周期内的数据进行全量计算，因此对于数据晚到的容忍度也是比较高的，而实时数仓使用的是增量计算，对于数据晚到的容忍度更低一些。
- 在**扩展性**方面，因为准实时数仓的计算和存储是一体的，因此相比于实时数仓，扩展性更弱一些。
- 在**适用场景**方面，准实时数仓主要用于有实时性要求但不太高、数据量不大以及多表关联复杂和业务变更频繁的场景，如交易类型的实时分析，实时数仓则更适用于实时性要求高、数据量大的场景，如实时特征、流量分发以及流量类型实时分析。

总结一下，基于 OLAP 引擎的建设方式是数据量不太大，业务流量不太高情况下为了提高时效性和开发效率的一个折中方案，从未来的发展趋势来看，基于流计算的实时数仓更具有发展前景。

一站式解决方案

从业务实践过程中，我们看到了业务建设实时数仓的共同需求，包括发现不同业务的元数据是割裂的，业务开发也倾向于使用 SQL 方式同时开发离线数仓和实时数仓，需要更多的运维工具支持。因此我们规划了一站式解决方案，希望能够将整个流程贯通。

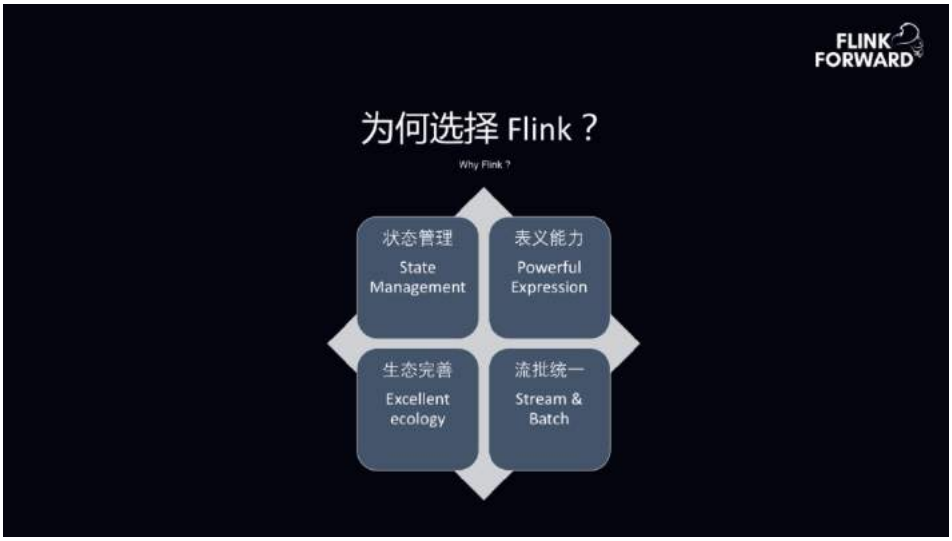
这里的一站式解决方案主要为用户提供了数据开发工作平台、元数据管理。同时我们考虑到业务从生产到应用过程中的问题，我们 OLAP 生产平台，从建模方式、生产任务管理和资源方面解决 OLAP 生产问题。左侧是我们已经具备数据安全体系、资源体系和数据治理，这些是离线数仓和实时数仓可以共用的。

为何选择 Flink？

实时数仓平台建设之所以选择 Flink 是基于以下四个方面的考虑，这也是实时数仓方面关注的比较核心的问题。

- **第一个是状态管理**，实时数仓里面会进行很多的聚合计算，这些都需要对于状态进行访问和管理，Flink 在这方面比较成熟。

- **第二个是表义能力**，Flink 提供极为丰富的多层次 API，包括 Stream API、Table API 以及 Flink SQL。
- **第三个是生态完善**，实时数仓的用途广泛，用户对于多种存储有访问需求，Flink 对于这方面的支持也比较完善。
- 最后一点就是 **Flink 提供了流批统一的可能性**。



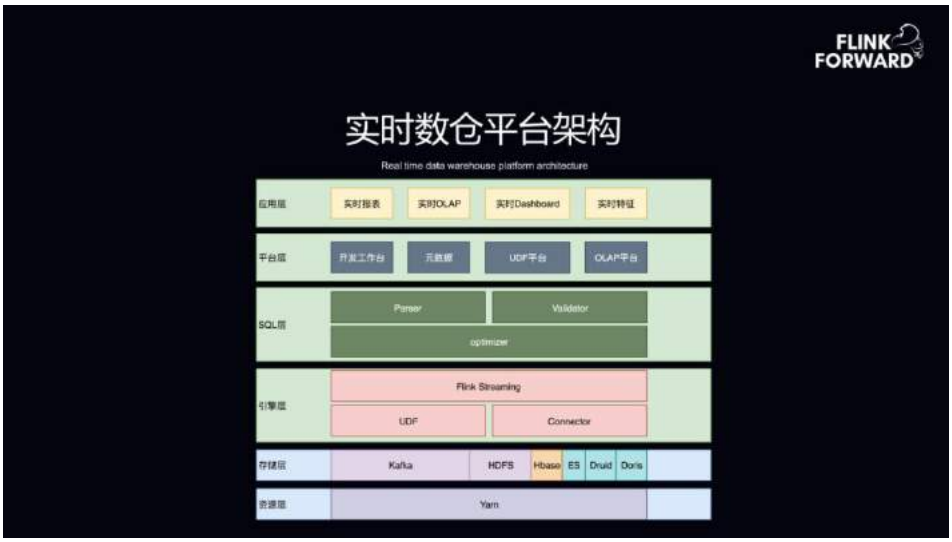
实时数仓平台

- **建设思路**

实时数仓平台的建设思路从外到内分为了四个层次，我们认为平台应该做的事情是为用户提供抽象的表达能力，分别是**消息表达、数据表达、计算表达以及流和批统一**。

- **实时数仓平台架构**

如下图所示的是美团点评的实时数仓平台架构，从下往上看，资源层和存储层复用了实时计算平台的能力，在引擎层则会基于 Flink Streaming 实现一些扩展能力，包括对 UDF 的集成和 Connector 的集成。再往上是基于 Flink SQL 独立出来的 SQL 层，主要负责解析、校验和优化。在这之上是平台层，包括开发工作台、元数据、UDF 平台以及 OLAP 平台。最上层则是平台所支持的实时数仓的应用，包括实时报表、实时 OLAP、实时 Dashboard 和实时特征等。



- **消息表达 – 数据接入**

在消息表达层面，因为 Binlog、埋点日志、后端日志以及 IoT 数据等的数据格式是不一致的，因此美团点评的实时数仓平台提供数据接入的流程，能够帮助大家把数据同步到 ODS 层。这里主要实现了两件事情，分别是统一消息协议和屏蔽处理细节。

如下图左侧是接入过程的一个例子，对于 Binlog 类型数据，实时数仓平台还为大家提供了分库分表的支持，能够将属于同一个业务的不同的分库分表数据根据业务规则收集到同一个 ODS 表中去。

- **计算表达 – 扩展 DDL**

美团点评实时数仓平台基于 Flink 扩展了 DDL，这部分工作的主要目的是建设元数据体系，打通内部的主流实时存储，包括 KV 数据、OLAP 数据等。由于开发工作台和元数据体系是打通的，因此很多数据的细节并不需要大家在 DDL 中明确地声明出来，只需要在声明中写上数据的名字，和运行时的一些设置，比如 MQ 从最新消费还是最旧消费或者从某个时间戳消费即可，其他的数据访问方式是一致的。

- **计算表达 –UDF 平台**

对于 UDF 平台而言，需要从三个层面考虑：

- 首先是**数据安全性**。之前的数仓建设过程中，用户可以上传 Jar 包去直接引用 UDF，这样做是有危险性存在的，并且我们无法知道数据的流向。从数据安全的角度来考虑，平台会进行代码审计和血缘关系分析，对于历史风险组件或者存在问题的组件可以进行组件收敛。
- 第二个层面，在数据安全基础上我们还会关注 **UDF 的运行质量**，平台将会为用户提供模板、用例以及测试的管理，为用户屏蔽编译打包、Jar 包管理的过程，并且会在 UDF 模板中进行指标日志的埋点和异常处理。
- 第三个层面是 **UDF 的复用能力**，因为一个业务方开发的 UDF，其他业务方很可能也会使用，但是升级过程中可能会带来不兼容的问题，因此，平台为业务提供了项目管理、函数管理和版本管理的能力。

UDF 的应用其实非常广泛，UDF 平台并不是只支持实时数仓，也会同时支持离线数仓、机器学习以及查询服务等应用场景。下图中右侧展示的是 UDF 的使用案例，左图是 UDF 的开发流程，用户只需要关心注册流程，接下来的编译打包、测试以及上传等都由平台完成；右图是 UDF 的使用流程中，用户只需要声明 UDF，平台会进行解析校验、路径获取以及在作业提交的时候进行集成。

- **实时数仓平台 –Web IDE**

最后介绍一下实时数仓平台的开发工作台，以 Web IDE 的形式集成了模型、作业以及 UDF 的管理，用户可以在 Web IDE 上以 SQL 方式开发。平台会对 SQL 做一些版本的管理，并且支持用户回退到已部署成功的版本上去。

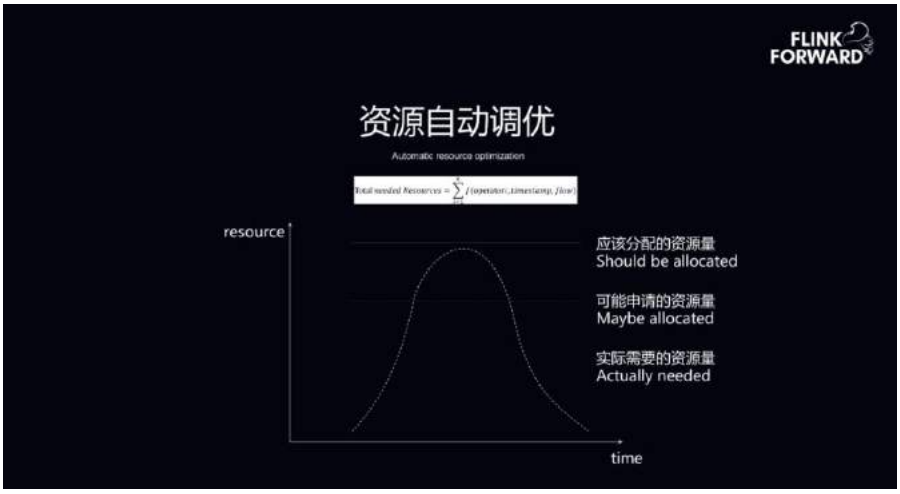
三、未来发展与思考

资源自动调优

从整个实时计算角度来考虑，目前美团点评的实时计算平台的节点数已经达到了几千台，未来很可能会达到上万台，因此资源优化这件事情很快就会被提上日程。由于业务本身的流量存在高峰和低谷，对于一个实时任务来说，可能在高峰时需要很多资源，但是在低谷时并不需要那么多资源。

另外一方面，波峰本身也是会发生变化的，有可能随着业务的上涨使得原来分配的资源数量不够用。因此，资源自动调优有两个含义，一个是指能够适配作业的高峰流量上涨，自动适配 Max 值；另外一个含义是指使得作业能够在高峰过去之后自动适应流量减少，能够快速缩容。我们可以通过每个任务甚至是算子的历史运行情况，拟合得到算子、流量与资源的关系函数，在流量变化时同步调整资源量。

以上是资源优化的思路，除此之外还需要考虑当资源完成优化之后应该如何利用。为了保证可用性，实时和离线任务一般会分开部署，否则带宽、IO 都可能被离线计算打满导致实时任务延迟。而从资源使用率角度出发，则需要考虑实时和离线的混合部署，或者以流的方式来处理一些实时性要求并不是非常高的任务。这就要求更细粒度的资源隔离和更快的资源释放。



推动实时数仓建设方式升级

实时数仓的建设一般分为几个步骤：

- 首先，业务提出需求，后续会进行设计建模、业务逻辑开发和底层技术实现。美团点评的实时数仓建设思路是将技术实现统一表达，让业务关注逻辑开发，而逻辑开发也可以基于配置化手段实现自动构建。
- 再上一层是可以根据业务需求实现智能建模，将设计建模过程实现自动化。

目前，美团点评的实时数仓平台建设工作还集中在统一表达的层次，距离理想状态仍然有比较长的一段路要走。

小米流式平台架构演进与实践

作者：夏军（小米流式平台负责人，高级研发工程师 ）

摘要：小米业务线众多，从信息流，电商，广告到金融等覆盖了众多领域，小米流式平台为小米集团各业务提供一体化的流式数据解决方案，主要包括数据采集，数据集成和流式计算三个模块。目前每天数据量达到 1.2 万亿条，实时同步任务 1.5 万，实时计算的数据 1 万亿条。

伴随着小米业务的发展，流式平台也经历三次大升级改造，满足了众多业务的各种需求。最新的一次迭代基于 Apache Flink，对于流式平台内部模块进行了彻底的重构，同时小米各业务也在由 Spark Streaming 逐步切换到 Flink。

1. 背景介绍
2. 小米流式平台发展历史
3. 基于 Flink 的实时数仓
4. 未来规划

背景介绍

小米流式平台的愿景是为小米所有的业务线提供流式数据的一体化、平台化解决方案。具体来讲包括以下三个方面：

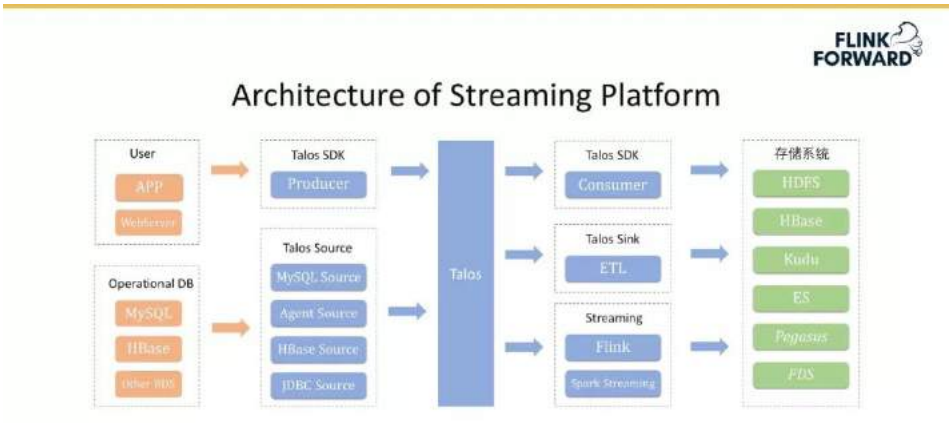
- **流式数据存储**：流式数据存储指的是消息队列，小米开发了一套自己的消息队列，其类似于 Apache kafka，但它有自己的特点，小米流式平台提供消息队列的存储功能；
- **流式数据接入和转储**：有了消息队列来做流式数据的缓存区之后，继而需要提供流式数据接入和转储的功能；
- **流式数据处理**：指的是平台基于 Flink、Spark Streaming 和 Storm 等计算引

擎对流式数据进行处理的过程。

下图展示了流式平台的整体架构。从左到右第一列橙色部分是数据源，包含两部分，即 User 和 Database。

- User 指的是用户各种各样的埋点数据，如用户 APP 和 WebServer 的日志，其次是 Database 数据，如 MySQL、HBase 和其他的 RDS 数据。
- 中间蓝色部分是流式平台的具体内容，其中 Talos 是小米实现的消息队列，其上层包含 Consumer SDK 和 Producer SDK。
- 此外小米还实现了一套完整的 Talos Source，主要用于收集刚才提到的用户和数据库的全场景的数据。

Talos Sink 和 Source 共同组合成一个数据流服务，主要负责将 Talos 的数据以极低的延迟转储到其他系统中；Sink 是一套标准化的服务，但其不够定制化，后续会基于 Flink SQL 重构 Talos Sink 模块。



下图展示了小米的业务规模。在存储层面小米每天大概有 1.2 万亿条消息，峰值流量可以达到 4300 万条每秒。转储模块仅 Talos Sink 每天转储的数据量就高达 1.6 PB，转储作业目前将近有 1.5 万个。每天的流式计算作业超过 800 个，Flink 作业超过 200 个，Flink 每天处理的消息量可以达到 7000 亿条，数据量在 1 PB 以上。

小米流式平台发展历史

小米流式平台发展历史分为如下三个阶段：

- **Streaming Platform 1.0**：小米流式平台的 1.0 版本构建于 2010 年，其最

初使用的是 Scribe、Kafka 和 Storm，其中 Scribe 是一套解决数据收集和数据转储的服务。

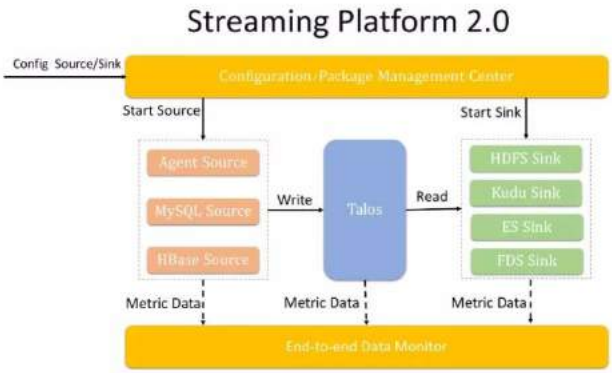
- **Streaming Platform 2.0**：由于 1.0 版本存在的种种问题，我们自研了小米自己的消息队列 Talos，还包括 Talos Source、Talos Sink，并接入了 Spark Streaming。
- **Streaming Platform 3.0**：该版本在上一个版本的基础上增加了 Schema 的支持，还引入了 Flink 和 Stream SQL。

Streaming Platform 1.0 整体是一个级联的服务，前面包括 Scribe Agent 和 Scribe Server 的多级级联，主要用于收集数据，然后满足离线计算和实时计算的场景。离线计算使用的是 HDFS 和 Hive，实时计算使用的是 Kafka 和 Storm。虽然这种离线加实时的方式可以基本满足小米当时的业务需求，但也存在一系列的问题。

- 首先是 Scribe Agent 过多，而配置和包管理机制缺乏，导致维护成本非常高；
- Scribe 采用的 Push 架构，异常情况下无法有效缓存数据，同时 HDFS / Kafka 数据相互影响；
- 最后数据链级联比较长的时候，整个全链路数据黑盒，缺乏监控和数据检验机制。

为了解决 Streaming Platform 1.0 的问题，小米推出了 Streaming Platform 2.0 版本。该版本引入了 Talos，将其作为数据缓存区来进行流式数据的存储，左侧是多种多样的数据源，右侧是多种多样的 Sink，即将原本的级联架构转换成星型架构，优点是方便地扩展。

- 由于 Agent 自身数量及管理的流较多（具体数据均在万级别），为此该版本实现了一套配置管理和包管理系统，可以支持 Agent 一次配置之后的自动更新和重启等。
- 此外，小米还实现了去中心化的配置服务，配置文件设定好后可以自动地分发到分布式结点上去。
- 最后，该版本还实现了数据的端到端监控，通过埋点来监控数据在整个链路上的数据丢失情况和数据传输延迟情况等。

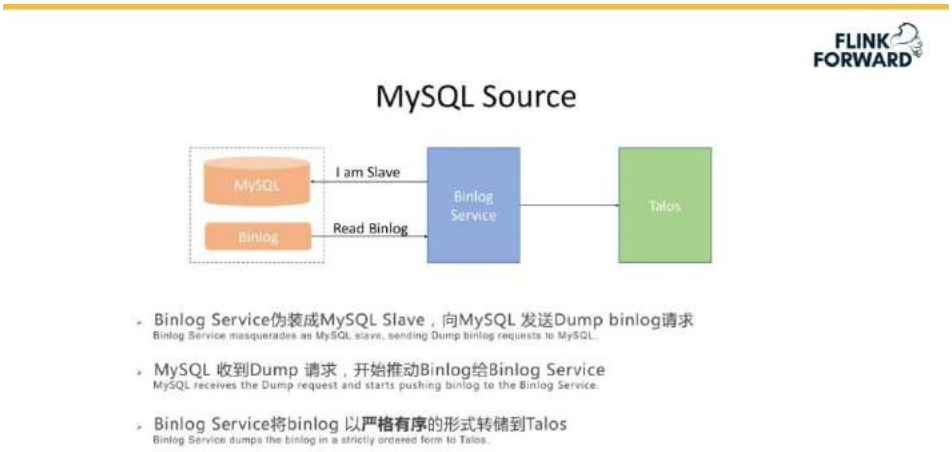


Streaming Platform 2.0 的优势主要有：

- 引入了 **Multi Source & Multi Sink**，之前两个系统之间导数据需要直接连接，现在的架构将系统集成复杂度由原来的 O(M*N) 降低为 O(M+N)；
- 引入**配置管理和包管理机制**，彻底解决系统升级、修改和上线等一系列问题，降低运维的压力；
- 引入**端到端数据监控机制**，实现全链路数据监控，量化全链路数据质量；
- 产品化解决方案，避免重复建设，解决业务运维问题。

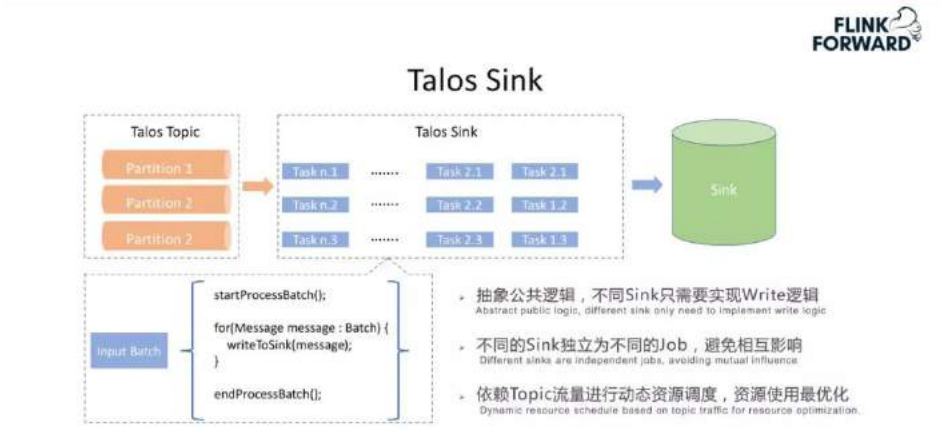
下图详细介绍一下 MySQL 同步的案例，场景是将 MySQL 的一个表通过上述的机制同步到消息队列 Talos。具体流程是 Binlog 服务伪装成 MySQL 的 Slave，

向 MySQL 发送 Dump binlog 请求；MySQL 收到 Dump 请求后，开始推动 Binlog 给 Binlog 服务；Binlog 服务将 binlog 以严格有序的形式转储到 Talos。之后会接入 Spark Streaming 作业，对 binlog 进行解析，解析结果写入到 Kudu 表中。目前平台支持写入到 Kudu 中的表的数量级超过 3000 个。

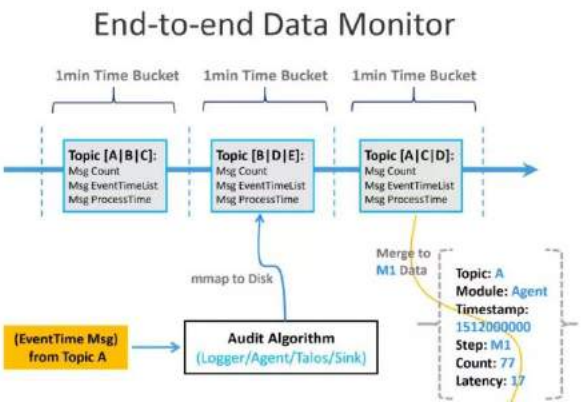


Agent Source 的功能模块如下图所示。其支持 RPC、Http 协议，并可以通过 File 来监听本地文件，实现内存和文件双缓存，保证数据的高可靠。平台基于 RPC 协议实现了 Logger Appender 和 RPC 协议的 SDK；对于 Http 协议实现了 HttpClient；对于文件实现了 File Watcher 来对本地文件进行自动地发现和扫描，Offset Manager 自动记录 offset；Agent 机制与 K8S 环境深度整合，可以很容易地和后端的流式计算等相结合。

下图是 Talos Sink 的逻辑流程图，其基于 Spark Streaming 来实现一系列流程。最左侧是一系列 Talos Topic 的 Partition 分片，基于每个 batch 抽象公共逻辑，如 startProcessBatch() 和 stopProcessBatch()，不同 Sink 只需要实现 Write 逻辑；不同的 Sink 独立为不同的作业，避免相互影响；Sink 在 Spark Streaming 基础上进行了优化，实现了根据 Topic 流量进行动态资源调度，保证系统延迟的前提下最大限度节省资源。



下图是平台实现的端到端数据监控机制。具体实现是为每个消息都有一个时间戳 EventTime，表示这个消息真正生成的时间，根据 EventTime 来划分时间窗口，窗口大小为一分钟，数据传输的每一跳统计当前时间窗口内接受到的消息数量，最后统计出消息的完整度。延迟是计算某一跳 ProcessTime 和 EventTime 之间的差值。



Streaming Platform 2.0 目前的问题主要有三点：

- Talos 数据缺乏 Schema 管理，Talos 对于传入的数据是不理解的，这种情况下无法使用 SQL 来消费 Talos 的数据；
- Talos Sink 模块不支持定制化需求，例如从 Talos 将数据传输到 Kudu 中，Talos 中有十个字段，但 Kudu 中只需要 5 个字段，该功能目前无法很好地支持；
- Spark Streaming 自身问题，不支持 Event Time，端到端 Exactly Once 语义。

基于 Flink 的实时数仓

为了解决 Streaming Platform 2.0 的上述问题，小米进行了大量调研，也和阿里的实时计算团队做了一系列沟通和交流，最终决定将使用 Flink 来改造平台当前的流程，下面具体介绍小米流式计算平台基于 Flink 的实践。

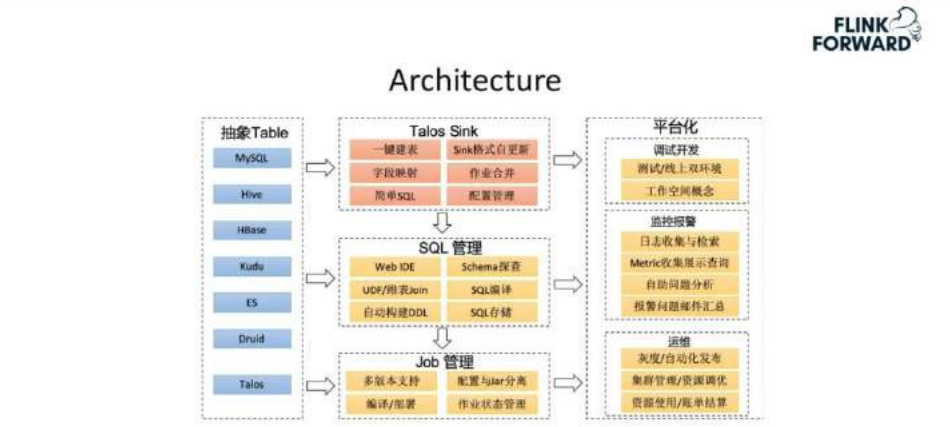
使用 Flink 对平台进行改造的设计理念如下：

- **全链路 Schema 支持**，这里的全链路不仅包含 Talos 到 Flink 的阶段，而是从最开始的数据收集阶段一直到后端的计算处理。需要实现数据校验机制，避免数据污染；字段变更和兼容性检查机制，在大数据场景下，Schema 变更频繁，兼容性检查很有必要，借鉴 Kafka 的经验，在 Schema 引入向前、向后或全兼容检查机制。
- 借助 Flink 社区的力量**全面推进 Flink 在小米的落地**，一方面 Streaming 实时计算的作业逐渐从 Spark、Storm 迁移到 Flink，保证原本的延迟和资源节省，目前小米已经运行了超过 200 个 Flink 作业；另一方面期望用 Flink 改造 Sink 的流程，提升运行效率的同时，支持 ETL，在此基础上大力推进 Streaming SQL；
- **实现 Streaming 产品化**，引入 Streaming Job 和 Streaming SQL 的平台化管理；

- **基于 Flink SQL 改造 Talos Sink**，支持业务逻辑定制化

下图是 Streaming Platform 3.0 版本的架构图，与 2.0 版本的架构设计类似，只是表达的角度不同。具体包含以下几个模块：

- **抽象 Table**：该版本中各种存储系统如 MySQL 和 Hive 等都会抽象成 Table，为 SQL 化做准备。
- **Job 管理**：提供 Streaming 作业的管理支持，包括多版本支持、配置与 Jar 分离、编译部署和作业状态管理等常见的功能。
- **SQL 管理**：SQL 最终要转换为一个 Data Stream 作业，该部分功能主要有 Web IDE 支持、Schema 探查、UDF/ 维表 Join、SQL 编译、自动构建 DDL 和 SQL 存储等。
- **Talos Sink**：该模块基于 SQL 管理对 2.0 版本的 Sink 重构，包含的功能主要有一键建表、Sink 格式自动更新、字段映射、作业合并、简单 SQL 和配置管理等。前面提到的场景中，基于 Spark Streaming 将 Message 从 Talos 读取出来，并原封不动地转到 HDFS 中做离线数仓的分析，此时可以直接用 SQL 表达很方便地实现。未来希望实现该模块与小米内部的其他系统如 ElasticSearch 和 Kudu 等进行深度整合，具体的场景是假设已有 Talos Schema，基于 Talos Topic Schema 自动帮助用户创建 Kudu 表。
- **平台化**：为用户提供一体化、平台化的解决方案，包括调试开发、监控报警和运维等。



Job 管理

Job 管理提供 Job 全生命周期管理、Job 权限管理和 Job 标签管理等功能；支持 Job 运行历史展示，方便用户追溯；支持 Job 状态与延迟监控，可以实现失败作业自动拉起。

SQL 管理

主要包括以下四个环节：

- 将外部表转换为 SQL DDL，对应 Flink 1.9 中标准的 DDL 语句，主要包含 Table Schema、Table Format 和 Connector Properities。
- 基于完整定义的外部 SQL 表，增加 SQL 语句，既可以得到完成的表达用户的需求。即 SQL Config 表示完整的用户预计表达，由 Source Table DDL、Sink Table DDL 和 SQL DML 语句组成。
- 将 SQL Config 转换成 Job Config，即转换为 Stream Job 的表现形式。
- 将 Job Config 转换为 JobGraph，用于提交 Flink Job。

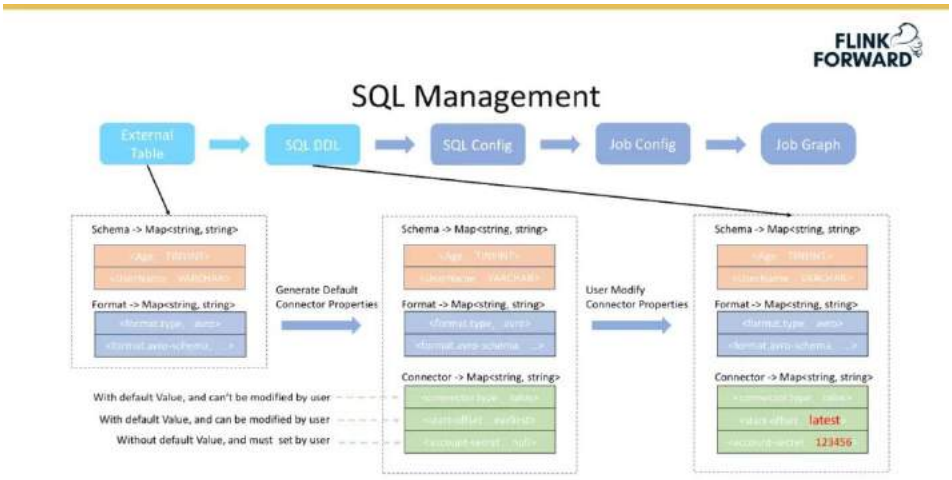
外部表转换成 SQL DDL 的流程如下图所示。

- 首先根据外部表获取 Table Schema 和 Table Format 信息，后者用于反解数据，如对于 Hive 数据反序列化；
- 然后再后端生成默认的 Connector 配置，该配置主要分为三部分，即不可修改的、带默认值的用户可修改的、不带默认值的用户必须配置的。

不可修改的配置情况是假设消费的是 Talos 组件，那么 connector.type 一定是 talos，则该配置不需要改；而默认值是从 Topic 头部开始消费，但用户可以设置从

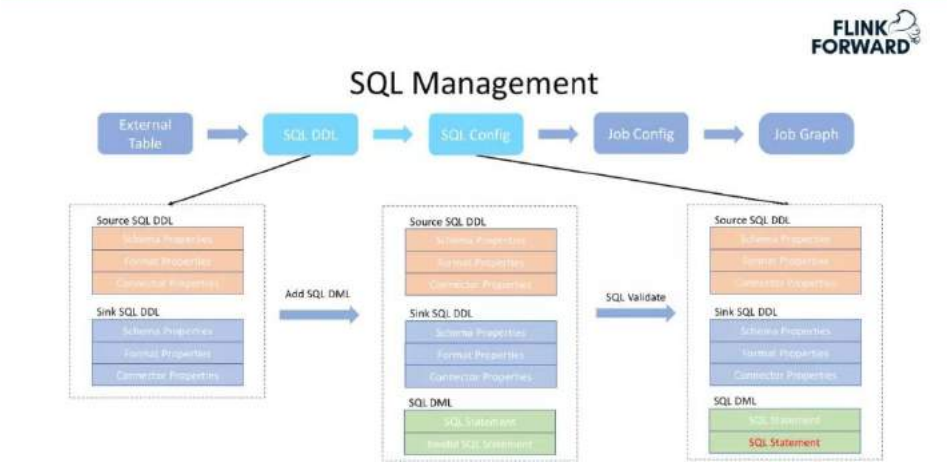
尾部开始消费，这种情况属于带默认值但是用户可修改的配置；而一些权限信息是用户必须配置的。

之所以做三层配置管理，是为了尽可能减少用户配置的复杂度。Table Schema、Table Format 和 Connector 1 其他配置信息，组成了 SQL DDL。将 SQL Config 返回给用户之后，对于可修改的需要用户填写，这样便可以完成从外部表到 SQL DDL 的转换，红色字体表示的是用户修改的信息。



SQL 管理引入了一个 External Table 的特性。假设用户在平台上选择消费某个 Topic 的时候，该特性会自动地获取上面提到的 Table 的 Schema 和 Format 信息，并且显示去掉了注册 Flink Table 的逻辑；获取 Schema 时，该特性会将外部表字段类型自动转换为 Flink Table 字段类型，并自动注册为 Flink Tab 了。同时将 Connector Properties 分成三类，参数带默认值，只有必须项要求用户填写；所有参数均采用 Map<string,string> 的形式表达，非常便于后续转化为 Flink 内部的 TableDescriptor。

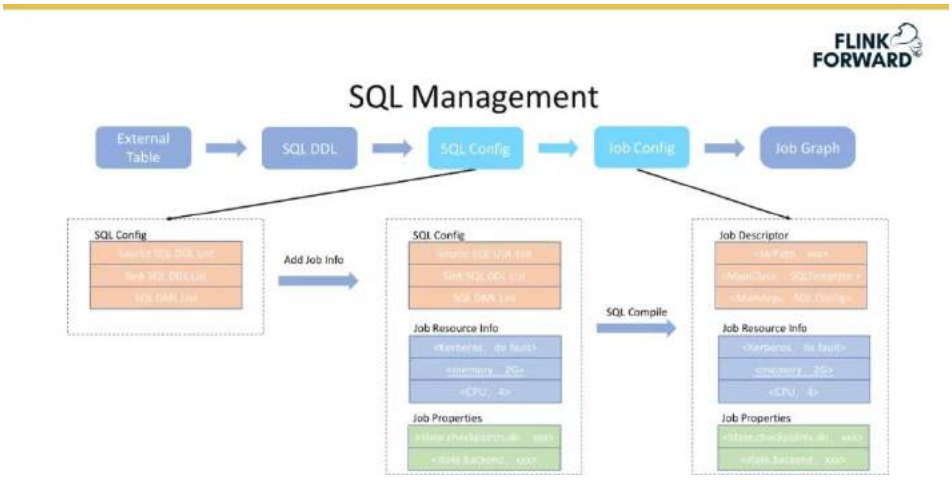
上面介绍了 SQL DDL 的创建过程，在已经创建的 SQL DDL 的基础上，如 Source SQL DDL 和 Sink SQL DDL，要求用户填写 SQL query 并返回给后端，后端会对 SQL 进行验证，然后会生成一个 SQL Config，即一个 SQL 语句的完整表达。



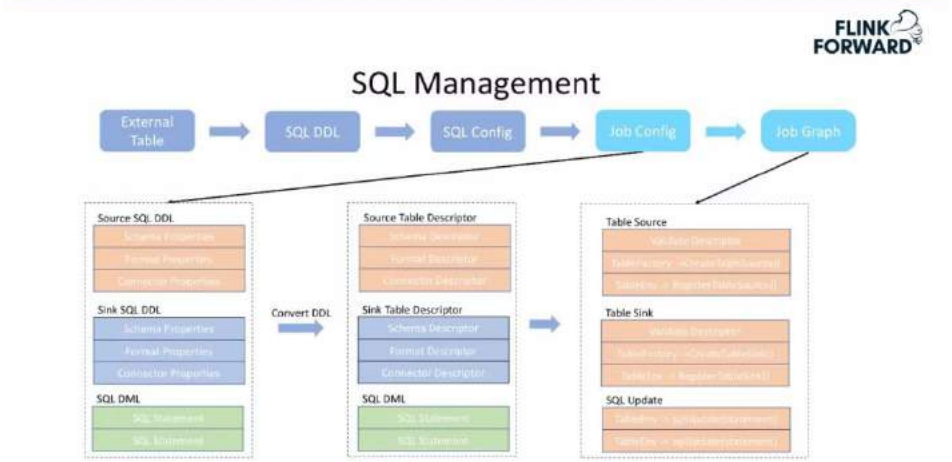
SQL Config 转换为 Job Config 的流程如下图所示。

- 首先在 SQL Config 的基础上增加作业所需要的资源、Job 的相关配置（Flink 的 state 参数等）；
- 然后将 SQLConfig 编译成一个 Job Descriptor，即 Job Config 的描述，如

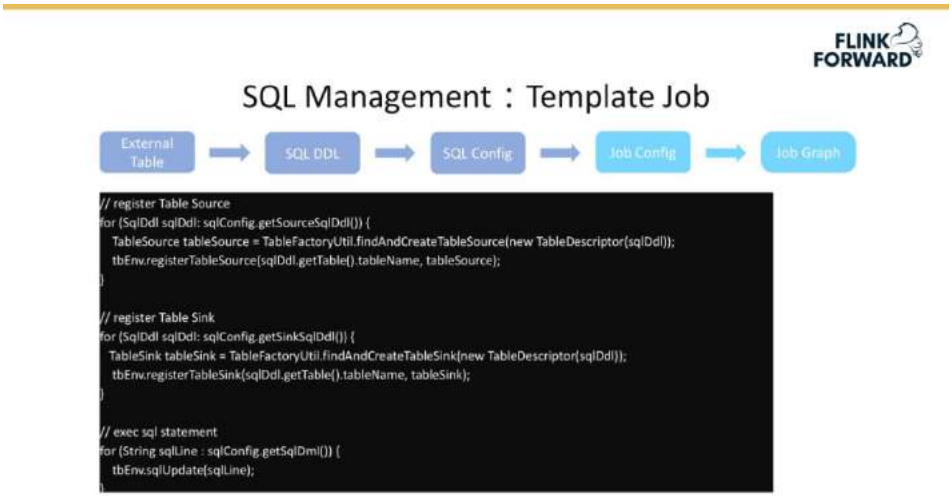
Job 的 Jar 包地址、MainClass 和 MainArgs 等。



下图展示了 Job Config 转换为 Job Graph 的过程。对于 DDL 中的 Schema、Format 和 Property 是和 Flink 中的 Table Descriptor 是一一对应的，这种情况下只需要调用 Flink 的相关内置接口就可以很方便地将信息转换为 Table Descriptor，如 CreateTableSource()、RegistorTableSource() 等。通过上述过程，DDL 便可以注册到 Flink 系统中直接使用。对于 SQL 语句，可以直接使用 TableEnv 的 sqlUpdate() 可以完成转换。

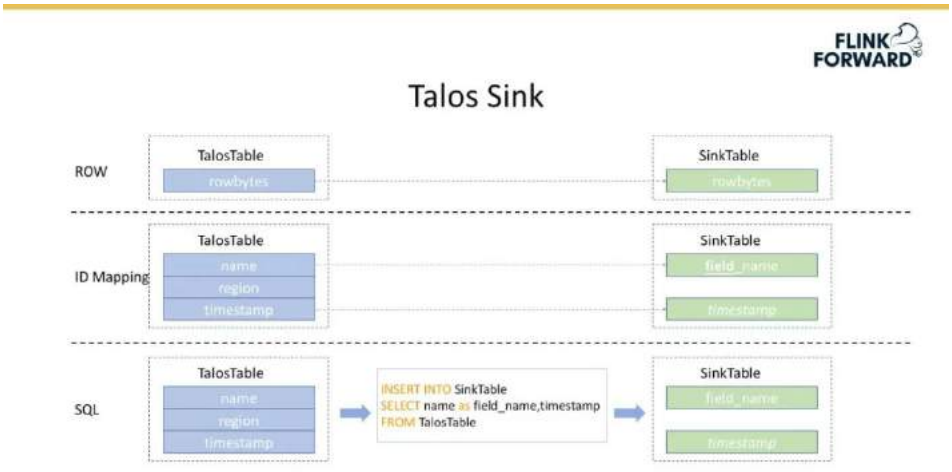


SQL Config 转换为一个 Template Job 的流程如下所示。前面填写的 Jar 包地址即该 Template 的 Jar 地址，MainClass 是该 Template Job。假设已经有了 SQL DDL，可以直接转换成 Table Descriptor，然后通过 TableFactorUtil 的 findAndCreateTableSource() 方法得到一个 Table Source，Table Sink 的转换过程类似。完成前两步操作后，最后进行 sqlUpdate() 操作。这样便可以将一个 SQL Job 转换为最后可执行的 Job Graph 提交到集群上运行。



Talos Sink 采用了下图所示的三种模式：

- **Row**：Talos 的数据原封不动地灌到目标系统中，这种模式的好处是数据读取和写入的时候无需进行序列化和反序列化，效率较高；
- **ID mapping**：即左右两边字段进行 mapping，name 对应 field_name，timestamp 对应 timestamp，其中 Region 的字段丢掉；
- **SQL**：通过 SQL 表达来表示逻辑上的处理。



未来规划

小米流式平台未来的计划主要有以下几点：

- 在 Flink 落地的时候持续推进 Streaming Job 和平台化建设；
- 使用 Flink SQL 统一离线数仓和实时数仓；
- 在 Schema 的基础上数据血缘分析和展示，包括数据治理方面的内容；
- 持续参与 Flink 社区的建设。

Netflix：Evolving Keystone to an Open Collaborative Real-time ETL Platform

作者：徐振中（Senior Software Engineer at Netflix）

摘要： Netflix 致力于我们会员的喜悦。我们不懈地专注于提高产品体验和高质量内容。近年来，我们一直在技术驱动的 Studio 和内容制作方面进行大量投资。在这个过程中，我们发现在实时数据平台的领域里中出现了许多独特并有意思的挑战。例如，在微服务架构中，领域对象分布在不同的 App 及其有状态存储中，这使得低延迟高一致性的实时报告和实体搜索发现特别具有挑战性。2019 年 11 月 28 日于北京举办的 Flink Forward Asia 大会实时数仓专场中，Netflix 高级软件工程师徐振中向大家分享了一些有趣的案例、分布式系统基础方面的各种挑战以及解决方案，此外还讨论了其在开发运维过程中的收获，对开放式自助式实时数据平台的一些新愿景，以及对 Realtime ETL 基础平台的一些新思考。

产品背景

Netflix 的长久愿景是把欢乐和微笑带给整个世界，通过在全球各地拍摄一些高质量、多元化的内容产品放在平台上，分享给平台超过一个亿级别的用户。更远大的目标为了给用户带来愉悦的体验，Netflix 的努力方向分为两个：一方面是通过数据整合知识来反馈回并用于提高用户的产品体验中去；另一方面通过建立一个技术驱动的 Studio 去帮助产出内容质量更高的产品。而作为一个数据平台团队，需要关注的是怎么帮助公司中不同的开发人员、数据分析人员等实现其在公司中的价值，最终为解决上述两方面问题做出自己的贡献。

简单地介绍一下该数据平台团队及相应的产品，Keystone。其主要功能是帮助公司在所有的微服务中埋点、建立 Agent、发布事件、收集事件信息，然后放到不同的数据仓库中进行存储，比如 Hive 或 ElasticSearch，最后帮助用户在数据实时存

储的情况下实现计算和分析。

从用户的角度来讲，Keystone 是一个完整的自容（Self-contained）的平台，支持多用户，用户可以通过所提供的 UI 很方便地声明并创建自己想要的 pipeline。从平台角度来说，Keystone 提供底层所有分布式系统中实现比较困难的解决方案，如容器编排（Container Orchestration）、工作流管理（Workflow Management）等等，其对于用户是不可见的。从产品的角度来说，主要有两个功能，一个是帮助用户将数据从边缘设备移到数仓，另一个是帮助用户实时计算的功能。从数字的角度来说，产品在 Netflix 的使用是非常有必要的，只要个数据打交道的开发者，一定会用到该产品，因此该产品在整个公司中有几千个用户，其有一百个 Kafka 的集群支持每天 10PB 数量级左右的数据。

Keystone 的整个架构分为两层，底层是 Kafka 和 Flink 作为底层引擎，底层对所有分布式系统中比较困难的技术方案进行抽象，对用户不可见，在上层构建整个应用；服务层会提供抽象的服务，UI 对于用户来讲比较简单，不需要关心底层实现。

下面介绍一下 Keystone 产品在过去四五年的发展历程。其最初的动机是收集所有设备的数据并将其存储到数据仓库中，当时使用的是 Kafka 技术，因为数据移动比较好解决，本质上来讲仅是一个多并发的问题。在此之后，用户给出了新的需求，即在数据移动的过程中对数据进行一些简单的处理操作，比如筛选（Filter），还有一个很通用的功能 – projection 为此产品推出了针对该需求推出了相应的功能特性。经过一段时间后，用户表示响应做更加复杂的 ETL，比如 Streaming Join 等，因此产品决定将底层的 API 提供给用户，并将底层的关于所有分布式系统的解决方案抽象化，让其更好地关注上层的内容。

产品功能

产品功能介绍将围绕 Netflix 中的两个“超级英雄”Elliot 和 Charlie 来展开。Elliot 是来自数据科学工程组织的一个数据科学家，她的需求是在非常大的数据中寻找响应的 pattern，以帮助提高用户体验；Charlie 是一个来自 Studio 组织的应用开

发者，其目标是通过开发一系列的应用来帮助周边的其他开发者产出更高质量的产品。这两人工作对于产品来讲都非常重要，Elliot 的数据分析结果可以帮助给出更好的推荐和个性化定制 , 最终提高用户体验；而 Charlie 的工作可以帮助周边的开发者提高效率。

Recommendation & Personalization

Elliot 作为一个数据科学家，需要的是一个简单易用的实时 ETL 操作平台，其不希望写非常复杂的编码，同时需要保证整个 pipeline 的低延时。其所从事的工作和相关需求主要有以下几个：

- **推荐和个性化定制。**该工作中可以根据个人特点的不同将同样的视频通过不同的形式推送给相应的用户，视频可以分为多个 row，每一个 row 可以是不同的分类，根据个人的喜好可以对不同的 row 进行更改。此外，每一个视频的题目都会有一个 artwork，不同国家、不同地域的不同用户对 artwork 的喜好也可能不同，也会通过算法进行计算并定制适合用户的 artwork。
- **A/B Testing。**Netflix 提供给非会员用户 28 天免费的视频观看机会，同时也相信给用户看到了适合自己的视频，其更有可能会购买 Netflix 的服务，而在进行 A/B Testing 的时候，就需要 28 天才能做完。对于 Elliot 来讲，进行 A/B Testing 的时候可能会犯错误，其所关心的是怎么样才能在不用等到 28 天结束的时候就可以提前发现问题。

当在设备上观看 Netflix 的时候，会以请求的形式和网关进行交互，然后网关会将这些请求分发给后端的微服务，比如说用户在设备上点击播放、暂停、快进、快退等操作，这些会有不同的微服务进行处理，因此需要将相应的数据收集起来，进一步处理。对于 Keystone 平台团队来讲，需要收集不同的微服务中产生的数据并进行存储。Elliot 需要将不同的数据整合起来，以解决她关注的问题。

至于为什么要使用流处理，主要有四方面的考量，即实时报告、实时告警、机器学习模型的快速训练以及资源效率。相比于前两点，机器学习模型的快速训练以及资

源效率对于 Elliot 的工作更加重要。尤其需要强调的是资源效率，针对前面的 28 天的 A/B Testing，目前的做法是每天将数据与前 27 天做 Batch Processing，这个过程中涉及了很多重复处理，使用流处理可以很好地帮助提高整体的效率。

产品会提供命令行的工具给用户，用户只需要在命令行中输入相应的命令来进行操作，工具最开始会询问用户一些简单的问题，如需要使用什么 repository 等，用户给出相应的回答后，会最终产生一个模板，用户便可以开始使用工具进行开发工作；产品还提供一系列简单的 SDK，目前支持的是 Hive、Iceberg、Kafka 和 ElasticSearch 等。需要强调的是 Iceberg，它是在 Netflix 主导的一个 Table Format，未来计划取代 Hive。其提供了很多特色功能来帮助用户做优化；产品向用户提供了简单的 API，可以帮助其直接生成 Source 和 Sink。

Elliot 在完成一系列的工作之后，可以选择将自己的代码提交到 repository 中，后台会自动启动一个 CI/CD pipeline，将所有的源代码和制品等包装在 Docker 镜像中，保证所有的版本一致性。Elliot 在 UI 处只需要选择想要部署哪一个版本，然后点击部署按钮可以将 jar 部署到生产环境中。产品会在后台帮助其解决底层分布式系统比较困难的问题，比如怎么做容器编排等，目前是基于资源的编排，未来计划向 K8S 方向发展。部署 Job（作业）包的过程中会部署一个 JobManager 的集群和一个 TaskManager 的集群，因此每一个 Job 对于用户来说是完全独立的。

产品提供默认的配置选项，同时也支持用户在平台 UI 上修改并覆盖配置信息，直接选择部署即可生效，而不需重写代码。Elliot 之前有一个需求是在 Stream Processing 的过程中，比如从不同的 Topic 中去读取数据，出现问题的情况下可能需要在 Kafka 中操作，也可能需要在数据仓库中操作，面对该问题，其需求是在不改动代码的情况下切换不同的 Source，而目前平台提供的 UI 很方便地完成该需求。此外平台还可以帮助用户在部署的时候选择需要多少资源来运行作业。

很多用户从 Batch Processing 转到 Stream Processing 的过程中，已经有了很多需要的制品，比如 Schema 等，因此平台还帮助其简单地实现这些制品的集成。

平台拥有很多需要在其之上写 ETL 工程的用户，当用户越来越多的时候，平台的可伸缩性显得尤为重要。为此，平台采用了一系列的 pattern 来解决该问题。具体来讲，主要有三个 pattern 正在使用，即 Extractor Pattern、Join Pattern 和 Enrichment Pattern。

Content Production

先简要介绍一下什么是 Content Production。其包括预测在视频制作方面的花费、制定 program、达成 deal、制作视频、视频后期处理、发布视频以及金融报告。

Charlie 所在的是 Studio 部门主要负责开发一系列的应用来帮助支持 Content Production。每一个应用都是基于微服务架构来开发部署的，每一个微服务应用会有自己的职责。举个最简单的例子，会有专门管理电影标题的微服务应用，会有专门管理 deals 和 contracts 的微服务应用等等。面对如此多的微服务应用，Charlie 面临的挑战问题是当其在进行实时搜索的过程中，比如搜索某一个电影的演员，需要将数据从不同的地方 join 起来；另外数据每天都在增加，保证实时更新的数据的一致性比较困难，这本质上是分布式微服务系统的特点导致，不同的微服务选择使用的数据库可能不同，这给数据一致性的保证又增加了一定的复杂度。针对该问题，常用的解决方案有以下三个：

1) **Dual writes:** 当开发者知道数据需要放到主要的数据库中的时候，同时也要放到另一个数据库中，可以很简单地选择分两次写入到数据库中，但是这种操作是不容错的，一旦发生错误，很有可能会导致数据的不一致；
2) **Change Data Table:** 需要数据库支持事务的概念，不管对数据库做什么操作，相应的变更会加到事务变更的 statement 中并存入单独的表中，之后可以查询该 change 表并获取相应的变更情况并同步到其他数据表；
3) **Distributed Transaction：** 指的是分布式事务，在多数据环境中实现起来比较复杂。

Charlie 的一个需求是将所有的电影从 Movie Datastore 复制到一个以

Elasticsearch 来支持的 movie search index 中，主要通过一个 Polling System 来做数据拉取和复制，数据一致性的保证采用的是上述的 Change data table 的方法。该方案的弊端是只支持定期数据拉取，另外 Polling System 和数据源直接紧密结合，一旦 Movie Search Datastore 的 Schema 改变，Polling System 就需要修改。为此，该架构在后来做了一次改进，引入了事件驱动的机制，读取数据库中所有实现的事务，通过 stream processing 的方式传递到下一个 job 进行处理。为了普适化该解决方案，在 source 端实现了不同数据库的 CDC（Change Data Capture）支持，包括 MySQL、PostgreSQL 和 Cassandra 等在 Netflix 中比较常用的数据库，通过 Keystone 的 pipeline 进行处理。

下面分享一下上述方案存在的挑战和相应的解决方案：

- **Ordering Semantics。**

  在变更数据事件中，必须要保证 Event ordering，比如一个事件包含 create、update 和 delete 是三个操作，需要返回给消费者侧一个严格遵守该顺序的操作事件。一个解决方案是通过 Kafka 来控制；另一个解决方案是在分布式系统中保证捕获的事件与实际从从数据库中读取数据的顺序是一致的，该方案中当所有的变更事件捕获出来后，会存在重复和乱序的情况，会通过 Flink 进行去重和重新排序。

- **Processing Contracts。**

  在写 stream processing 的时候，很多情况下不知道 Schema 的具体信息，因此需要在消息上定义一个契约 contract，包括 Wire Format 以及在不同的层级上定义与 Schema 相关的信息，如基础设施（Infrastructure）、平台（Platform）等。Processor Contract 的目的是帮助用户将不同的 processor metadata 组合起来，尽量减少其写重复代码的可能。举一个具体的案例，比如 Charlie 希望有新的 deal 的时候被及时通知，平台通过将相关的不同组件组合起来，DB Connector、Filter 等，通过用户定义契约的方式帮助其实现一个开放的可组合的流数据平台。

以往所看到的 ETL 工程大多数适用于数据工程师或数据科学家。但从经验上来讲，ETL 的整个过程，即 Extract、Transform 和 Load，其实是有被更广泛应用的可能。最早的 Keystone 简单易用，但灵活性不高，之后的发展过程中虽然提高了灵活性，但复杂性也相应地增大了。因此未来团队计划在目前的基础上进一步优化，推出开放的、合作的、可组合的、可配置的 ETL 工程平台，帮助用户在极短的时间解决问题。

OPPO 基于 Apache Flink 的实时数仓实践

作者：张俊（Apache Flink Contributor，OPPO 大数据平台研发负责人）

文章整理自 2019 年 4 月 13 日在深圳举行的 Flink Meetup 会议，分享嘉宾张俊，目前担任 OPPO 大数据平台研发负责人，也是 Apache Flink contributor。本文主要内容如下：

OPPO 实时数仓的演进思路；

基于 Flink SQL 的扩展工作；

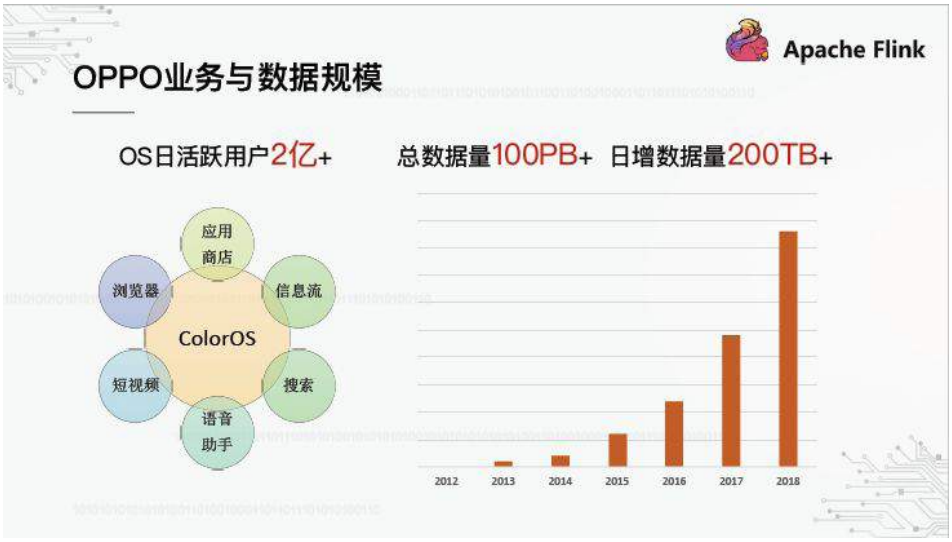
构建实时数仓的应用案例；

未来工作的思考和展望。

一、OPPO 实时数仓的演进思路

1.1 OPPO 业务与数据规模

大家都知道 OPPO 是做智能手机的，但并不知道 OPPO 与互联网以及大数据有什么关系，下图概要介绍了 OPPO 的业务与数据情况：



OPPO 作为手机厂商，基于 Android 定制了自己的 ColorOS 系统，当前日活跃用户超过 2 亿。围绕 ColorOS，OPPO 构建了很多互联网应用，比如应用商店、浏览器、信息流等。在运营这些互联网应用的过程中，OPPO 积累了大量的数据，上图右边是整体数据规模的演进：从 2012 年开始每年都是 2~3 倍的增长速度，截至目前总数据量已经超过 100PB，日增数据量超过 200TB。

要支撑这么大的一个数据量，OPPO 研发出一整套的数据系统与服务，并逐渐形成了自己的数据中台体系。

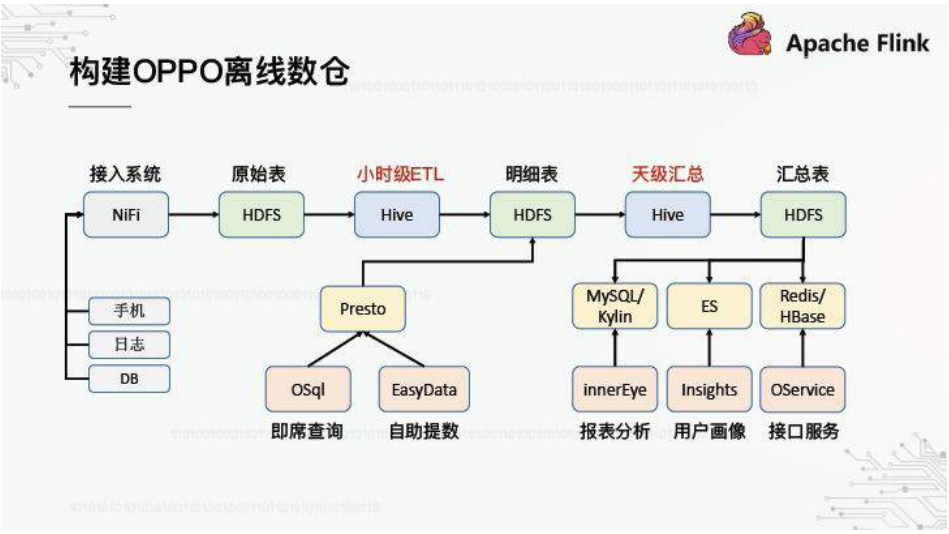
1.2 OPPO 数据中台

今年大家都在谈数据中台，OPPO 是如何理解数据中台的呢？我们把它分成了 4 个层次：

- 最下层是统一工具体系，涵盖了 " 接入 – 治理 – 开发 – 消费 " 全数据链路；
- 基于工具体系之上构建了数据仓库，划分成 " 原始层 – 明细层 – 汇总层 – 应用层 "，这也是经典的数仓架构；
- 再往上是全域的数据体系，什么是全域呢？就是把公司所有的业务数据都打通，形成统一的数据资产，比如 ID-Mapping、用户标签等；
- 最终，数据要能被业务用起来，需要场景驱动的数据产品与服务。

以上就是 OPPO 数据中台的整个体系，而数据仓库在其中处于非常基础与核心的位置。

1.3 构建 OPPO 离线数仓



过往 2、3 年，我们的重点聚焦在离线数仓的构建。上图大致描述了整个构建过程：首先，数据来源基本是手机、日志文件以及 DB 数据库，我们基于 Apache NiFi 打造了高可用、高吞吐的接入系统，将数据统一落入 HDFS，形成原始层；紧接着，基于 Hive 的小时级 ETL 与天级汇总 Hive 任务，分别负责计算生成明细层与汇总层；最后，应用层是基于 OPPO 内部研发的数据产品，主要是报表分析、用户画像以及接口服务。此外，中间的明细层还支持基于 Presto 的即席查询与自助提数。

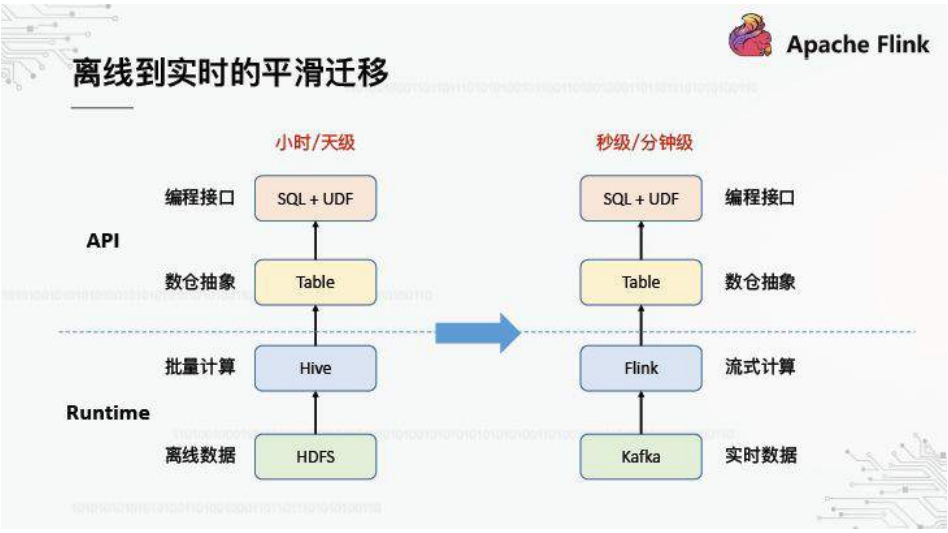
伴随着离线数仓的逐步完善，业务对实时数仓的诉求也愈发强烈。

1.4 数仓实时化的诉求

对于数仓实时化的诉求，大家通常都是从业务视角来看，但其实站在平台的角度，实时化也能带来切实的好处。首先，从业务侧来看，报表、标签、接口等都会有实时的应用场景，分别参见上图左边的几个案例；其次，对平台侧来说，我们可以从三个案例来看：第一，OPPO 大量的批量任务都是从 0 点开始启动，都是通过 T+1 的方式去做数据处理，这会导致计算负载集中爆发，对集群的压力很大；第二，标签导入也属于一种 T+1 批量任务，每次全量导入都会耗费很长的时间；第三，数据质量的监控也必须是 T+1 的，导致没办法及时发现数据的一些问题。

既然业务侧和平台侧都有实时化的这个诉求，那 OPPO 是如何来构建自己的实时数仓呢?

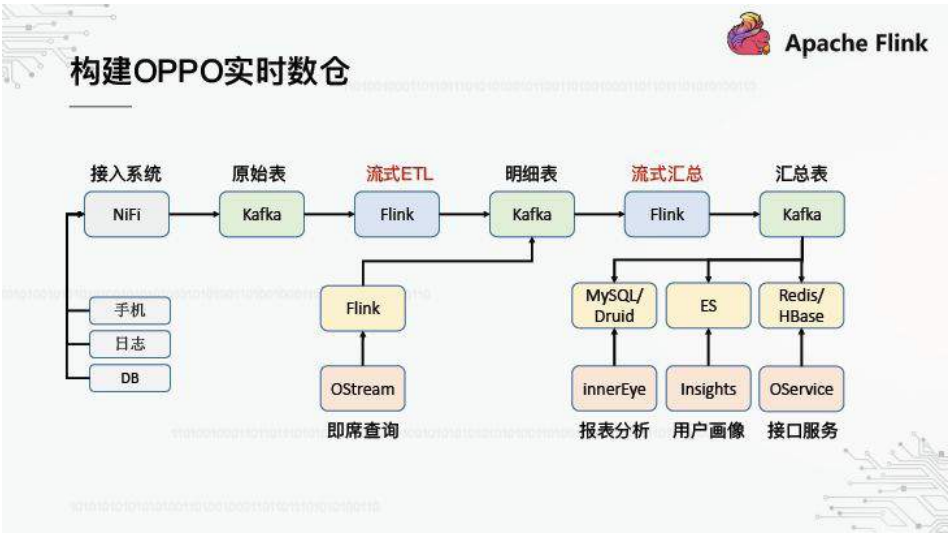
1.5 离线到实时的平滑迁移



无论是一个平台还是一个系统，都离不开上下两个层次的构成：上层是 API，是面向用户的编程抽象与接口；下层是 Runtime，是面向内核的执行引擎。我们希望从离线到实时的迁移是平滑的，是什么意思呢？从 API 这层来看，数仓的抽象是 Table、编程接口是 SQL+UDF，离线数仓时代用户已经习惯了这样的 API，迁移到实时数仓后最好也能保持一致。而从 Runtime 这层来看，计算引擎从 Hive 演进到了 Flink，存储引擎从 HDFS 演进到了 Kafka。

基于以上的思路，只需要把之前提到的离线数仓 pipeline 改造下，就得到了实时数仓 pipeline。

1.6 构建 OPPO 实时数仓



从上图可以看到，整个 pipeline 与离线数仓基本相似，只是把 Hive 替换为 Flink，把 HDFS 替换为 Kafka。从总体流程来看，基本模型是不变的，还是由原始层、明细层、汇总层、应用层的级联计算来构成。

因此，这里的核心问题是如何基于 Flink 构建出这个 pipeline，下面就介绍下我们基于 Flink SQL 所做的一些工作。

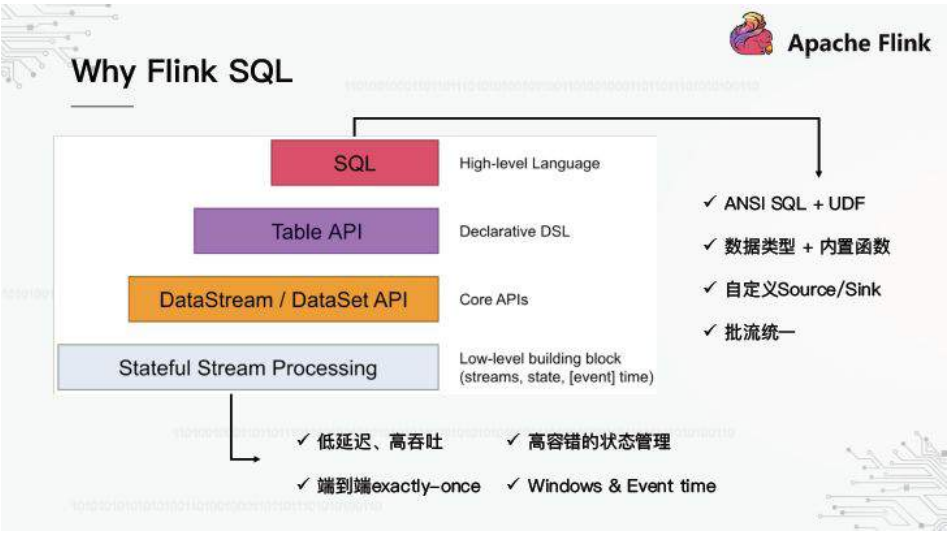
二、基于 Flink SQL 的扩展工作

2.1 Why Flink SQL

首先，为什么要用 Flink SQL？下图展示了 Flink 框架的基本结构，最下面是 Runtime，这个执行引擎我们认为最核心的优势是四个：第一，低延迟，高吞吐；第二，端到端的 Exactly-once；第三，可容错的状态管理；第四，Window & Event time 的支持。基于 Runtime 抽象出 3 个层次的 API，SQL 处于最上层。

Flink SQL API 有哪些优势呢？我们也从四个方面去看：第一，支持 ANSI SQL

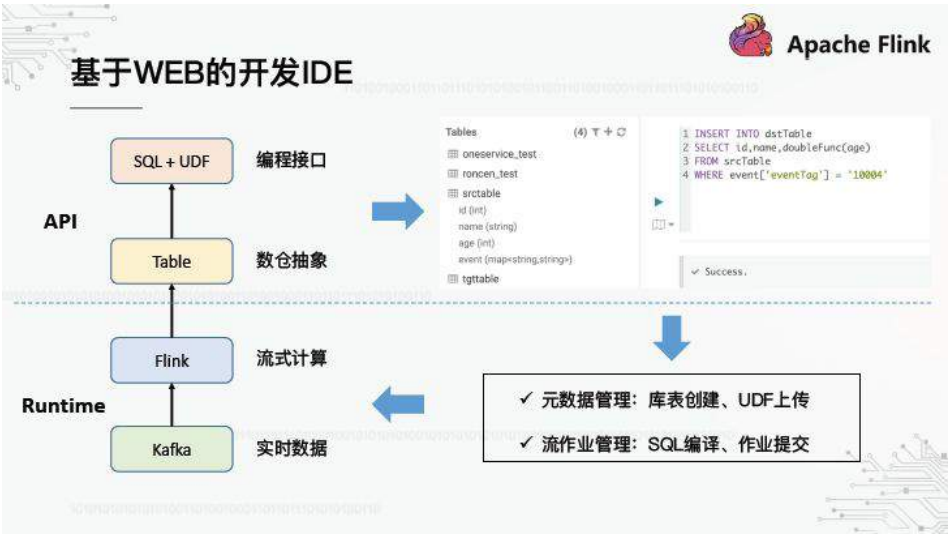
的标准；第二，支持丰富的数据类型与内置函数，包括常见的算术运算与统计聚合；第三，可自定义 Source/Sink，基于此可以灵活地扩展上下游；第四，批流统一，同样的 SQL，既可以跑离线也可以跑实时。



那么，基于 Flink SQL API 如何编程呢？下面是一个简单的演示：

首先是定义与注册输入 / 输出表，这里创建了 2 张 Kakfa 的表，指定 kafka 版本是什么、对应哪个 topic；接下来是注册 UDF，篇幅原因这里没有列出 UDF 的定义；最后是才是执行真正的 SQL。可以看到，为了执行 SQL，需要做这么多的编码工作，这并不是我们希望暴露给用户的接口。

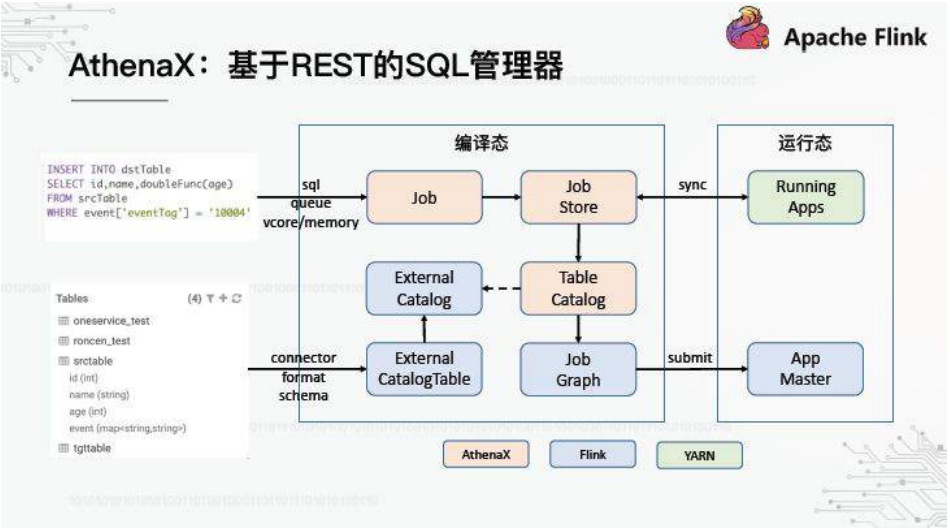
2.2 基于 WEB 的开发 IDE



前面提到过，数仓的抽象是 Table，编程接口是 SQL+UDF。对于用户来说，平台提供的编程界面应该是类似上图的那种，有用过 HUE 做交互查询的应该很熟悉。左边的菜单是 Table 列表，右边是 SQL 编辑器，可以在上面直接写 SQL，然后提交执行。要实现这样一种交互方式，Flink SQL 默认是无法实现的，中间存在 gap，总结下来就 2 点：第一，元数据的管理，怎么去创建库表，怎么去上传 UDF，使得之后在 SQL 中可直接引用；第二，SQL 作业的管理，怎么去编译 SQL，怎么去提交作业。

在技术调研过程中，我们发现了 Uber 在 2017 年开源的 AthenaX 框架。

2.3 AthenaX：基于 REST 的 SQL 管理器



AthenaX 可以看作是一个基于 REST 的 SQL 管理器，它是怎么实现 SQL 作业与元数据管理的呢?

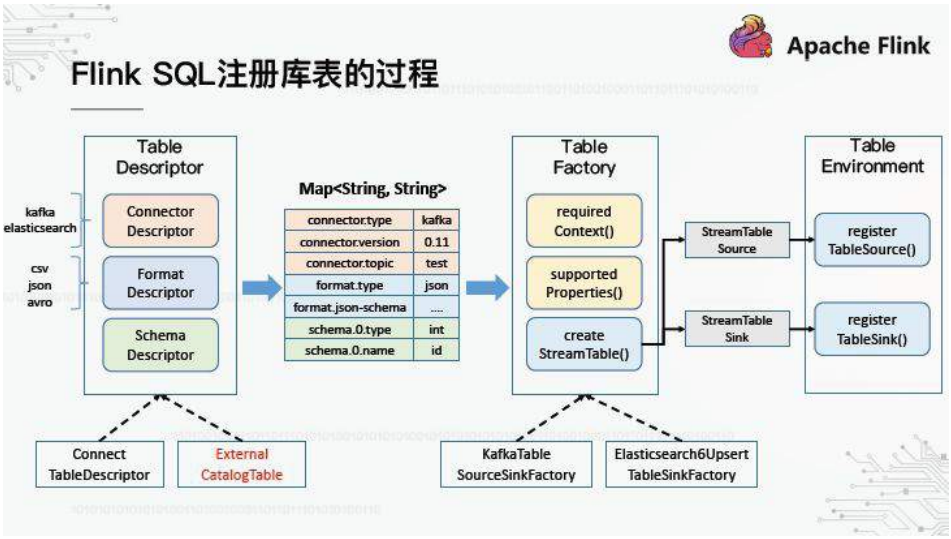
- 对于 SQL 作业提交，AthenaX 中有一个 Job 的抽象，封装了要执行的 SQL 以及作业资源等信息。所有的 Job 由一个 JobStore 来托管，它定期跟 YARN 当中处于 Running 状态的 App 做一个匹配。如果不一致，就会向 YARN 提交对应的 Job。
- 对于元数据管理，核心的问题是如何将外部创建的库表注入 Flink，使得 SQL 中可以识别到。实际上，Flink 本身就预留了与外部元数据对接的能力，分别提供了 ExternalCatalog 和 ExternalCatalogTable 这两个抽象。AthenaX 在此基础上再封装出一个 TableCatalog，在接口层面做了一定的扩展。在提交 SQL 作业的阶段，AthenaX 会自动将 TableCatalog 注册到 Flink，再调用 Flink SQL 的接口将 SQL 编译为 Flink 的可执行单元 JobGraph，并最终提交到 YARN 生成新的 App。

AthenaX 虽然定义好了 TableCatalog 接口，但并没有提供可直接使用的实现。

那么，我们怎么来实现，以便对接到我们已有的元数据系统呢?

2.4 Flink SQL 注册库表的过程

首先，我们得搞清楚 Flink SQL 内部是如何注册库表的。整个过程涉及到三个基本的抽象：TableDescriptor、TableFactory 以及 TableEnvironment。



TableDescriptor 顾名思义，是对表的描述，它由三个子描述符构成：第一是 Connector，描述数据的来源，比如 Kafka、ES 等；第二是 Format，描述数据的格式，比如 csv、json、avro 等；第三是 Schema，描述每个字段的名称与类型。TableDescriptor 有两个基本的实现——ConnectTableDescriptor 用于描述内部表，也就是编程方式创建的表；ExternalCatalogTable 用于描述外部表。

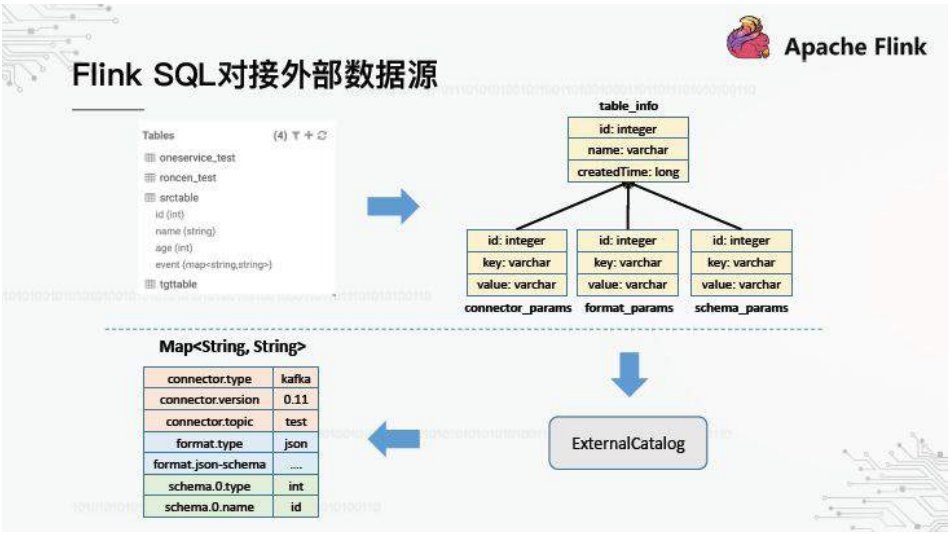
有了 TableDescriptor，接下来需要 TableFactory 根据描述信息来实例化 Table。不同的描述信息需要不同的 TableFactory 来处理，Flink 如何找到匹配的 TableFactory 实现呢? 实际上，为了保证框架的可扩展性，Flink 采用了 Java SPI 机制来加载所有声明过的 TableFactory，通过遍历的方式去寻找哪个 TableFactory 是匹配该 TableDescriptor 的。TableDescriptor 在传递给 TableFactory 前，被转

换成一个 map，所有的描述信息都用 key-value 形式来表达。TableFactory 定义了两个用于过滤匹配的方法——一个是 requiredContext()，用于检测某些特定 key 的 value 是否匹配，比如 connector.type 是否为 kakfa；另一个是 supported-Properties()，用于检测 key 是否能识别，如果出现不识别的 key，说明无法匹配。

匹配到了正确的 TableFactory，接下来就是创建真正的 Table，然后将其通过 TableEnvironment 注册。最终注册成功的 Table，才能在 SQL 中引用。

2.5 Flink SQL 对接外部数据源

搞清楚了 Flink SQL 注册库表的过程，给我们带来这样一个思路：如果外部元数据创建的表也能被转换成 TableFactory 可识别的 map，那么就能被无缝地注册到 TableEnvironment。基于这个思路，我们实现了 Flink SQL 与已有元数据中心的对接，大致过程参见下图：



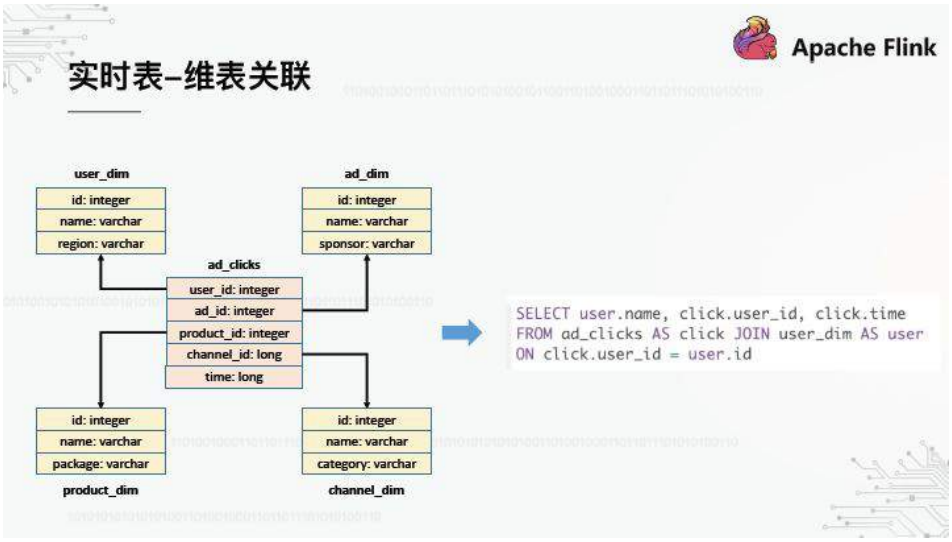
通过元数据中心创建的表，都会将元数据信息存储到 MySQL，我们用一张表来记录 Table 的基本信息，然后另外三张表分别记录 Connector、Format、Schema 转换成 key-value 后的描述信息。之所以拆开成三张表，是为了能够能独立的更新

这三种描述信息。接下来是定制实现的 ExternalCatalog，能够读取 MySQL 这四张表，并转换成 map 结构。

2.6 实时表 – 维表关联

到目前为止，我们的平台已经具备了元数据管理与 SQL 作业管理的能力，但是要真正开放给用户使用，还有一点基本特性存在缺失。通过我们去构建数仓，星型模型是无法避免的。这里有一个比较简单的案例：中间的事实表记录了广告点击流，周边是关于用户、广告、产品、渠道的维度表。

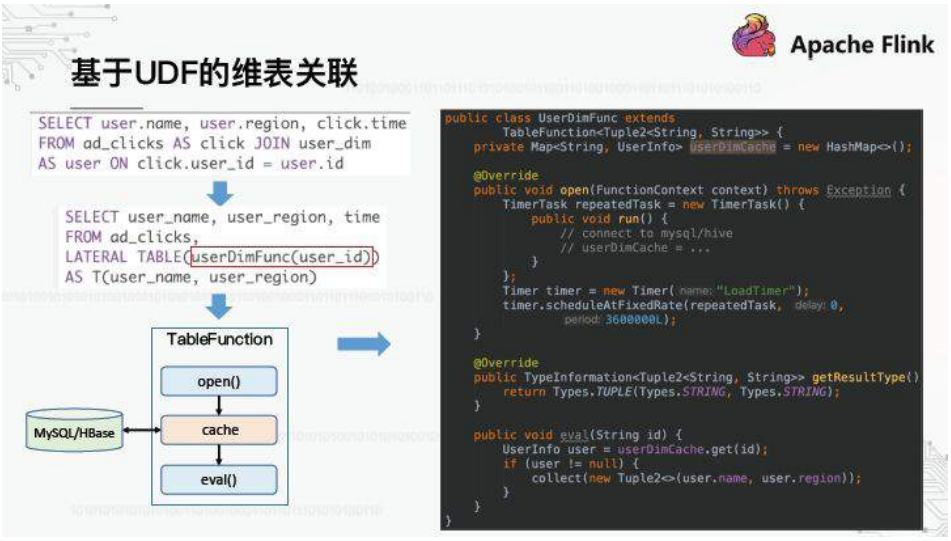
假定我们有一个 SQL 分析，需要将点击流表与用户维表进行关联，这个目前在 Flink SQL 中应该怎么来实现？我们有两种实现方式，一个基于 UDF，一个基于 SQL 转换，下面分别展开来讲一下。



2.7 基于 UDF 的维表关联

首先是基于 UDF 的实现，需要用户将原始 SQL 改写为带 UDF 调用的 SQL，这里是 userDimFunc，上图右边是它的代码实现。UserDimFunc 继承了 Flink

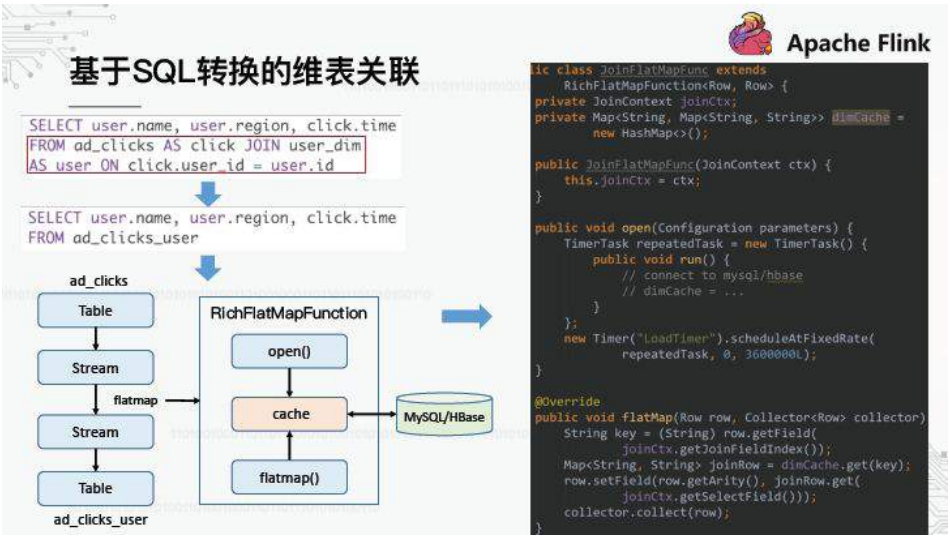
SQL 抽象的 TableFunction，它是其中一种 UDF 类型，可以将任意一行数据转换成一行或多行数据。为了实现维表关联，在 UDF 初始化时需要从 MySQL 全量加载维表的数据，缓存在内存 cache 中。后续对每行数据的处理，TableFunction 会调用 eval() 方法，在 eval() 中根据 user_id 去查找 cache，从而实现关联。当然，这里是假定维表数据比较小，如果数据量很大，不适合全量的加载与缓存，这里不做展开了。



基于 UDF 的实现，对用户和平台来说都不太友好：用户需要写奇怪的 SQL 语句，比如图中的 LATERAL TABLE；平台需要为每个关联场景定制特定的 UDF，维护成本太高。有没有更好的方式呢？下面我们来看看基于 SQL 转换的实现。

2.8 基于 SQL 转换的维表关联

我们希望解决基于 UDF 实现所带来的问题，用户不需要改写原始 SQL，平台不需要开发很多 UDF。有一种思路是，是否可以在 SQL 交给 Flink 编译之前，加一层 SQL 的解析与改写，自动实现维表的关联？经过一定的技术调研与 POC，我们发现是行得通的，所以称之为基于 SQL 转换的实现。下面将该思路展开解释下。



首先，增加的 SQL 解析是为了识别 SQL 中是否存在预先定义的维度表，比如上图中的 user_dim。一旦识别到维表，将触发 SQL 改写的流程，将红框标注的 join 语句改写成新的 Table，这个 Table 怎么得到呢？我们知道，流计算领域近年来发展出“流表二象性”的理念，Flink 也是该理念的践行者。这意味着，在 Flink 中 Stream 与 Table 之间是可以相互转换的。我们把 ad_clicks 对应的 Table 转换成 Stream，再调用 flatmap 形成另一个 Stream，最后再转换回 Table，就得到了 ad_clicks_user。最后的问题是，flatmap 是如何实现维表关联的？

Flink 中对于 Stream 的 flatmap 操作，实际上是执行一个 RichFlatmap-Funciton，每来一行数据就调用其 flatmap() 方法做转换。那么，我们可以定制一个 RichFlatmapFunction，来实现维表数据的加载、缓存、查找以及关联，功能与基于 UDF 的 TableFunction 实现类似。

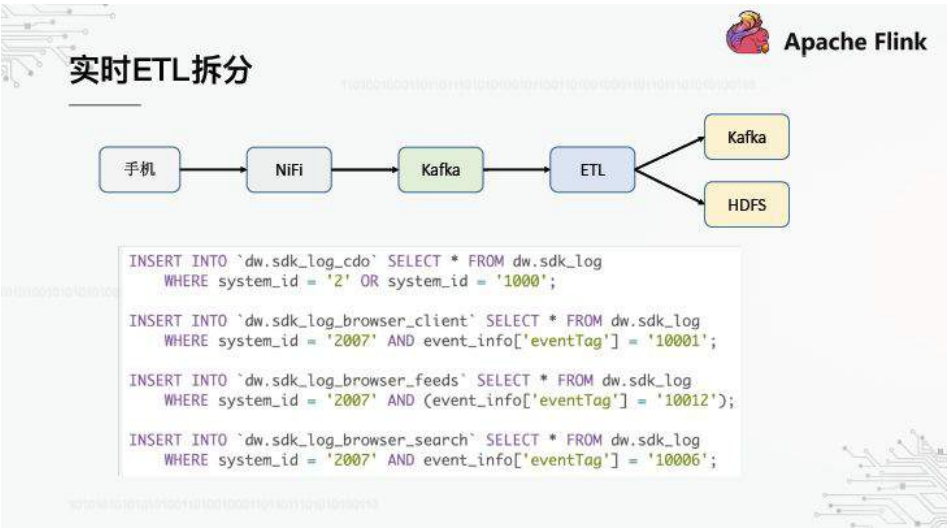
既然 RichFlatmapFunciton 的实现逻辑与 TableFunction 相似，那为什么相比基于 UDF 的方式，这种实现能更加通用呢？核心的点在于多了一层 SQL 解析，可以将维表的信息获取出来（比如维表名、关联字段、select 字段等），再封装成 JoinContext 传递给 RichFlatmapFunciton，使得的表达能力就具备通用性了。

三、构建实时数仓的应用案例

下面分享几个典型的应用案例，都是在我们的平台上用 Flink SQL 来实现的。

3.1 实时 ETL 拆分

这里是一个典型的实时 ETL 链路，从大表中拆分出各业务对应的小表：



OPPO 的最大数据来源是手机端埋点，从手机 APP 过来的数据有一个特点，所有的数据是通过统一的几个通道上报过来。因为不可能每一次业务有新的埋点，都要去升级客户端，去增加新的通道。比如我们有个 sdk_log 通道，所有 APP 应用的埋点都往这个通道上报数据，导致这个通道对应的原始层表巨大，一天几十个 TB。但实际上，每个业务只关心它自身的那部分数据，这就要求我们在原始层进行 ETL 拆分。

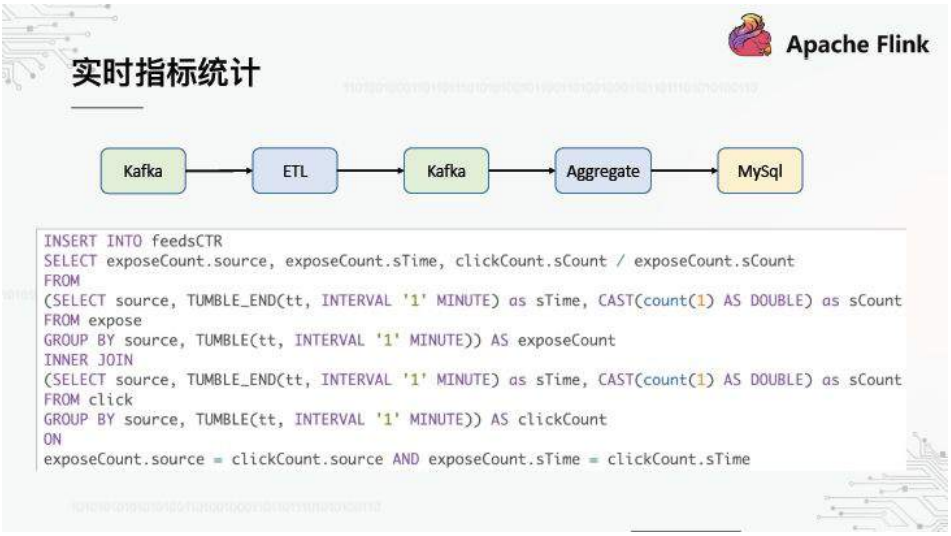
这个 SQL 逻辑比较简单，无非是根据某些业务字段做筛选，插入到不同的业务表中去。它的特点是，多行 SQL 最终合并成一个 SQL 提交给 Flink 执行。大家担心的是，包含了 4 个 SQL，会不会对同一份数据重复读取 4 次？其实，在 Flink 编译

SQL 的阶段是会做一些优化的，因为最终指向的是同一个 kafka topic，所以只会读取 1 次数据。

另外，同样的 Flink SQL，我们同时用于离线与实时数仓的 ETL 拆分，分别落入 HDFS 与 Kafka。Flink 中本身支持写入 HDFS 的 Sink，比如 RollingFileSink。

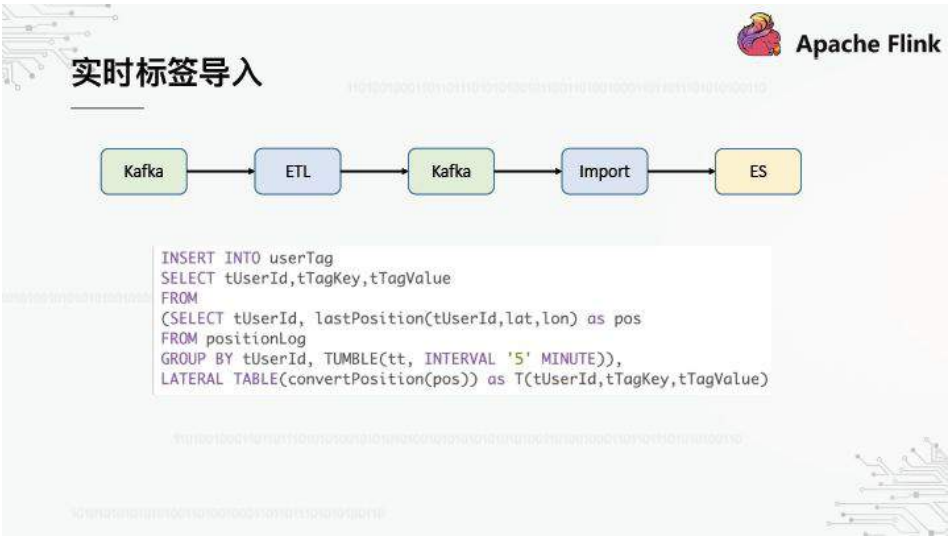
3.2 实时指标统计

这里是一个典型的计算信息流 CTR 的这个案例，分别计算一定时间段内的曝光与点击次数，相除得到点击率导入 Mysql，然后通过我们内部的报表系统来可视化。这个 SQL 的特点是它用到了窗口 (Tumbling Window) 以及子查询。

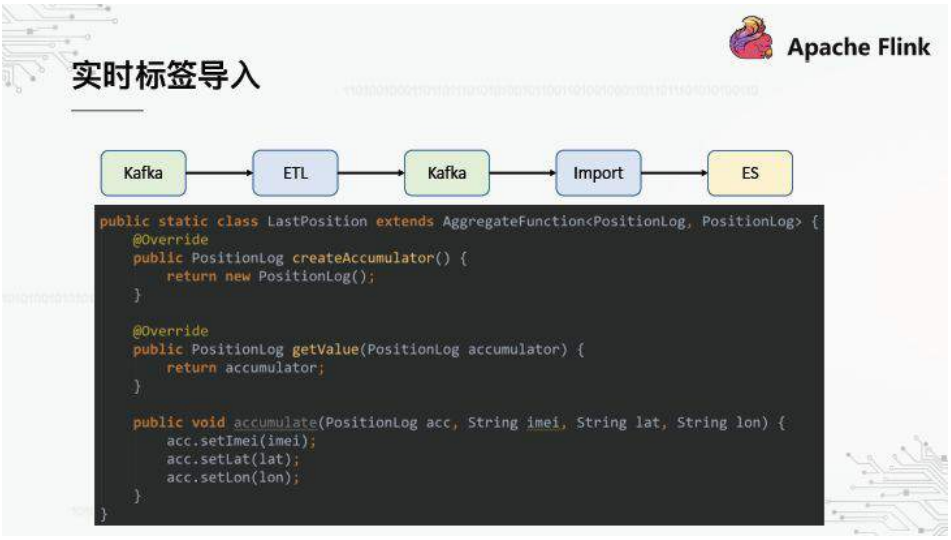


3.3 实时标签导入

这里是一个实时标签导入的案例，手机端实时感知到当前用户的经纬度，转换成具体 POI 后导入 ES，最终在标签系统上做用户定向。



这个 SQL 的特点是用了 AggregateFunction，在 5 分钟的窗口内，我们只关心用户最新一次上报的经纬度。AggregateFunction 是一种 UDF 类型，通常是用于聚合指标的统计，比如计算 sum 或者 average。在这个示例中，由于我们只关心最新的经纬度，所以每次都替换老的数据即可。



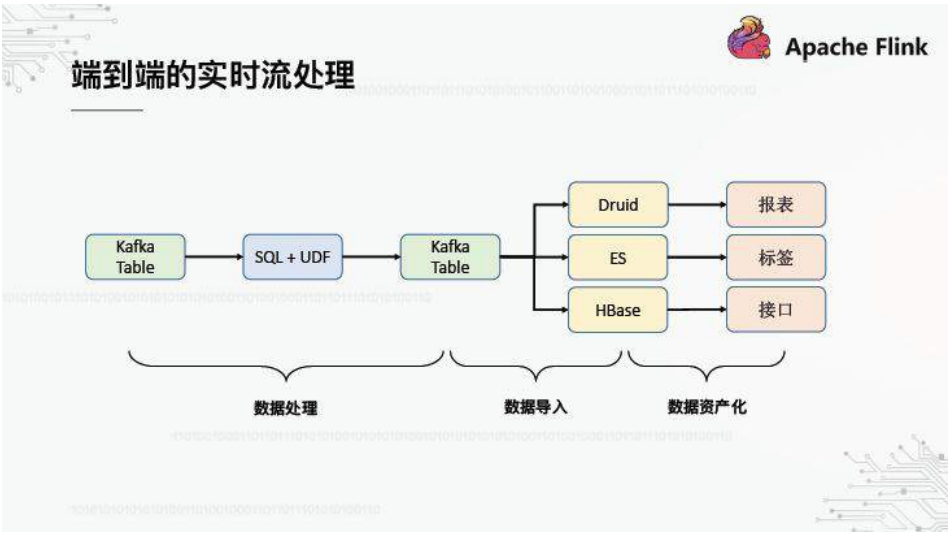
四、未来工作的思考和展望

最后，给大家分享一下关于未来工作，我们的一些思考与规划，还不是太成熟，抛出来和大家探讨一下。

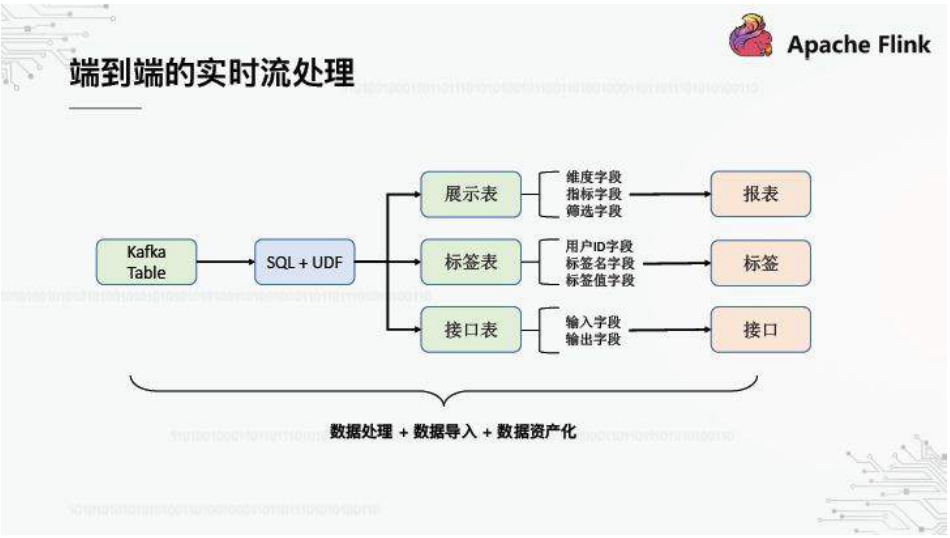
4.1 端到端的实时流处理

什么是端到端？一端是采集到的原始数据，另一端是报表 / 标签 / 接口这些对数据的呈现与应用，连接两端的是中间实时流。当前我们基于 SQL 的实时流处理，源表是 Kafka，目标表也是 Kafka，统一经过 Kafka 后再导入到 Druid/ES/HBase。这样设计的目的是提高整体流程的稳定性与可用性：首先，kafka 作为下游系统的缓冲，可以避免下游系统的异常影响实时流的计算（一个系统保持稳定，比起多个系统同时稳定，概率上更高点）；其次，kafka 到 kafka 的实时流，exactly-once 语义是比较成熟的，一致性上有保证。

然后，上述的端到端其实是由割裂的三个步骤来完成的，每一步可能需要由不同角色人去负责处理：数据处理需要数据开发人员，数据导入需要引擎开发人员，数据资产化需要产品开发人员。



我们的平台能否把端到端给自动化起来，只需要一次 SQL 提交就能打通处理、导入、资产化这三步？在这个思路下，数据开发中看到的不再是 Kafka Table，而应该是面向场景的展示表 / 标签表 / 接口表。比如对于展示表，创建表的时候只要指定维度、指标等字段，平台会将实时流结果数据从 Kafka 自动导入 Druid，再在报表系统自动导入 Druid 数据源，甚至自动生成报表模板。

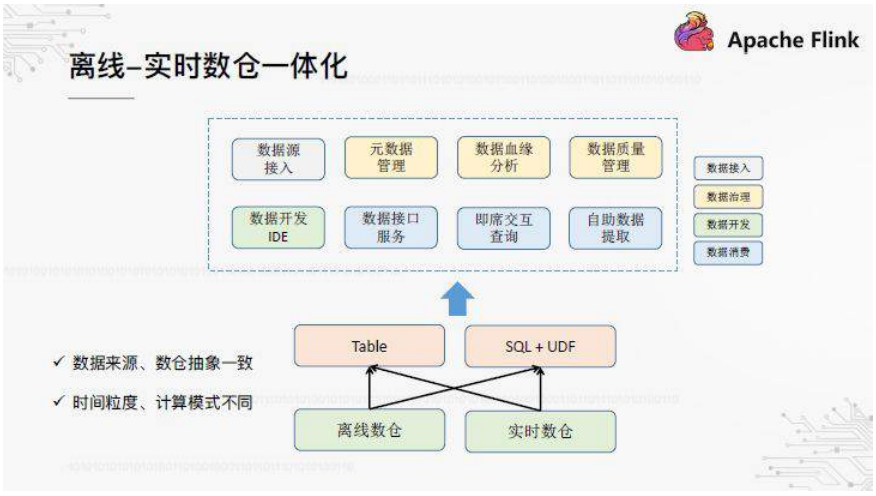


4.2 实时流的血缘分析

关于血缘分析，做过离线数仓的朋友都很清楚它的重要性，它在数据治理中都起着不可或缺的关键作用。对于实时数仓来说也莫不如此。我们希望构建端到端的血缘关系，从采集系统的接入通道开始，到中间流经的实时表与实时作业，再到消费数据的产品，都能很清晰地展现出来。基于血缘关系的分析，我们才能评估数据的应用价值，核算数据的计算成本。

4.3 离线 – 实时数仓一体化

最后提一个方向是离线实时数仓的一体化。我们认为短期内，实时数仓无法替代离线数仓，两者并存是新常态。在离线数仓时代，我们积累的工具体系，如何去适配实时数仓，如何实现离线与实时数仓的一体化管理？理论上来讲，它们的数据来源是一致的，上层抽象也都是 Table 与 SQL，但本质上也有不同的点，比如时间粒度以及计算模式。对于数据工具与产品来说，需要做哪些改造来实现完全的一体化，这也是我们在探索和思考的。



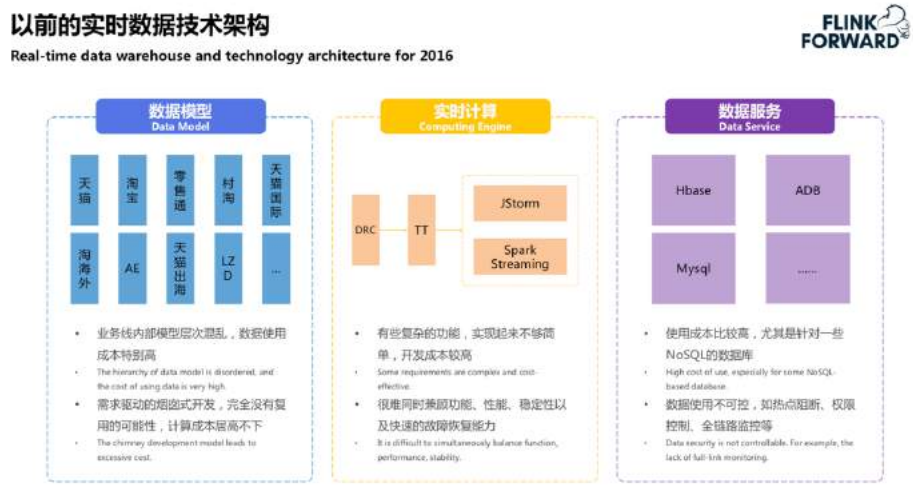
菜鸟供应链实时数仓的架构演进及应用场景

作者：贾元乔（菜鸟高级数据技术专家）

摘要：在 Flink Forward Asia 大会实时数仓专场中，菜鸟数据 & 规划部高级数据技术专家贾元乔从数据模型、数据计算、数据服务等几个方面介绍了菜鸟供应链数据团队在实时数据技术架构上的演进，以及在供应链场景中典型的实时应用场景和 Flink 的实现方案。

首先从三个方面简要介绍一下菜鸟早在 2016 年采用的实时数据技术架构：数据模型、实时计算和数据服务。

- **数据模型。**菜鸟最初使用的是需求驱动的、纵向烟囱式的开发模式，其计算成本高且完全没有复用的可能性，同时也会导致数据一致性的问题；整个数据模型没有分层，业务线内部模型层次混乱，使得数据使用成本特别高。
- **实时计算。**该部分使用的是阿里的 JStorm 和 Spark Streaming，大多数情况下，二者可以满足实时计算的需求，但是对于有些复杂的功能，如物流和供应链场景，实现起来不够简单，开发成本较高；同时很难兼顾功能、性能、稳定性以及快速的故障恢复能力。
- **数据服务。**数据主要存储在 Hbase、MySQL 和 ADB 等不同类型的数据库中，然而对于很多运营人员来说，查询数据库的频率并不高，其使用数据库的成本较高，尤其针对一些 NoSQL 的数据库；数据使用不可控，如热点阻断、权限控制以及全链路监控等。

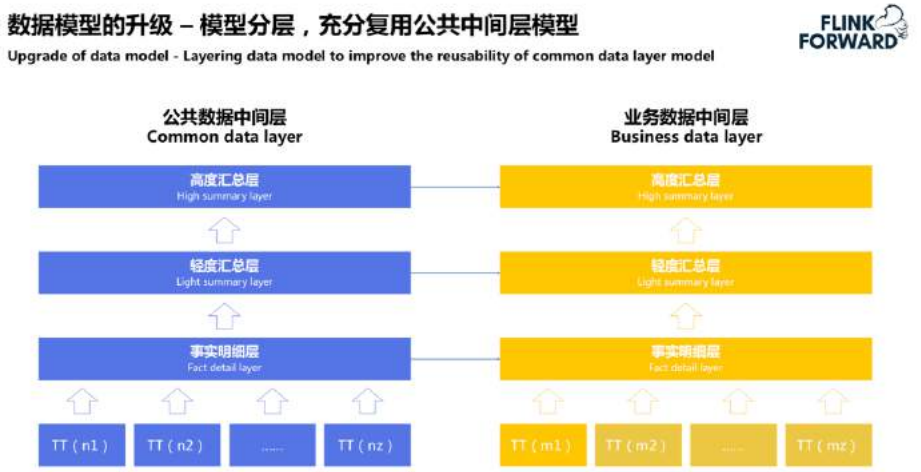


针对以上问题，菜鸟在 2017 年对数据技术架构进行了一次比较大的升级改造，以下将详细介绍。

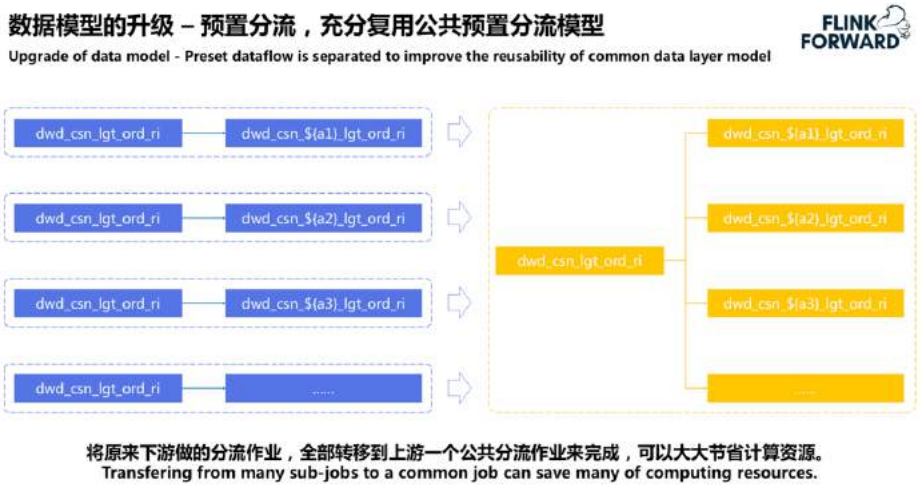
数据模型升级

数据模型的升级主要是模型分层，充分复用公共中间层模型。之前的模式是从数据源 TT（如 Kafka）中抽取数据并进行加工，产生一层式的表结构。新版本的数据模型进行了分层，第一层是数据采集，支持多种数据库中的数据采集，同时将采集到的数据放入消息中间件中；第二层是事实明细层，基于 TT 的实时消息产生事实明细表，然后再写入 TT 的消息中间件中，通过发布订阅的方式汇总到第三、四层，分别是轻度汇总层和高度汇总层。轻度汇总层适合数据维度、指标信息比较多的情况，如大促统计分析的场景，该层的数据一般存入阿里自研的 ADB 数据库中，用户可以根据自己的需求筛选出目标指标进行聚合；而高度汇总层则沉淀了一些公共粒度的指标，并将其写入 Hbase 中，支持大屏的实时数据显示场景，如媒体大屏、物流大屏等。原本采用的开发模式各个业务线独立开发，不同业务线之间不考虑共性的问题，但物流场景中，很多功能需求其实是类似的，这样往往会造成资源的浪费，针对该问题进行的改造首先是抽象出横向的公共数据中间层（左侧蓝色），然后各个业务线在此基础上

分流自己的业务数据中间层（右侧黄色）。



前面介绍的业务线分流由预置的公共分流任务来实现，即将原来下游做的分流作业，全部转移到上游的一个公共分流作业来完成，充分复用公共预置分流模型，大大节省计算资源。



下面介绍一个数据模型升级的具体案例—菜鸟供应链实时数据模型。下图左侧是前面介绍的公共数据中间层，包括整个菜鸟横向的物流订单、大盘物流详情和公共粒度的一些数据，在此基础上菜鸟实现了预置公共分流，从物流订单、物流详情中拆分出个性化业务线的公共数据中间层，包括国内供应链、进口供应链以及出口供应链等。基于已经分流出来的公共逻辑，再加上业务线个性化 TT 的消息，产出各业务线的业务数据中间层。以进口供应链为例，其可能从公共业务线中分流出物流订单和物流详情，但是海关信息、干线信息等都在自己的业务线进口供应链的 TT 中，基于这些信息会产生该业务线的业务数据中间层。借助前面所述的设计理念，再加上实时的模型设计规范和实时的开发规范，大大提升了数据模型的易用性。

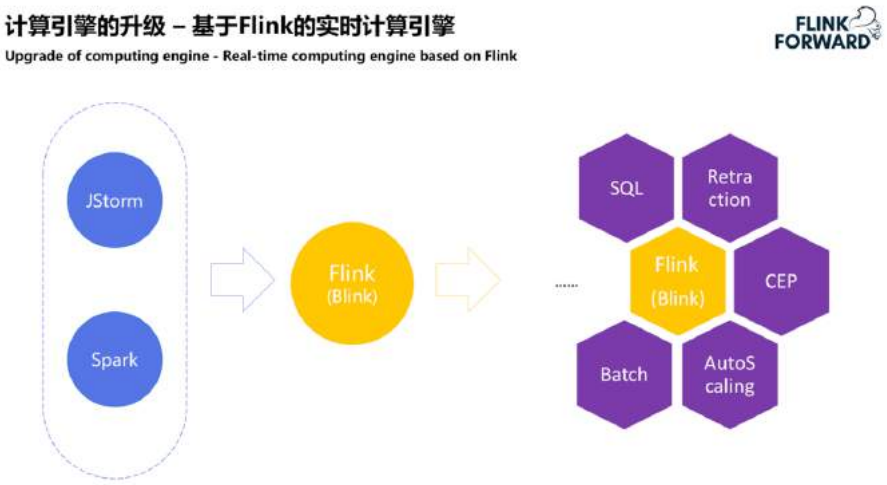
计算引擎升级

菜鸟最初的计算引擎采用的是阿里内部研发的 JStorm 和 Spark Streaming，可以满足大多数场景的需求，但针对一些复杂的场景，如供应链、物流等，会存在一些问题。因此，菜鸟在 2017 年全面升级为基于 Flink 的实时计算引擎。当时选择 Flink 的主要原因是：

- Flink 提供的很多功能非常适用于解决供应链场景下的需求，菜鸟内部提炼了

一套 Flink 的 SQL 语法，简单易用且标准化，大大提升了开发效率。

- 此外，Flink 内置的基于 state 的 Retraction 的机制可以很好地支持供应链场景下的取消订单、换配需求的实现；
- 后来推出的 CEP 功能使得物流、供应链中实时超时统计需求的实现变得更加简单；
- AutoScaling 等自动优化的方案可以使得菜鸟省去了一些资源配置等方面的复杂性和成本；
- 半智能功能如批流混合等也较好地满足菜鸟业务的实际需求。



下面介绍三个与计算引擎升级相关的案例。

案例 1：基于 state 的 Retraction

下图左侧是一个物流订单表，包含了四列数据，即物流订单号、创建时间、是否取消和计划配送公司。假设有一个需求是统计某个配送公司计划履行的有效单量是多少，该需求看起来简单，实际实现过程中有有一些问题需要注意。

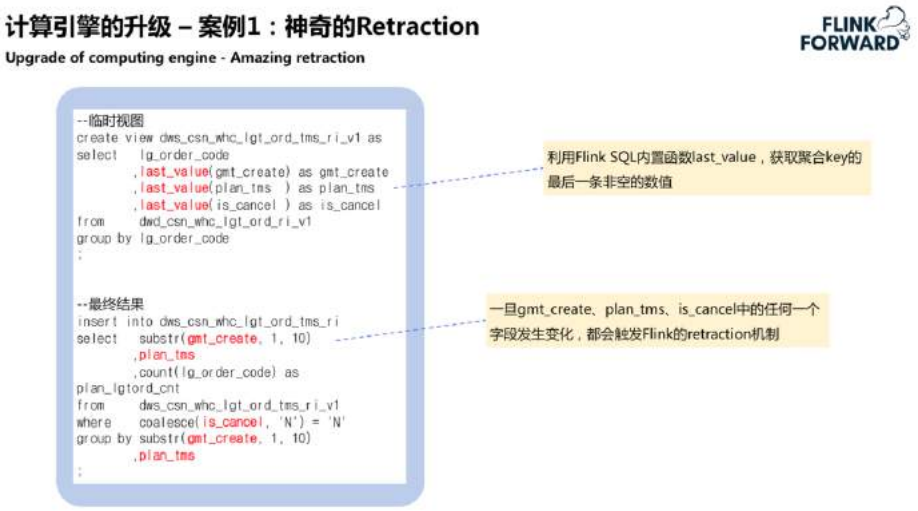
- 一个问题是针对表中 LP3 订单，在开始的时候是有效的（18 分的时候“是否取消”应该是 N，表写错），然而最后该订单却被取消了（最后一行“是否取

消”应该是 Y，表写错），这种情况该订单被视为无效订单，统计的时候不应该考虑在内。

- 另外，配送公司的转变也需要注意，LP1 订单在 1 分钟的时候计划配送公司还是 tmsA，而之后计划配送公司变成了 tmsB 和 tmsC，按照离线的计算方式（如 Storm 或增量）会得出右上角的结果，tmsA、tmsB 和 tmsC 与 LP1 订单相关的记录都会被统计，事实上 tmsA 和 tmsB 都未配送该订单，因此该结果实际上是错误的，正确的结果应该如图右下角表格所示。

针对该场景，Flink 内置提供了基于 state 的 Retraction 机制，可以帮助轻松实现流式消息的回撤统计。

下图展示了 Retraction 机制的伪代码实现。第一步是利用 Flink SQL 内置行数 last_value，获取聚合 key 的最后一条非空的数值，针对上述表中的 LP1 订单，使用 last_value 得到的结果是 tmsC，是符合预期的结果。需要强调的一点是，左侧使用 last_value 统计的字段 gmt_create、plan_tms、is_cancel，一旦其中的任何一个字段发生变化，都会发生出发 Flink 的 Retraction 机制。



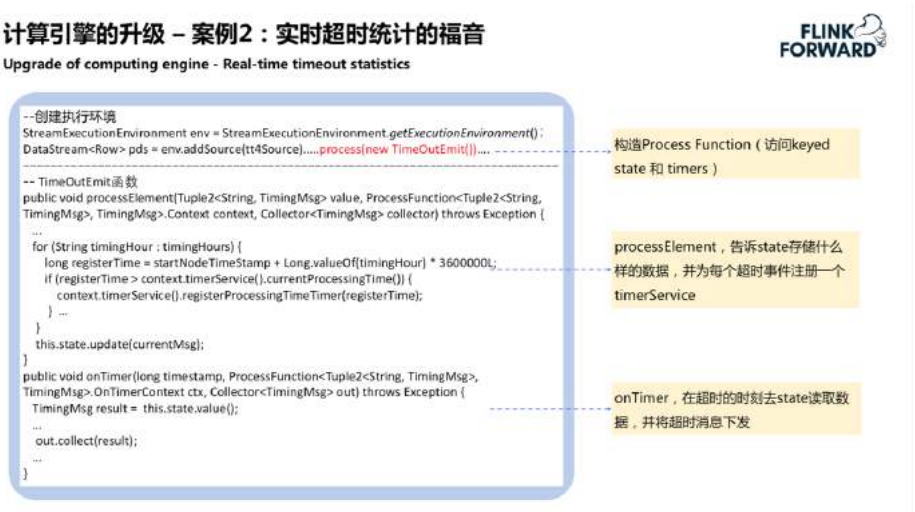
案例 2：超时统计

物流是菜鸟中比较常见的业务场景，物流业务中经常会有实时超时统计的需求，比如统计仓出库超过六个小时未被揽收的单量。用到的数据表如下图左侧所示，其中包含日志时间、物流订单号、出库时间和揽收时间。该需求如果在离线的小时表或天表中比较好实现，但是在实时的场景下，其实现面临一定的挑战。因为如果仓出库后未被揽收，意味着没有新的消息流入，如果没有消息就没有办法进行超时消息的计算。为了解决该问题，菜鸟从 2017 年初就开始了一系列的探索，发现一些消息中间件（如 Kafka）和 Flink CEP 等本身会提供超时消息下发的功能，引入消息中间件的维护成本比较高，而 Flink CEP 的应用会出现回传不准确的问题。

针对上述需求，菜鸟选择了 Flink Timer Service 来进行实现。具体来讲，菜鸟对 Flink 底层的 ProcessFunction 中的 ProcessElement 函数进行了改写，该函数中，由 Flink 的 state 存储原始消息，相同的主键只存一次，一旦 endNode 已实操，则 state 消息置为无效，已超时的消息直接下发。此外，重写编写一个 OnTimer 函数，主要负责在每个超时的时刻读取 state 消息，然后下发 state 中仍然有效的消息，基于下游和正常游的关联操作便可以统计出超时消息的单量。

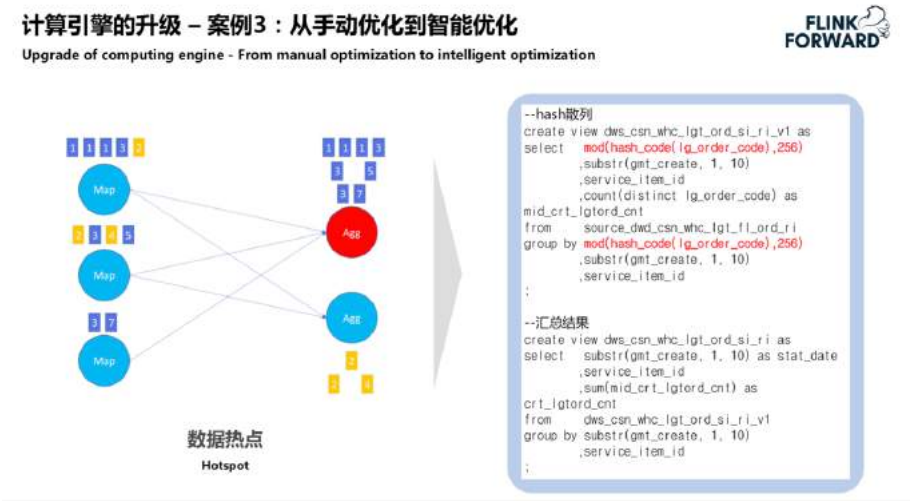
使用 Flink Timer Service 进行超时统计的伪代码实现如下图所示。

- 首先需要创建执行环境，构造 Process Function（访问 keyed state 和 times）；
- 其次是 processElement 函数的编写，主要用于告诉 state 存储什么样的数据，并为每个超时消息注册一个 timerService，代码中 timingHour 存储超时时间，比如前面的提到六小时，
- 然后启动 timerService；
- 最后是 onTimer 函数的编写，作用是在超时的时刻读取 state 的数据，并将超时消息下发。



案例 3：从手动优化到智能优化

实时数仓中会经常遇到数据热点和数据清洗的问题。下图左侧展示了数据热点的流程，蓝色部分 Map 阶段经过 Shuffle 后，转到红色部分 Agg，此时便会出现数据热点。针对该问题，菜鸟最初的解决方案的伪代码实现如下图右侧所示。假设对 lg_order_code 进行清洗，首先会对其进行 hash 散列操作，然后针对散列的结果进行二次聚合，这样便可以在一定程度上减轻倾斜度，因为可能会多一个 Agg 的操作。

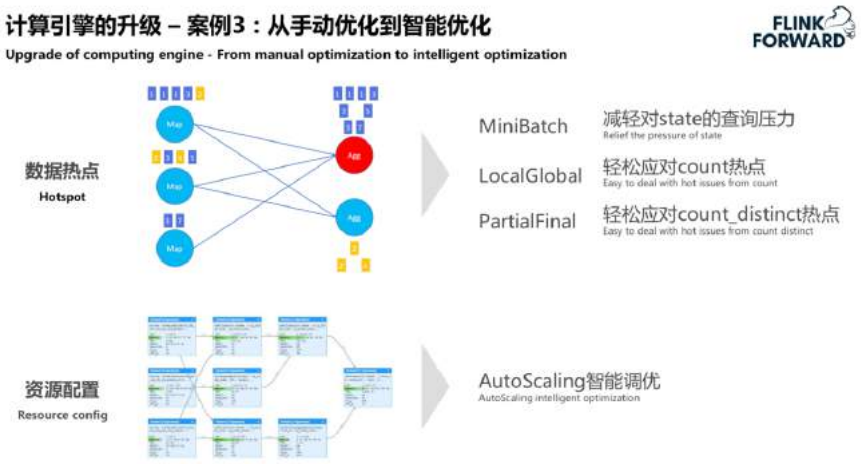


菜鸟内部目前使用的 Flink 最新版本提供了解决数据热点问题的智能化特性：

- **MiniBatch。**原来每进来一条数据，就需要去 state 中查询并写入，该功能可以将数据进行聚合后再写入 state 或从 state 中读取，从而减轻对 state 的查询压力。
- **LocalGlobal。**类似于 Hive 中 Map 阶段的聚合，通过该参数可以实现数据读取阶段的聚合，轻松应对 count 热点。
- **PartialFinal。**面对更复杂的场景，比如 count_distinct 的热点，使用该参数可以轻松应对，实现两次聚合，类似于 Hive 中的两次 Reduce 操作。

智能化功能支持的另一个场景是资源配置。在进行实时 ETL 过程中，首先要定义 DDL，然后 编写 SQL，之后需要进行资源配置。针对资源配置问题，菜鸟之前的方案是对每一个节点进行配置，包括并发量、是否会涉及消息乱序操作、CPU、内存等，一方面配置过程非常复杂，另一方面无法提前预知某些节点的资源消耗量。Flink 目前提供了较好的优化方案来解决该问题：

- **大促场景：**该场景下，菜鸟会提前预估该场景下的 QPS，会将其配置到作业中并重启。重启后 Flink 会自动进行压测，测试该 QPS 每个节点所需要的资源。
- **日常场景：**日常场景的 QPS 峰值可能远远小于大促场景，此时逐一配置 QPS 依然会很复杂。为此 Flink 提供了 AutoScaling 智能调优的功能，除了可以支持大促场景下提前设置 QPS 并压测获取所需资源，还可以根据上游下发的 QPS 的数据自动预估需要的资源。大大简化了资源配置的复杂度，使得开发人员可以更好地关注业务逻辑本身的开发。



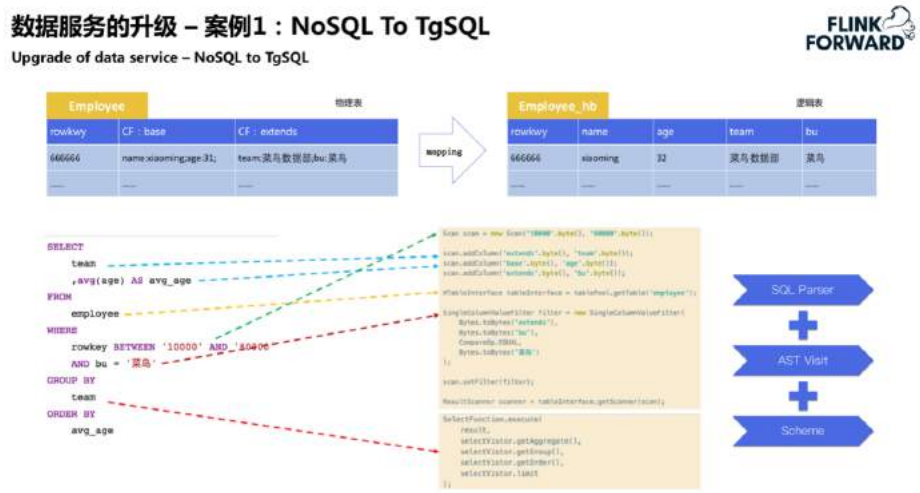
数据服务升级

菜鸟在做数仓的过程中也会提供开发一系列的数据产品来提供数据服务，原来是采用 Java Web 提供多种连接 DB 的方式。但是实际应用过程中，经常用到的数据库无非是 Hbase、MySQL 和 OpenSearch 等，因此后来菜鸟联合数据服务团队建立了一个统一的数据服务中间件“天工数据服务”。它可以提供统一的数据库接入、统一的权限管理、统一的数据保障以及统一的全链路监控等中心化的功能，将 SQL 作为一等公民，作为数据服务的 DSL，提供标准化的服务接入方式（HSF）。

作为菜鸟数据服务的践行者，天工还提供了很多贴近业务的功能。接下来通过具体的案例场景来介绍。

案例 1：NoSQL to TgSQL

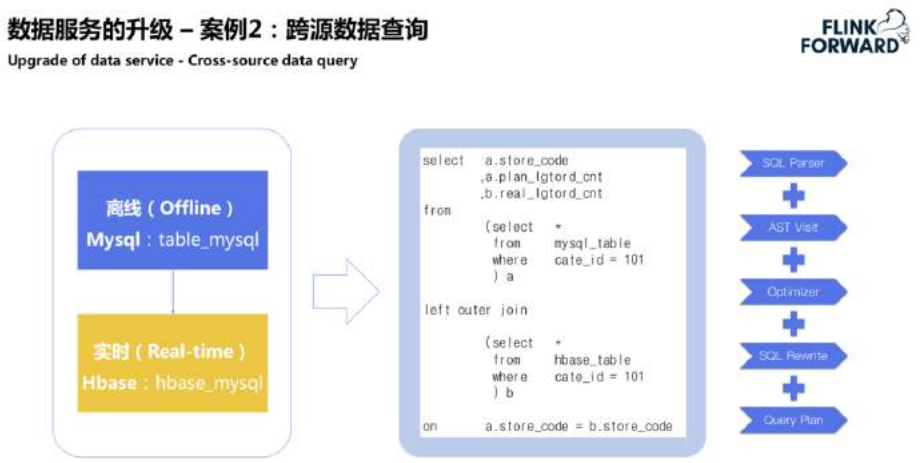
Hbase 等 NoSQL 类型的数据库，对于运营人员来讲编写代码是比较困难的，这种情况下其急需一套标准的语法。为了解决该问题，天工提供了 TgSQL，用于标准化 NoSQL 的转换。下图展示了转换的过程，Employee 转换成一个二维表，这里只是逻辑转换而非物理转换。天工中间件会解析 SQL，并在后台自动转换成查询的语言对数据进行查询。



案例 2：跨源数据查询

菜鸟在开发数据产品的过程中，会经常遇到实时和离线分不开的情况。比如菜鸟每年都会统计 KPI 的实时完成率，计算方式是已经完成的单量与计划单量之间的比值，该计算依赖的数据源有两部分：一部分是已经计划好的离线 KPI 表，另一部分是已经计算好的写入 Hbase 的实时表。原本的实现方案是通过 Java 取两次接口，然后在前端进行加减乘除的计算操作。针对该问题，菜鸟提供了标准的 SQL，用

针对跨数据源的查询，如 MySQL 离线表和 Hbase 实时表，用户只需要按照标准 SQL 的方式来写，通过升级的数据服务进行解析，再从对应的数据库中进行数据的查询操作。



案例 3：服务保障升级

菜鸟最初对于服务的保障比较缺失，一个任务发布后并不确定是否有问题，有些问题直到用户反馈的时候才能发现。另外，当并发量比较大的时候，也没有办法及时地做限流和主备切换等应对措施。

为此，天工的中间件提供了数据保障功能，除了主备切换，还包括主备双活、动态负载、热点服务阻断以及白名单限流等功能。

对于主备切换，前面提到的左右两侧分别是物理表和逻辑表的场景中，一个逻辑表可以映射成主备链路，当主链路出现问题时，可以一键切换到备链上；

此外，大促期间一些非常重要的业务，如大屏业务、内部统计分析等，会通过主备链路同时进行操作，此时完全读写其中一个库不合适，所期望的两条链路均有流量，而天工则实现了主备双活的功能支持，即将大流量切到主链，小流量切到备链；

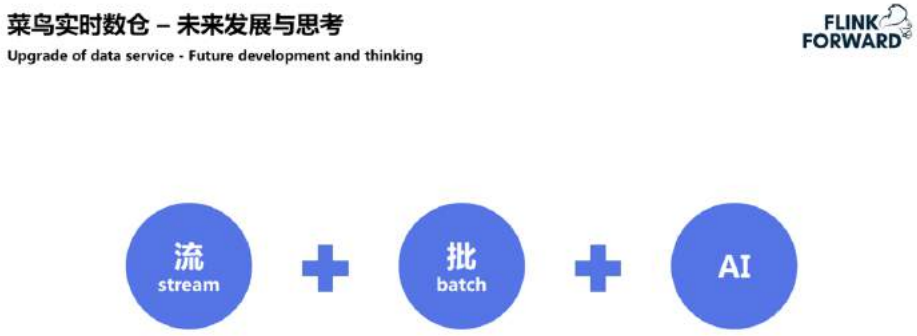
当主链上受到其中一个任务影响时，该任务会被移到备链上；对于比较复杂、执行较慢的查询，会对整个任务的性能造成影响，此时会对这种类型的热点服务进行阻断。

除了数据模型、计算引擎和数据服务，菜鸟还在其他方面进行了探索和创新，包括实时压测、过程监控、资源管理和数据质量等。实时压测在大促期间比较常用，通过实时压测来模拟大促期间的流量，测试特定的 QPS 下任务是否可以成功执行。原本的做法是重启备链上的作业，然后将备链作业的 source 改为压测的 source，sink 改为压测 source 的动作，这种方案在任务特别多的时候实现起来非常复杂。为此，阿里云团队开发了实时压测的工具，可以做到一次启动所有的需要的压测的作业，并自动生成压测的 source 和 sink，执行自动压测，生成压测报告。采用 Flink 后，还实现了作业过程监控的功能，包括延迟监控和告警监控，比如超过特定的时间无响应会进行告警，TPS、资源预警等。

菜鸟目前在实时数仓方面更多的是基于 Flink 进行一系列功能的开发，未来的发展方向计划向批流混合以及 AI 方向演进。

- Flink 提供了 batch 的功能后，菜鸟很多中小型的表分析不再导入到 Hbase 中，而是在定义 source 的时候直接将 MaxCompute 的离线维表读到内存中，直接去做关联，如此一来很多操作不需要再进行数据同步的工作。
- 针对一些物流的场景，如果链路比较长，尤其是双十一支付的订单，在十一月十七号可能还存在未签收的情况，这时候如果发现作业中有一个错误，如果重启的话，作业的 state 将会丢失，再加之整个上游的 source 在 TT 中只允许保存三天，使得该问题的解决变得更加困难。菜鸟之后发现 Flink 提供的 batch 功能可以很好地解决该问题，具体来讲是定义 TT 的 source，作为三天的实时场景的应用，TT 数据写到离线数据库进行历史数据备份，如果存在重启的情况，会读取并整合离线的数据，即使 Flink 的 state 丢失，因为离线数据的加入，也会生成新的 state，从而不必担心双十一的订单如果在十七号签收之前重启导致无法获取十一号的订单信息。当然，在上述问题的解决上，菜鸟也踩了很多的小坑。其中的一个是整合实时数据和离线数据的时候，数据乱序的问题。菜鸟实现了一系列的 UDF 来应对该问题，比如实时数据和离线数据的读取优先级设置。

- 针对日志型的业务场景，比如曝光、网站流量等，其一条日志下来后，基本不会再发生变化。菜鸟目前在考虑将所有解析的工作交给 Flink 来处理，然后再写入到 batch 中，从而无需在 MaxCompute 的 ODPS 中进行批处理的操作。
- 在智能化方面，前面提到的数据倾斜隐患的规避、资源的优化等，都用到了 Flink 提供的智能化功能。菜鸟也期望在实时 ETL 过程中的一些场景中，比如去重，也使用 Flink 相应的智能化解决方案来进行优化。此外，在数据服务保障上，如主备切换等，目前仍然依赖人工对数据库进行监控，菜鸟也期望 Flink 之后能提供全链路实时保障的策略。最后是业务场景的智能化，阿里 Alink 对于业务智能化的支持也是之后探索的方向。



感兴趣的朋友可以通过下面的联系方式扫码添加好友，共同探讨更优的实现方案。

Alibaba Group
阿里巴巴集团

扫一扫二维码图案，关注我吧

开发者社区

阿里云实时计算

实时计算交流钉钉群

Flink 社区微信公众号