

# 不仅仅是流计算 Apache Flink® 实践



Apache Flink

InfoQ

2018 Q4



MORE

THAN

STREAMING

# 卷首语 :Apache Flink 对我们意味着什么 ?

作者 徐川

最近 Qubole 的一份调查报告显示，Apache Flink 是 2018 年大数据和 Hadoop 生态系统中发展最快的引擎，与 2017 年的类似调查相比，采用量增长了 125%。这表明 Flink 正在获得越来越多人的认可。

之所以增长这么快，和 Flink 在设计上的先进性是分不开的。流计算一直是大数据计算引擎的一个痛点，在 Apache Storm 出现以前，大家都是采用批处理的方式来计算，其中最典型的代表就是 Apache Spark，它推出了 Spark Streaming 试图用快速的批处理及“微批处理”来模拟流计算，这种尝试现在被证明限制太多，Spark 本身也在尝试连续执行模式（Continuous Processing），然而进展较为缓慢。

Storm 本身也存在问题，一个是性能，无法支持高吞吐低延迟的场景，其次是功能，对于 Exactly Once 模式和窗口支持较弱，使用的场景有限。与此相比，Flink 已经克服了流处理方面大部分的问题，包括更好的状态管理、利用分布式一致性快照实现的检查点容错机制，让 Flink 在流处理方面的能力趋于完善。

然而，就如本书标题所示，Flink 并不想将自己仅仅局限于流处理引擎，而是用流处理来模拟批处理，以及支持交互式查询、机器学习等大部分数据处理场景。这已经进入了通用计算引擎的领域，与 Spark 展开了正面竞争，而如果你阅读了本书的案例部分，你会发现 Flink 除了生态和社区方面与老牌计算引擎相比尚有不如，其它部分几乎都能很好的支持。连在易用性方面，阿里也给 Flink 贡献了 Flink SQL。各大公司对 Flink 的支持，为 Flink 的发展打下了坚实的根基。

从 MapReduce，到 Spark、Storm，再到 Flink，大数据计算技术已经经历了三代发展，我们正处于第三代的前半段，相信计算技术的不断创新，会推动上层应用的革新和进化，亲自参与技术的变革，这就是今天我们大数据技术人的历史机遇。

## 卷首语：愿更多的开发者融入 Apache Flink 社区



Apache Flink 是德国柏林工业大学的几个博士生和研究生从学校开始做起来的项目，之前叫做 Stratosphere。他们在 2014 年开源了这个项目，起名为 Flink。我从 2015 年开始接触 Apache Flink，完成并见证了 Apache Flink 作为一款卓越的流计算引擎在阿里集团的落地，连续多年帮助阿里平稳的度过了一个又一个双十一大促。在刚刚过去的 2018 年双十一，Flink 引擎完美的支撑了高达 17 亿每秒的流量洪峰。

作者 王绍翻（花名：大沙）

阿里巴巴 资深技术专家

为了让大家更为全面的了解 Flink，我和 InfoQ 的徐川老师一起合作制作了这本介绍 Apache Flink 的中文专刊。它融合了 Apache Flink 在国内各大顶级互联网公司的大规模实践。在这本专刊里你可以了解到：Flink 如何为整个阿里集团平稳度过双十一立下汗马功劳？如何为满足滴滴极为复杂的业务需求提供简单直观的 API 支持？如何在字节跳动逐步取代原有的 JStorm 引擎，成为公司内部流式数据处理唯一标准？

Apache Flink 已经被业界公认是最好的流计算引擎。然而 Flink 其实并不是一个仅仅局限于做流处理的引擎。Apache Flink 的定位是一套兼具流、批、机器学习等多种计算功能的大数据引擎。在最近的一段时间，Flink 在批处理以及机器学习等诸多大数据场景都有长足的突破。一方面 Flink 的批计算在经过阿里的优化后有了数量级的提升。另一方面，Flink 社区在 tableAPI, Python，以及 ML 等诸多领域都在逐步发力，持续提升用户做 Data science 和 AI 计算的体验。此外，Flink 也在逐步提升和其他开源软件融合的体验，包括 Hive，还有 Notebook ( Zeppelin, Jupyter ) 等等。由于准备时间的仓促，本次专刊并没有收录很多关于 Flink 在这些新场景的进展的介绍。我们后续还会组织发布更多关于 Apache Flink 的系列专刊。

Apache Flink 自 2014 年开源至今也才 4 年，我们期待更多的企业和开发者们和我们一起参与到 Apache Flink 的社区和生态建设中来，共同把它打造成为全球最一流的开源大数据引擎。

# 目录

## 案例篇

阿里巴巴为什么选择 Apache Flink ? .....	1
Apache Flink 在滴滴出行的应用与实践 .....	11
字节跳动 Jstorm 到 Apache Flink 的迁移实践 .....	20
Apache Flink 在美团的实践与应用 .....	32
Apache Flink 在唯品会的实践 .....	47
携程基于 Apache Flink 的实时特征平台 .....	57

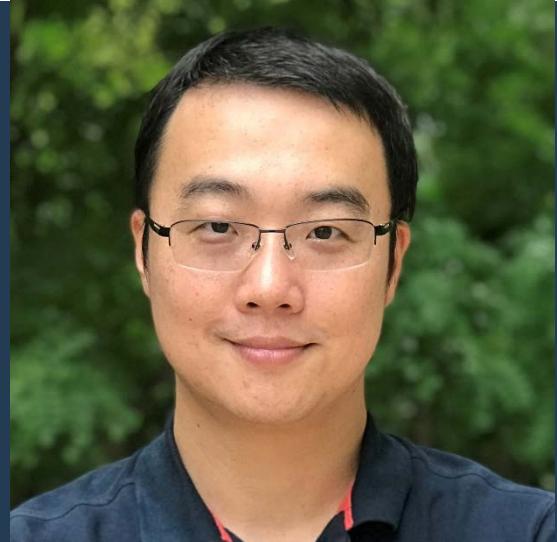
## 技术篇

一文了解 Apache Flink 核心技术 .....	66
流计算框架 Flink 与 Storm 的性能对比 .....	73
Spark VS Flink – 下一代大数据计算引擎之争，谁主沉浮？ .....	95
5 分钟从零构建第一个 Apache Flink 应用 .....	109
Apache Flink 零基础实战教程：如何计算实时热门商品 .....	114
Apache Flink SQL 概览 .....	124
Apache Flink 类型和序列化机制简介 .....	140
深度剖析阿里巴巴对 Apache Flink 的优化与改进 .....	151

# 阿里巴巴为什么选择 Apache Flink ?

作者 王峰

整理 韩非



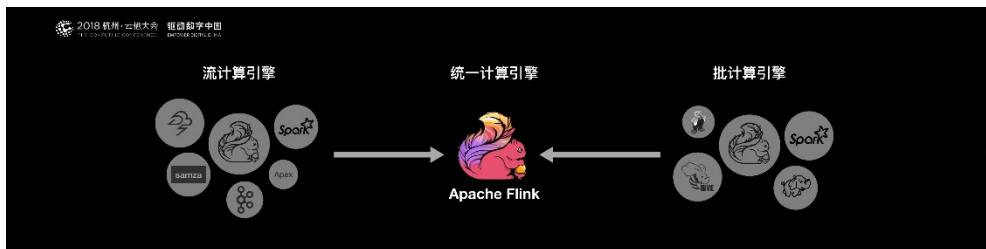
本文主要整理自云栖大会阿里巴巴计算平台事业部资深技术专家王峰（花名：莫问）在云栖大会‘开发者生态峰会’上发表的演讲。



伴随着海量增长的数据，数字化时代的未来感扑面而至。不论是结绳记事的小数据时代，还是我们正在经历的大数据时代，计算的边界正在被无限拓宽，而数据的价值，再也难以被计算。时下，谈及大数据，不得不提到最热门的下一代大数据计算引擎 Apache Flink（以下简称 Flink）。本文将结合 Flink 的前世今生，从业务角度出发，向大家娓娓道来：为什么阿里选择了 Flink？

## 合抱之木，生于毫末

随着人工智能时代的降临，数据量的爆发，在典型的大数据的业务场景下数据业务最通用的做法是：选用批处理的技术处理全量数据，采用流式计算处理实时增量数据。在绝大多数的业务场景之下，用户的业务逻辑在批处理和流处理之中往往是相同的。但是，用户用于批处理和流处理的两套计算引擎是不同的。因此，用户通常需要写两套代码。毫无疑问，这带来了一些额外的负担和成本。阿里巴巴的商品数据处理就经常需要面对增量和全量两套不同的业务流程问题，所以阿里就在想，我们能不能有一套统一的大数据引擎技术，用户只需要根据自己的业务逻辑开发一套代码。这样在各种不同的场景下，不管是全量数据还是增量数据，亦或者实时处理，一套方案即可全部支持，这就是阿里选择 Flink 的背景和初衷。

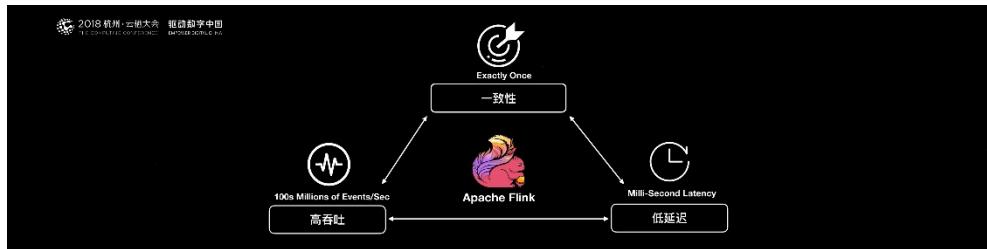


目前开源大数据计算引擎有很多选择，流计算如 Storm、Samza、Flink、Kafka Stream 等，批处理如 Spark、Hive、Pig、Flink 等。而同时支持流处理和批处理的计算引擎，只有两种选择：一个是 Apache Spark, 一个是 Apache Flink。

从技术，生态等各方面的综合考虑，首先，Spark 的技术理念是基于批来模拟流的计算。而 Flink 则完全相反，它采用的是基于流计算来模拟批计算。

从技术发展方向看，用批来模拟流有一定的技术局限性，并且这个局限性可能很难突破。而 Flink 基于流来模拟批，在技术上有更好的扩展性。从长远来看，阿里决定用 Flink 做一个统一的、通用的大数据引擎作为未来的选型。

Flink 是一个低延迟、高吞吐、统一的大数据计算引擎。在阿里巴巴的生产环境中，Flink 的计算平台可以实现毫秒级的延迟情况下，每秒钟处理上亿次的消息或者事件。同时 Flink 提供了一个 Exactly-once 的一致性语义。保证了数据的正确性。这样就使得 Flink 大数据引擎可以提供金融级的数据处理能力。



## Flink 在阿里的现状

基于 Apache Flink 在阿里巴巴搭建的平台于 2016 年正式上线，并从阿里巴巴的搜索和推荐这两大场景开始实现。目前阿里巴巴所有的业务，包括阿里巴巴所有子公司都采用了基于 Flink 搭建的实时计算平台。同时 Flink 计算平台运行在开源的 Hadoop 集群之上。采用 Hadoop 的 YARN 做为资源管理调度，以 HDFS 作为数据存储。因此，Flink 可以和开源大数据软件 Hadoop 无缝对接。



目前，这套基于 Flink 搭建的实时计算平台不仅服务于阿里巴巴集团内部，而且通过阿里云的云产品 API 向整个开发者生态提供基于 Flink 的云产品支持。

## Flink 在阿里巴巴的大规模应用，表现如何？

- 规模：一个系统是否成熟，规模是重要指标，Flink 最初上线阿里巴巴只有数百台服务器，

目前规模已达上万台，此等规模在全球范围内也是屈指可数；

- 状态数据：基于 Flink，内部积累起来的状态数据已经是 PB 级别规模；
- Events：如今每天在 Flink 的计算平台上，处理的数据已经超过万亿条；
- TPS：在峰值期间可以承担每秒超过 4.72 亿次的访问，最典型的应用场景是阿里巴巴双 11 大屏；



## Flink 的发展之路

接下来从开源技术的角度，来谈一谈 Apache Flink 是如何诞生的，它是如何成长的？以及在成长的这个关键的时间点阿里是如何进入的？并对它做出了那些贡献和支持？

Flink 诞生于欧洲的一个大数据研究项目 StratoSphere。该项目是柏林工业大学的一个研究性项目。早期，Flink 是做 Batch 计算的，但是在 2014 年，StratoSphere 里面的核心成员孵化出 Flink，同年将 Flink 捐赠 Apache，并在后来成为 Apache 的顶级大数据项目，同时 Flink 计算的主流方向被定位为 Streaming，即用流式计算来做所有大数据的计算，这就是 Flink 技术诞生的背景。



2014 年 Flink 作为主攻流计算的大数据引擎开始在开源大数据行业内崭露头角。区别于 Storm、Spark Streaming 以及其他流式计算引擎的是：它不仅是一个高吞吐、低延迟的计算引擎，同时还提供很多高级的功能。比如它提供了有状态的计算，支持状态管理，支持强一致性的数据语

义以及支持 Event Time, WaterMark 对消息乱序的处理。



## Flink 核心概念以及基本理念

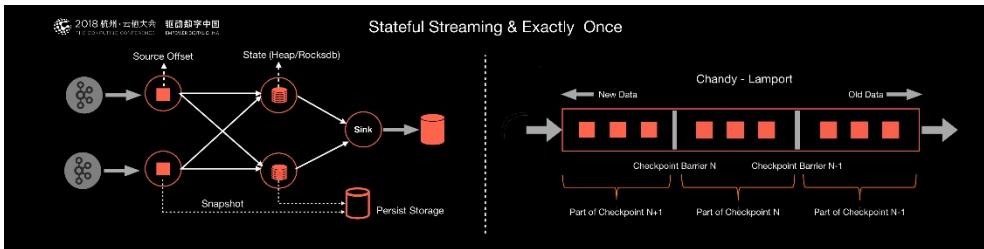
Flink 最区别于其他流计算引擎的，其实就是状态管理。

什么是状态？例如开发一套流计算的系统或者任务做数据处理，可能经常要对数据进行统计，如 Sum、Count、Min、Max,这些值是需要存储的。因为要不断更新，这些值或者变量就可以理解为一种状态。如果数据源是在读取 Kafka、RocketMQ，可能要记录读取到什么位置，并记录 Offset，这些 Offset 变量都是要计算的状态。

Flink 提供了内置的状态管理，可以把这些状态存储在 Flink 内部，而不需要把它存储在外部系统。这样做好处是第一降低了计算引擎对外部系统的依赖以及部署，使运维更加简单；第二，对性能带来了极大的提升：如果通过外部去访问，如 Redis, HBase，它一定是通过网络及 RPC。如果通过 Flink 内部去访问，它只通过自身的进程去访问这些变量。同时 Flink 会定期将这些状态做 Checkpoint 持久化，把 Checkpoint 存储到一个分布式的持久化系统中，比如 HDFS。这样的话，当 Flink 的任务出现任何故障时，它都会从最近的一次 Checkpoint 将整个流的状态进行恢复，然后继续运行它的流处理。对用户没有任何数据上的影响。

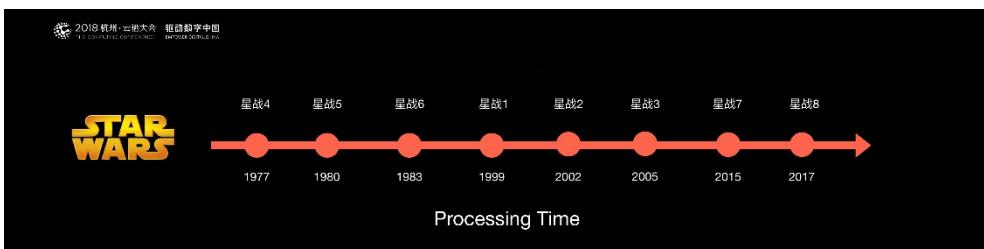
Flink 是如何做到在 Checkpoint 恢复过程中没有任何数据的丢失和数据的冗余？来保证精准计算的？

这其中原因是 Flink 利用了一套非常经典的 Chandy-Lamport 算法，它的核心思想是把这个流计算看成一个流式的拓扑，定期从这个拓扑的头部 Source 点开始插入特殊的 Barriers，从上游开始不断的向下游广播这个 Barriers。每一个节点收到所有的 Barriers,会将 State 做一次 Snapshot，当每个节点都做完 Snapshot 之后，整个拓扑就算完整的做完了一次 Checkpoint。接下来不管出现任何故障，都会从最近的 Checkpoint 进行恢复。



Flink 利用这套经典的算法，保证了强一致性的语义。这也是 Flink 与其他无状态流计算引擎的核心区别。

下面介绍 Flink 是如何解决乱序问题的。比如星球大战的播放顺序，如果按照上映的时间观看，可能会发现故事在跳跃。



在流计算中，与这个例子是非常类似的。所有消息到来的时间，和它真正发生在源头，在线系统 Log 当中的时间是不一致的。在流处理当中，希望是按消息真正发生在源头的顺序进行处理，不希望是真正到达程序里的时间来处理。Flink 提供了 Event Time 和 WaterMark 的一些先进技术来解决乱序的问题。使得用户可以有序的处理这个消息。这是 Flink 一个很重要的特点。

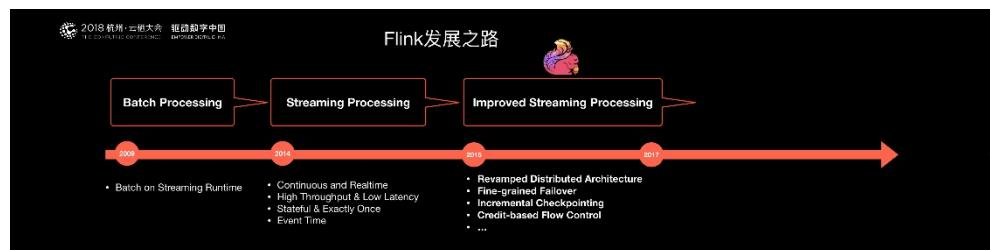


接下来要介绍的是 Flink 启动时的核心理念和核心概念，这是 Flink 发展的第一个阶段；第二个

阶段时间是 2015 年和 2017 年，这个阶段也是 Flink 发展以及阿里巴巴介入的时间。故事源于 2015 年年中，我们在搜索事业部的一次调研。当时阿里有自己的批处理技术和流计算技术，有自研的，也有开源的。但是，为了思考下一代大数据引擎的方向以及未来趋势，我们做了很多新技术的调研。

结合大量调研结果，我们最后得出的结论是：解决通用大数据计算需求，批流融合的计算引擎，才是大数据技术的发展方向，并且最终我们选择了 Flink。

但 2015 年的 Flink 还不够成熟，不管是规模还是稳定性尚未经历实践。最后我们决定在阿里内部建立一个 Flink 分支，对 Flink 做大量的修改和完善，让其适应阿里巴巴这种超大规模的业务场景。在这个过程当中，我们团队不仅对 Flink 在性能和稳定性上做出了很多改进和优化，同时在核心架构和功能上也进行了大量创新和改进，并将其贡献给社区，例如：Flink 新的分布式架构，增量 Checkpoint 机制，基于 Credit-based 的网络流控机制和 Streaming SQL 等。



## 阿里巴巴对 Flink 社区的贡献

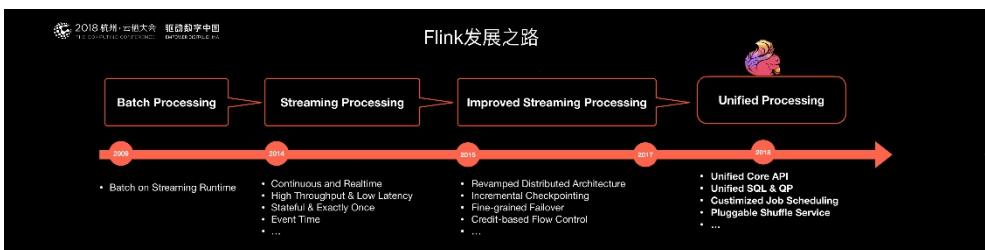
我们举两个设计案例，第一个是阿里巴巴重构了 Flink 的分布式架构，将 Flink 的 Job 调度和资源管理做了一个清晰的分层和解耦。这样做的首要好处是 Flink 可以原生的跑在各种不同的开源资源管理器上。经过这套分布式架构的改进，Flink 可以原生地跑在 Hadoop Yarn 和 Kubernetes 这两个最常见的资源管理系统之上。同时将 Flink 的任务调度从集中式调度改为了分布式调度，这样 Flink 就可以支持更大规模的集群，以及得到更好的资源隔离。



另一个是实现了增量的 Checkpoint 机制，因为 Flink 提供了有状态的计算和定期的 Checkpoint 机制，如果内部的数据越来越多，不停地做 Checkpoint, Checkpoint 会越来越大，最后可能导致做不出来。提供了增量的 Checkpoint 后，Flink 会自动地发现哪些数据是增量变化，哪些数据是被修改了。同时只将这些修改的数据进行持久化。这样 Checkpoint 不会随着时间的运行而越来越难做，整个系统的性能会非常地平稳，这也是我们贡献给社区的一个很重大的特性。



经过 2015 年到 2017 年对 Flink Streaming 的能力完善，Flink 社区也逐渐成熟起来。Flink 也成为在 Streaming 领域最主流的计算引擎。因为 Flink 最早期想做一个流批统一的大数据引擎，2018 年已经启动这项工作，为了实现这个目标，阿里巴巴提出了新的统一 API 架构，统一 SQL 解决方案，同时流计算的各种功能得到完善后，我们认为批计算也需要各种各样的完善。无论在任务调度层，还是在数据 Shuffle 层，在容错性，易用性上，都需要完善很多工作。



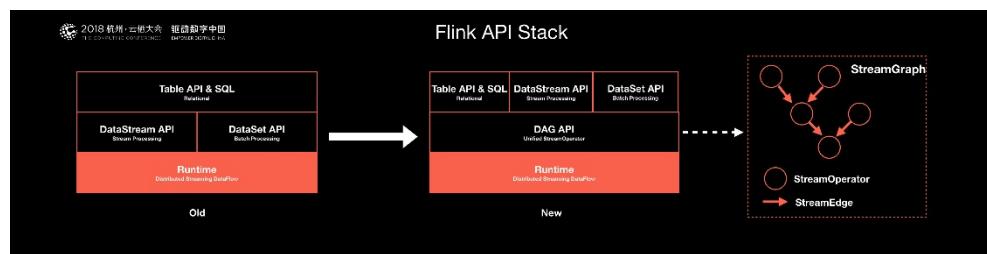
篇幅原因，下面主要和大家分享两点：

- 统一 API Stack
- 统一 SQL 方案

先来看下目前 Flink API Stack 的一个现状，调研过 Flink 或者使用过 Flink 的开发者应该知道。Flink 有 2 套基础的 API，一套是 DataStream，一套是 DataSet。DataStream API 是针对流式处理的用户提供，DataSet API 是针对批处理用户提供，但是这两套 API 的执行路径是完全不一样的，

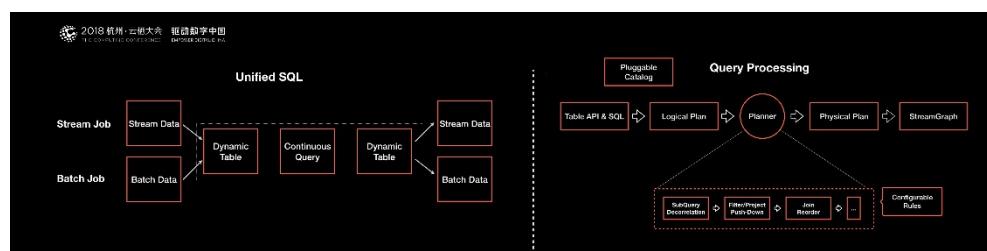
甚至需要生成不同的 Task 去执行。所以这跟得到统一的 API 是有冲突的，而且这个也是不完善的，不是最终的解法。在 Runtime 之上首先是要有一个批流统一融合的基础 API 层，我们希望可以统一 API 层。

因此，我们在新架构中将采用一个 DAG (有限无环图) API，作为一个批流统一的 API 层。对于这个有限无环图，批计算和流计算不需要泾渭分明的表达出来。只需要让开发者在不同的节点，不同的边上定义不同的属性，来规划数据是流属性还是批属性。整个拓扑是可以融合批流统一的语义表达，整个计算无需区分是流计算还是批计算，只需要表达自己的需求。有了这套 API 后，Flink 的 API Stack 将得到统一。



除了统一的基础 API 层和统一的 API Stack 外，同样在上层统一 SQL 的解决方案。流和批的 SQL，可以认为流计算有数据源，批计算也有数据源，我们可以将这两种源都模拟成数据表。可以认为流数据的数据源是一张不断更新的数据表，对于批处理的数据源可以认为是一张相对静止的表，没有更新的数据表。整个数据处理可以当做 SQL 的一个 Query，最终产生的结果也可以模拟成一个结果表。

对于流计算而言，它的结果表是一张不断更新的结果表。对于批处理而言，它的结果表是相当于一次更新完成的结果表。从整个 SQL 语义上表达，流和批是可以统一的。此外，不管是流式 SQL，还是批处理 SQL，都可以用同一个 Query 来表达复用。这样以来流批都可以用同一个 Query 优化或者解析。甚至很多流和批的算子都是可以复用的。



## Flink 的未来方向

首先，阿里巴巴还是要立足于 Flink 的本质，去做一个全能的统一大数据计算引擎。将它在生态和场景上进行落地。目前 Flink 已经是一个主流的流计算引擎，很多互联网公司已经达成了共识：Flink 是大数据的未来，是最好的流计算引擎。下一步很重要的工作是让 Flink 在批计算上有所突破。在更多的场景下落地，成为一种主流的批计算引擎。然后进一步在流和批之间进行无缝的切换，流和批的界限越来越模糊。用 Flink，在一个计算中，既可以有流计算，又可以有批计算。

第二个方向就是 Flink 的生态上有更多语言的支持，不仅仅是 Java, Scala 语言，甚至是机器学习下用的 Python, Go 语言。未来我们希望能用更多丰富的语言来开发 Flink 计算的任务，来描述计算逻辑，并和更多的生态进行对接。



最后不得不说 AI，因为现在很多大数据计算的需求和数据量都是在支持很火爆的 AI 场景，所以在 Flink 流批生态完善的基础上，将继续往上走，完善上层 Flink 的 Machine Learning 算法库，同时 Flink 往上层也会向成熟的机器学习，深度学习去集成。比如可以做 Tensorflow On Flink，让大数据的 ETL 数据处理和机器学习的 Feature 计算和特征计算，训练的计算等进行集成，让开发者能够同时享受到多种生态给大家带来的好处。

# Apache Flink 在滴滴出行 的应用与实践

作者 余海林

整理 赵明远



本文来自于余海林在 2018 年 8 月 11 日 Flink China 社区线下 Meetup · 北京站的分享。余海林目前在滴滴出行负责实时流计算相关工作，研发主要是集中在 Apache Flink 上。之前任职于阿里巴巴，主要负责 TCP/IP 协议栈以及手淘的无线网络优化。

本文主要内容主要包括以下几个方面：

- 1、Apache Flink 在滴滴的背景
- 2、Apache Flink 在滴滴的平台化
- 3、Apache Flink 在滴滴的生产实践
- 4、StreamSQL
- 5、展望规划

## Apache Flink 在滴滴

在滴滴，所有的数据基本上可以分为四个大块：

- 1、轨迹数据：轨迹数据和订单数据往往是业务方特别关心的。同时因为每一个用户在打车以后，都必须要实时的看到自己的轨迹，所以这些数据有强烈的实时需求。
- 2、交易数据：滴滴的交易数据，

- 3、埋点数据：滴滴各个业务方的埋点数据，包括终端以及后端的所有业务数据，
- 4、日志数据：整个的日志系统都有一些特别强烈的实时需求。



- 1、在滴滴应用发展的过程中，有一些对延迟性要求特别高的应用场景。比如说滴滴的轨迹数据，以及滴滴网关的日志监控，都对我们的引擎提出了非常大的挑战，要求我们在一个秒级或者说在一个很短的时间内能够给业务方一个反馈。在调研以及对比各个流计算引擎以后，由于 Apache Flink (以下简称 Flink) 是一个纯流式的处理引擎。发现 Flink 比较满足我们的业务场景。
- 2、在滴滴的内部，一个业务形态是事业部特别多，然后有很多业务需要进行实时处理，很多业务部门选择自己搭建 storm 或者 Spark Streaming 小机群。但是一个个小机群会带来一定的问题，例如：由于业务方不会有人专门去做维护流式计算引擎这些相关工作，所以每一次业务方出问题以后，实时计算团队做的最多事情就是进行重启集群，减少这样的一些成本也是对我们一个很大的挑战，
- 3、实时计算团队需要能够掌握住流计算引擎，也就是说我们必须要有一个统一的入口，来供大家更方便或者是更快捷更稳定的让业务方使用流计算服务。所以综上考虑，我们最终选择了 Flink 来作为流计算引擎的一个统一入口。

## Flink 在滴滴的平台化

### 平台化的优点

- 平台化能给带来什么样的好处呢？很明显就是业务方不再需要自己去维护自己的小机群，也不需要过多的去关心流计算引擎相关的一些问题，业务方只需要专注于业务即可，这显然能够降低业务方的成本。
- 然后各个业务方如果自己去维护一个小集群的话，就相当于是说每个业务方这里有十台机器，另外一个业务可能也有个七八台机器，然后每个集群上的机器可能就跑了很少的几个应用，业务方的机器的利用率根本上不去，这对公司内部和机器资源来说都是浪费。
- 第三个就是如果每个业务方自己维护一个小集群的话，无法也没人给业务方任何的稳定性保障，如果将流计算进行平台化以后，平台会给每个业务方承诺一个稳定性保障，并且会有一个稳定性的一个保障体系。总之流计算平台化的优点可以归结为以下三点：
  - 1、降低流计算使用门槛
  - 2、统一流计算平台，降低机器运维成本，提升机器利用率
  - 3、稳定性保障

### 平台化整体架构



通过看上面这一张图，很明显滴滴平台化可以分为以下几个部分：

- 第一个是上游的数据源，在滴滴内部，数据源用的比较多的差不多有两类，第一类是 Kafka，Kafka 作为滴滴的一个大型的日志系统，因此 Kafka 用的会比较多，然后还有 DDMQ（滴滴内部自研的一个消息队列），这两类中件间在数据流输入方面用的比较多。
- 然后对于中间这一块，是滴滴流计算平台的核心部分，应用管控、StreamSQL、WebIDE、诊断系统都是围绕着这个核心来做的。在滴滴内部现在主要维护了两个引擎，一个是 Flink，还有一个是 Spark Streaming，滴滴流计算平台上的这两个引擎，用户都是能够非常方便的使用到的。
- 再往下，用户提交上来的流计算应用都是由平台去做应用管理的，无论是 Flink 还是 Spark Streaming 应用都是以 On Yarn 模式运行的，流计算平台使用 Yarn 来管理计算资源和集群。对于需要持久化的一些依赖，在底层平台是存储在 HDFS 上的。
- 最后是流计算平台的下游，在下游当然也包括上游的一些中间件，比如 Kafka 和 DDMQ，当然在流计算的过程中，不可避免地要使用到 HBase 或者 MySQL，KV 数据库等下游存储。综上所述这就是滴滴的一个整体平台化的架构。

## 引擎改进

对于引擎我们主要做了一下这些优化：

- 平台化我们第一个做的工作就是将整个任务提交以及任务管控的各个方面都进行服务化了，既然要流计算平台化，服务化是肯定要做的。
- 第二是在流计算平台化的过程中，为了能够更好的去限制每一个应用，更好的管理应用的资源，流计算平台限制了每个 Yarn-session 上只能提交一个 Job，如果在一个 Yarn-session 上提交多个 Job，平台会进行提示或报错，保证 Job 提交不上去。
- 然后是应用在使用的过程中无法避免的会去做一些升级的操作，比如说一个 Flink Application 在今天使用的时候，很可能没有预估到明天流量会涨很多，这就导致应用在启动的过程中申请到的资源不够，用户可能要重启去修改代码，修改算子的并行度等。但是重启总是会带来一定的业务延迟，因此流计算平台提供了支持动态扩容的新特性。Flink Application 在重启的时候，以前已经在使用的资源不会被释放，而是会被重新利用，平台会根据新的资源使用情况来进行动态的缩扩。
- 最后一个是在使用官方 Flink 版本的过程中，碰到比较多的问题，例如在 Zookeeper 这一层面就碰到了不少的问题，平台内部修复了很多围绕 Zookeeper 相关的一些问题。例如 Zookeeper 抖动会导致获取不到 CheckPoint 的 ID，在官方的版本里面会存留一些 bug，平台内部已经进行修复了。

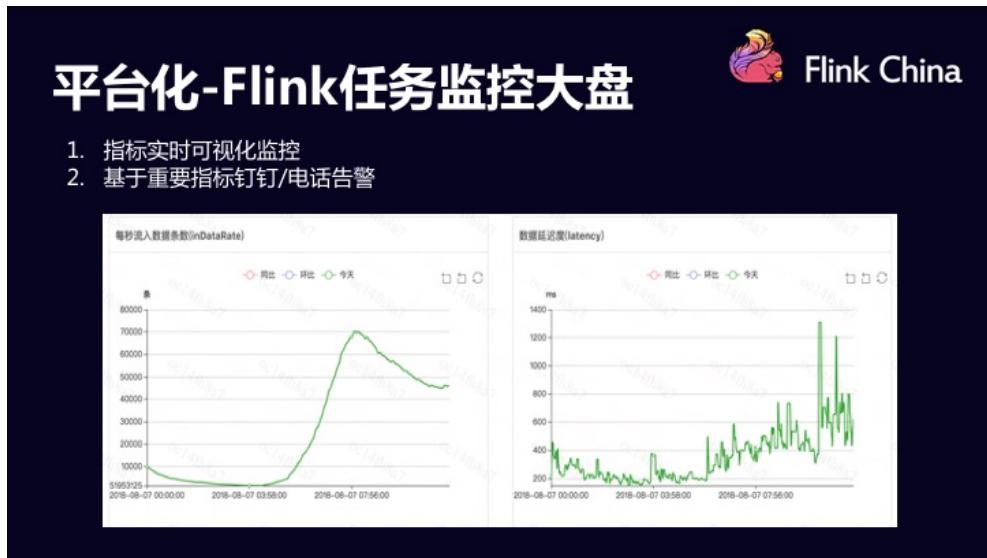
## 流计算任务开发

- 流计算平台化很大的一个目标，就是让用户开发更简单，能够更加便捷的去使用平台，因此流计算平台提供了多元化的开发方式。在早期主要有两种，第一种是用户在 WebIDE 上进行开发，第二种就是用户在本地的 IDE 中进行开发。现在流计算平台提供了第三种方式：StreamSQL IDE，流计算平台希望通过 StreamSQL 大大的降低用户开发使用流计算的门槛。



## Flink 任务监控

- 对于流计算平台，用户非常关心任务每时每刻的运行情况，并且用户需要非常实时的进行查看和确认，既然是流式任务，自然对实时要求比较高，因此用户特别关心应用的延迟有多少。所以流计算平台提供了一个完善的监控大盘，让用户来可以实时的看到他们所关心的每一个指标，当然用户还可以去自定义更个性化的指标。在下面的图中，分别给出了延迟，和吞吐量（就是应用最大能够消费多大的一个数据量，极限是多少）的实时数据。同时对用户来说，不可能实时的去盯着监控大盘，查看这个任务到底有没有出问题，因此流计算平台也提供了针对各个指标的报警服务，平台会根据适当的策略进行实时告警。



## 任务诊断体系

- 虽然流计算平台提供了监控报警的服务，但是用户看到报警数据以后，有可能没法及时有效的去判断自己的实时计算作业到底发生了什么，出现了什么问题。因此流计算平台还提供了任务诊断的服务，流计算平台会把用户任务的一些日志，包括流计算引擎里面的日志进行实时的采集，然后实时的接入到 ES 里面，这样用户就可以实时的查到指定应用的日志了。然后对于监控大盘里面看到的监控数据，流计算平台还会在 Druid 中保存一段时间。然后流计算平台修复了 Watermark 没法正常显示等 Flink UI 上面的问题。这样可以让用户能够更好地去查看监控，以及对问题进行诊断。

## Flink 在滴滴的生产实践

### 生产实践

滴滴的流计算业务在滴滴内部来讲，对于用户认可的业务场景来说，简单的归纳一下，主要是以下四种：

- 实时 ETL
- 实时数据报表

- 实时业务监控。
- 然后还有一个就是 CEP 在线业务。

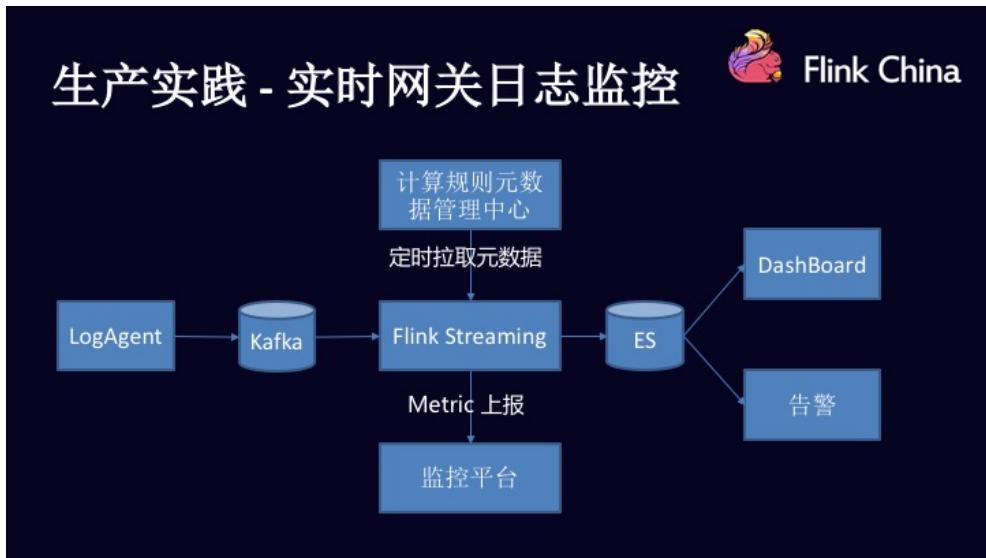
## 业务场景——实时网管监控

### 背景

- 相信很多公司都会有一个业务网关，从网关上面可以看到的各个业务线，网关上面会对每一个业务线去做一些像业务分发这样的逻辑，如果业务线非常庞大，例如滴滴就有很多业务线。
- 如果某一个业务在某一时刻出现了故障，我们怎么能够快速的发现，同时怎么快速的定位到问题。例如网关后面的每一个业务都会有相关的调用关系，一个 Service A 有可能会依赖于 Service B 或者是 Service C，然后如果一个服务出现故障以后，依赖这个服务的其他服务也有可能会出问题。
- 但是从应用最上层来看，某个业务曲线出现了下跌，或者是说曲线毛刺很高，这是不符合预期，是异常的。对于这样的一些问题，对内部系统来说，如果一个个模块去排查，是很难排查的，相当于说需要将链路上面的每一个调用关系都一个一个的进行排查，这个过程是相当复杂的。
- 因此滴滴内部做了一套实时的日志监控系统，能够实时的按业务线进行监控。每一个业务都会细化到每一个子业务，实时的去反映一个系统的服务到底是好还是坏。为了能够支持这样的一些业务场景，我们进行了适当的抽象，把所有的网络日志全部采集到 Kafka 的一个 topic 里面，Topic 里面的日志能够覆盖到滴滴 90% 的业务，然后我们会按照业务和服务去做一些 Filter, Group By 以及一定范围内的 Window 聚合等计算服务。

### 架构

在前面是介绍了我们这个系统的背景，然后现在来看看滴滴这个系统的架构设计。最前面是滴滴的数据采集服务，然后日志数据会被统一收集到 Kafka 中，在中间这一块，主要由 Flink Streaming 来进行处理，这里面是一个 Pipeline，例如在这个 Pipeline 里面会进行一系列操作：数据展开，数据展开以后，会根据具体的规则进行实时匹配，同时因为规则会动态更新，所以匹配的过程中是需要考虑的。对于规则的动态更新，在滴滴是通过配置流来实现的。配置流更新以后，会广播到下游的算子中去，下游的算子接收到规则更新以后，会对主数据流进行相应的变更。数据处理完以后，会把数据落到后端的一些系统里面去。比如 ES，数据进入 ES 以后，会有各种各样的使用方式，比如说实时的进行展示，基于这些数据进行判断是否需要进行告警。从整个链路上面来讲，整个实时网关日志架构还是非常清晰的。



## StreamSQL

滴滴内部的 StreamSQL 正在开发中，以后会作为滴滴内部流计算主要的使用方式，滴滴内部的 StreamSQL 的核心功能如下：

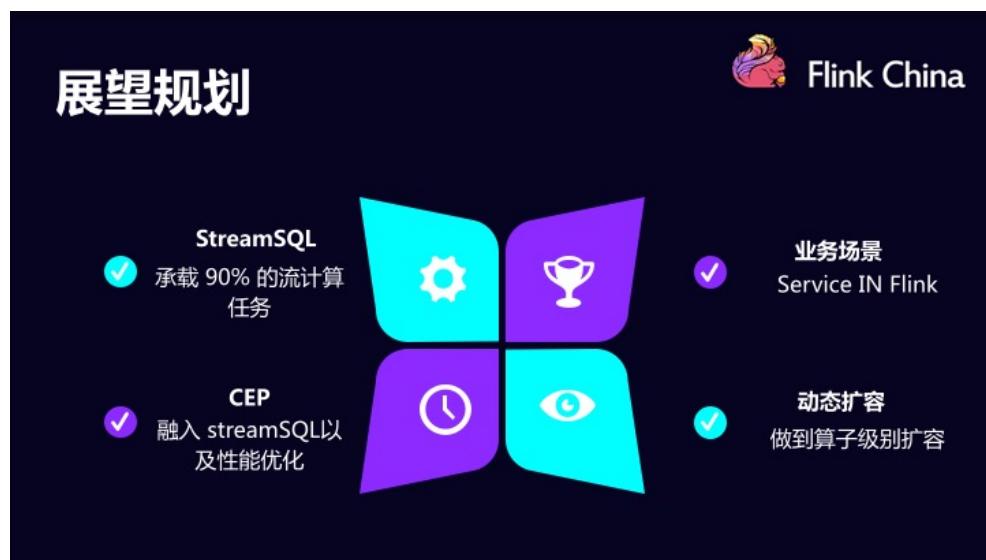
- 第一个就是支持 DDL。滴滴内部使用的数据比较多，格式也比较多，所以滴滴 StreamSQL 的 DDL 具有支持多格式以及多数据源的特点。
- 第二个就是支持 DML，对于 DML 在滴滴，只有一种即：INSERT INTO TABLE，就是插入流数据到某一张表，这个表的一定是一张 Sink 表，并且只能插入到要输出的一个下游。
- 然后是一些常用的、核心的一些功能点。比如 Group Agg、Window Agg、Join。Join 的场景主要有两种：一种是双流上面 no-window join 以及流和维度表的 Join，同时也支持 UDF、UDTF、UDAF 等用户自定义函数。

在这里简单的介绍一下滴滴定义数据源的一种方式，比如说现在要从 Kafka 中加载数据，我们的元数据具有各种各样的格式，比如说是 JSON 的，需要用户去指定所定义的数据流的 Schema，同时定义 Schema 的时候，必须要指定数据类型。然后在滴滴用的比较多的一个业务场景是分流 SQL，也就是说一条数据可能会往多个地方写，例如既要写 Hbase 又要写 Kafka 这样的一些需求。Flink 官方的 Stream SQL 是不支持这么去做的，原因可能是因为 SQL 的一些限制导致的，但是滴滴的 Stream SQL 支持分流这一新特性。同时 Stream Join 也是我们正在着力推进的一个

功能点，双流 noWindow 的 join，在滴滴也是准备支持的，也是滴滴正在不断研发的一个新特性。当然 noWindow 是滴滴给出的一个复杂概念，真正的数据当然还是有一定的状态，Window 里面的数据还是会有一定的过期时间的，只是说滴滴正在尝试天级别的一个过期时间。在用户设置以后，会在指定的一个时间，比如说每天凌晨或者说固定的一个时间点，将一些过去的数据一次性的清空掉。最后对于维度表，滴滴 Stream SQL Join 的永远是当前表，并且只支持当前表，不支持和历史表进行 Join，也不支持数据的回撤。

## 展望规划

前面讲到的 StreamSQL，滴滴内部正在不断推进。下图是 Flink 在滴滴内部的一个大致的规划和展望。

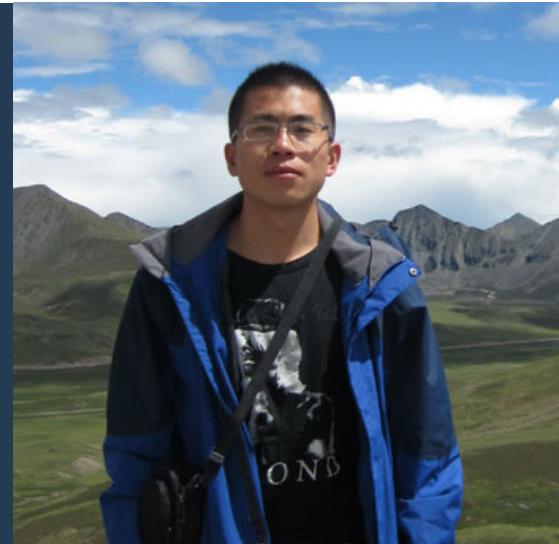


- 1、我们希望 StreamSQL 以后会承载滴滴内部至少 90% 的流计算任务，越来越多的任务都会慢慢的往 StreamSQL 上面迁移，比如说增加的新任务，以及历史遗留的一些任务。
- 2、第二个是关于 CEP，滴滴也会将其融入到 StreamSQL 的体系中，同时会不断的进行这方面性能优化。
- 3、第三点是关于业务场景的，在滴滴，监控和实时报表这样的一些业务场景会占比较多的一个部分。以后滴滴会探索开发更多业务场景，让 Flink 不断成长。
- 4、第四点是为了去应对流量突发带来的稳定性的一些问题，滴滴会在动态扩容上做更多的事情，同时滴滴也正在尝试在算子级别进行资源的自动缩扩。

# 字节跳动 Jstorm 到 Apache Flink 的迁移实践

作者 张光辉

整理 张刘毅



本文将为大家展示字节跳动公司将 Jstorm 任务迁移到 Apache Flink 上的整个过程以及后续计划。你可以借此了解到字节跳动公司引入 Apache Flink 的背景，Apache Flink 集群的构建过程，如何兼容以前的 Jstorm 作业以及基于 Apache Flink 构建一个流式任务管理平台，本文将一一为你揭开这些神秘的面纱。

本文内容如下：

- 引入 Apache Flink 的背景
- Apache Flink 集群的构建过程
- 构建流式管理平台
- 近期规划

## 一、以引入 Apache Flink 的背景

下面这幅图展示的是字节跳动公司的业务场景



首先，应用层有广告，AB 测试，推送，数据仓库等业务；其次中间层针对 python 用户抽象出来一个模板，用户只需要在模板里写自己的业务代码，结合一个 yaml 配置将 spout,bolt 组成 DAG 图；最后将其跑在 Jstorm 计算引擎上。

大概在 17 年 7 月份左右，当时 Jstorm 集群个数大概 20 左右，集群规模达到 5000 机器。



当时使用 Jstorm 集群遇到了以下几个问题：

## Jstorm 遇到的问题

- Worker 间没有内存限制
- 业务团队没有 Quota 管理，预算审核无头绪
- 重要业务使用独立集群，集群过多，运维成本高
- Python topology 性能较差，推广 Java 也比较难

- 第一个问题：单个 worker 没有内存限制，因此整个集群是没有内存隔离的。经常会出现单个作业内存使用过高，将整台机器的内存占满。
- 第二个问题：业务团队之间没有 Quota 管理，平台做预算和审核是无头绪的。当时几乎大部分业务方都跑在一个大集群上面，资源不足时，无法区分出来哪些作业优先级高，哪些作业优先级低。
- 第三个问题：集群过多，运维工具平台化做得不太好，都是靠脚本来运维的。
- 第四个问题：业务方普遍使用 python，某些情况下性能有些差。其次由于平台针对 Java Jstorm 的一些 Debug 工具，SDK 较弱，故推广 Java Jstorm 作业较难。

针对上面的问题，有两个解决方案：(1) 在 Jstorm 的基础上支持内存限制，业务 Quota 管理，集群运维；(2) Flink on yarn，也能够解决内存限制，业务 Quota 管理，Yarn 队列运维。

最终选择方案（2）也是考虑到 Apache Flink （以下简称 Flink）除了解决上述问题之外，能将运维工作交付给 yarn，节省人力；Flink 在 exactly once, time window, table/sql 等特性上支持更好；一些公司，例如阿里，在 Flink 上已经有了生产环境的实践；Flink 可以兼容 Jstorm，因此历史作业可以无缝迁移到新框架上，没有历史包袱，不需要维护两套系统。

## Flink 的优势



- Flink on yarn 解决了 worker 粒度的物理资源限制
- Yarn 支持队列，解决了 Quota 问题
- Yarn 支持物理隔离队列，解决高优作业要求物理隔离的需求
- Flink 兼容 storm，历史作业可迁移
- Flink 支持 high level api , SQL , 窗口等特性

以上就是 Flink 的优势，于是我们就决定从 Jstorm 往 Flink 迁移。

## 二、Flink 集群的构建过程

### 构建 Flink Cluster



#### 构建独立 Yarn 集群

构建独立 Yarn 集群，与离线业务隔离，主要是考虑稳定性

#### 构建 Hdfs NameSpace

提交作业依赖独立 HDFS NameSpace，主要考虑稳定性



#### 按业务划分队列

梳理现有流式作业用户，按照组织架构构建队列

#### 构建物理隔离队列

针对高优，特殊业务，建议独立物理队列

在迁移的过程中，第一件事情是要先把 Flink 集群建立起来。一开始肯定要是追求稳定性，需要把流式 yarn 集群和离线集群隔离开；提交作业，checkpoint 等依赖的 HDFS 也独立 namespace；然后跟业务方梳理旧 Jstorm 作业，根据不同的业务团队，创建不同的 Yarn 队列；同时也支持了一下最重要的作业跑在独立 label yarn 队列上，与其他业务物理隔离。

### 三、Jstorm->Flink 作业迁移

#### 兼容 Jstorm

**兼容 Jstorm**


Flink China

- 借鉴 Flink-storm，实现了 Flink-jstorm，支持将 Jstorm topology 结构转换为 Flink job
- 构建脚本，能够提交 Job，停止 Job
- 构建 Flink Job 外围工具，自动注册报警，自动注册 Dashboard，Log Service 等
- 构建迁移脚本，获取 Jstorm 作业资源使用情况，自动生成 Flink 资源配置。

当时使用的 Flink 版本是 1.3.2，Flink 官方提供了一个 flink-storm module，用来支持将一个 Storm topology 转换为 Flink 作业，借鉴 flink-storm 实现了一个 flink-jstorm，完成将 Jstorm topology 转换为 Flink 作业。

仅仅做完这件事情还是不够的，因为有一批外围工具也需要修改。例如提交作业脚本；自动注册消费延迟报警；自动注册作业状态的 Dashboard 等。

完成上面事情后，还有一件最重要的事情就是资源配置的转换。Jstorm 和 Flink 在资源配置管理方面还是有些不同，Jstorm 没有 slot 的概念，Jstorm 没有 network buffer 等，因此为了方便用户迁移作业，我们完成了一个资源配置脚本，自动根据用户的资源使用情况，以及 Topology

结构创建适合 Flink 作业的资源配置信息。

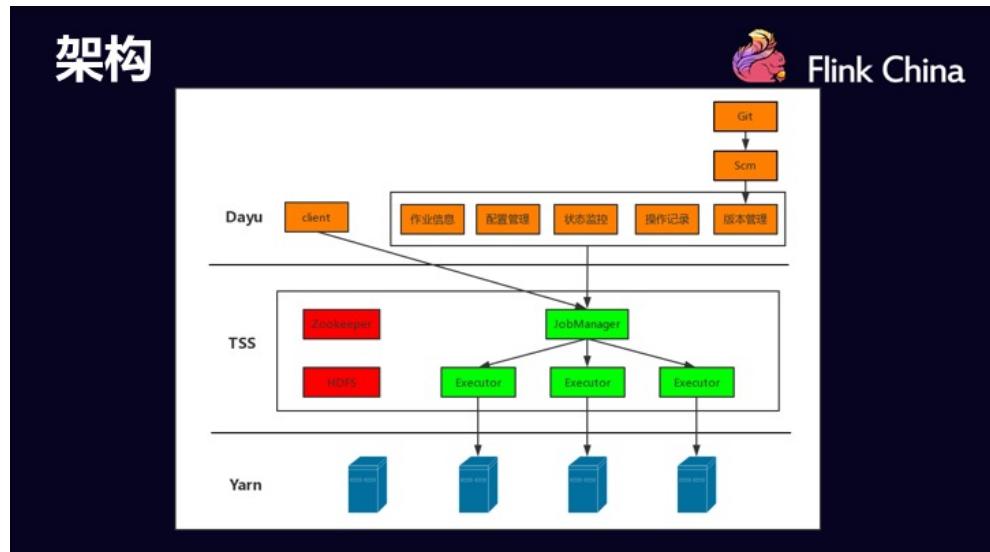
## 迁移 Jstorm

上述工作全部准备完成之后，开始推动业务迁移，截止到当前，基本已经完成迁移。

在迁移的过程中我们也有一些其他优化，比如说 Jstorm 是能够支持 task 和 work 维度故障恢复，Flink 这一块做得不是特别好，在现有 Flink 故障恢复的基础上，实现了 single task 和 single tm 维护故障恢复，这样就解决部分作业因为单 task 故障导致整个作业全部重启。

## 四、构建流式管理平台

在迁移过程中，开始着手构建了一个流式管理平台。这个平台和其他管理平台是一样的，主要提供作业配置管理，版本管理，监控，重启，回滚，Debug 功能，操作记录等功能。



不同的是，我们在架构上分两层实现的，上面一层是面向用户端的产品，称作大禹（取自大禹治水）；下面一层是用来执行具体和 Yarn, Flink 交互的工作，称作 TSS (Toutiao Streaming Service)。这样的好处是，未来有一些产品也可以构造自己面向用户端的产品，这样他直接对接

TSS 层就可以了。

下面给大家介绍一下，在字节跳动实现一个流式作业的流程。

## 创建流式作业

创建一个作业模板，使用 maven 提供的脚手架创建一个任务模板，重要内容是 pom.xml 文件。

生成的作业模板 pom.xml 已经将 Flink lib 下面的 Jar 包都 exclude 掉了，降低版本冲突的可能性。



## 测试作业

写完作业之后，可以测试作业。可以支持本地测试，也可以提交到 stage 环境测试。

## 测试作业

创建作业模板  
书写代码，完成业务逻辑  
测试作业  
在平台上注册作业  
添加配置和参数  
对代码升级版本  
启动作业  
查看作业运行状态

## 测试方式1

使用 IDEA 进行本地测试

## 测试方式2

使用 tss-client 提交到 stage 环境



Flink China

## 增加配置信息

测试完成后，需要在 dayu 平台上注册作业，添加一些配置信息。

## 增加配置信息

创建作业模板  
书写代码，完成业务逻辑  
测试作业  
在平台上注册作业  
添加配置和参数  
对代码升级版本  
启动作业  
查看作业运行状态

基本信息      配置详情      监控信息      操作历史

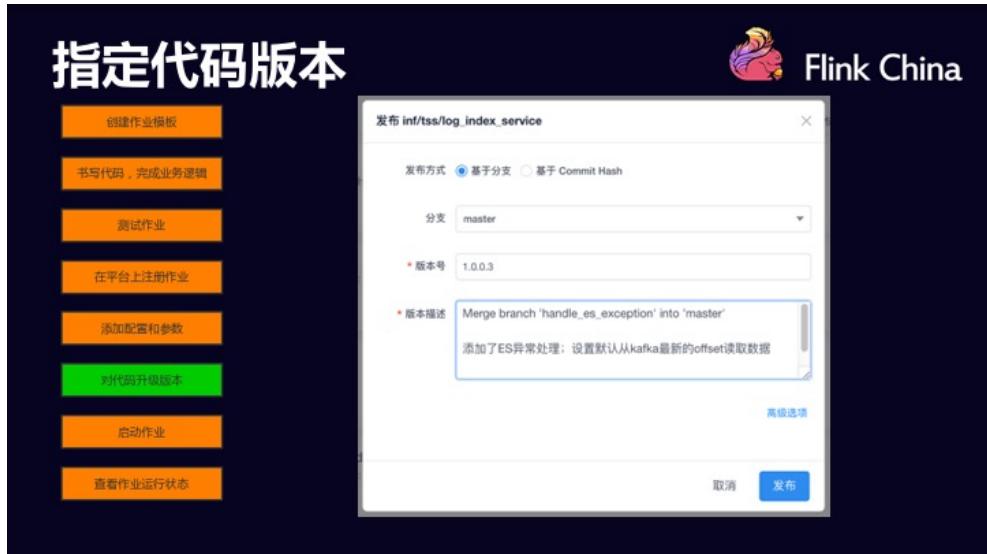
```
{  
    "tm_num":16,  
    "tm_memory_mb":10240,  
    "container_vcores":8,  
    "slots_per_tm":8,  
    "jn_memory_mb":4096,  
    "main_class":"com/bytedance/data/lis.LogIndexService",  
    "custom":{}  
}
```



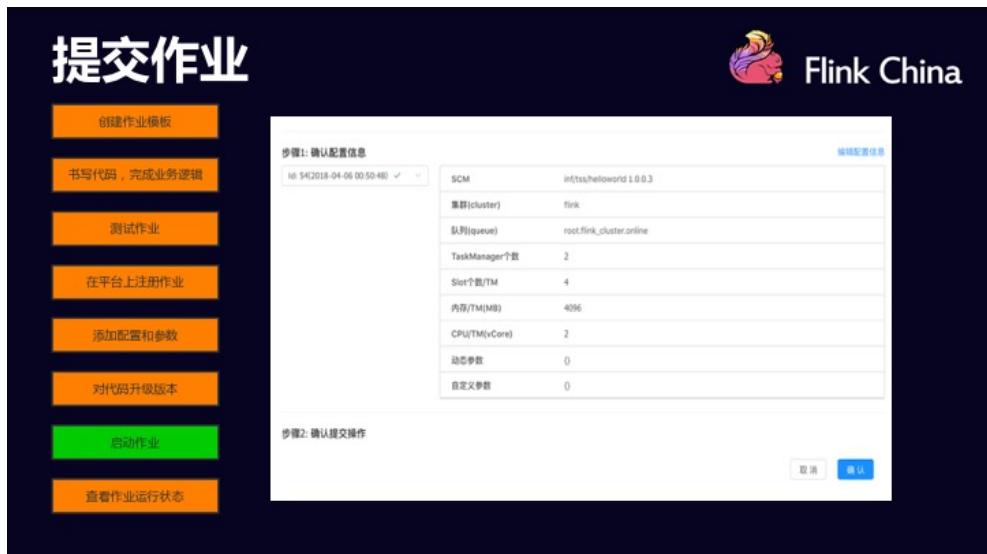
Flink China

## 指定代码版本

将自己 git 上的代码，打包，升级到最新版本，在 dayu 页面上选择版本信息，方便回滚。



## 提交作业



## 查看作业运行状态

提交完作业后，用户需要查看作业运行的状态怎么样，提供四种方式供用户查看作业状态

## 查看作业运行状态



1. Flink UI, 官方自带的 Web UI
2. Dashboard, 将作业的重要 metrics 汇总到一起，方便查看
3. 错误日志，将作业的错误日志收集写到 ES 上，用 kibana 查看
4. Jobtrace，通过提取作业异常日志，匹配异常类型，提供给用户解决方案

Flink China

第一个是 Flink UI，也就是官方自带的 UI，用户可以去看。

第二个是 Dashboard，展示作业 task qps 和 latency 以及 task 之间的网络 buffer，将这些重要信息汇总到一个页面，追查问题时清晰明了。



第三个是错误日志，将作业的错误日志都收集在一起，写入到 ES 上，方便用户查看。

## JobTrace



Flink China

将每个 Flink Job 的 ERROR 日志收集汇总，根据 Error 日志来诊断作业问题，将已知问题的解决 方案反馈给用户，下面是常见异常日志。

- java.lang.OutOfMemoryError
- Insufficient number of network buffers
- ImportError: No module named
- is running beyond physical memory limits
- diagnostics=Container released on a .\*lost.\* node
- ERROR not support cluster name
- Unable to allocate further port in port range
- java.lang.IndexOutOfBoundsException
- No such file or directory

流式计算报文

第四个是 Jobtrace 工具，就是把 Flink 框架层面产生的异常日志匹配出来，直接判断故障，告知用户处理方法。例如当作业 OOM 了，则告知用户如何扩大内存。

## 五、近期规划

最后跟大家分享一下近期规划

### 近期规划



Flink China

- SmartResource，能够做到动态调整 Job 资源，以及每次重启给 Job 一个初始资源。
- 优化作业重启速度。
- Flink 1.3.2->Flink 1.5 升级
- Flink SQL 刚上线，需要投入精力推广。

- 用户资源配置是否合理，一直是用户比较头疼的一件事，因此希望能够根据该作业的历史表现，告知用户合理的资源配置信息。
- Flink 1.3 -> 1.5 版本升级
- 优化作业重启速度，缩短用户重启作业数据流中断时间。
- Flink SQL 平台刚上线，需要投入一些精力去了解 SQL 工作机制。

以上就是我本次分享的主要内容，感谢 Flink 的举办者和参与者，感谢我的同事，因为以上的分享内容是我和同事一起完成的。

# Apache Flink 在美团的实践与应用

作者 刘迪珊

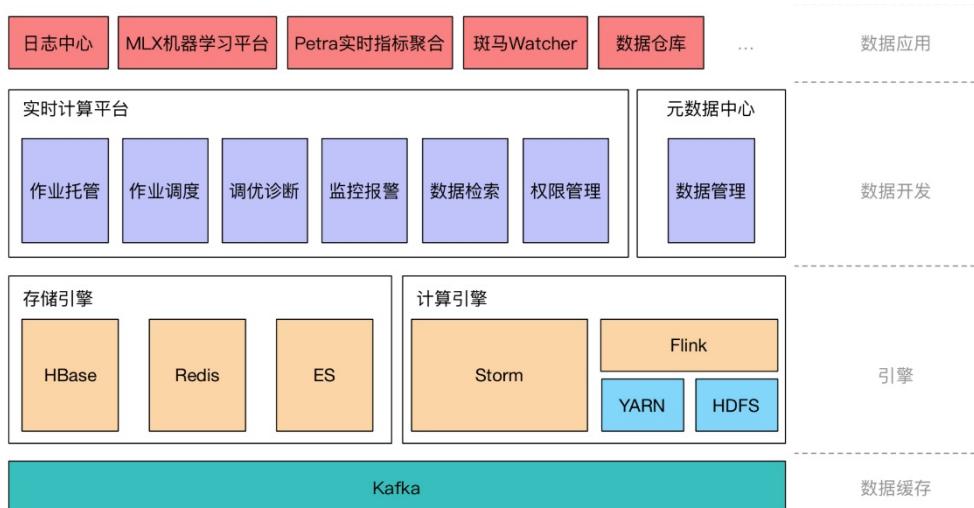
整理 徐前进



本文整理自 8 月 11 日在北京举行的 Flink Meetup，分享嘉宾刘迪珊(2015 年加入美团数据平台。致力于打造高效、易用的实时计算平台，探索不同场景下实时应用的企业级解决方案及统一化服务)。

## 美团实时计算平台现状和背景

### 实时平台架构



上图呈现的是当前美团实时计算平台的简要架构。最底层是数据缓存层，可以看到美团测的所有日志类的数据，都是通过统一的日志收集系统收集到 Kafka。Kafka 作为最大的数据中转层，支撑了美团线上的大量业务，包括离线拉取，以及部分实时处理业务等。在数据缓存层之上，是一个引擎层，这一层的左侧是我们目前提供的实时计算引擎，包括 Storm 和 Apache Flink（以下简称 Flink）。Storm 在此之前是 standalone 模式的部署方式，Flink 由于其现在运行的环境，美团选择的是 On YARN 模式，除了计算引擎之外，我们还提供一些实时存储功能，用于存储计算的中间状态、计算的结果、以及维度数据等，目前这一类存储包含 Hbase、Redis 以及 ES。在计算引擎之上，是趋于五花八门的一层，这一层主要面向数据开发的同学。实时数据开发面临诸多问题，例如在程序的调试调优方面就要比普通的程序开发困难很多。在数据平台这一层，美团面向用户提供的实时计算平台，不仅可以托管作业，还可以实现调优诊断以及监控报警，此外还有实时数据的检索以及权限管理等功能。除了提供面向数据开发同学的实时计算平台，美团现在正在做的事情还包括构建元数据中心。这也是未来我们想做 SQL 的一个前提，元数据中心是承载实时流系统的一个重要环节，我们可以把它理解为实时系统中的大脑，它可以存储数据的 Schema, Meta。架构的最顶层就是我们现在实时计算平台支撑的业务，不仅包含线上业务日志的实时查询和检索，还涵盖当下十分热门的实时机器学习。机器学习经常会涉及到搜索和推荐场景，这两个场景最显著特点：一、会产生海量实时数据；二、流量的 QPS 相当高。此时就需要实时计算平台承载部分实时特征的提取工作，实现应用的搜索推荐服务。还有一类是比较常见的场景，包括实时的特征聚合，斑马 Watcher（可以认为是一个监控类的服务），实时数仓等。

以上就是美团目前实时计算平台的简要架构。

## 实时平台现状

美团实时计算平台的现状是作业量现在已经达到了近万，集群的节点的规模是千级别的，天级消息量已经达到了万亿级，高峰期的消息量能够达到千万条每秒。

近万 千级 万亿级 千万条/s

作业量

集群节点规模

天级消息量

高峰期消息量

## 痛点和问题

美团在调研使用 Flink 之前遇到了一些痛点和问题：

- 实时计算精确性问题：在调研使用 Flink 之前美团很大规模的作业是基于 Storm 去开发的，Storm 主要的计算语义是 At-Least-Once，这种语义在保证正确性上实际上是有一些问题的，在 Trident 之前 Storm 是无状态的处理。虽然 Storm Trident 提供了一个维护状态的精确的开发，但是它是基于串行的 Batch 提交的，那么遇到问题在处理性能上可能会有一点瓶颈。并且 Trident 是基于微批的处理，在延迟上没有达到比较高的要求，所以不能满足一些对延迟比较高需求的业务。
- 流处理中的状态管理问题：基于之前的流处理过程中状态管理的问题是非常大的一类问题。状态管理除了会影响到比如说计算状态的一致性，还会影响到实时计算处理的性能以及故障恢复时候的能力。而 Flink 最突出的一个优势就是状态管理。
- 实时计算表意能力的局限性：在实时计算之前很多公司大部分的数据开发还是面向离线的场景，近几年实时的场景也慢慢火热起来了。那与离线的处理不同的是，实时的场景下，数据处理的表意能力可能有一定的限制，比如说他要进行精确计算以及时间窗口都是需要在此之上去开发很多功能性的东西。
- 开发调试成本高：近千结点的集群上已经跑了近万的作业，分布式的处理的引擎，手工写代码的方式，给数据开发的同学也带来了很高开发和调试的成本，再去维护的时候，运维成本也比较高。

## Flink 探索关注点

在上面这些痛点和问题的背景下，美团从去年开始进行 Flink 的探索，关注点主要有以下 4 个方面：

- ExactlyOnce 计算能力
- 状态管理能力
- 窗口/Join/时间处理等等
- SQL/TableAPI

## Flink 在美团的实践

下面带大家来看一下，美团从去年投入生产过程中都遇到了哪些问题，以及一些解决方案，分为下面三个部分：

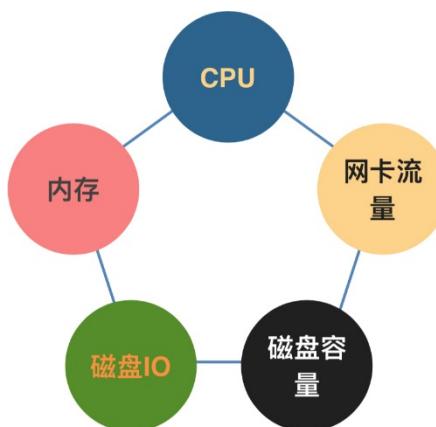
## 稳定性实践

### 稳定性实践-资源隔离

1. 资源隔离的考虑：分场景、按业务
  - 高峰期不同，运维时间不同；
  - 可靠性、延迟需求不同；
  - 应用场景，重要性不同；
2. 资源隔离的策略：
  - YARN 打标签，节点物理隔离；
  - 离线 DataNode 与实时计算节点的隔离；



### 稳定性实践-智能调度



智能调度目的也是为了解决资源不均的问题，现在普通的调度策略就是基于 CPU，基于内存去调度的。除此之外，在生产过程中也发现了一些其他的问题，比如说 Flink 是会依赖本地磁盘，进行依赖本地磁盘做本地的状态的存储，所以磁盘 IO，还有磁盘的容量，也是一类考虑的问题点，除此之外还包括网卡流量，因为每个业务的流量的状态是不一样的，分配进来会导致流量的高峰，把某一个网卡打满，从而影响其他业务，所以期望的话是说做一些智能调度化的事情。目前暂时能做到的是从 cpu 和内存两方面，未来会从其他方面做一些更优的调度策略。

## 稳定性实践-故障容错

### 1. 节点/网络故障

- JobManagerHA
- 自动拉起

与 Storm 不同的是，知道 Storm 在遇到异常的时候是非常简单粗暴的，比如说有发生了异常，可能用户没有在代码中进行比较规范的异常处理，但是没有关系，因为 worker 会重启作业还会继续执行，并且他保证的是 At-Least-Once 这样的语义，比如说一个网络超时的异常对他而言影响可能并没有那么大，但是 Flink 不同的是他对异常的容忍度是非常的苛刻的，那时候就考虑的是比如说会发生节点或者是网络的故障，那 JobManager 单点问题可能就是一个瓶颈，JobManager 那个如果挂掉的话，那么可能对整个作业的影响就是不可回复的，所以考虑了做 HA，另外一个是会去考虑一些由于运维的因素而导致的那作业，还有除此之外，可能有一些用户作业是没有开启 CheckPoint，但如果是因为节点或者是网络故障导致挂掉，希望会在平台内层做一些自动拉起的策略，去保证作业运行的稳定性。

### 2. 上下游容错

- FlinkKafka08 异常重试

我们的数据源主要是 Kafka，读写 Kafka 是一类非常常见的实时流处理避不开的一个内容，而 Kafka 本身的集群规模是非常比较大的，因此节点的故障出现是一个常态问题，在此基础上我们对节点故障进行了一些容错，比如说节点挂掉或者是数据均衡的时候，Leader 会切换，那本身 Flink 的读写对 Leader 的切换容忍度没有那么高，在此基础上我们对一些特定场景的，以及一些特有的异常做的一些优化，进行了一些重试。

### 3. 容灾

- 多机房
- 流热备

容灾可能大家对考虑的并不多，比如说有没有可能一个机房的所有的节点都挂掉了，或者是无法访问了，虽然它是一个小概率的事件，但它也是会发生的。所以现在也会考虑做多机房的一些部署，包括还有 Kafka 的一些热备。

## Flink 平台化

### Flink 平台化-作业管理

在实践过程中，为了解决作业管理的一些问题，减少用户开发的一些成本，我们做了一些平台化的工作，下图是一个作业提交的界面展示，包括作业的配置，作业生命周期的管理，报警的一些配置，延迟的展示，都是集成在实时计算平台的。

The screenshot displays two overlapping configuration panels for a job named [app\_data\_flinktest\_biz].

**作业基本配置 [app\_data\_flinktest\_biz]** (Top Panel):

- TaskSet: FlinkDeoug - iudishan
- 用户组: data
- Git仓库: https://github.com/...
- 报警方式: 大象 邮件 短信
- 负责人邮箱: 123456789@qq.com
- 备注信息: 拓扑备注信息,描述拓扑功能
- 修改基本配置

**作业生产配置 [app\_data\_flinktest\_biz]** (Bottom Panel):

- FlinkFrame配置:
  - 主函数: (empty)
  - 并发度: 1
  - TaskManager数量: 1
  - Flink版本: 1.4.0
  - 从上一次结束 Savepoint自动:
  - Flink arguments: 选项
- JobManager内存/MB: 1024
- TaskManager内存/MB: 1024
- JDK版本: jdk1.8.0\_112
- 同步调试配置
- 保存生产配置

### Flink 平台化-监控报警

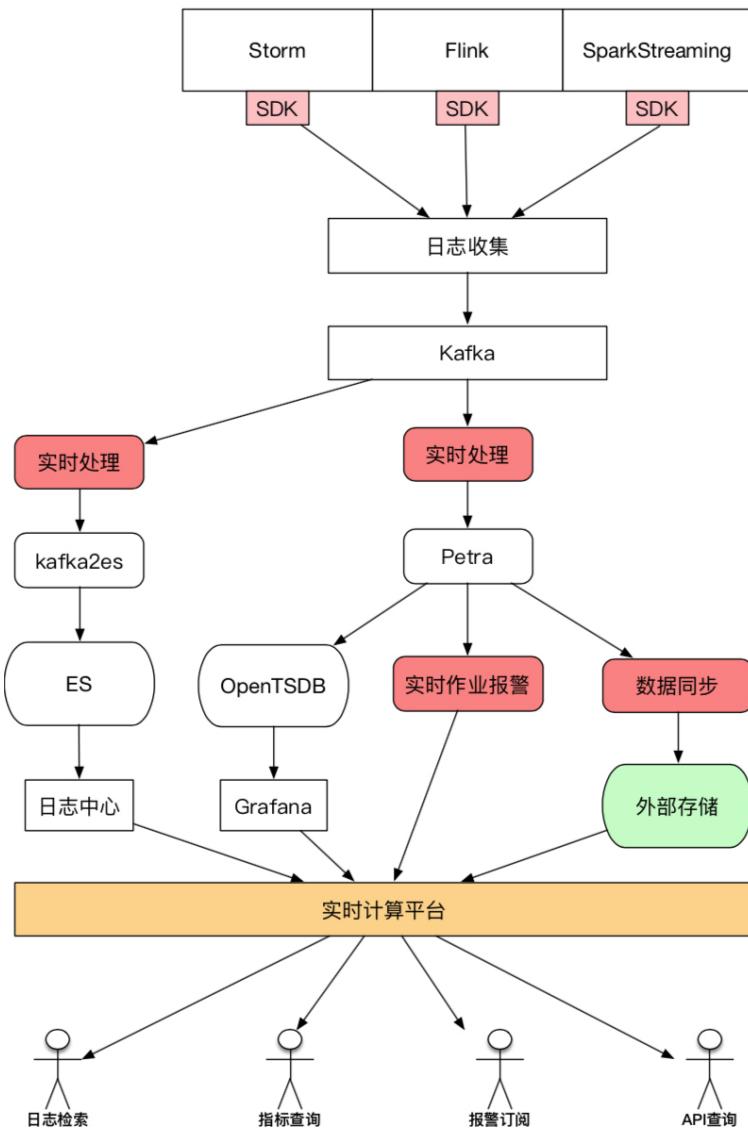
在监控上我们也做了一些事情，对于实时作业来讲，对监控的要求会更高，比如说在作业延迟的时候对业务的影响也比较大，所以做了一些延迟的报警，包括作业状态的报警，比如说作业存活的状态，以及作业运行的状态，还有未来会做一些自定义 Metrics 的报警。自定义 Metrics

是未来会考虑基于作业处理本身的内容性，做一些可配置化的一些报警。

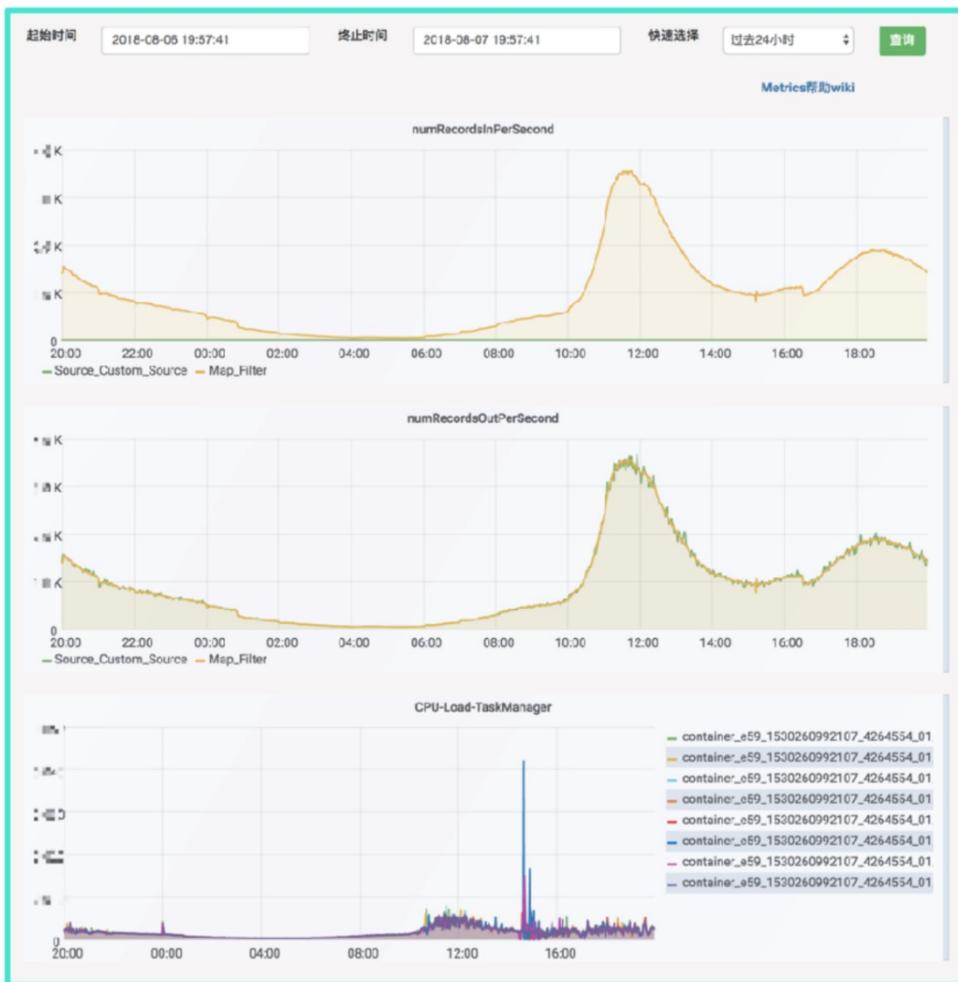
## Flink 平台化-调优诊断

- 实时计算引擎提供统一日志和 Metrics 方案
- 为业务提供按条件过滤的日志检索
- 为业务提供自定义时间跨度的指标查询
- 基于日志和指标，为业务提供可配置的报警

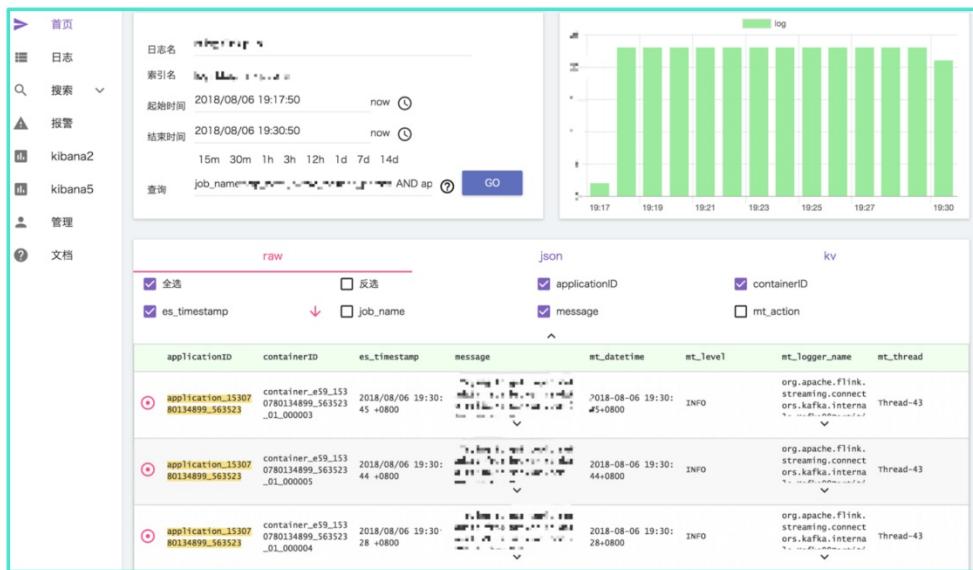
另外就是刚刚提到说在开发实时作业的时候，调优和诊断是一个比较难的痛点，就是用户不是很难去查看分布式的日志，所以也提供了一套统一的解决方案。这套解决方案主要是针对日志和 Metrics，会在针对引擎那一层做一些日志和 Metrics 的上报，那么它会通过统一的日志收集系统，将这些原始的日志，还有 Metrics 汇集到 Kafka 那一层。今后 Kafka 这一层大家可以发现它有两个下游，一方面是做日志到 ES 的数据同步，目的的话是说能够进入日志中心去做一些日志的检索，另外一方面是通过一些聚合处理流转到写入到 OpenTSDB 把数据做依赖，这份聚合后的数据会做一些查询，一方面是 Metrics 的查询展示，另外一方面就是包括实做的一些相关的报警。



下图是当前某一个作业的一个可支持跨天维度的 Metrics 的一个查询的页面。可以看到说如果是能够通过纵向的对比，可以发现除了作业在某一个时间点是因为什么情况导致的？比如说延迟啊这样容易帮用户判断一些他的做作业的一些问题。除了作业的运行状态之外，也会先就是采集一些节点的基本信息作为横向的对比



下图是当前的日志的一些查询，它记录了，因为作业在挂掉之后，每一个 ApplicationID 可能会变化，那么基于作业唯一的唯一的主键作业名去搜集了所有的作业，从创建之初到当前运行的日志，那么可以允许用户的跨 Application 的日志查询。



## 生态建设

为了适配这两类 MQ 做了不同的事情，对于线上的 MQ，期望去做一次同步多次消费，目的是避免对线上的业务造成影响，对于的生产类的 Kafka 就是线下的 Kafka，做了一些地址的屏蔽，还有基础基础的一些配置，包括一些权限的管理，还有指标的采集。

## Flink 在美团的应用

下面会给大家讲两个 Flink 在美团的真实使用的案例。第一个是 Petra，Petra 其实是一个实时指标的一个聚合的系统，它其实是面向公司的一个统一化的解决方案。它主要面向的业务场景就是基于业务的时间去统计，还有计算一些实时的指标，要求的话是低时延，还有一个就是说，因为它是面向的是通用的业务，由于业务可能是各自会有各自不同的维度，每一个业务可能包含了包括应用通道机房，还有其他的各自应用各个业务特有的一些维度，而且这些维度可能涉及到比较多，另外一个就是说它可能是就是业务需要去做一些复合的指标的计算，比如说最常见的交易成功率，他可能需要去计算支付的成功数，还有和下单数的比例。另外一个就是说统一化的指标聚合可能面向的还是一个系统，比如说是一些 B 端或者是 R 端的一些监控类的系统，那么系统对于指标系统的诉求，就是说我希望指标聚合能够最真最实时最精确的能够产生

一些结果，数据保证说它的下游系统能够真实的监控到当前的信息。右边图是我当一个 Metrics 展示的一个事例。可以看到其他其实跟刚刚讲也是比较类似的，就是说包含了业务的不同维度的一些指标汇聚的结果。

## Petra 实时指标聚合

### 1. 业务场景：

- 基于业务时间（事件时间）
- 多业务维度：如应用、通道、机房等
- 复合指标计算：如交易成功率=支付成功数/下单数
- 低延迟：秒级结果输出



### 2. Exactlyonce 的精确性保障

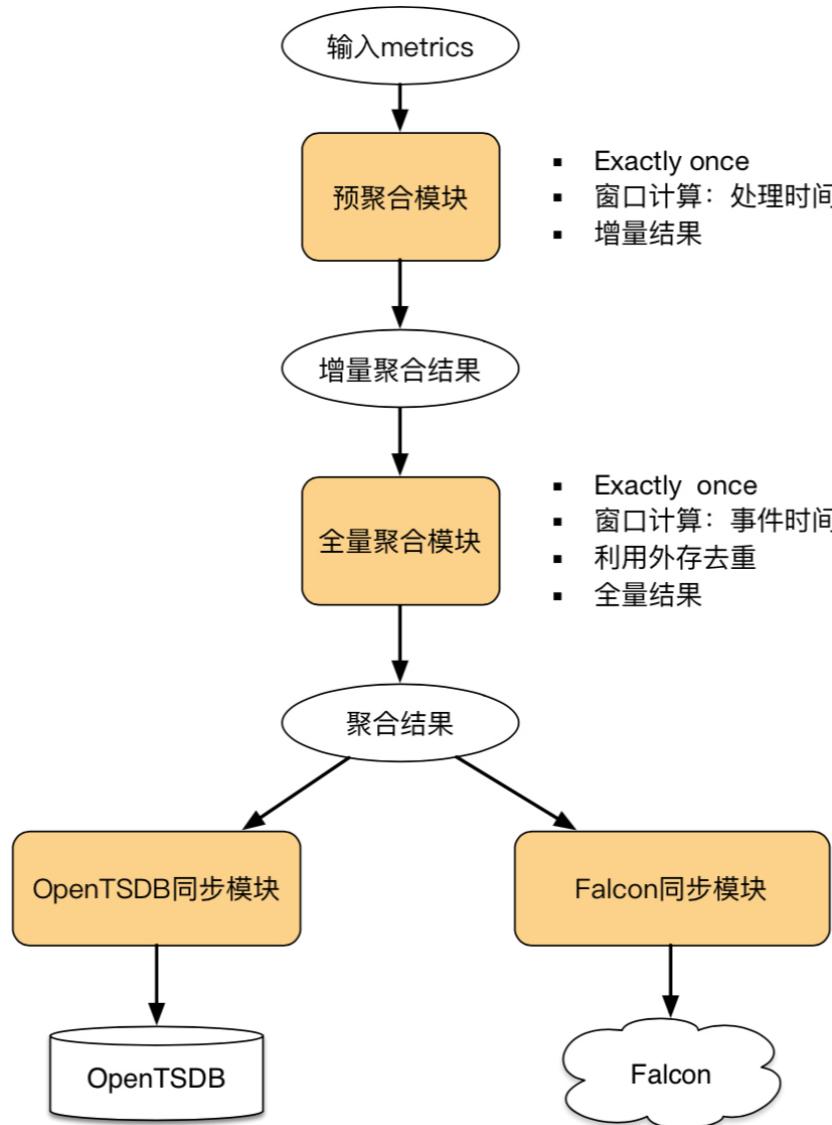
- Flinkcheckpoint 机制

### 3. 维度计算中数据倾斜

- 热点 key 散列

### 4. 对晚到数据的容忍能力

- 窗口的设置与资源的权衡



在用 Flink 做实时指标复核的系统的时候，着重从这几方面去考虑了。第一个方面是说精确的计算，包括使用了 FLink 和 CheckPoint 的机制去保证说我能做到不丢不重的计算，第一个首先是由统一化的 Metrics 流入到一个预聚合的模块，预聚合的模块主要去做一些初始化的一些聚合，其中的为什么分预聚合和全量聚合主要的解决一类问题，包括就刚刚那位同学问的一个

问题，就是数据倾斜的问题，比如说在热点 K 发生的时候，当前的解决方案也是通过预聚合的方式去做一些缓冲，让尽量把 K 去打散，再聚合全量聚合模块去做汇聚。那其实也是只能解决一部分问题，所以后面也考虑说在性能的优化上包括去探索状态存储的性能。下面的话还是包含晚到数据的容忍能力，因为指标汇聚可能刚刚也提到说要包含一些复合的指标，那么符合的指标所依赖的数据可能来自于不同的流，即便来自于同一个流，可能每一个数据上报的时候，可能也会有晚到的情况发生，那时候需要去对数据关联做晚到的容忍，容忍的一方面是说可以设置晚到的 Lateness 的延迟，另一方面是可以设置窗口的长度，但是其实在现实的应用场景上，其实还有一方面考虑就是说除了去尽量的去拉长时间，还要考虑真正的计算成本，所以在这方面也做了一些权衡，那么指标基本就是经过全量聚合之后，聚合结果会回写 Kafka，经过数据同步的模块写到 OpenTSDB 做，最后去 grafana 那做指标的展示，另一方面可能去应用到通过 Facebook 包同步的模块去同步到报警的系统里面去做一些指标，基于指标的报警。

下图是现在提供的产品化的 Petra 的一个展示的机示意图，可以看到目前的话就是定义了某一些常用的算子，以及维度的配置，允许用户进行配置话的处理，直接去能够获取到他期望要的指标的一个展示和汇聚的结果。目前还在探索说为 Petra 基于 Sql 做一些事情，因为很多用户也比较就是在就是习惯上也可以倾向于说我要去写 Sql 去完成这样的统计，所以也会基于此说依赖 Flink 的本身的对 SQL 还有 TableAPI 的支持，也会在 Sql 的场景上进行一些探索。

The screenshot shows the Petra application configuration interface. At the top, there is a table with basic application information:

应用名称	unionid
责任人	.....
创建时间	2016-03-07 14:34:08
状态	上线
修改时间	2018-03-21 12:48:15
应用描述	.....
文档链接	.....

Below this is the "应用配置" (Application Configuration) section. It includes tabs for "输入配置" (Input Configuration), "简单聚合配置" (Simple Aggregation Configuration), "复杂聚合配置" (Complex Aggregation Configuration), "Falcon同步配置" (Falcon Sync Configuration), "结果信息查看" (Result Information View), and "线上数据追踪" (Online Data Tracking). The "简单聚合配置" tab is selected.

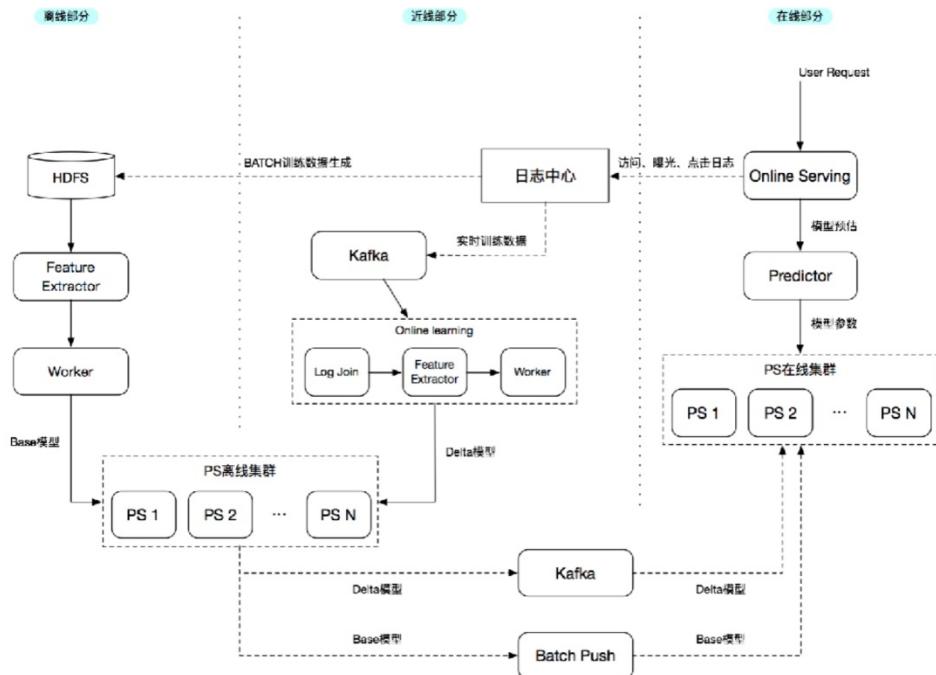
The "聚合规则配置" (Aggregation Rule Configuration) section contains a table for managing aggregation rules:

<input type="checkbox"/>	metric名称	聚合类型	聚合维度	聚合粒度	数据延迟时间	同步falcon	操作
<input type="checkbox"/>	count	counter	host, method, sub_name	60	Empty	False	

Below the table, it says "Showing 1 to 1 of 1 rows". There are also sections for "数据过滤配置" (Data Filter Configuration) and "流量细分配置" (Traffic Fine-grained Configuration).

## MLX 机器学习平台

第二类应用就是机器学习的一个场景，机器学习的场景可能会依赖离线的特征数据以及实时的特征数据。一个是基于现有的离线场景下的特征提取，经过了批处理，流转到了离线的集群。另外一个就是近线模式，近线模式出的数据就是现有的从日志收集系统流转过来的统一的日志，经过 Flink 的处理，就是包括流的关联以及特征的提取，再做模型的训练，流转到最终的训练的集群，训练的集群会产出 P 的特征，还有都是 Delta 的特征，最终将这些特征影响到线上的线上的特征的一个训练的一个服务上。这是一个比较常见的，比如说比较就是通用的也是比较通用的一个场景，目前的话主要应用的方可能包含了搜索还有推荐，以及一些其他的业务。



## 未来展望

未来的话可能也是通过也是期望在这三方面进行做一些更多的事情，刚刚也提到了包括状态的管理，第一个是状态的统一的，比如说 Sql 化的统一的管理，希望有统一的配置，帮用户去选

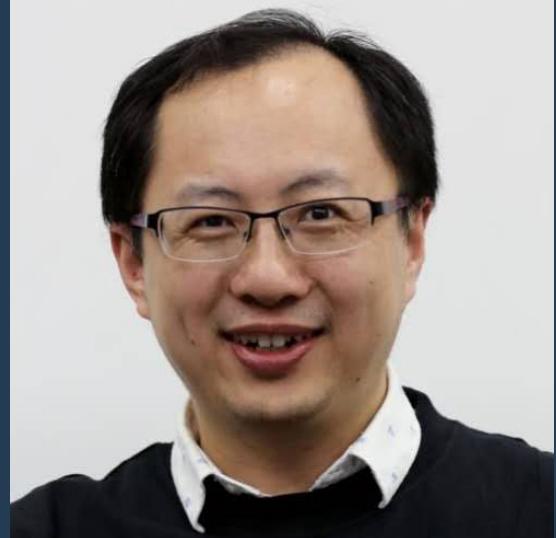
择一些期望的回滚点。另外一个就是大状态的性能优化，因为比如说像做一些流量数据的双流的关联的时候，现在也遇到了一些性能瓶颈的问题，对于说啊基于内存型的状态，基于内存型的数据的处理，以及基于 RocksDB 的状态的处理，做过性能的比较，发现其实性能的差异还是有一些大的，所以说在基于 RocksDBBackend 的上面能够去尽量去更多的做一些优化，从而提升作业处理的性能。第二方面就是 Sql，Sql 的话应该是每一个位就是当前可能各个公司都在做的一个方向，因为之前也有对 Sql 做一些探索，包括提供了基于 Storm 的一些 Sql 的表示，但是可能对于之前的话对于语义的表达可能会有一些欠缺，所以说在基于 Flink 可以去解决这些方面的事情，以及包括 Sql 的并发度的一些配置的优化，包括 Sql 的查询的一些优化，都希望说在 Flink 未来能够去优化更多的东西，去真正能使 Sql 应用到生产的环境。

另外一方面的话就是会进行新的场景的也在做新的场景的一些探索，期望是比如说包括刚刚也提到说除了流式的处理，也期望说把离线的场景下的数据进行一些合并，通过统一的 Sql 的 API 去提供给业务做更多的服务，包括流处理，还有批处理的结合。

# Apache Flink 在唯品会的实践

作者 王新春

整理 郭旭策



本文来自于王新春在 2018 年 7 月 29 日 Flink China 社区线下 Meetup · 上海站的分享。王新春目前在唯品会负责实时平台相关内容，主要包括实时计算框架和提供实时基础数据，以及机器学习平台的工作。之前在美团点评，也是负责大数据平台工作。他已经在大数据实时处理方向积累了丰富的工作经验。

本文主要内容主要包括以下几个方面：

1. 唯品会实时平台现状
2. Apache Flink（以下简称 Flink）在唯品会的实践
3. Flink On K8S
4. 后续规划

## 一、唯品会实时平台现状

目前在唯品会实时平台并不是一个统一的计算框架，而是包括 Storm, Spark, Flink 在内的三个主要计算框架。由于历史原因，当前在 Storm 平台上的 job 数量是最多的，但是从去年开始，业务重心逐渐切换到 Flink 上面，所以今年在 Flink 上面的应用数量有了大幅增加。

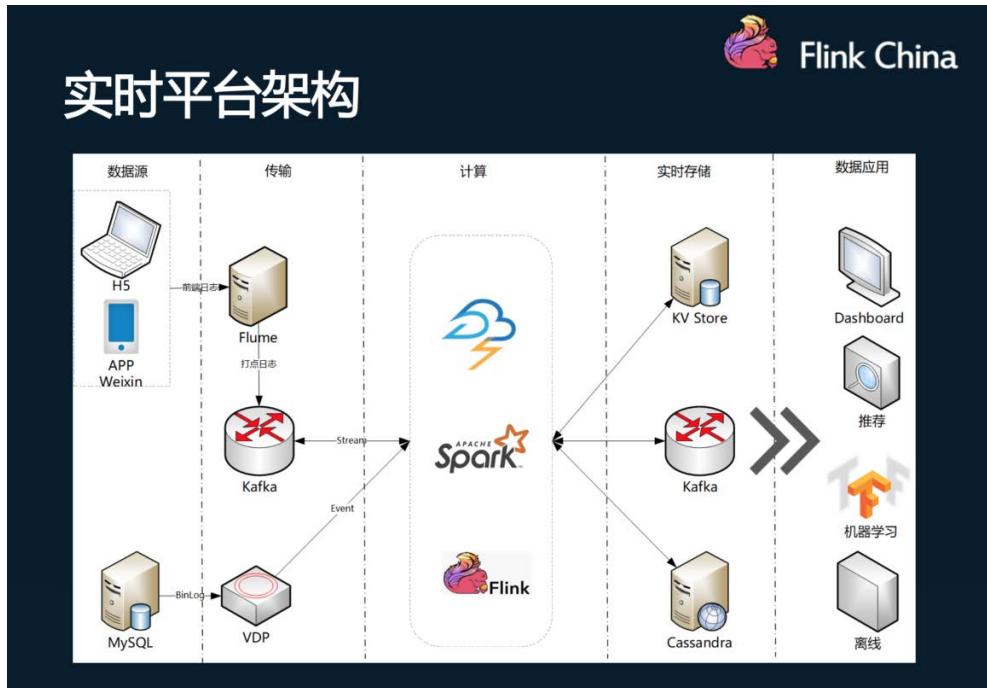
实时平台的核心业务包含八大部分：实时推荐作为电商的重点业务，包含多个实时特征；大促

看板，包含各种维度的统计指标（例如：各种维度的订单、UV、转化率、漏斗等），供领导层、运营、产品决策使用；实时数据清洗，从用户埋点收集来数据，进行实时清洗和关联，为下游的各个业务提供更好的数据；此外还有互联网金融、安全风控、与友商比价等业务，以及 Logview、Mercury、Titan 作为内部服务的监控系统、VDRC 实时数据同步系统等。



实时平台的职责主要包括实时计算平台和实时基础数据。实时计算平台在 Storm、Spark、Flink 等计算框架的基础上，为监控、稳定性提供了保障，为业务开发提供了数据的输入与输出。实时基础数据包含对上游埋点的定义和规范化，对用户行为数据、MySQL 的 Binlog 日志等数据进行清洗、打宽等处理，为下游提供质量保证的数据。

在架构设计上，包括两大数据源。一种是在 App、微信、H5 等应用上的埋点数据，原始数据收集后发送到在 kafka 中；另一种是线上实时数据的 MySQL Binlog 日志。数据在计算框架里面做清洗关联，把原始的数据通过实时 ETL 为下游的业务应用（包括离线宽表等）提供更易于使用的数据。

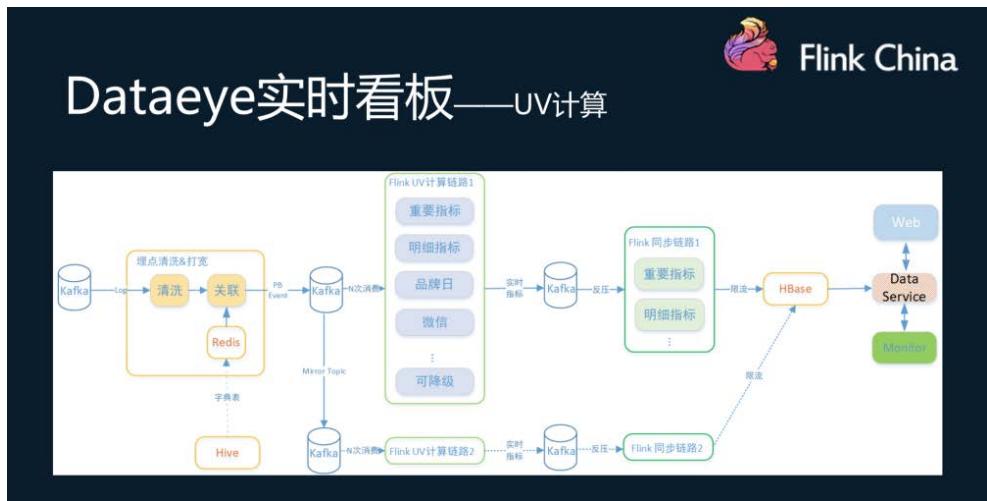


## 二、Flink 在唯品会的实践

### 场景一：Dataeye 实时看板

Dataeye 实时看板是支持需要对所有的埋点数据、订单数据等进行实时计算时，具有数据量大的特点，并且需要统计的维度有很多，例如全站、二级平台、部类、档期、人群、活动、时间维度等，提高了计算的复杂程度，统计的数据输出指标每秒钟可以达到几十万。

以 UV 计算为例，首先对 Kafka 内的埋点数据进行清洗，然后与 Redis 数据进行关联，关联好的数据写入 Kafka 中；后续 Flink 计算任务消费 Kafka 的关联数据。通常任务的计算结果的量也很大（由于计算维度和指标特别多，可以达到上千万），数据输出通过也是通过 Kafka 作为缓冲，最终使用同步任务同步到 HBase 中，作为实时数据展示。同步任务会对写入 HBase 的数据限流和同类型的指标合并，保护 HBase。与此同时还有另一路计算方案作为容灾。



在以 Storm 进行计算引擎中进行计算时，需要使用 Redis 作为中间状态的存储，而切换到 Flink 后，Flink 自身具备状态存储，节省了存储空间；由于不需要访问 Redis，也提升了性能，整体资源消耗降低到了原来的 1/3。

在将计算任务从 Storm 逐步迁移到 Flink 的过程中，对两路方案先后进行迁移，同时将计算任务和同步任务分离，缓解了数据写入 HBase 的压力。

切换到 Flink 后也需要对一些问题进行追踪和改进。对于 FlinkKafkaConsumer，由于业务原因对 kafka 中的 Aotu Commit 进行修改，以及对 offset 的设定，需要自己实现支持 kafka 集群切换的功能。对不带 window 的 state 数据需要手动清理。还有计算框架的通病——数据倾斜问题需要处理。同时对于同步任务追数问题，Storm 可以从 Redis 中取值，Flink 只能等待。

## 场景二：Kafka 数据落地 HDFS

之前都是通过 Spark Streaming 的方式去实现，现在正在逐步切换到 Flink 上面，通过 **OrcBucketingTableSink** 将埋点数据落地到 HDFS 上的 Hive 表中。在 Flink 处理中单 Task Write 可达到 3.5K/s 左右，使用 Flink 后资源消耗降低了 90%，同时将延迟 30s 降低到了 3s 以内。目前还在做 Flink 对 Spark Bucket Table 的支持。

## 场景三：实时的 ETL

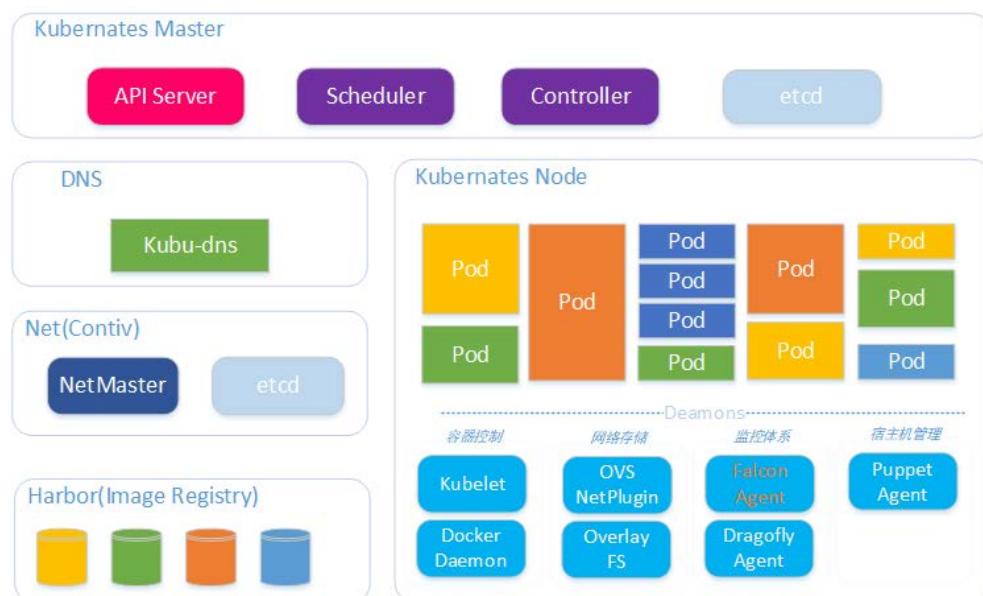
对于 ETL 处理工作而言，存在一个痛点就是字典表存储在 HDFS 中，并且是不断变化的，而实时的数据流需要与字典表进行 join。字典表的变化是由离线批处理任务引起的，目前的做法是使用 **ContinuousFileMonitoringFunction** 和 **ContinuousFileReaderOperator** 定时监听 HDFS 数据变化，不断地将新数据刷入，使用最新的数据去做 join 实时数据。

我们计划做更加通用的方式，去支持 Hive 表和 stream 的 join，实现 Hive 表数据变化之后，数据自动推送的效果。

## 三、Flink On K8S

在唯品会内部有一些不同的计算框架，有实时计算的，有机器学习的，还有离线计算的，所以需要一个统一的底层框架来进行管理，因此将 Flink 迁移到了 K8S 上。

在 K8S 上使用了思科的网络组件，每个 docker 容器都有独立的 ip，对外也是可见的。实时平台的融合器整体架构如下图所示。



唯品会在 K8S 上的实现方案与 Flink 社区提供的方案差异还是很大的。唯品会使用 K8S StatefulSet 模式部署，内部实现了 cluster 相关的一些接口。一个 job 对应一个 mini cluster，并且支持 HA。对于 Flink 来说，使用 StatefulSet 的最大的原因是 pod 的 hostname 是有序的；这样潜在的好处有：

1. hostname 为-0 和-1 的 pod 可以直接指定为 jobmanager；可以使用一个 statefulset 启动一个 cluster，而 deployment 必须 2 个；Jobmanager 和 TaskManager 分别独立的 deployment。
2. pod 由于各种原因 fail 后，由于 StatefulSet 重新拉起的 pod 的 hostname 不变，集群 recover 的速度理论上可以比 deployment 更快（deployment 每次主机名随机）。

## 容器的 entrypoint

由于要由主机名来判断是启动 jobmanager 还是 taskmanager，因此需要在 entrypoint 中去匹配设置的 jobmanager 的主机名是否有一致。

传入参数为："cluster ha"；则自动根据主机名判断启动那个角色；也可以直接指定角色名称

docker-entrypoint.sh 的脚本内容如下：

```
bash
#!/bin/sh
# If unspecified, the hostname of the container is taken as the JobManager
address
ACTION_CMD="$1"
# if use cluster model, pod ${JOB_CLUSTER_NAME}-0,${JOB_CLUSTER_NAME}-1 as
jobmanager
if [ ${ACTION_CMD} == "cluster" ]; then
    jobmanagers=(${JOB_MANGER_HOSTS//,/ })
    ACTION_CMD="taskmanager"
    for i in ${!jobmanagers[@]}
    do
        if [ "$(hostname -s)" == "${jobmanagers[i]}" ]; then
            ACTION_CMD="jobmanager"
            echo "pod hostname match jobmanager config host, change action to
jobmanager."
        fi
    done
fi
# if ha model, replace ha configuration
if [ "$2" == "ha" ]; then
    sed -i -e "s|high-availability.cluster-id: cluster-id|high-availability.cluster-
id: ${FLINK_CLUSTER_IDENT}|g" "$FLINK_CONF_DIR/flink-conf.yaml"
    sed -i -e "s|high-availability.zookeeper.quorum: localhost:2181|high-
availability.zookeeper.quorum: ${FLINK_ZK_QUORUM}|g" "$FLINK_CONF_DIR/flink-
conf.yaml"
```

```

sed -i -e "s|state.backend.fs.checkpointdir:
checkpointdir|state.backend.fs.checkpointdir: hdfs://user/flink/flink-
checkpoints/${FLINK_CLUSTER_IDENT}|g" "$FLINK_CONF_DIR/flink-conf.yaml"
sed -i -e "s|high-availability.storageDir: hdfs:///flink/ha/|high-
availability.storageDir: hdfs:///user/flink/ha/${FLINK_CLUSTER_IDENT}|g"
"$FLINK_CONF_DIR/flink-conf.yaml"
fi
if [ ${ACTION_CMD} == "help" ]; then
    echo "Usage: $(basename $0) (cluster ha|jobmanager|taskmanager|local|help)"
    exit 0
elif [ ${ACTION_CMD} == "jobmanager" ]; then
    JOB_MANAGER_RPC_ADDRESS=${JOB_MANAGER_RPC_ADDRESS:-$(hostname -f)}
    echo "Starting Job Manager"
    sed -i -e "s/jobmanager.rpc.address: localhost/jobmanager.rpc.address:
${JOB_MANAGER_RPC_ADDRESS}/g" "$FLINK_CONF_DIR/flink-conf.yaml"
    sed -i -e "s/jobmanager.heap.mb: 1024/jobmanager.heap.mb:
${JOB_MANAGER_HEAP_MB}/g" "$FLINK_CONF_DIR/flink-conf.yaml"
    echo "config file: " && grep '^[\^n#]' "$FLINK_CONF_DIR/flink-conf.yaml"
    exec "$FLINK_HOME/bin/jobmanager.sh" start-foreground cluster
elif [ ${ACTION_CMD} == "taskmanager" ]; then
    TASK_MANAGER_NUMBER_OF_TASK_SLOTS=${TASK_MANAGER_NUMBER_OF_TASK_SLOTS:-$(grep
-c ^processor /proc/cpuinfo)}
    echo "Starting Task Manager"
    sed -i -e "s/taskmanager.heap.mb: 1024/taskmanager.heap.mb:
${TASK_MANAGER_HEAP_MB}/g" "$FLINK_CONF_DIR/flink-conf.yaml"
    sed -i -e "s/taskmanager.numberOfTaskSlots: 1/taskmanager.numberOfTaskSlots:
${TASK_MANAGER_NUMBER_OF_TASK_SLOTS}/g" "$FLINK_CONF_DIR/flink-conf.yaml"
    echo "config file: " && grep '^[\^n#]' "$FLINK_CONF_DIR/flink-conf.yaml"
    exec "$FLINK_HOME/bin/taskmanager.sh" start-foreground
elif [ ${ACTION_CMD} == "local" ]; then
    echo "Starting local cluster"
    exec "$FLINK_HOME/bin/jobmanager.sh" start-foreground local
fi
exec "$@"

```

## entrypoint 变量说明

镜像的 docker entrypoint 脚本里面需要设置的环境变量设置说明：

环境变量名称	参数	示例内容	说明
JOB_MANGER_HOSTS	StatefulSet.name-0,StatefulSet.name-1	flink-cluster-0,flink- cluster-1	JM的主机名，短主机名；可以不用FQDN
FLINK_CLUSTER_IDENT	namespace/StatefulSet.name	default/flink-cluster	用来做zk ha设置和hdfs checkpoint的根目录
TASK_MANAGER_NUMBER_OF_TAS K_SLOTS	containers.resources.cpu.limits	000002	TM的slot数量，根据 resources.cpu.limits来设置
FLINK_ZK_QUORUM	env:FLINK_ZK_QUORUM	*.*.*:2181	HA ZK的地址
JOB_MANAGER_HEAP_MB	env:JOB_MANAGER_HEAP_MB value:cont ainers.resources.memory.limit -1024	004096	JM的Heap大小，由于存在堆外 内存，需要小于 container.resources.memory.lim its；否则容易OOM kill
TASK_MANAGER_HEAP_MB	env:TASK_MANAGER_HEAP_MB value: containers.resources.memory.limit -1024	004096	TM的Heap大小，由于存在 Netty的堆外内存，需要小于 container.resources.memory.lim its；否则容易OOM kill

## 使用 ConfigMap 维护配置

对应 Flink 集群所依赖的 HDFS 等其他配置，则通过创建 configmap 来管理和维护。

```
kubectl create configmap hdfs-conf --from-file=hdfs-site.xml --from-file=core-site.xml
```

```
bash
[hadoop@flink-jm-0 hadoop]$ ll /home/vipshop/conf/hadoop
total 0
lrwxrwxrwx. 1 root root 20 Apr  9 06:54 core-site.xml -> ..data/core-site.xml
lrwxrwxrwx. 1 root root 20 Apr  9 06:54 hdfs-site.xml -> ..data/hdfs-site.xml
```

## 四、后续计划

当前实时系统，机器学习平台要处理的数据分布在各种数据存储组件中，如 Kafka、Redis、Tair 和 HDFS 等，如何方便高效的访问，处理，共享这些数据是一个很大的挑战，对于当前的数据访问和解析常常需要耗费很多的精力，主要的痛点包括：

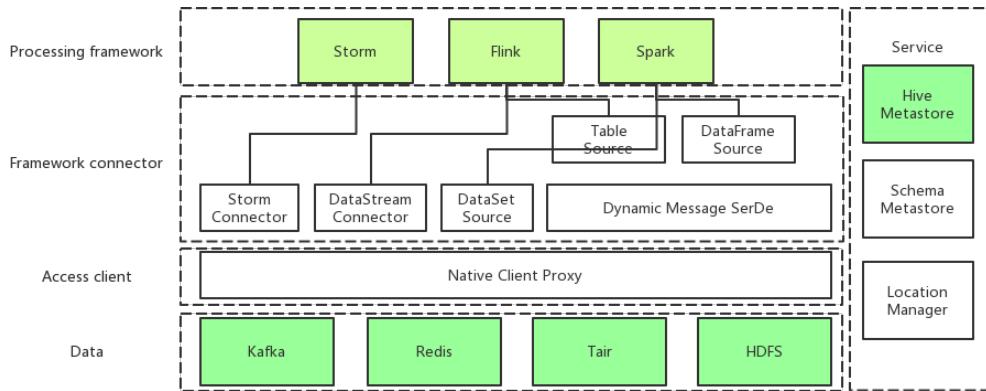
1. 对于 Kafka、Redis、Tair 中的 binary (PB/Avro 等格式) 数据，使用者无法快速直接的了解数据的 schema 与数据内容，采集数据内容及与写入者的沟通成本很高。
2. 由于缺少独立的统一数据系统服务，对 Kafka、Redis、Tair 等中的 binary 数据访问需要依赖写入者提供的信息，如 proto 生成类，数据格式 wiki 定义等，维护成本高，容易出错。
3. 缺乏 relational schema 使得使用者无法直接基于更高效易用的 SQL 或 LINQ 层 API 开发业务。
4. 无法通过一个独立的服务方便的发布和共享数据。
5. 实时数据无法直接提供给 Batch SQL 引擎使用。
6. 此外，对于当前大部分的数据源的访问也缺少审计，权限管理，访问监控，跟踪等特性。

UDM(统一数据管理系统) 包括 Location Manager, Schema Metastore 以及 Client Proxy 等模块，主要的功能包括：

1. 提供从名字到地址的映射服务，使用者通过抽象名字而不是具体地址访问数据。
2. 用户可以方便的通过 Web GUI 界面方便的查看数据 Schema，探查数据内容。
3. 提供支持审计，监控，溯源等附加功能的 Client API Proxy。

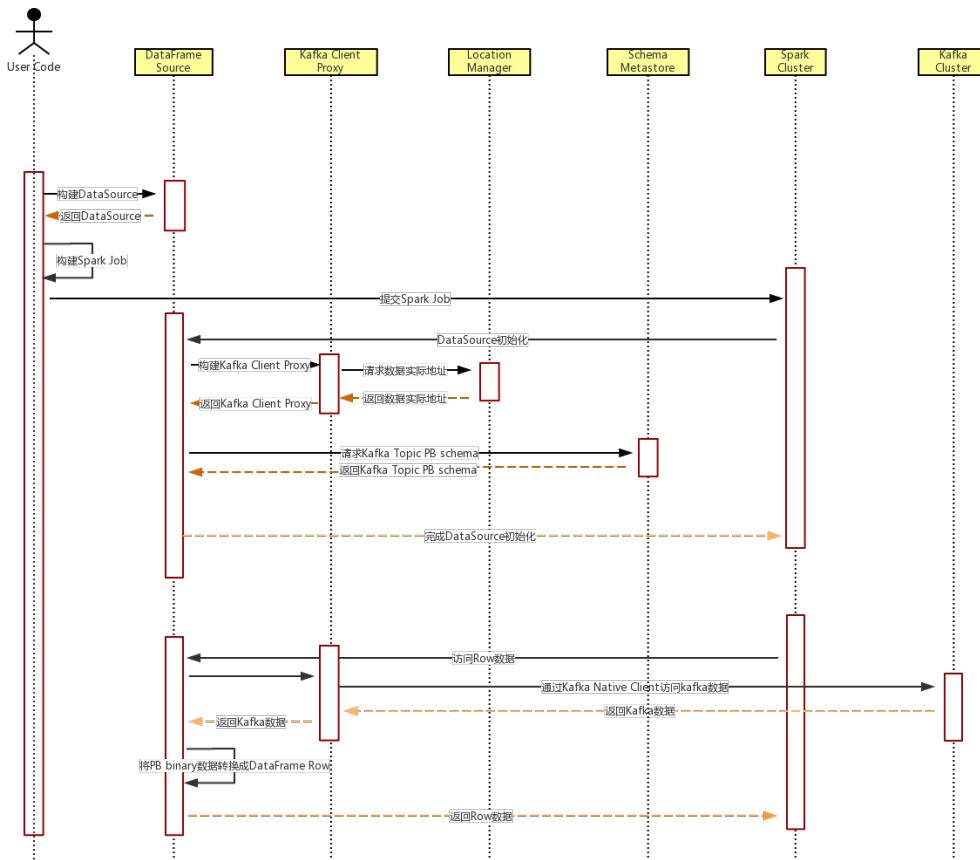
4. 在 Spark/Flink/Storm 等框架中，以最适合使用的形式提供这些数据源的封装。

UDM 的整体架构如下图所示。



UDM 的使用者包括实时，机器学习以及离线平台中数据的生产者和使用者。在使用 Sql API 或 Table API 的时候，首先完成 Schema 的注册，之后使用 Sql 进行开发，降低了开发代码量。

以 Spark 访问 Kafka PB 数据的时序图来说明 UDM 的内部流程



在 Flink 中，使用 UDMExternalCatalog 来打通 Flink 计算框架和 UDM 之间的桥梁，通过实现 ExternalCatalog 的各个接口，以及实现各自数据源的 TableSourceFactory，完成 Schema 和接入管控等各项功能。同时增强 Flink 的 SQLClient 的各项功能，可以通过调用 API 查询 UDM 的 Schema，完成 SQL 任务的生成和提交。

# 携程基于 Apache Flink 的 实时特征平台

作者 刘康

整理 张宏柯



本文来自 7 月 26 日在上海举行的 Flink Meetup 会议，分享来自于刘康，目前在大数据平台部从事模型生命周期相关平台开发，现在主要负责基于 Apache Flink（以下简称 Flink）开发实时模型特征计算平台。熟悉分布式计算，在模型部署及运维方面有丰富实战经验和深入的理解，对模型的算法及训练有一定的了解。

本文主要内容如下：

- 在公司实时特征开发的现状基础上，说明实时特征平台的开发背景、目标以及现状
- 选择 Flink 作为平台计算引擎的原因
- Flink 的实践：有代表性的使用示例、为兼容 Aerospike（平台的存储介质）的开发以及碰到的坑
- 当前效果&未来规划

## 一、在公司实时特征开发的现状基础上，说明实时特征平 台的开发背景、目标以及现状

### 1、原实时特征作业的开发运维；

1.1、选择实时计算平台：依据项目的性能指标要求（latency, throughput 等），在已有的实时计算平台：Storm Spark flink 进行选择

1.2、主要的开发运维过程：

- 80%以上的作业需要用到消息队列数据源，但是消息队列为非结构化数据且没有统一的数据字典。所以需要通过消费对应的 topic，解析消息并确定所需的内容
- 基于需求中的场景，设计开发计算逻辑
- 在实时数据不能完全满足数据需求的情况下，另外开发单独的离线作业以及融合逻辑；

例如：在需要 30 天数据的场景下，但消息队列中只有七天内的数据时（kafka 中消息的默认保留时间），剩下 23 天就需要用离线数据来补充。

- 设计开发数据的校验和纠错逻辑

消息的传输需要依赖网络，消息丢失和超时难以完全避免，所以需要有一个校验和纠错的逻辑。

- 测试上线
- 监控和预警

## 2、原实时特征作业的开发痛点

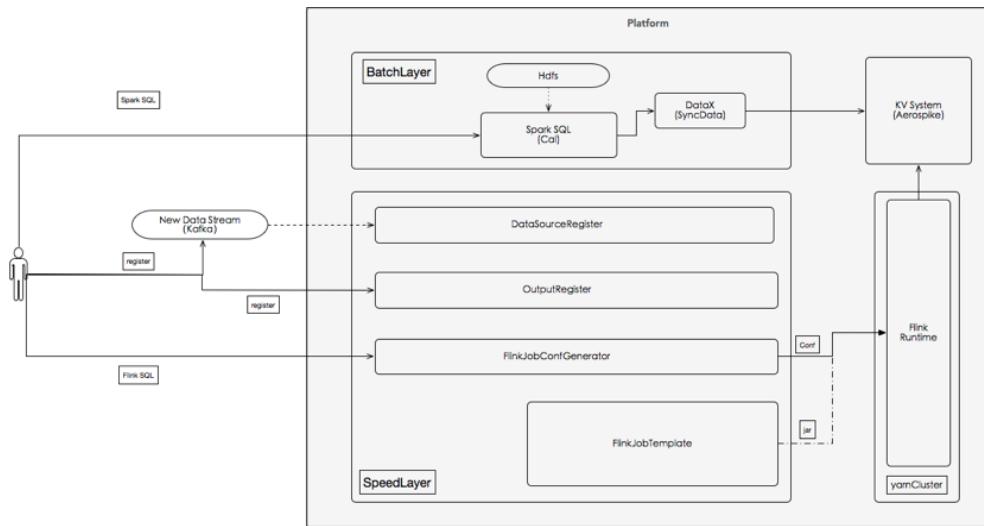
- 消息队列数据源结构没有统一的数据字典
- 特征计算逻辑高度定制化，开发测试周期长
- 实时数据不能满足需求时，需要定制离线作业和融合逻辑
- 校验和纠错方案没有形成最佳实践，实际效果比较依赖个人能力
- 监控和预警方案需要基于业务逻辑定制

## 3、基于整理的痛点，确定下来的平台目标

- 实时数据字典：提供统一的数据源注册、管理功能，支持单一结构消息的 topic 和包含多种不同结构消息的 topic
- 逻辑抽象：抽象为 SQL，减少工作量&降低使用门槛
- 特征融合：提供融合特征的功能，解决实时特征不能完全满足数据需求的情况
- 数据校验和纠错：提供利用离线数据校验和纠错实时特征的功能
- 实时计算延迟：ms 级

- 实时计算容错：端到端 exactly-once
- 统一的监控预警和 HA 方案

## 4、特征平台系统架构



现在的架构是标准 lambda 架构，离线部分由 spark sql + dataX 组成。现在使用的是 KV 存储系统 Aerospike , 跟 redis 的主要区别是使用 SSD 作为主存, 我们压测下来大部分场景读写性能跟 redis 在同一个数据量级。

实时部分：使用 flink 作为计算引擎，介绍一下用户的使用方式：

- 注册数据源：目前支持的实时数据源主要是 Kafka 和 Aerospike，其中 Aerospike 中的数据如果是在平台上配置的离线或者实时特征，会进行自动注册。Kafka 数据源需要上传对应的 schemaSample 文件
- 计算逻辑：通过 SQL 表达
- 定义输出：定义输出的 Aerospike 表和可能需要的 Kafka Topic, 用于推送 Update 或者 Insert 的数据的 key

用户完成上面的操作后，平台将所有信息写入到 json 配置文件。下一步平台将配置文件和之前准备好的 flinkTemplate.jar(包含所有平台所需的 flink 功能)提交给 yarn，启动 flink job。

## 5、平台功能展示

## 1) 平台功能展示-数据源注册

注册数据源信息

数据源名称\*  数据源名称  
数据源名称必填

Topic类型\*  HermesKafka  Kafka  
Topic编码\*  Avro  Json

上传JsonSchema文件  vcar\_test.json

Field名称	Tab列名	类型	自定义类型类型	默认EventTime属性
bu	bu	STRING		
custom_key	custom_key	STRING		
dtcityid	dtcityid	INT		
dtcityname:string	dtcityname	STRING		
pkid:int	pkid	INT		
slcityid	slcityid	INT		
slcityname:string	slcityname	STRING		
ts:long	ts	LONG		

显示 10 行 搜索: 上一页 1 下一页

数据源类型\*  注册  转换

**完成** 取消

## 2) 实时特征编辑-基本信息

特征编辑与测试

首页 / 特征管理 / 特征编辑与测试

在线特征 离线特征

1 基本信息 2 数据源信息 3 计算信息 4 存储信息

Tip: 首次创建特征请先在配置管理中配置数据源,分类匀特征输出等配置

实时特征名称\*  实时特征名称必填

实时特征别名\*  实时特征别名必填

是否公开\*  公开

版本信息\*  1

所属分类

**完成** 取消

### 3) 实时特征编辑-数据源选择

特征编辑与测试

首页 / 特征管理 / 特征编辑与测试

1 基本信息 2 数据源信息 3 计算信息 4 存储信息

KAFKA 数据源位置信息

选择数据源\*

test

AeroSpike 数据源 (可选)

已经存储到As的表

default\_online\_features.ops\_user\_info

立即创建 取消

### 4) 实时特征编辑-SQL 计算

特征编辑与测试

首页 / 特征管理 / 特征编辑与测试

1 基本信息 2 数据源信息 3 计算信息 4 存储信息

计算引擎:

FLINK

实时计算描述:

计算描述

实时计算sql:

```
1 select ID, cityId, cityName from test
2 LEFT JOIN LATERAL TABLE(asGetBins(PRODUCTID)) as T(cityId,cityName) ON TRUNC
```

#	列名	类型	映射主键
0	ID	Key/Value	userId
1	cityId	Key/Value	city
2	cityName	Key/Value	

是否进行Checkpoint:

checkPoint触发间隔(ms):

100  
checkPoint间隔小时间(ms):  
50

## 5) 实时特征编辑-选择输出

特征编辑与测试

首页 / 特征管理 / 特征编辑与测试

The screenshot shows a step-by-step process for feature editing. Step 1: Basic Information (Output to AeroSpike [必填]). Step 2: Data Source Information (Output to Kafka [可选]). Step 3: Calculation Information (Storage Configuration). Step 4: Persistence Information (未显示). The storage configuration includes selecting a table 'default\_online\_features.ops\_forTest' and setting a TTL of '1'. Buttons at the bottom include '完成' (Finish) and '取消' (Cancel).

## 二、选择 Flink 的原因

我们下面一个我们说一下我们选择 flink 来做这个特征平台的原因。



Flink China

	storm V1.2.2	spark Structured Streaming v2.3.1	flink v1.5
延迟	Streaming ms级	MicroBatch 100ms级 Streaming ms级 (实验)	Streaming ms级
容错	Ack atLeastOnce	CheckPoint&WAL exactlyOnce	CheckPoint&SavePoint exactlyOnce
SQL成熟度 (Unsupported Functions)	aggregation、join	distinct、limit、order by(partial)	distinct aggregate

分为三个维度：最高延迟、容错、sql 功能成熟度

- 延迟: storm 和 flink 是纯流式，最低可以达到毫秒级的延迟。spark 的纯流式机制是 continuous 模式，也可以达最低毫秒级的延迟
- 容错: storm 使用异或 ack 的模式，支持 atLeastOnce。消息重复解决不。spark 通过 checkpoint 和 WAL 来提供 exactlyOnce。flink 通过 checkpoint 和 SavePoint 来做到 exactlyOnce。
- sql 成熟度: storm 现在的版本中 SQL 还在一个实验阶段，不支持聚合和 join。spark 现在可以提供绝大部分功能，不支持 distinct、limit 和聚合结果的 order by。flink 现在社区版中提供的 sql，不支持 distinct aggregate

## 三、Flink 实践

### 1、实用示例

The slide features a dark blue header with the Flink China logo (a cartoon character) and the text "Flink China". Below the header, the title "使用用例" is displayed in large white font. The main content area contains two examples, each consisting of a teal chevron icon pointing right, followed by the feature name, a teal circular icon with a checkmark, and a detailed description.

**SessionWin** 会话窗口用途广泛，可以用于推荐召回、用户召回等；原设计方案需要使用分布式锁做进程间的并发控制，复杂度高  
select user,count(ts) from vac\_ts group by SESSION(userEventTime\_ts, INTERVAL '10' SECOND),user

**External CheckPoint** 实时计算job执行异常设置的重启策略无法恢复需要修复bug时，如果设置了 externalCheckPoint，很大几率保证exactly Once的语义

2、兼容开发：flink 现在没有对 Aerospike 提供读写支持，所以需要二次开发



## 兼容开发

**AS=Apache**

- AS Async Upsert TableSink**
  - 基于flink Asynchronous I/O开发 ( 支持exactlyOnce语义 )
  - 异步写入AS
  - 写入AS成功后可选择push key至Kafka
  - 支持Upert Stream的TableSink
- UDF-AS Table Functions**
  - 通过定义Table Function类型的UDF支持，SQL中用到AS数据的情况
  - 示例：
  - SELECT ID, cityId, cityName FROM product LEFT JOIN LATERAL TABLE(asGetBins(productId)) as T(cityId,cityName) ON TRUE

3、碰到的坑



## 碰到的坑

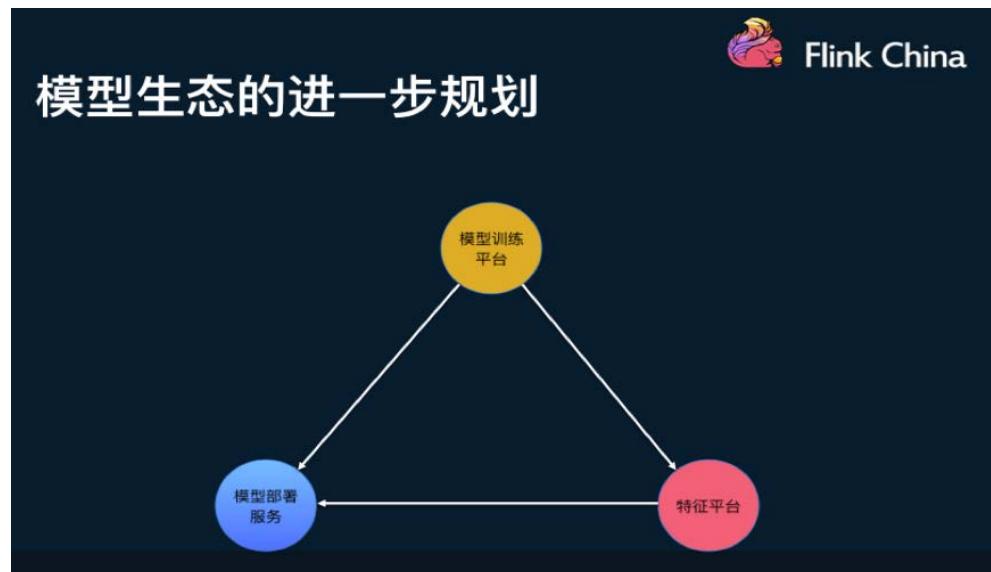
- SQL\_TIMESTAMP**
  - 内部默认反序列化JSON中SQL\_TIMESTAMP类型时，值必须是格式为yyyy-MM-dd' T'HH:mm:ss.SSS'Z的字符串
  - 场景：直接使用StreamTableEnvironment的方法registerTableSource注册表时，表中的时间属性 ( eventTime ) 字段的类型需要配置为SQL\_TIMESTAMP
- SlidingWin Offset**
  - 窗口的offset需要小于winSize，不能用于时区适配
  - 下面是当前计算win的start值的公式
  - $$\text{timestamp} - (\text{timestamp} - \text{offset} + \text{windowSize}) \% \text{windowSize}$$

## 四、平台当前效果&未来规划

当前效果：将实时特征上线周期从原平均 3 天-5 天降至小时级。未来规划：

- 完善特征平台的功能：融合特征等
- 简化步骤，提高用户体验
- 根据需求，进一步完善 SQL 的功能例如支持 win 的开始时间 offset，可以通过 countTrigger 的 win 等

下一步的规划是通过 sql 或者 DSL 来描述模型部署和模型训练



# 一文了解 Apache Flink 核心技术

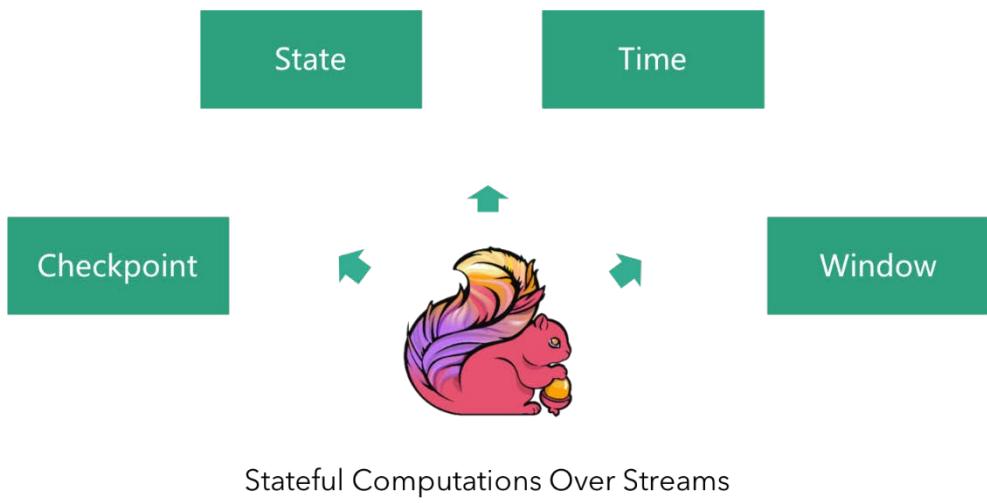
作者 伍翀



## Apache Flink 介绍

Apache Flink（以下简称 Flink）是近年来越来越流行的一款开源大数据计算引擎，它同时支持了批处理和流处理，也能用来做一些基于事件的应用。使用官网的语句来介绍 Flink 就是 "**Stateful Computations Over Streams**"。

首先 Flink 是一个纯流式的计算引擎，它的基本数据模型是数据流。流可以是无边界的无限流，即一般意义上的流处理。也可以是有边界的有限流，这样就是批处理。因此 Flink 用一套架构同时支持了流处理和批处理。其次，Flink 的一个优势是支持有状态的计算。如果处理一个事件（或一条数据）的结果只跟事件本身的内容有关，称为无状态处理；反之结果还和之前处理过的事件有关，称为有状态处理。稍微复杂一点的数据处理，比如说基本的聚合，数据流之间的关联都是有状态处理。

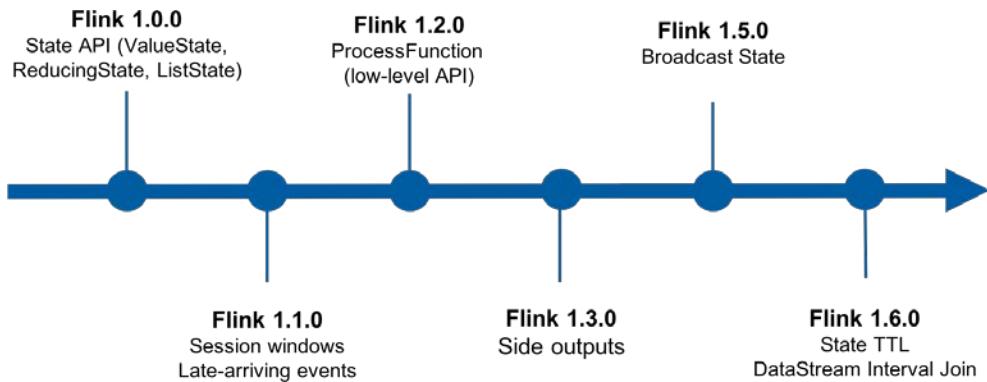


## Apache Flink 基石

Apache Flink 之所以能越来越受欢迎, 我们认为离不开它最重要的四个基石: Checkpoint、State、Time、Window。

首先是 Checkpoint 机制, 这是 Flink 最重要的一个特性。Flink 基于 Chandy-Lamport 算法实现了分布式一致性的快照, 从而提供了 exactly-once 的语义。在 Flink 之前的流计算系统 (如 Strom, Samza) 都没有很好地解决 exactly-once 的问题。提供了一致性的语义之后, Flink 为了让用户在编程时能够更轻松、更容易地去管理状态, 引入了托管状态 (managed state) 并提供了 API 接口, 让用户使用起来感觉就像在用 Java 的集合类一样。除此之外, Flink 还实现了 watermark 的机制, 解决了基于事件时间处理时的数据乱序和数据迟到的问题。最后, 流计算中的计算一般都会基于窗口来计算, 所以 Flink 提供了一套开箱即用的窗口操作, 包括滚动窗口、滑动窗口、会话窗口, 还支持非常灵活的自定义窗口以满足特殊业务的需求。

## Flink API 历史变迁



在 **Flink 1.0.0** 时期，加入了 State API，即 ValueState、ReducingState、ListState 等等。State API 可以认为是 Flink 里程碑式的创新，它能够让用户像使用 Java 集合一样地使用 Flink State，却能够自动享受到状态的一致性保证，不会因为故障而丢失状态。包括后来 Apache Beam 的 State API 也从中借鉴了很多。

在 **Flink 1.1.0** 时期，支持了 Session Window 并且能够正确的处理乱序的迟到数据，使得最终结果是正确的

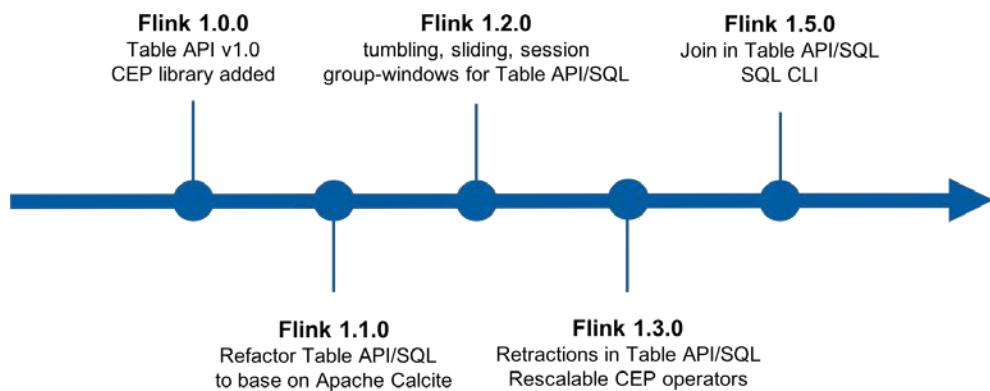
在 **Flink 1.2.0** 时期，提供了 ProcessFunction，这是一个 Lower-level 的 API，用于实现更高级更复杂的功能。它除了能够注册各种类型的 State 外，还支持注册定时器（支持 EventTime 和 ProcessingTime），常用于开发一些基于事件、基于时间的应用程序。

在 **Flink 1.3.0** 时期，提供了 Side Output 功能。算子的输出一般只有一种输出类型，但是有些时候可能需要输出另外的类型，比如除了输出主流外，还希望把一些异常数据、迟到数据以侧边流的形式进行输出，并分别交给下游不同节点进行处理。简而言之，Side Output 支持了多路输出的功能。

在 **Flink 1.5.0** 时期，加入了 BroadcastState。BroadcastState 是对 State API 的一个扩展。它用来存储上游被广播过来的数据，这个 operator 的每个并发上存的 BroadcastState 里面的数据都是一模一样的，因为它是从上游广播而来的。基于这种 State 可以比较好地去解决 CEP 中的动态规则的功能，以及 SQL 中不等值 Join 的场景。

在 **Flink 1.6.0** 时期，提供了 State TTL 功能、DataStream Interval Join 功能。State TTL 实现了在申请某个 State 时候可以在指定一个生命周期参数 (TTL)，指定该 state 过了多久之后需要被系统自动清除。在这个版本之前，如果用户想要实现这种状态清理操作需要使用 ProcessFunction 注册一个 Timer，然后利用 Timer 的回调手动把这个 State 清除。从该版本开始，Flink 框架可以基于 TTL 原生地解决这件事情。DataStream Interval Join 使得区间 Join 成为可能。例如左流的每一条数据去 Join 右流前后 5 分钟之内的数据，这种就是 5 分钟的区间 Join。

## Flink High-Level API 历史变迁



在 **Flink 1.0.0** 时期，Table API（结构化数据处理 API）和 CEP（复杂事件处理 API）这两个框架被首次加入到仓库中。Table API 是一种结构化的高级 API，支持 Java 语言和 Scala 语言，类似于 Spark 的 DataFrame API。Table API 和 SQL 非常相近，他们都是一种处理结构化数据的语言，实现上可以共用很多内容。所以在 **Flink 1.1.0** 里面，社区基于 Apache Calcite 对整个 Table 模块做了重构，使得同时支持了 Table API 和 SQL 并共用了大部分代码。

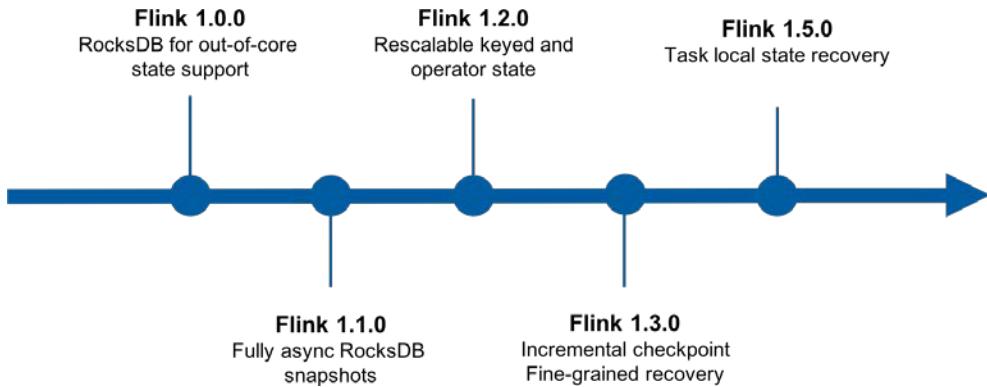
在 **Flink 1.2.0** 时期，社区在 Table API 和 SQL 上支持丰富的内置窗口操作，包括 Tumbling Window、Sliding Window、Session Window。

在 **Flink 1.3.0** 时期，社区首次提出了 Dynamic Table 这个概念，借助 Dynamic Table，流和批之间可以相互进行转换。流可以是一张表，表也可以是一张流，这是流批统一的基础之一。其中 Retraction 机制是实现 Dynamic Table 的基础之一，基于 Retraction 才能够正确地实现多级 Aggregate、多级 Join，才能够保证流式 SQL 的语义与结果的正确性。另外，在该版本中还支持

了 CEP 算子的可伸缩容（即改变并发）。

在 **Flink 1.5.0** 时期，在 Table API 和 SQL 上支持了 Join 操作，包括无限流的 Join 和带窗口的 Join。还添加了 SQL CLI 支持。SQL CLI 提供了一个类似 Shell 命令的对话框，可以交互式执行查询。

## Flink Checkpoint & Recovery 历史变迁



Checkpoint 机制在 Flink 很早期的时候就已经支持，是 Flink 一个很核心的功能，Flink 社区也一直努力提升 Checkpoint 和 Recovery 的效率。

在 **Flink 1.0.0** 时期，提供了 RocksDB 状态后端的支持，在这个版本之前所有的状态数据只能存在进程的内存里面，JVM 内存是固定大小的，随着数据越来越多总会发生 FullGC 和 OOM 的问题，所以在生产环境中很难应用起来。如果想要存更多数据、更大的 State 就要用到 RocksDB。RocksDB 是一款基于文件的嵌入式数据库，它会把数据存到磁盘，同时又提供高效的读写性能。所以使用 RocksDB 不会发生 OOM 这种事情。

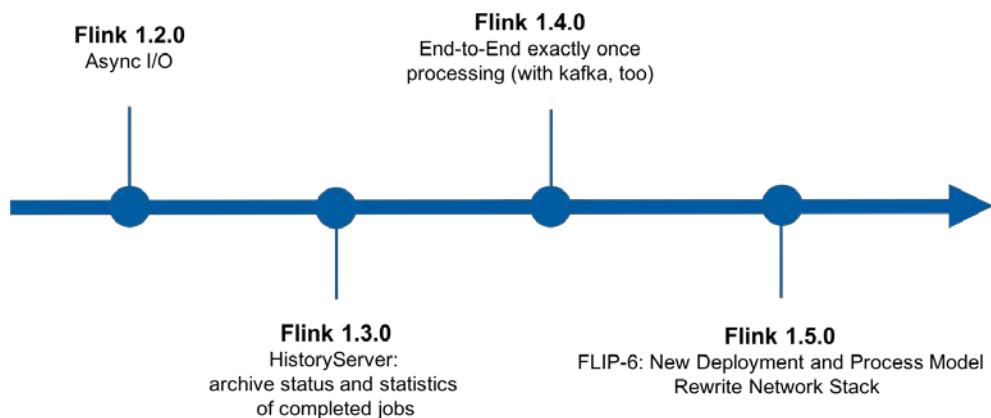
在 **Flink 1.1.0** 时期，支持了 RocksDB Snapshot 的异步化。在之前的版本，RocksDB 的 Snapshot 过程是同步的，它会阻塞主数据流的处理，很影响吞吐量。在支持异步化之后，吞吐量得到了极大的提升。

在 **Flink 1.2.0** 时期，通过引入 KeyGroup 的机制，支持了 KeyedState 和 OperatorState 的可扩缩容。也就是支持了对带状态的流计算任务改变并发的功能。

在 **Flink 1.3.0** 时期，支持了 Incremental Checkpoint（增量检查点）机制。Incremental Checkpoint 的支持标志着 Flink 流计算任务正式达到了生产就绪状态。增量检查点是每次只将本次 checkpoint 期间新增的状态快照并持久化存储起来。一般流计算任务，GB 级别的状态，甚至 TB 级别的状态是非常常见的，如果每次都把全量的状态都刷到分布式存储中，这个效率和网络代价是很大的。如果每次只刷新新增的数据，效率就会高很多。在这个版本里面还引入了细粒度的 recovery 的功能，细粒度的 recovery 在做恢复的时候，只需要恢复失败节点的联通子图，不用对整个 Job 进行恢复，这样便能够提高恢复效率。

在 **Flink 1.5.0** 时期，引入了本地状态恢复的机制。因为基于 checkpoint 机制，会把 State 持久化地存储到某个分布式存储，比如 HDFS，当发生 failover 的时候需要重新把数据从远程 HDFS 再下载下来，如果这个状态特别大那么下载耗时就会较长，failover 恢复所花的时间也会拉长。本地状态恢复机制会提前将状态文件在本地也备份一份，当 Job 发生 failover 之后，恢复时可以在本地直接恢复，不需从远程 HDFS 重新下载状态文件，从而提升了恢复的效率。

## Flink Runtime 历史变迁



在 **Flink 1.2.0** 时期，提供了 Async I/O 功能。Async I/O 是阿里巴巴贡献给社区的一个呼声非常高的特性，主要目的是为了解决与外部系统交互时网络延迟成为了系统瓶颈的问题。例如，为了关联某些字段需要查询外部 HBase 表，同步的方式是每次查询的操作都是阻塞的，数据流会被频繁的 I/O 请求卡住。当使用异步 I/O 之后就可以同时地发起 N 个异步查询的请求，不会阻塞主数据流，这样便提升了整个 job 的吞吐量，提升 CPU 利用率。

在 **Flink 1.3.0** 时期，引入了 HistoryServer 的模块。HistoryServer 主要功能是当 job 结束以后，会把 job 的状态以及信息都进行归档，方便后续开发人员做一些深入排查。

在 **Flink 1.4.0** 时期，提供了端到端的 exactly-once 的语义保证。Exactly-once 是指每条输入的数据只会作用在最终结果上有且只有一次，即使发生软件或硬件的故障，不会有丢数据或者重复计算发生。而在该版本之前，exactly-once 保证的范围只是 Flink 应用本身，并不包括输出给外部系统的部分。在 failover 时，这就有可能写了重复的数据到外部系统，所以一般会使用幂等的外部系统来解决这个问题。在 Flink 1.4 的版本中，Flink 基于两阶段提交协议，实现了端到端的 exactly-once 语义保证。内置支持了 Kafka 的端到端保证，并提供了 TwoPhaseCommitSinkFunction 供用于实现自定义外部存储的端到端 exactly-once 保证。

在 **Flink 1.5.0** 时期，Flink 发布了新的部署模型和处理模型（FLIP6）。新部署模型的开发工作已经持续了很久，该模型的实现对 Flink 核心代码改动特别大，可以说是自 Flink 项目创建以来，Runtime 改动最大的一次。简而言之，新的模型可以在 YARN, MESOS 调度系统上更好地动态分配资源、动态释放资源，并实现更高的资源利用率，还有提供更好的作业之间的隔离。

除了 FLIP6 的改进，在该版本中，还对网站栈做了重构。重构的原因是在老版本中，上下游多个 task 之间的通信会共享同一个 TCP connection，导致某一个 task 发生反压时，所有共享该连接的 task 都会被阻塞，反压的粒度是 TCP connection 级别的。为了改进反压机制，Flink 应用了在解决网络拥塞时一种经典的流控方法——基于 Credit 的流量控制。使得流控的粒度精细到具体某个 task 级别，有效缓解了反压对吞吐量的影响。

## 总结

Flink 同时支持了流处理和批处理，目前流计算的模型已经相对比较成熟和领先，也经历了各个公司大规模生产的验证。社区在接下来将继续加强流计算方面的性能和功能，包括对 Flink SQL 扩展更丰富的功能和引入更多的优化。另一方面也将加大力量提升批处理、机器学习等生态上的能力。

# 流计算框架 Flink 与 Storm 的性能对比

作者 孙梦瑶



## 1. 背景

Apache Flink 和 Apache Storm 是当前业界广泛使用的两个分布式实时计算框架。其中 [Apache Storm](#) (以下简称“Storm”) 在美团点评实时计算业务中已有较为成熟的运用 (可参考 [Storm 的可靠性保证测试](#))，有管理平台、常用 API 和相应的文档，大量实时作业基于 Storm 构建。而 [Apache Flink](#) (以下简称“Flink”) 在近期倍受关注，具有高吞吐、低延迟、高可靠和精确计算等特性，对事件窗口有很好的支持，目前在美团点评实时计算业务中也已有一定应用。

为深入熟悉了解 Flink 框架，验证其稳定性和可靠性，评估其实时处理性能，识别该体系中的缺点，找到其性能瓶颈并进行优化，给用户提供最适合的实时计算引擎，我们以实践经验丰富的 Storm 框架作为对照，进行了一系列实验测试 Flink 框架的性能，计算 Flink 作为确保“至少一次”和“恰好一次”语义的实时计算框架时对资源的消耗，为实时计算平台资源规划、框架选择、性能调优等决策及 Flink 平台的建设提出建议并提供数据支持，为后续的 SLA 建设提供一定参考。

Flink 与 Storm 两个框架对比：

	Storm	Flink
状态管理	无状态，需用户自行进行状态管理	有状态
窗口支持	对事件窗口支持较弱，缓存整个窗口的所有数据，窗口结束时一起计算	窗口支持较为完善，自带一些窗口聚合方法，并且会自动管理窗口状态。
消息投递	At Most Once At Least Once	At Most Once At Least Once <b>Exactly Once</b>
容错方式	<b>ACK 机制</b> ：对每个消息进行全链路跟踪，失败或超时进行重发。	<b>检查点机制</b> ：通过分布式一致性快照机制，对数据流和算子状态进行保存。在发生错误时，使系统能够进行回滚。
应用现状	在美团点评实时计算业务中已有较为成熟的运用，有管理平台、常用 API 和相应的文档，大量实时作业基于 Storm 构建。	在美团点评实时计算业务中已有一定应用，但是管理平台、API 及文档等仍需进一步完善。

## 2. 测试目标

评估不同场景、不同数据压力下 Flink 和 Storm 两个实时计算框架目前的性能表现，获取其详细性能数据并找到处理性能的极限；了解不同配置对 Flink 性能影响的程度，分析各种配置的适用场景，从而得出调优建议。

### 2.1 测试场景

#### “输入-输出”简单处理场景

通过对“输入-输出”这样简单处理逻辑场景的测试，尽可能减少其它因素的干扰，反映两个框架本身性能。

同时测算框架处理能力的极限，处理更加复杂的逻辑的性能不会比纯粹“输入-输出”更高。

#### 用户作业耗时较长的场景

如果用户的处理逻辑较为复杂，或是访问了数据库等外部组件，其执行时间会增大，作业的性能会受到影响。因此，我们测试了用户作业耗时较长的场景下两个框架的调度性能。

## 窗口统计场景

实时计算中常有对时间窗口或计数窗口进行统计的需求，例如一天中每五分钟的访问量，每 100 个订单中有多少个使用了优惠等。Flink 在窗口支持上的功能比 Storm 更加强大，API 更加完善，但是我们同时也想了解在窗口统计这个常用场景下两个框架的性能。

### 精确计算场景（即消息投递语义为“恰好一次”）

Storm 仅能保证“至多一次”（At Most Once）和“至少一次”（At Least Once）的消息投递语义，即可能存在重复发送的情况。有很多业务场景对数据的精确性要求较高，希望消息投递不重不漏。Flink 支持“恰好一次”（Exactly Once）的语义，但是在限定的资源条件下，更加严格的精确度要求可能带来更高的代价，从而影响性能。因此，我们测试了在不同消息投递语义下两个框架的性能，希望为精确计算场景的资源规划提供数据参考。

## 2.2 性能指标

### 吞吐量（Throughput）

- 单位时间内由计算框架成功地传送数据的数量，本次测试吞吐量的单位为：条/秒。
- 反映了系统的负载能力，在相应的资源条件下，单位时间内系统能处理多少数据。
- 吞吐量常用于资源规划，同时也用于协助分析系统性能瓶颈，从而进行相应的资源调整以保证系统能达到用户所要求的处理能力。假设商家每小时能做二十份午餐（吞吐量 20 份/小时），一个外卖小哥每小时只能送两份（吞吐量 2 份/小时），这个系统的瓶颈就在小哥配送这个环节，可以给该商家安排十个外卖小哥配送。

### 延迟（Latency）

- 数据从进入系统到流出系统所用的时间，本次测试延迟的单位为：毫秒。
- 反映了系统处理的实时性。
- 金融交易分析等大量实时计算业务对延迟有较高要求，延迟越低，数据实时性越强。
- 假设商家做一份午餐需要 5 分钟，小哥配送需要 25 分钟，这个流程中用户感受到了 30

分钟的延迟。如果更换配送方案后延迟变成了 60 分钟，等送到了饭菜都凉了，这个新的方案就是无法接受的。

## 3. 测试环境

为 Storm 和 Flink 分别搭建由 1 台主节点和 2 台从节点构成的 Standalone 集群进行本次测试。其中为了观察 Flink 在实际生产环境中的性能，对于部分测内容也进行了 on Yarn 环境的测试。

### 3.1 集群参数

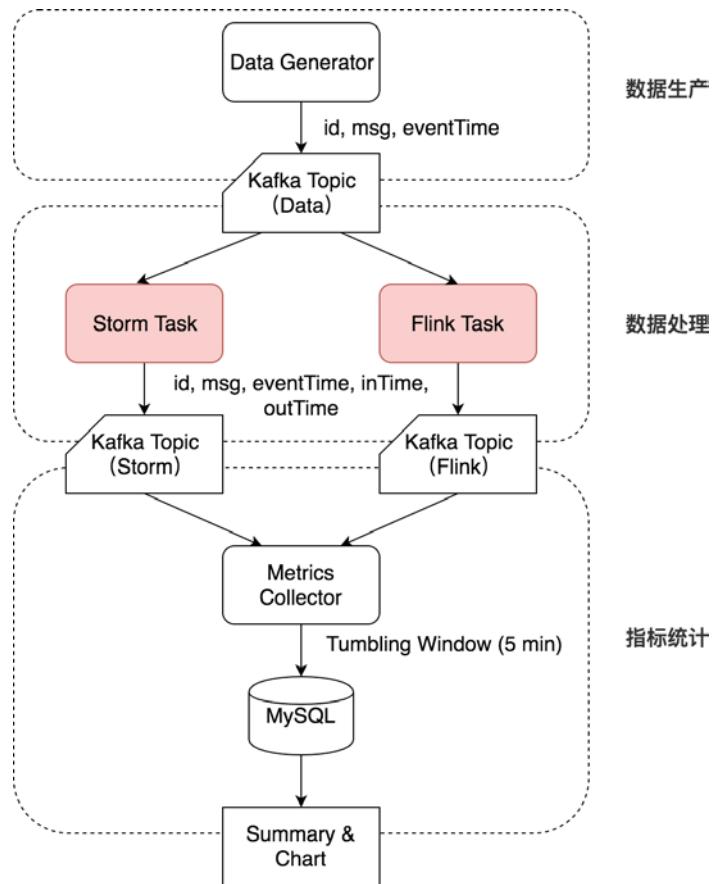
参数项	参数值
CPU	QEMU Virtual CPU version 1.1.2 2.6GHz
Core	8
Memory	16GB
Disk	500G
OS	CentOS release 6.5 (Final)

### 3.2 框架参数

参数项	Storm 配置	Flink 配置
Version	Storm 1.1.0-mt002	Flink 1.3.0
Master Memory	2600M	2600M
Slave Memory	1600M * 16	12800M * 2
Parallelism	2 supervisor 16 worker	2 Task Manager 16 Task slots

## 4. 测试方法

### 4.1 测试流程



## 数据生产

Data Generator 按特定速率生成数据，带上自增的 id 和 eventTime 时间戳写入 Kafka 的一个 Topic (Topic Data)。

## 数据处理

Storm Task 和 Flink Task (每个测试用例不同) 从 Kafka Topic Data 相同的 Offset 开始消费，并将结果及相应 inTime、outTime 时间戳分别写入两个 Topic (Topic Storm 和 Topic Flink) 中。

## 指标统计

Metrics Collector 按 outTime 的时间窗口从这两个 Topic 中统计测试指标，每五分钟将相应的指标写入 MySQL 表中。

Metrics Collector 按 outTime 取五分钟的滚动时间窗口，计算五分钟的平均吞吐（输出数据的条数）、五分钟内的延迟 (outTime - eventTime 或 outTime - inTime) 的中位数及 99 线等指标，写入 MySQL 相应的数据表中。最后对 MySQL 表中的吞吐计算均值，延迟中位数及延迟 99 线选取中位数，绘制图像并分析。

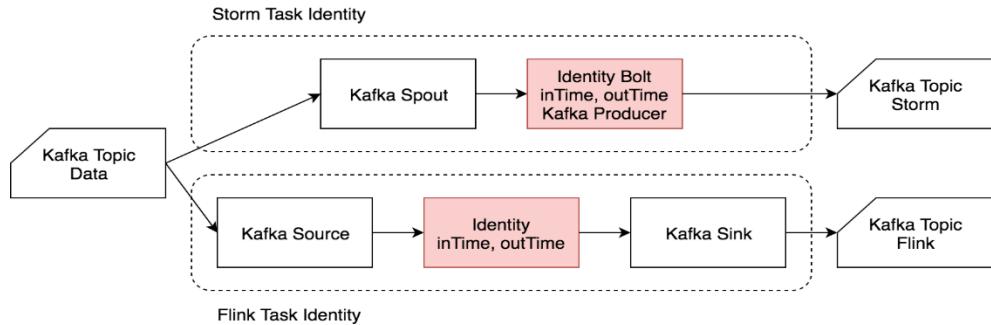
## 4.2 默认参数

- Storm 和 Flink 默认均为 **At Least Once** 语义。
  - Storm 开启 ACK，ACKer 数量为 1。
  - Flink 的 Checkpoint 时间间隔为 30 秒，默认 StateBackend 为 Memory。
- 保证 Kafka 不是性能瓶颈，尽可能排除 Kafka 对测试结果的影响。
- 测试延迟时数据生产速率小于数据处理能力，假设数据被写入 Kafka 后立刻被读取，即 eventTime 等于数据进入系统的时间。
- 测试吞吐量时从 Kafka Topic 的最旧开始读取，假设该 Topic 中的测试数据量充足。

## 4.3 测试用例

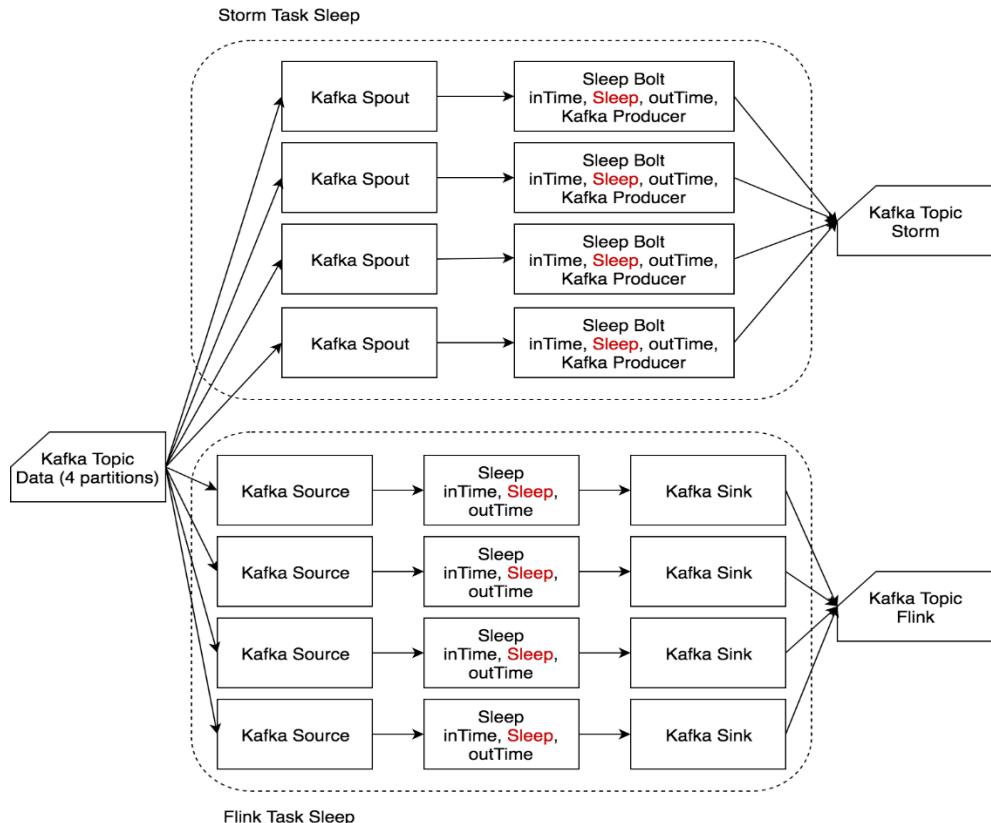
### Identity

- Identity 用例主要模拟“输入-输出”简单处理场景，反映两个框架本身的性能。
- 输入数据为“msgId, eventTime”，其中 eventTime 视为数据生成时间。单条输入数据约 20 B。
- 进入作业处理流程时记录 inTime，作业处理完成后（准备输出时）记录 outTime。
- 作业从 Kafka Topic Data 中读取数据后，在字符串末尾追加时间戳，然后直接输出到 Kafka。
- 输出数据为“msgId, eventTime, inTime, outTime”。单条输出数据约 50 B。



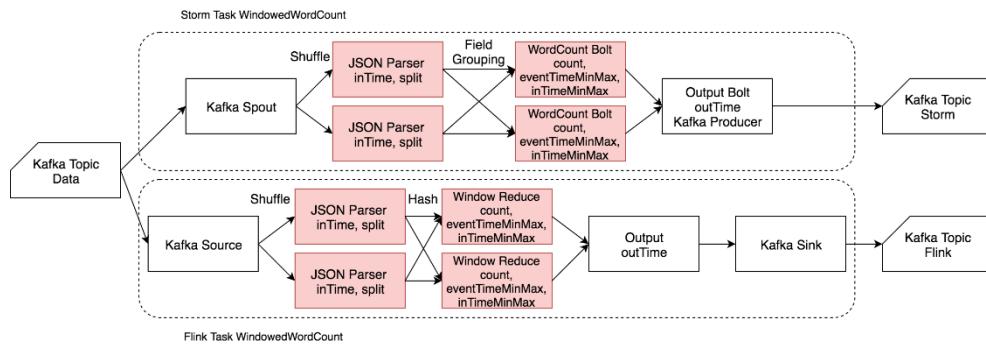
## Sleep

- Sleep 用例主要模拟用户作业耗时较长的场景，反映复杂用户逻辑对框架差异的削弱，比较两个框架的调度性能。
- 输入数据和输出数据均与 Identity 相同。
- 读入数据后，等待一定时长（1 ms）后在字符串末尾追加时间戳后输出



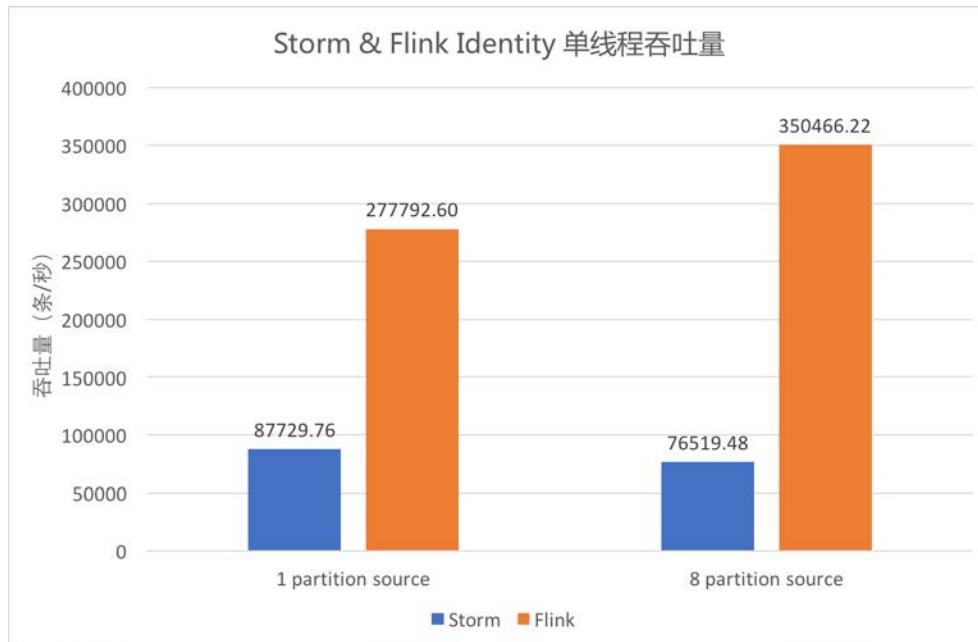
## Windowed Word Count

- Windowed Word Count 用例主要模拟窗口统计场景，反映两个框架在进行窗口统计时性能的差异。
- 此外，还用其进行了精确计算场景的测试，反映 Flink 恰好一次投递的性能。
- 输入为 JSON 格式，包含 msgId、eventTime 和一个由若干单词组成的句子，单词之间由空格分隔。单条输入数据约 150 B。
- 读入数据后解析 JSON，然后将句子分割为相应单词，带 eventTime 和 inTime 时间戳发给 CountWindow 进行单词计数，同时记录一个窗口中最大最小的 eventTime 和 inTime，最后带 outTime 时间戳输出到 Kafka 相应的 Topic。
- Spout/Source 及 OutputBolt/Output/Sink 并发度恒为 1，增大并发度时仅增大 JSONParser、CountWindow 的并发度。
- 由于 Storm 对 window 的支持较弱，CountWindow 使用一个 HashMap 手动实现，Flink 用了原生的 CountWindow 和相应的 Reduce 函数。



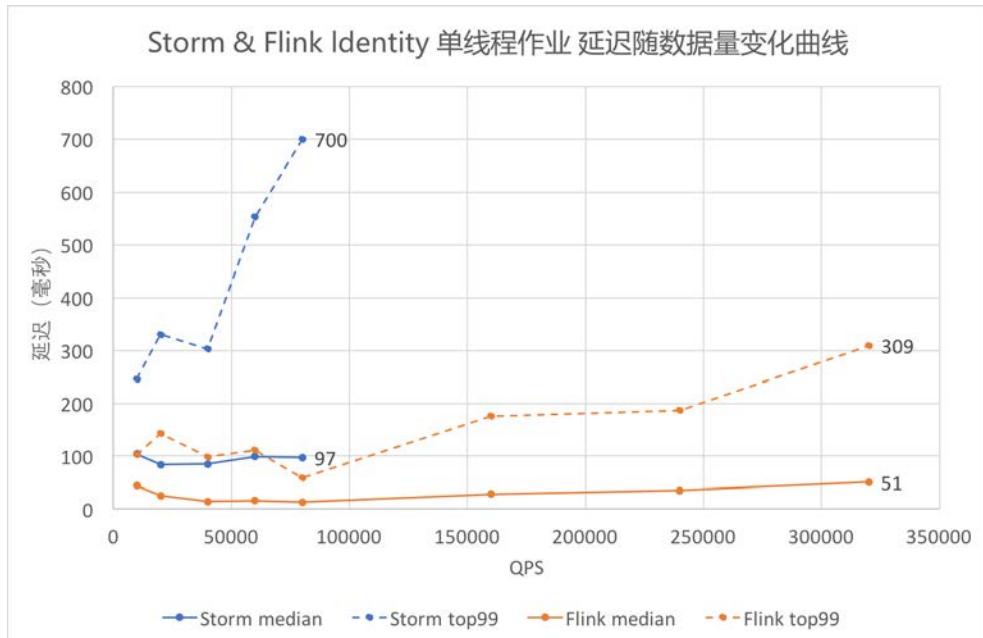
## 5. 测试结果

### 5.1 Identity 单线程吞吐量



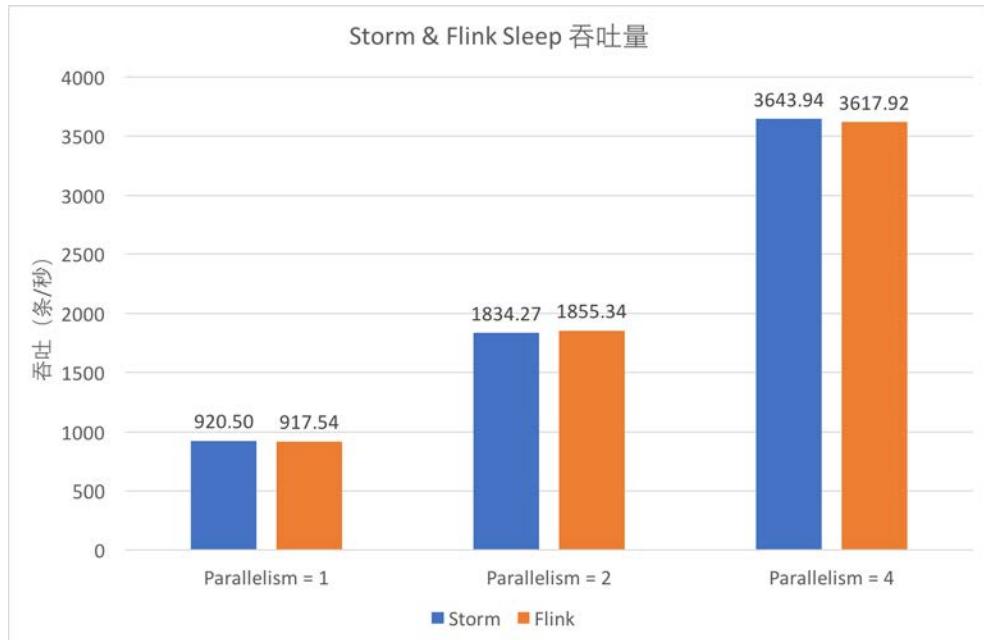
- 上图中蓝色柱形为单线程 Storm 作业的吞吐，橙色柱形为单线程 Flink 作业的吞吐。
- Identity 逻辑下，Storm 单线程吞吐为 **8.7** 万条/秒，Flink 单线程吞吐可达 **35** 万条/秒。
- 当 Kafka Data 的 Partition 数为 1 时，Flink 的吞吐约为 Storm 的 3.2 倍；当其 Partition 数为 8 时，Flink 的吞吐约为 Storm 的 4.6 倍。
- 由此可以看出，**Flink** 吞吐约为 **Storm** 的 **3-5** 倍。

## 5.2 Identity 单线程作业延迟



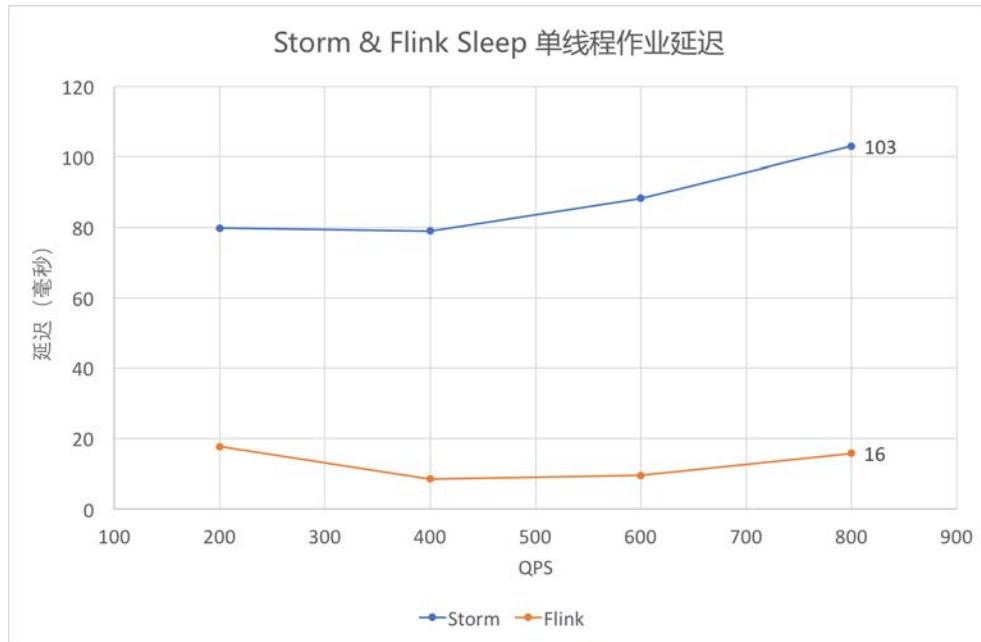
- 采用 `outTime - eventTime` 作为延迟，图中蓝色折线为 Storm，橙色折线为 Flink。虚线为 99 线，实线为中位数。
- 从图中可以看出随着数据量逐渐增大，Identity 的延迟逐渐增大。其中 99 线的增大速度比中位数快，Storm 的 增大速度比 Flink 快。
- 其中 QPS 在 80000 以上的测试数据超过了 Storm 单线程的吞吐能力，无法对 Storm 进行测试，只有 Flink 的曲线。
- 对比折线最右端的数据可以看出，Storm QPS 接近吞吐时延迟中位数约 100 毫秒，99 线约 700 毫秒，Flink 中位数约 50 毫秒，99 线约 300 毫秒。Flink 在满吞吐时的延迟约为 Storm 的一半。

### 5.3 Sleep 吞吐量



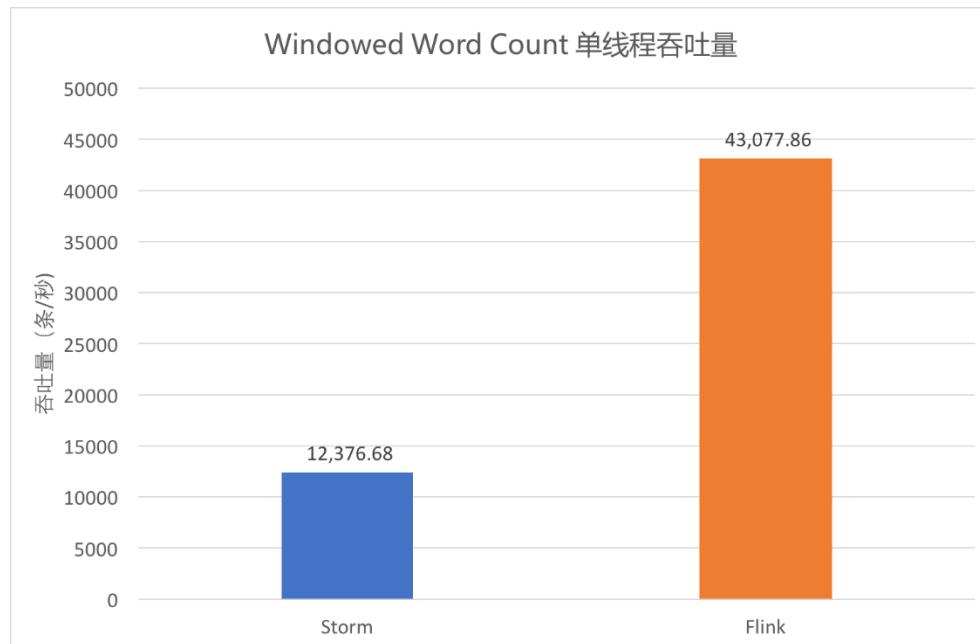
- 从图中可以看出，Sleep 1 毫秒时，Storm 和 Flink 单线程的吞吐均在 900 条/秒左右，且随着并发增大基本呈线性增大。
- 对比蓝色和橙色的柱形可以发现，此时两个框架的吞吐能力基本一致。

## 5.4 Sleep 单线程作业延迟 ( 中位数 )



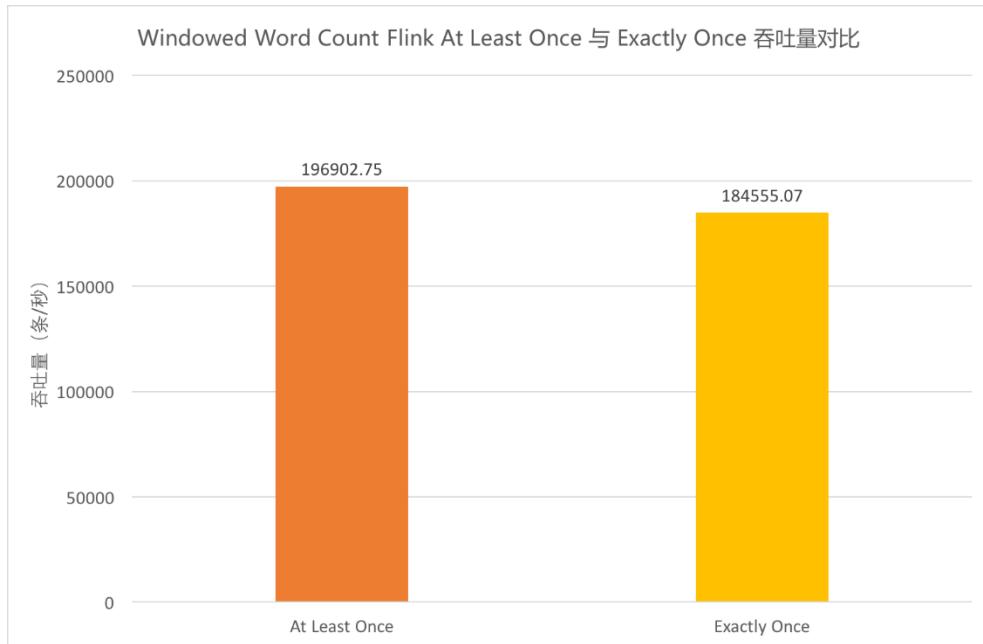
- 依然采用 `outTime - eventTime` 作为延迟，从图中可以看出，Sleep 1 毫秒时，Flink 的延迟仍低于 Storm。

## 5.5 Windowed Word Count 单线程吞吐量



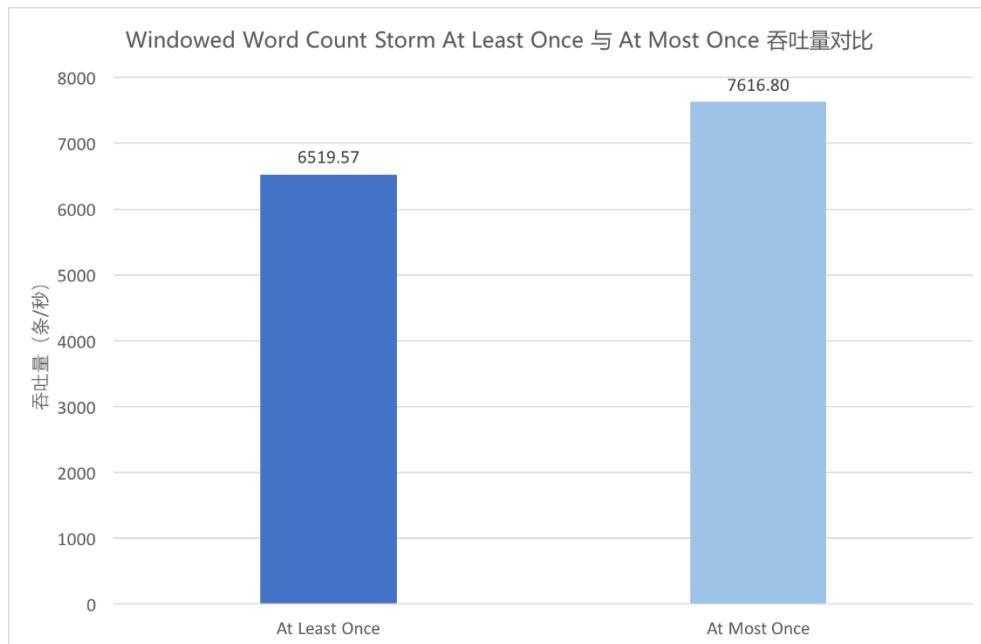
- 单线程执行大小为 10 的计数窗口，吞吐量统计如图。
- 从图中可以看出，Storm 吞吐约为 1.2 万条/秒，Flink Standalone 约为 4.3 万条/秒。Flink 吞吐依然为 **Storm** 的 **3** 倍以上。

## 5.6 Windowed Word Count Flink At Least Once 与 Exactly Once 吞吐量对比



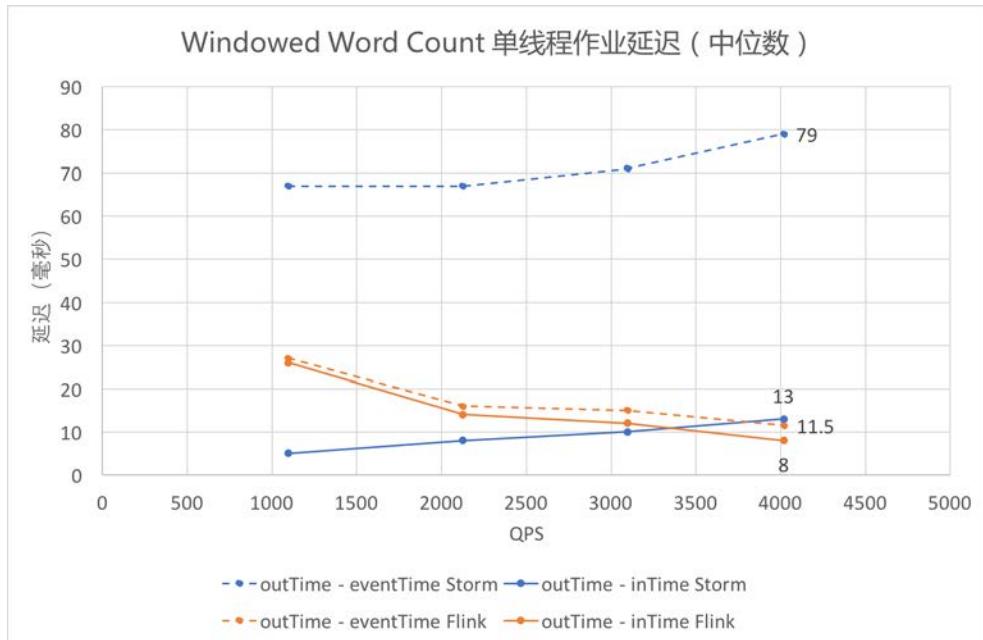
- 由于同一算子的多个并行任务处理速度可能不同，在上游算子中不同快照里的内容，经过中间并行算子的处理，到达下游算子时可能被计入同一个快照中。这样一来，这部分数据会被重复处理。因此，Flink 在 Exactly Once 语义下需要进行对齐，即当前最早的快照中所有数据处理完之前，属于下一个快照的数据不进行处理，而是在缓存区等待。当前测试用例中，在 JSON Parser 和 CountWindow、CountWindow 和 Output 之间均需要进行对齐，有一定消耗。为体现出对齐场景，Source/Output/Sink 并发度的并发度仍为 1，提高了 JSONParser/CountWindow 的并发度。具体流程细节参见前文 Windowed Word Count 流程图。
- 上图中橙色柱形为 At Least Once 的吞吐量，黄色柱形为 Exactly Once 的吞吐量。对比两者可以看出，在当前并发条件下，Exactly Once 的吞吐较 At Least Once 而言下降了 6.3%

## 5.7 Windowed Word Count Storm At Least Once 与 At Most Once 吞吐量对比



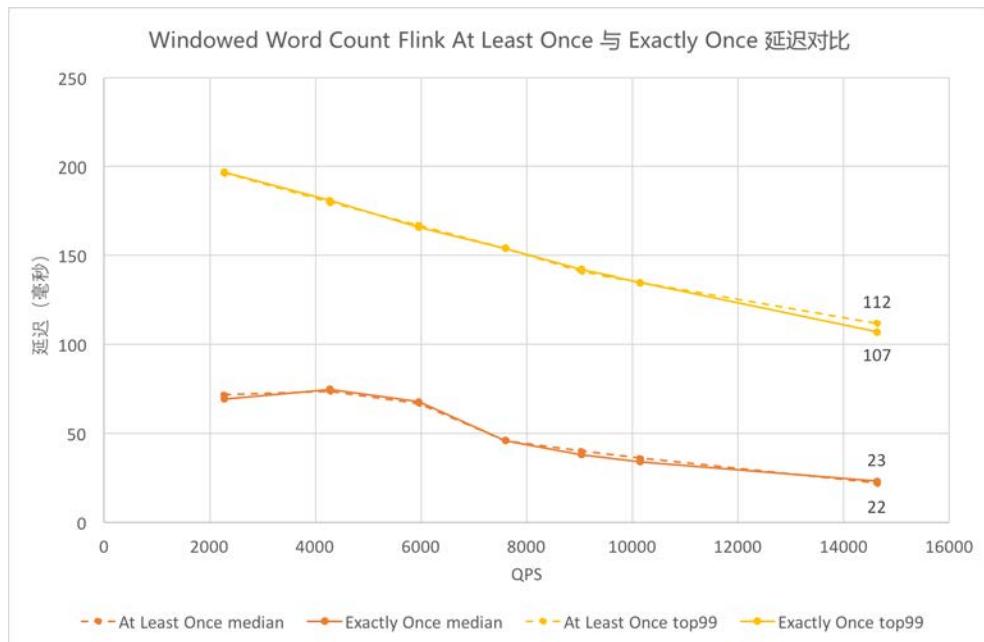
- Storm 将 ACKer 数量设置为零后, 每条消息在发送时就自动 ACK, 不再等待 Bolt 的 ACK, 也不再重发消息, 为 At Most Once 语义。
- 上图中蓝色柱形为 At Least Once 的吞吐量, 浅蓝色柱形为 At Most Once 的吞吐量。对比两者可以看出, 在当前并发条件下, At Most Once 语义下的吞吐较 At Least Once 而言提高了 **16.8%**

## 5.8 Windowed Word Count 单线程作业延迟



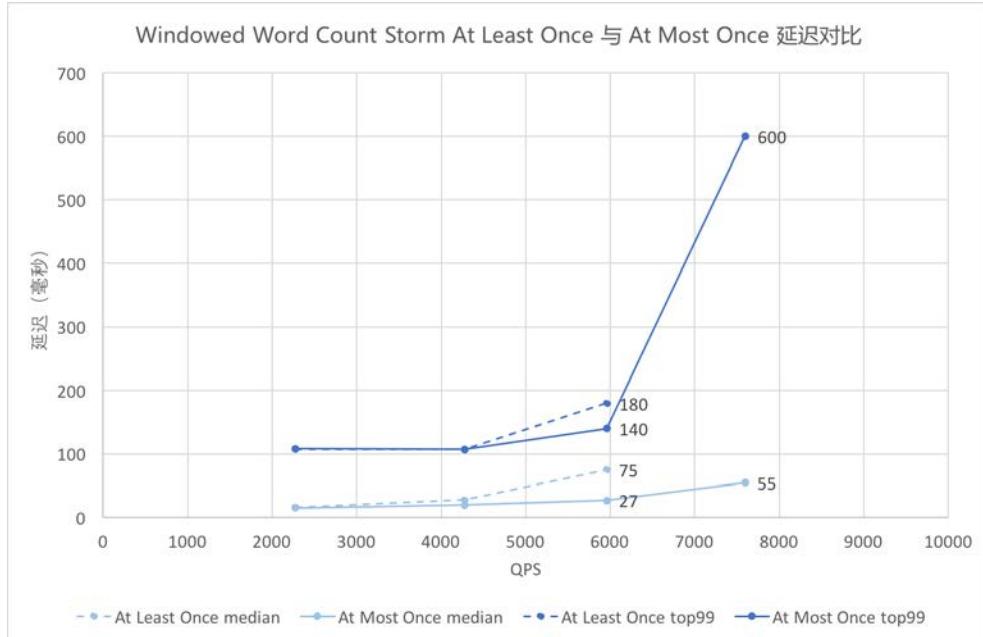
- Identity 和 Sleep 观测的都是 outTime - eventTime, 因为作业处理时间较短或 Thread.sleep() 精度不高, outTime - inTime 为零或没有比较意义; Windowed Word Count 中可以有效测得 outTime - inTime 的数值, 将其与 outTime - eventTime 画在同一张图上, 其中 outTime - eventTime 为虚线, outTime - inTime 为实线。
- 观察橙色的两条折线可以发现, Flink 用两种方式统计的延迟都维持在较低水平; 观察两条蓝色的曲线可以发现, Storm 的 outTime - inTime 较低, outTime - eventTime 一直较高, 即 inTime 和 eventTime 之间的差值一直较大, 可能与 Storm 和 Flink 的数据读入方式有关。
- 蓝色折线表明 Storm 的延迟随数据量的增大而增大, 而橙色折线表明 Flink 的延迟随着数据量的增大而减小 (此处未测至 Flink 吞吐量, 接近吞吐时 Flink 延迟依然会上升)。
- 即使仅关注 outTime - inTime (即图中实线部分), 依然可以发现, 当 QPS 逐渐增大的时候, Flink 在延迟上的优势开始体现出来。

## 5.9 Windowed Word Count Flink At Least Once 与 Exactly Once 延迟对比



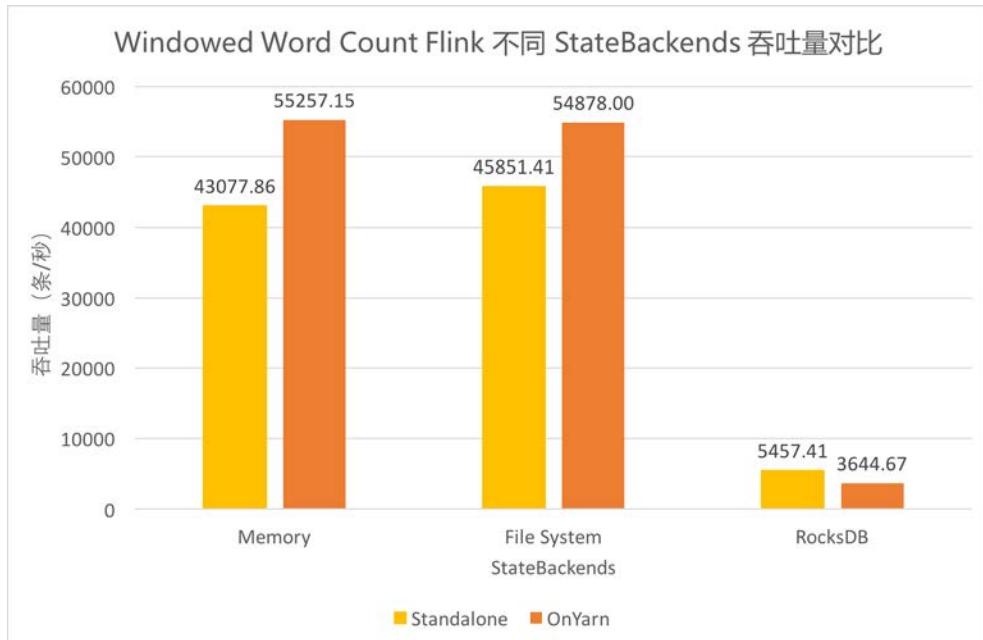
- 图中黄色为 99 线，橙色为中位数，虚线为 At Least Once，实线为 Exactly Once。图中相应颜色的虚实曲线都基本重合，可以看出 Flink Exactly Once 的延迟中位数曲线与 At Least Once 基本贴合，在延迟上性能没有太大差异。

## 5.10 Windowed Word Count Storm At Least Once 与 At Most Once 延迟对比



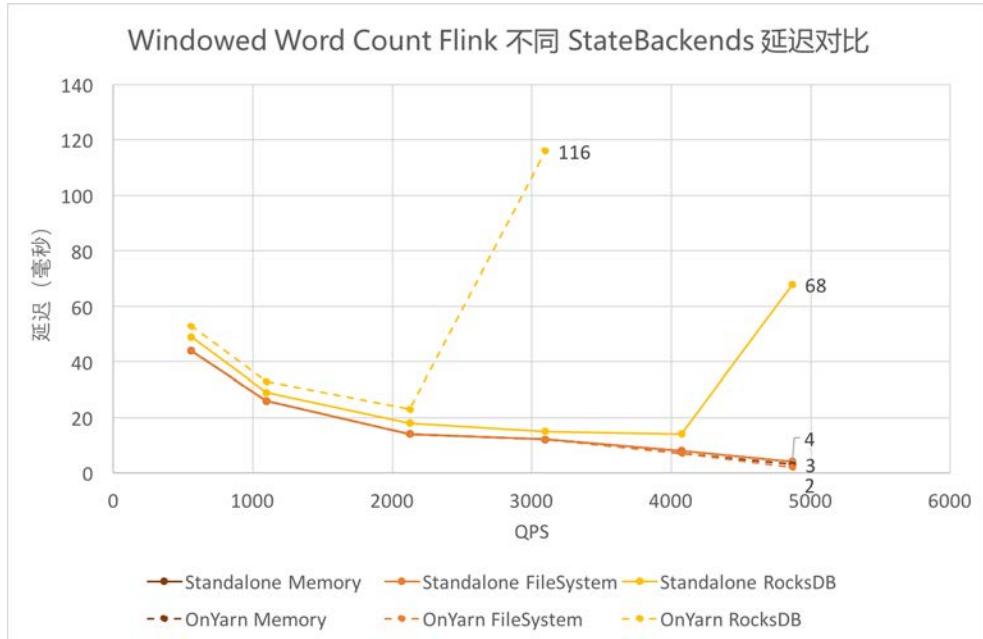
- 图中蓝色为 99 线，浅蓝色为中位数，虚线为 At Least Once，实线为 At Most Once。QPS 在 4000 及以前的时候，虚线实线基本重合；QPS 在 6000 时两者已有差异，虚线略高；QPS 接近 8000 时，已超过 At Least Once 语义下 Storm 的吞吐，因此只有实线上的点。
- 可以看出，QPS 较低时 Storm At Most Once 与 At Least Once 的延迟观察不到差异，随着 QPS 增大差异开始增大，At Most Once 的延迟较低。

## 5.11 Windowed Word Count Flink 不同 StateBackends 吞吐量对比



- Flink 支持 Standalone 和 on Yarn 的集群部署模式，同时支持 Memory、FileSystem、RocksDB 三种状态存储后端（StateBackends）。由于线上作业需要，测试了这三种 StateBackends 在两种集群部署模式上的性能差异。其中，Standalone 时的存储路径为 JobManager 上的一个文件目录，on Yarn 时存储路径为 HDFS 上一个文件目录。
- 对比三组柱形可以发现，使用 **FileSystem** 和 **Memory** 的吞吐差异不大，使用 **RocksDB** 的吞吐仅其余两者的十分之一左右。
- 对比两种颜色可以发现，**Standalone** 和 **on Yarn** 的总体差异不大，使用 **FileSystem** 和 **Memory** 时 **on Yarn** 模式下吞吐稍高，使用 **RocksDB** 时 **Standalone** 模式下的吞吐稍高。

## 5.12 Windowed Word Count Flink 不同 StateBackends 延迟对比



- 使用 FileSystem 和 Memory 作为 Backends 时，延迟基本一致且较低。
- 使用 RocksDB 作为 Backends 时，延迟稍高，且由于吞吐较低，在达到吞吐瓶颈前的延迟陡增。其中 on Yarn 模式下吞吐更低，接近吞吐时的延迟更高。

## 6. 结论及建议

### 6.1 框架本身性能

- 由 5.1、5.5 的测试结果可以看出，Storm 单线程吞吐约为 8.7 万条/秒，Flink 单线程吞吐可达 35 万条/秒。Flink 吞吐约为 Storm 的 3-5 倍。
- 由 5.2、5.8 的测试结果可以看出，Storm QPS 接近吞吐时延迟（含 Kafka 读写时间）中位数约 100 毫秒，99 线约 700 毫秒，Flink 中位数约 50 毫秒，99 线约 300 毫秒。Flink 在满吞吐时的延迟约为 Storm 的一半，且随着 QPS 逐渐增大，Flink 在延迟上的优势开始体现出来。
- 综上可得，Flink 框架本身性能优于 Storm。

## 6.2 复杂用户逻辑对框架差异的削弱

- 对比 5.1 和 5.3、5.2 和 5.4 的测试结果可以发现，单个 Bolt Sleep 时长达到 1 毫秒时，Flink 的延迟仍低于 Storm，但吞吐优势已基本无法体现。
- 因此，用户逻辑越复杂，本身耗时越长，针对该逻辑的测试体现出来的框架的差异越小。

## 6.3 不同消息投递语义的差异

- 由 5.6、5.7、5.9、5.10 的测试结果可以看出，Flink Exactly Once 的吞吐较 At Least Once 而言下降 6.3%，延迟差异不大；Storm At Most Once 语义下的吞吐较 At Least Once 提升 16.8%，延迟稍有下降。
- 由于 Storm 会对每条消息进行 ACK，Flink 是基于一批消息做的检查点，不同的实现原理导致两者在 At Least Once 语义的花费差异较大，从而影响了性能。而 Flink 实现 Exactly Once 语义仅增加了对齐操作，因此在算子并发量不大、没有出现慢节点的情况下对 Flink 性能的影响不大。Storm At Most Once 语义下的性能仍然低于 Flink。

## 6.4 Flink 状态存储后端选择

- Flink 提供了内存、文件系统、RocksDB 三种 StateBackends，结合 5.11、5.12 的测试结果，三者的对比如下：

StateBackend	过程状态存储	检查点存储	吞吐	推荐使用场景
Memory	TM Memory	JM Memory	高（3-5 倍 Storm）	调试、无状态或对数据是否丢失重复无要求
FileSystem	TM Memory	FS/HDFS	高（3-5 倍 Storm）	普通状态、窗口、KV 结构 (建议作为默认 Backend)
RocksDB	RocksDB on TM	FS/HDFS	低（0.3-0.5 倍 Storm）	超大状态、超长窗口、大型 KV 结构

## 6.5 推荐使用 Flink 的场景

综合上述测试结果，以下实时计算场景建议考虑使用 Flink 框架进行计算：

- 要求消息投递语义为 **Exactly Once** 的场景；
- 数据量较大，要求高吞吐低延迟的场景；
- 需要进行状态管理或窗口统计的场景。

## 7. 展望

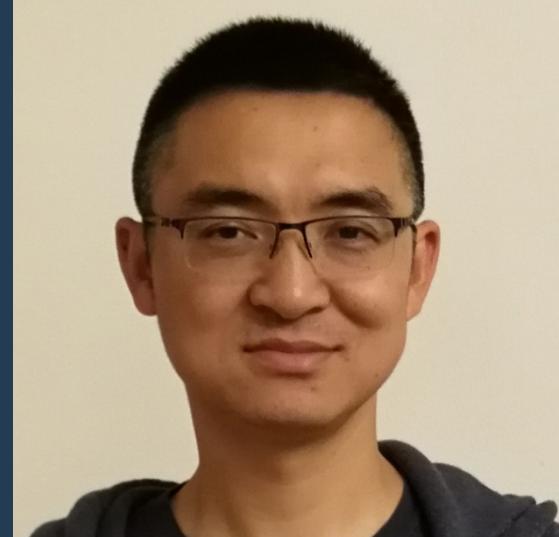
- 本次测试中尚有一些内容没有进行更加深入的测试，有待后续测试补充。例如：
  - Exactly Once 在并发量增大的时候是否吞吐会明显下降？
  - 用户耗时到 1ms 时框架的差异已经不再明显（`Thread.sleep()` 的精度只能到毫秒），用户耗时在什么范围内 Flink 的优势依然能体现出来？
- 本次测试仅观察了吞吐量和延迟两项指标，对于系统的可靠性、可扩展性等重要的性能指标没有在统计数据层面进行关注，有待后续补充。
- Flink 使用 RocksDBStateBackend 时的吞吐较低，有待进一步探索和优化。
- 关于 Flink 的更高级 API，如 Table API & SQL 及 CEP 等，需要进一步了解和完善。

## 8. 参考内容

- 分布式流处理框架——功能对比和性能评估.
- intel-hadoop/HiBench: HiBench is a big data benchmark suite.
- Yahoo 的流计算引擎基准测试.
- Extending the Yahoo! Streaming Benchmark.

# Spark VS Flink – 下一代大数据 计算引擎之争，谁主沉浮？

作者 王海涛



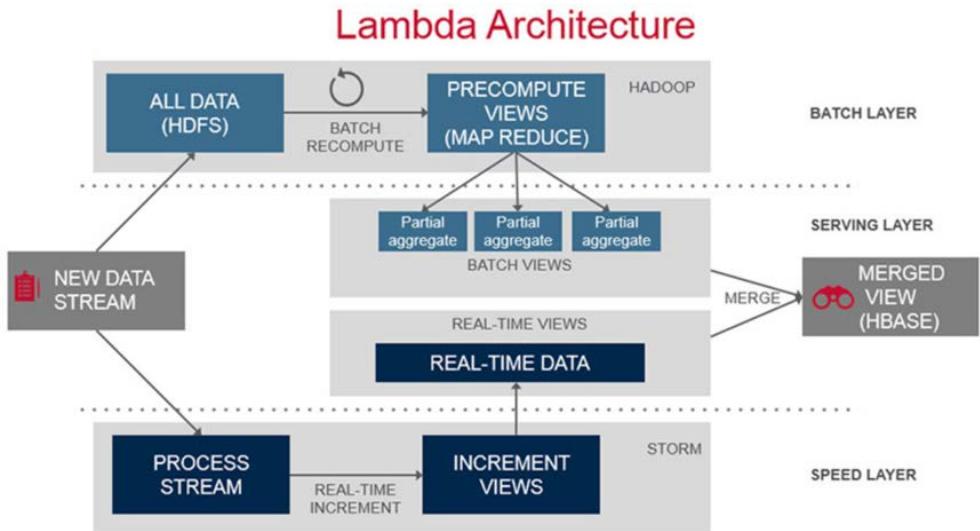
简介：本文主要整理自作者 2018 年 7 月 23 日发表在公众号“AI 前线”的文章

## 下一代大数据计算引擎

自从数据处理需求超过了传统数据库能有效处理的数据量之后，Hadoop 等各种基于 MapReduce 的海量数据处理系统应运而生。从 2004 年 Google 发表 MapReduce 论文开始，经过近 10 年的发展，基于 Hadoop 开源生态或者其它相应系统的海量数据处理已经成为业界的基本需求。

但是，很多机构在开发自己的数据处理系统时都会发现需要面临一系列的问题。从数据中获取价值需要的投入远远超过预期。常见的问题包括：

- 非常陡峭的学习曲线。刚接触这个领域的人经常会被需要学习的技术的数量砸晕。不像经过几十年发展的数据库一个系统可以解决大部分数据处理需求，Hadoop 等大数据生态里的一个系统往往在一些数据处理场景上比较擅长，另一些场景凑合能用，还有一些场景完全无法满足需求。结果就是需要好几个系统来处理不同的场景。



(来源: <https://mapr.com/developercentral/lambda-architecture/>)

上图是一个典型的 lambda 架构，只是包含了批处理和流处理两种场景，就已经牵涉到至少四五种技术了，还不算每种技术的可替代选择。再加上实时查询，交互式分析，机器学习等场景，每个场景都有几种技术可以选择，每个技术涵盖的领域还有不同方式的重叠。结果就是一个业务经常需要使用四五种以上的技术才能支持好一个完整的数据处理流程。加上调研选型，需要了解的数目还要多得多。

下图是大数据领域的全景。有没有晕？



大数据和 AI 全景 - 2018 (来源: <http://mattturck.com/bigdata2018/>)

- 开发和运行效率低下。因为牵涉到多种系统，每种系统有自己的开发语言和工具，开发效率可想而知。而因为采用了多套系统，数据需要在各个系统之间传输，也造成了额外的开发和运行代价，数据的一致也难以保证。在很多机构，实际上一半以上的开发精力花在了数据在各个系统之间的传输上。
  - 复杂的运维。多个系统，每个需要自己的运维，带来更高的运维代价的同时也提高了系统出问题的可能。
  - 数据质量难以保证。数据出了问题难以跟踪解决。
  - 最后，还有人的问题。在很多机构，由于系统的复杂性，各个子系统的支持和使用落实在不同部门负责。

了解了这些问题以后，对 Spark 从 2014 年左右开始迅速流行就比较容易理解了。Spark 在当时除了在某些场景比 Hadoop MapReduce 带来几十到上百倍的性能提升外，还提出了用一个统一的引擎支持批处理，流处理，交互式查询，机器学习等常见的数据处理场景。看过在一个 Notebook 里完成上述所有场景的 Spark 演示后，对比之前的数据流程开发，对很多开发者来说不难做出选择。经过几年的发展，Spark 已经被视为可以完全取代 Hadoop 中的 MapReduce。

引擎。

正在 Spark 如日中天高速发展的时候，2016 年左右 Apache Flink（以下简称 Flink）开始进入大众的视野并逐渐广为人知。为什么呢？原来在人们开始使用 Spark 之后，发现 Spark 虽然支持各种常见场景，但并不是每一种都同样好用。数据流的实时处理就是其中相对较弱的一环。Flink 凭借更优的流处理引擎，同时也支持各种处理场景，成为 Spark 的有力挑战者。

Spark 和 Flink 是怎么做到这些的，它们之间又有那些异同，下面我们来具体看一下。

## Spark 和 Flink 的引擎技术

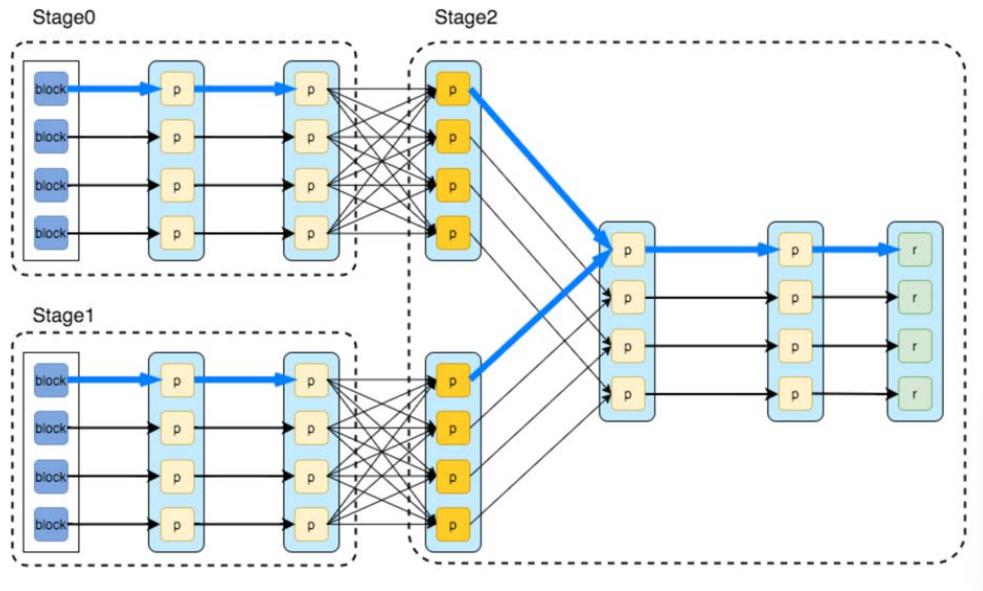
这一部分主要着眼于 Spark 和 Flink 引擎的架构方面，更看重架构带来的潜力和限制。

### 数据模型和处理模型

要理解 Spark 和 Flink 的引擎特点，首先从数据模型开始。

Spark 的数据模型是弹性分布式数据集 RDD（Resilient Distributed Datasets）。比起 MapReduce 的文件模型，RDD 是一个更抽象的模型，RDD 靠血缘（lineage）等方式来保证可恢复性。很多时候 RDD 可以实现为分布式共享内存或者完全虚拟化（即有的中间结果 RDD 当下游处理完全在本地时可以直接优化省略掉）。这样可以省掉很多不必要的 I/O，是早期 Spark 性能优势的主要原因。

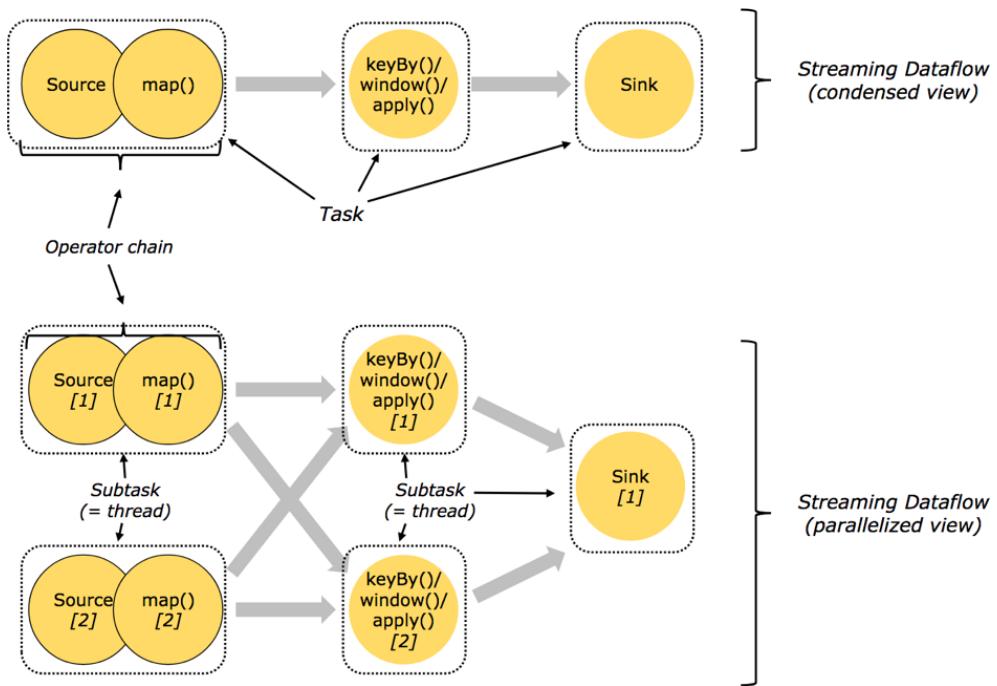
Spark 用 RDD 上的变换（算子）来描述数据处理。每个算子（如 map, filter, join）生成一个新的 RDD。所有的算子组成一个有向无环图（DAG）。Spark 比较简单地把边分为宽依赖和窄依赖。上下游数据不需要 shuffle 的即为窄依赖，可以把上下游的算子放在一个阶段（stage）里在本地连续处理，这时上游的结果 RDD 可以省略。下图展示了相关的基本概念。更详细的介绍在网上比较容易找到，这里就不花太多篇幅了。



Spark DAG ( 来源: <http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/> )

Flink 的基本数据模型是数据流，及事件（Event）的序列。数据流作为数据的基本模型可能没有表或者数据块直观熟悉，但是可以证明是完全等效的。流可以是无边界的无限流，即一般意义上的流处理。也可以是有边界的有限流，这样就是批处理。

Flink 用数据流上的变换（算子）来描述数据处理。每个算子生成一个新的数据流。在算子，DAG，和上下游算子链接（chaining）这些方面，和 Spark 大致等价。Flink 的节点（vertex）大致相当于 Spark 的阶段（stage），划分也会和上图的 Spark DAG 基本一样。



Flink 任务图（来源：<https://ci.apache.org/projects/flink/flink-docs-release-1.5/concepts/runtime.html>）

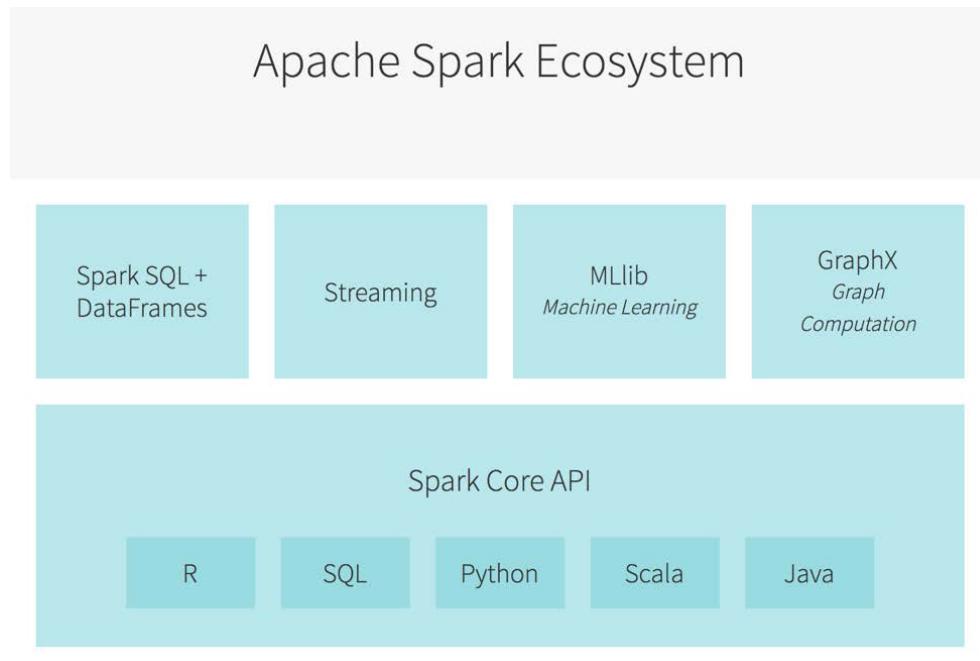
在 DAG 的执行上，Spark 和 Flink 有一个比较显著的区别。在 Flink 的流执行模式中，一个事件在一个节点处理完后的输出就可以发到下一个节点立即处理。这样执行引擎并不会引入额外的延迟。与之相应的，所有节点是需要同时运行的。而 Spark 的 micro batch 和一般的 batch 执行一样，处理完上游的 stage 得到输出之后才开始下游的 stage。

在 Flink 的流执行模式中，为了提高效率也可以把多个事件放在一起传输或者计算。但这完全是执行时的优化，可以在每个算子独立决定，也不用像 RDD 等批处理模型中一样和数据集边界绑定，可以做更加灵活的优化同时可以兼顾低延迟需求。

Flink 使用异步的 checkpoint 机制来达到任务状态的可恢复性，以保证处理的一致性，所以在处理的主流程上可以做到数据源和输出之间数据完全不用落盘，达到更高的性能和更低的延迟。

## 数据处理场景

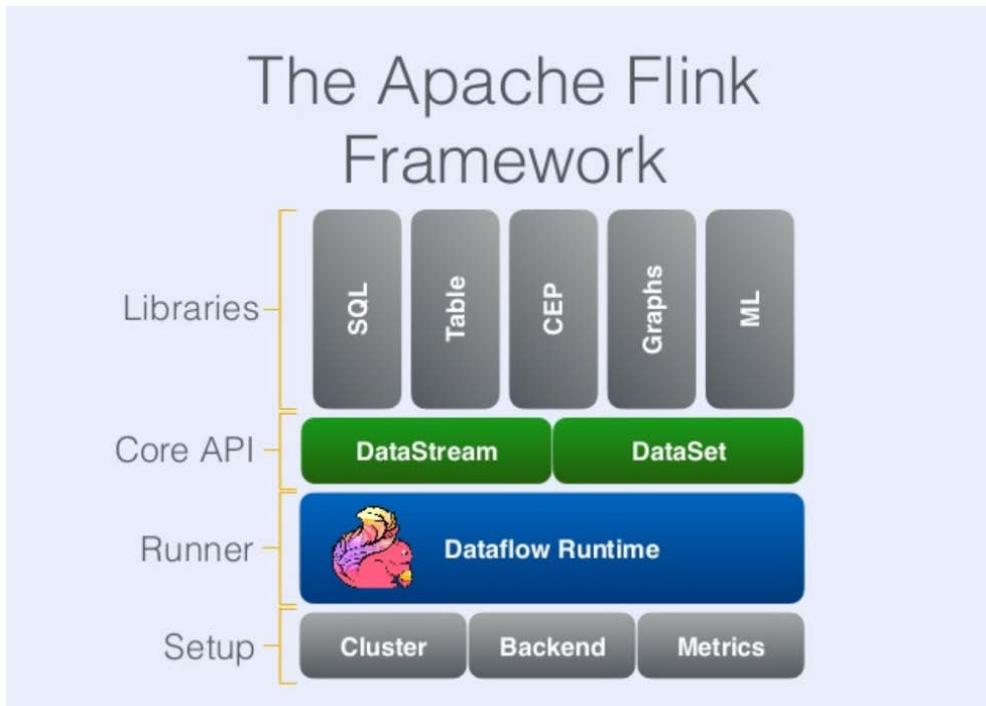
除了批处理之外，Spark 还支持实时数据流处理，交互式查询，和机器学习，图计算等。



(来源：<https://databricks.com/spark/about> )

- 实时数据流处理和批处理主要区别就是对低延时的要求。Spark 因为 RDD 是基于内存的，可以比较容易切成较小的块来处理。如果能对这些小块处理得足够快，就能达到低延时的效果。
- 交互式查询场景，如果数据能全在内存，处理得足够快的话，就可以支持交互式查询。
- 机器学习和图计算其实是和前几种场景不同的 RDD 算子类型。Spark 提供了库来支持常用的操作，用户或者第三方库也可以自己扩展。值得一提的是，Spark 的 RDD 模型和机器学习模型训练的迭代计算非常契合，从一开始就在有的场景带来了非常显著的性能提升。

从这些可以看出来，比起 Hadoop MapReduce，Spark 本质上就是基于内存的更快的批处理。然后用足够快的批处理来实现各种场景。



(来源: <https://www.slideshare.net/ParisCarbone/state-management-in-apache-flink-consistent-stateful-distributed-stream-processing>)

前面说过，在 Flink 中，如果输入数据流是有边界的，就自然达到了批处理的效果。这样流和批的区别完全是逻辑上的，和处理实现独立，用户需要实现的逻辑也完全一样，应该是更干净的一种抽象。

Flink 也提供了库来支持机器学习，图计算等场景。从这方面来说和 Spark 没有太大区别。

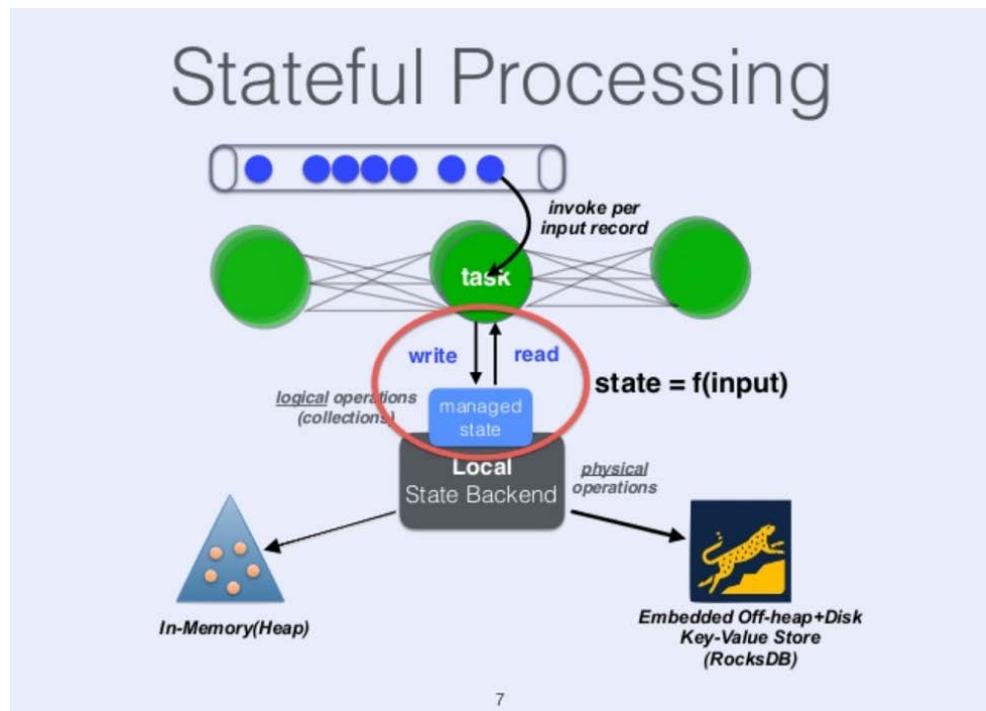
一个有意思的事情是用 Flink 的底层 API 可以支持只用 Flink 集群实现一些数据驱动的分布式服务。有一些公司用 Flink 集群实现了社交网络，网络爬虫等服务。这个也体现了 Flink 作为计算引擎的通用性，并得益于 Flink 内置的灵活的状态支持。

总的来说，Spark 和 Flink 都瞄准了一个执行引擎上同时支持大多数数据处理场景，也应该都能做到这一点。主要区别就在于因为架构本身的局限在一些场景会受到限制。比较突出的地方就是 Spark Streaming 的 micro batch 执行模式。Spark 社区应该也意识到了这一点，最近在

持续执行模式 (continuous processing) 方面开始发力。具体情况会在后面介绍。

## 有状态处理 (Stateful Processing)

Flink 还有一个非常独特的地方是在引擎中引入了托管状态 (managed state)。要理解托管状态，首先要从有状态处理说起。如果处理一个事件 (或一条数据) 的结果只跟事件本身的内容有关，称为无状态处理；反之结果还和之前处理过的事件有关，称为有状态处理。稍微复杂一点的数据处理，比如说基本的聚合，都是有状态处理。Flink 很早就认为没有好的状态支持是做不好留处理的，因此引入了 managed state 并提供了 API 接口。



Flink 中的状态支持 (来源: <https://www.slideshare.net/ParisCarbone/state-management-in-apache-flink-consistent-stateful-distributed-stream-processing>)

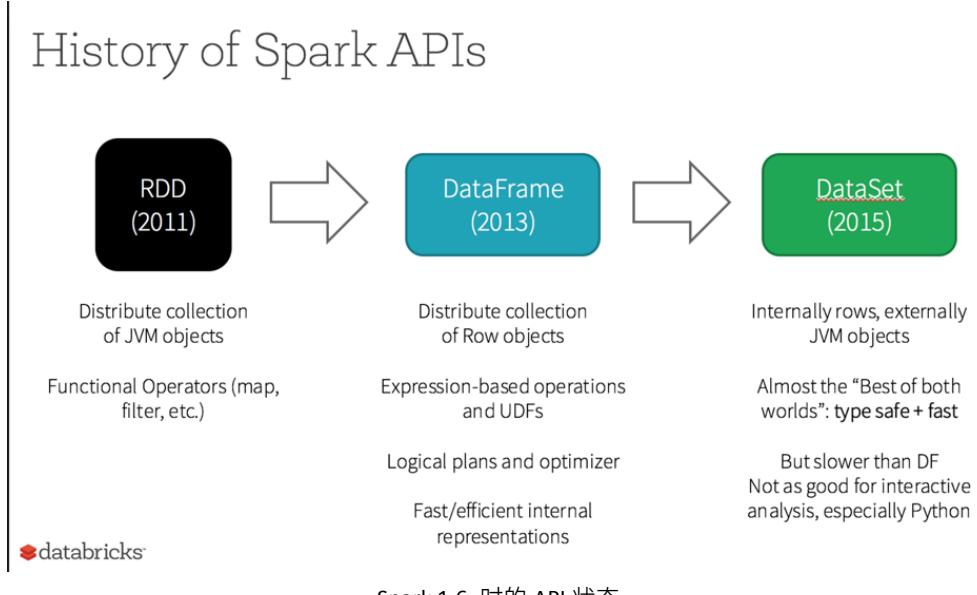
一般在流处理的时候会比较关注有状态处理，但是仔细看的话批处理也是会受到影响的。比如

常见的窗口聚合，如果批处理的数据时间段比窗口大，是可以不考虑状态的，用户逻辑经常会忽略这个问题。但是当批处理时间段变得比窗口小或相当的时候，一个批的结果实际上依赖于以前处理过的批。这时，因为批处理引擎一般没有这个需求不会有很好的内置支持，维护状态就成为了用户需要解决的事情。比如窗口聚合的情况用户就要加一个中间结果表记住还没有完成的窗口的结果。这样当用户把批处理时间段变短的时候就会发现逻辑变复杂了。这是早期 Spark Streaming 用户经常碰到的问题。直到 Structured Streaming 出来才得到缓解。

而像 Flink 这样以流处理为基本模型的引擎，因为一开始就避不开这个问题，所以引入了 managed state 来提供了一个通用的解决方案。比起用户实现的特定解决方案，不但用户开发更简单，而且能提供更好的性能。最重要的是能更好地保证处理结果的一致性。

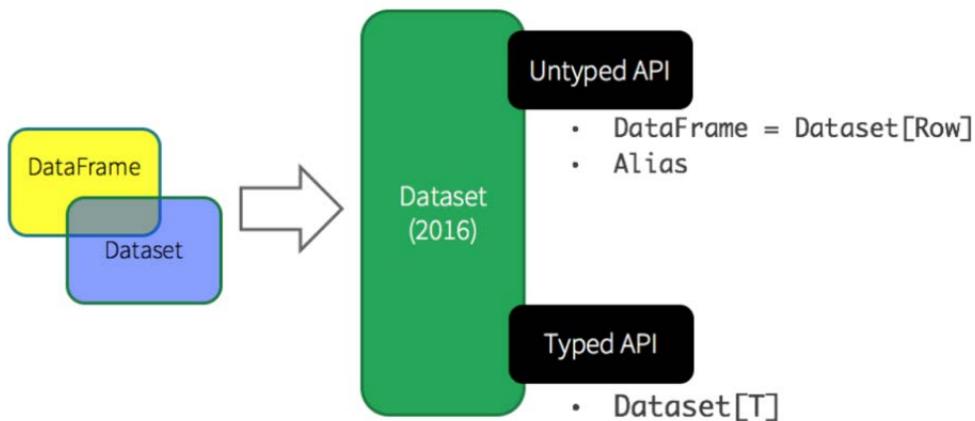
简单来说，就是有一些内秉的数据处理逻辑，在批处理中容易被忽略或简化处理掉也能得到可用的结果，而在流处理中问题被暴露出来解决掉了。所以流计算引擎用有限流来处理批在逻辑上比较严谨，能自然达到正确性。主要做一些不同的实现来优化性能就可以了。而用更小的批来模拟流需要处理一些以前没有的问题。当计算引擎还没有通用解决方案的时候就需要用户自己解决了。类似的问题还有维表的变化（比如用户信息的更新），批处理数据的边界和迟到数据等等。

## 编程模型



Spark 的初衷之一就是用统一的编程模型来解决用户的各种需求。在这方面一直很下功夫。最初基于 RDD 的 API 就可以做各种类型的数据处理。后来为了简化用户开发，逐渐推出了更高层的 DataFrame（在 RDD 中加了列变成结构化数据）和 Datasets（在 DataFrame 的列上加了类型），并在 Spark 2.0 中做了整合（`DataFrame = Dataset[Row]`）。Spark SQL 的支持也比较早就引入了。在加上各个处理类型 API 的不断改进，比如 Structured Streaming 以及和机器学习深度学习的交互，到了今天 Spark 的 API 可以说是非常好用的，也是 Spark 最强的方面之一。

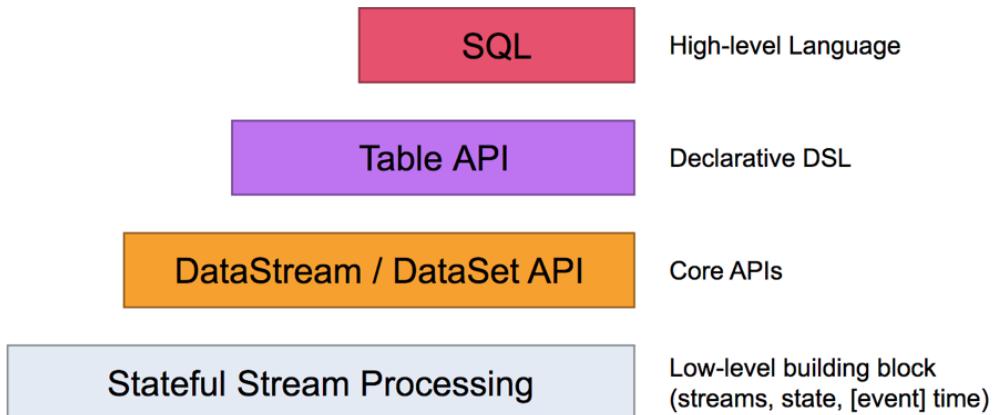
### Unified Apache Spark 2.0 API



◆ databricks

Spark 2.0 API （来源：<https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html>）

Flink 的 API 也有类似的目标和发展路线。Flink 和 Spark 的核心 API 可以说是可以基本对应的。今天 Spark API 总体上更完备一下，比如说最近一两年大力投入的和机器学习深度学习的整合方面。Flink 在流处理相关的方面还是领先一些，比如对 watermark, window, trigger 的各种支持。



Flink API (来源: <https://ci.apache.org/projects/flink/flink-docs-release-1.5/concepts/programming-model.html>)

## 小结

Spark 和 Flink 都是通用的能够支持超大规模数据处理，支持各种处理类型的计算引擎。两个系统都有很多值得探讨的方面在这里没有触及，比如 SQL 的优化，和机器学习的集成等等。这里主要是试图从最基本的架构和设计方面来比较一下两个系统。因为上层的功能在一定程度上是可以互相借鉴的，有足够的投入应该都能做好。而基本的设计改变起来会伤筋动骨，更困难一些。

Spark 和 Flink 的不同执行模型带来的最大的区别应该还是在对流计算的支持上。最开始的 Spark Streaming 对流计算想得过于简单，对复杂一点的计算用起来会有不少问题。从 Spark 2.0 开始引入的 Structured Streaming 重新整理了流计算的语义，支持按事件时间处理和端到端的一致性。虽然在功能上还有不少限制，比之前已经有了长足的进步。不过 micro batch 执行方式带来的问题还是存在，特别在规模上去以后性能问题会比较突出。最近 Spark 受一些应用场景的推动，也开始开发持续执行模式。2.3 里的实验性发布还只支持简单的 map 类的操作。从最近 Spark+AI Summit 大会上的介绍来看（下图），会发展成一个和 Flink 的流处理模式比较相似的执行引擎。不过从下图来看，主要的功能都还在开发中或者待开发。对将来能做到什么程度，和 Spark 原来的 batch 执行引擎怎么结合，我们拭目以待。



Continuous Processing in Structured Streami... - Databricks

## Ongoing And Future Work

- Shuffles (SPARK-24036)
- Event time (SPARK-24459)
- Metrics (SPARK-23887)
  
- Exactly-once semantics mode (SPARK-24460)
- Performance testing (TBD)
- Additional data sources (TBD)



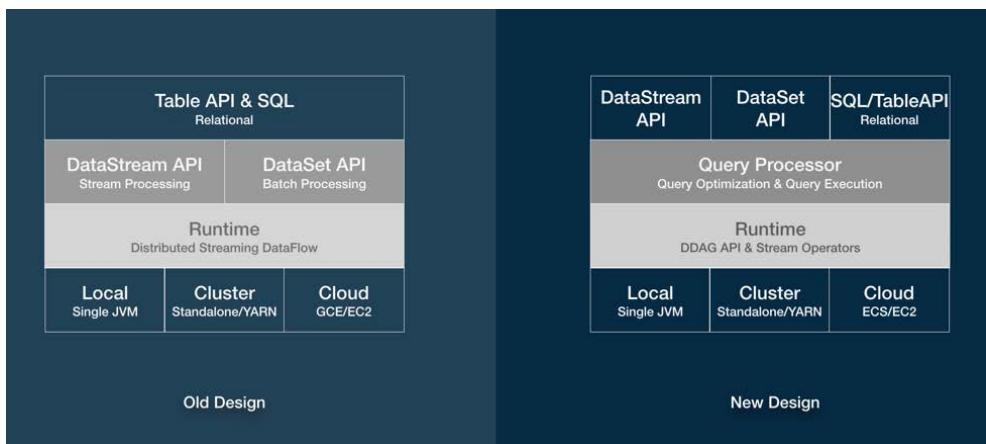
19

Spark 持续执行模式状态 (来源: <https://www.slideshare.net/databricks/continuous-processing-in-structured-streaming-with-jose-torres>)

## 近况与更新

从本文最初完成又过了近半年。总体来看感觉 Spark 在引擎上的新投入不是很积极。特别是在流处理方面，持续执行模式（Continuous Processing）在社区讨论中还没有看到一个好的设计可以扩展到支持通用的有状态处理，也没有看到如何解决大规模下 micro batch 模式自身带来的性能瓶颈。前文提到的几个关键性 JIRA 基本没有看到进展。

反观 Flink，阿里在 9 月的云栖大会上发布了不少批处理方面的进展。对流和批在算子层的统一可以进一步改进流和批处理的性能，同时表现了对引擎做持续深度改进的雄心。加上其它批处理的优化，在批处理的 benchmark 上也有不错的成绩。流处理方面 Flink 继续保持领先，业界有大规模流处理需求的大多采用了 Flink。如果社区和生态跟上的话，Flink 前景应该相当可观。



Flink 流批统一的新架构（来源：2018 云栖大会）

## 笔者简介

王海涛，曾经在微软的 SQL Server 和大数据平台组工作多年。带领团队建立了微软对内的 Spark 服务，主打 Spark Streaming。2017 年 3 月加入阿里实时计算部门，参与改进阿里基于 Apache Flink 的 Blink 平台。

# 5 分钟从零构建第一个 Apache Flink 应用

作者 伍翀



在本文中，我们将从零开始，教您如何构建第一个 Apache Flink（以下简称 Flink）应用程序。

## 开发环境准备

Flink 可以运行在 Linux, Max OS X, 或者是 Windows 上。为了开发 Flink 应用程序，在本地机器上需要有 **Java 8.x** 和 **maven** 环境。

如果有 Java 8 环境，运行下面的命令会输出如下版本信息：

```
bash
$ java -version
java version "1.8.0_65"
Java(TM) SE Runtime Environment (build 1.8.0_65-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)
```

如果有 maven 环境，运行下面的命令会输出如下版本信息：

```
bash
$ mvn -version
Apache Maven 3.5.4 (1eedded0938998edf8bf061f1ceb3cfdeccf443fe; 2018-06-
18T02:33:14+08:00)
Maven home: /Users/wuchong/dev/maven
Java version: 1.8.0_65, vendor: Oracle Corporation, runtime:
/Library/Java/JavaVirtualMachines/jdk1.8.0_65.jdk/Contents/Home/jre
```

```
Default locale: zh_CN, platform encoding: UTF-8
OS name: "mac os x", version: "10.13.6", arch: "x86_64", family: "mac"
```

另外我们推荐使用 IntelliJ IDEA（社区免费版已够用）作为 Flink 应用程序的开发 IDE。Eclipse 虽然也可以，但是 Eclipse 在 Scala 和 Java 混合型项目下会有些已知问题，所以不太推荐 Eclipse。下一章节，我们会介绍如何创建一个 Flink 工程并将其导入 IntelliJ IDEA。

## 创建 Maven 项目

我们将使用 Flink Maven Archetype 来创建我们的项目结构和一些初始的默认依赖。在你的工作目录下，运行如下命令来创建项目：

```
bash
mvn archetype:generate \
-DarchetypeGroupId=org.apache.flink \
-DarchetypeArtifactId=flink-quickstart-java \
-DarchetypeVersion=1.6.1 \
-DgroupId=my-flink-project \
-DartifactId=my-flink-project \
-Dversion=0.1 \
-Dpackage=myflink \
-DinteractiveMode=false
```

你可以编辑上面的 groupId, artifactId, package 成你喜欢的路径。使用上面的参数，Maven 将自动为你创建如下所示的项目结构：

```
bash
$ tree my-flink-project
my-flink-project
├── pom.xml
└── src
    ├── main
    │   ├── java
    │   │   └── myflink
    │   │       ├── BatchJob.java
    │   │       └── StreamingJob.java
    │   └── resources
    │       └── log4j.properties
```

我们的 pom.xml 文件已经包含了所需的 Flink 依赖，并且在 src/main/java 下有几个示例程序框架。接下来我们将开始编写第一个 Flink 程序。

## 编写 Flink 程序

启动 IntelliJ IDEA, 选择 "Import Project"(导入项目), 选择 my-flink-project 根目录下的 pom.xml。根据引导, 完成项目导入。

在 src/main/java/myflink 下创建 SocketWindowWordCount.java 文件:

```
java
package myflink;
public class SocketWindowWordCount {
    public static void main(String[] args) throws Exception {
    }
}
```

现在这程序还很基础, 我们会一步步往里面填代码。注意下文中我们不会将 import 语句也写出来, 因为 IDE 会自动将他们添加上去。在本节末尾, 我会将完整的代码展示出来, 如果你想跳过下面的步骤, 可以直接将最后的完整代码粘到编辑器中。

Flink 程序的第一步是创建一个 StreamExecutionEnvironment。这是一个入口类, 可以用来设置参数和创建数据源以及提交任务。所以让我们把它添加到 main 函数中:

```
java
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
```

下一步我们将创建一个从本地端口号 9000 的 socket 中读取数据的数据源:

```
java
DataStream<String> text = env.socketTextStream("localhost", 9000, "\n");
```

这创建了一个字符串类型的 DataStream。DataStream 是 Flink 中做流处理的核心 API, 上面定义了非常多常见的操作(如, 过滤、转换、聚合、窗口、关联等)。在本示例中, 我们感兴趣的是每个单词在特定时间窗口中出现的次数, 比如说 5 秒窗口。为此, 我们首先要将字符串数据解析成单词和次数(使用 Tuple2<String, Integer>表示), 第一个字段是单词, 第二个字段是次数, 次数初始值都设置成了 1。我们实现了一个 flatmap 来做解析的工作, 因为一行数据中可能有多个单词。

```
java
DataStream<Tuple2<String, Integer>> wordCounts = text
    .flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {
        @Override
        public void flatMap(String value, Collector<Tuple2<String,
Integer>> out) {
```

```

        for (String word : value.split("\s")) {
            out.collect(Tuple2.of(word, 1));
        }
    });
}
});
```

接着我们将数据流按照单词字段（即 0 号索引字段）做分组，这里可以简单地使用 keyBy(int index)方法，得到一个以单词为 key 的 Tuple2<String, Integer>数据流。然后我们可以在流上指定想要的窗口，并根据窗口中的数据计算结果。在我们的例子中，我们想要每 5 秒聚合一次单词数，每个窗口都是从零开始统计的：。

```

java
DataStream<Tuple2<String, Integer>> windowCounts = wordCounts
    .keyBy(0)
    .timeWindow(Time.seconds(5))
    .sum(1);
```

第二个调用的.timeWindow()指定我们想要 5 秒的翻滚窗口（Tumble）。第三个调用为每个 key 每个窗口指定了 sum 聚合函数，在我们的例子中是按照次数字段（即 1 号索引字段）相加。得到的结果数据流，将每 5 秒输出一次这 5 秒内每个单词出现的次数。

最后一件事就是将数据流打印到控制台，并开始执行：

```

java
windowCounts.print().setParallelism(1);
env.execute("Socket Window WordCount");
```

最后的 env.execute 调用是启动实际 Flink 作业所必需的。所有算子操作（例如创建源、聚合、打印）只是构建了内部算子操作的图形。只有在 execute()被调用时才会在提交到集群上或本地计算机上执行。

下面是完整的代码，部分代码经过简化（代码在 [GitHub](#) 上也能访问到）：

```

java
package myflink;
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.util.Collector;
public class SocketWindowWordCount {
    public static void main(String[] args) throws Exception {
        // 创建 execution environment
        final StreamExecutionEnvironment env =
        StreamExecutionEnvironment.getExecutionEnvironment();
        // 通过连接 socket 获取输入数据，这里连接到本地 9000 端口，如果 9000 端口已被占用，请换一
```

## 个端口

```
DataStream<String> text = env.socketTextStream("localhost", 9000, "\n");
// 解析数据，按 word 分组，开窗，聚合
DataStream<Tuple2<String, Integer>> windowCounts = text
    .flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {
        @Override
        public void flatMap(String value, Collector<Tuple2<String,
Integer>> out) {
            for (String word : value.split("\s+")) {
                out.collect(Tuple2.of(word, 1));
            }
        }
    })
    .keyBy(0)
    .timeWindow(Time.seconds(5))
    .sum(1);
// 将结果打印到控制台，注意这里使用的是单线程打印，而非多线程
windowCounts.print().setParallelism(1);
env.execute("Socket Window WordCount");
}
```

## 运行程序

要运行示例程序，首先我们在终端启动 netcat 获得输入流：

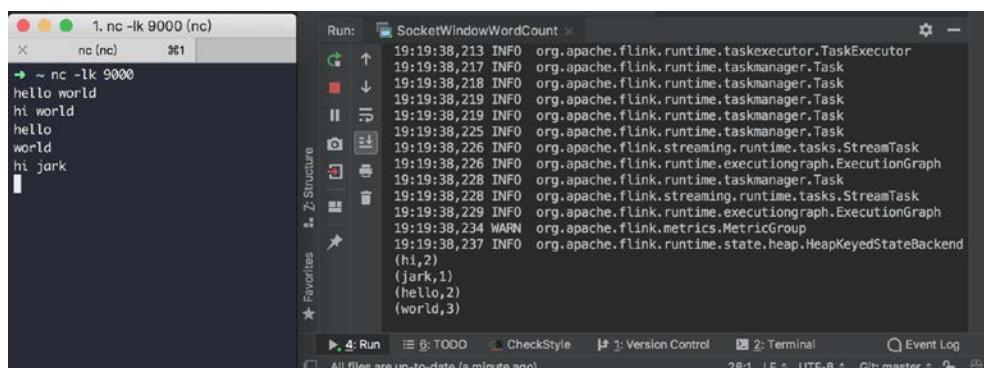
```
bash
nc -lk 9000
```

如果是 Windows 平台，可以通过 <https://nmap.org/ncat/> 安装 ncat 然后运行：

```
bash
ncat -lk 9000
```

然后直接运行 SocketWindowWordCount 的 main 方法。

只需要在 netcat 控制台输入单词，就能在 SocketWindowWordCount 的输出控制台看到每个单词的词频统计。如果想看到大于 1 的计数，请在 5 秒内反复键入相同的单词。



# Apache Flink 零基础实战教程： 如何计算实时热门商品

作者 伍翀



在[上一篇入门教程](#)中，我们已经能够快速构建一个基础的 Apache Flink（以下简称 Flink）程序了。本文会一步步地带领你实现一个更复杂的 Flink 应用程序：实时热门商品。在开始本文前我们建议你先实践一遍上篇文章，因为本文会沿用上文的 my-flink-project 项目框架。

通过本文你将学到：

1. 如何基于 EventTime 处理，如何指定 Watermark
2. 如何使用 Flink 灵活的 Window API
3. 何时需要用到 State，以及如何使用
4. 如何使用 ProcessFunction 实现 TopN 功能

## 实战案例介绍

本案例将实现一个“实时热门商品”的需求，我们可以将“实时热门商品”翻译成程序员更好理解的需求：每隔 5 分钟输出最近一小时内点击量最多的前 N 个商品。将这个需求进行分解我们大概要做这么几件事情：

- 抽取出业务时间戳，告诉 Flink 框架基于业务时间做窗口

- 过滤出点击行为数据
- 按一小时的窗口大小，每 5 分钟统计一次，做滑动窗口聚合（Sliding Window）
- 按每个窗口聚合，输出每个窗口中点击量前 N 名的商品

## 数据准备

这里我们准备了一份淘宝用户行为数据集（来自[阿里云天池公开数据集](#)，特别感谢）。本数据集包含了淘宝上某一天随机一千万用户的所有行为（包括点击、购买、加购、收藏）。数据集的组织形式和 MovieLens-20M 类似，即数据集的每一行表示一条用户行为，由用户 ID、商品 ID、商品类目 ID、行为类型和时间戳组成，并以逗号分隔。关于数据集中每一列的详细描述如下：

列名称	说明
用户 ID	整数类型，加密后的用户 ID
商品 ID	整数类型，加密后的商品 ID
商品类目 ID	整数类型，加密后的商品所属类目 ID
行为类型	字符串，枚举类型，包括('pv', 'buy', 'cart', 'fav')
时间戳	行为发生的时间戳，单位秒

你可以通过下面的命令下载数据集到项目的 resources 目录下：

```
$ cd my-flink-project/src/main/resources  
$ curl https://raw.githubusercontent.com/wuchong/my-flink-  
project/master/src/main/resources/UserBehavior.csv > UserBehavior.csv
```

这里是否使用 curl 命令下载数据并不重要，你也可以使用 wget 命令或者直接访问链接下载数据。关键是，将数据文件保存到项目的 resources 目录下，方便应用程序访问。

## 编写程序

在 src/main/java/myflink 下创建 HotItems.java 文件：

```
java  
package myflink;  
public class HotItems {  
    public static void main(String[] args) throws Exception {  
    }  
}
```

与上文一样，我们会一步步往里面填充代码。第一步仍然是创建一个 StreamExecutionEnvironment，我们把它添加到 main 函数中。

```
java
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
// 为了打印到控制台的结果不乱序，我们配置全局的并发为 1，这里改变并发对结果正确性没有影响
env.setParallelism(1);
```

## 创建模拟数据源

在数据准备章节，我们已经将测试的数据集下载到本地了。由于是一个 csv 文件，我们将使用 CsvInputFormat 创建模拟数据源。

注：虽然一个流式应用应该是一个一直运行着的程序，需要消费一个无限数据源。但是在本案例教程中，为了省去构建真实数据源的繁琐，我们使用了文件来模拟真实数据源，这并不影响下文要介绍的知识点。这也是一种本地验证 Flink 应用程序正确性的常用方式。

我们先创建一个 UserBehavior 的 POJO 类（所有成员变量声明成 public 便是 POJO 类），强类型化后能方便后续的处理。

```
java
/** 用户行为数据结构 */
public static class UserBehavior {
    public long userId;          // 用户 ID
    public long itemId;          // 商品 ID
    public int categoryId;       // 商品类目 ID
    public String behavior;      // 用户行为，包括("pv", "buy", "cart", "fav")
    public long timestamp;        // 行为发生的时间戳，单位秒
}
```

接下来我们就可以创建一个 PojoCsvInputFormat 了，这是一个读取 csv 文件并将每一行转成指定 POJO 类型（在我们案例中是 UserBehavior）的输入器。

```
java
// UserBehavior.csv 的本地文件路径
URL fileUrl = HotItems2.class.getClassLoader().getResource("UserBehavior.csv");
Path filePath = Path.fromLocalFile(new File(fileUrl.toURI()));
// 抽取 UserBehavior 的 TypeInformation，是一个 PojoTypeInfo
PojoTypeInfo<UserBehavior> pojoType = (PojoTypeInfo<UserBehavior>)
TypeExtractor.createTypeInfo(UserBehavior.class);
// 由于 Java 反射抽取出的字段顺序是不确定的，需要显式指定下文件中字段的顺序
String[] fieldOrder = new String[]{"userId", "itemId", "categoryId", "behavior",
"timestamp"};
// 创建 PojoCsvInputFormat
PojoCsvInputFormat<UserBehavior> csvInput = new PojoCsvInputFormat<>(filePath,
pojoType, fieldOrder);
```

下一步我们用 PojoCsvInputFormat 创建输入源。

```
java
DataStream<UserBehavior> dataSource = env.createInput(csvInput, pojoType);
```

这就创建了一个 UserBehavior 类型的 DataStream。

## EventTime 与 Watermark

当我们说“统计过去一小时内点击量”，这里的“一小时”是指什么呢？在 Flink 中它可以是指 ProcessingTime，也可以是 EventTime，由用户决定。

- ProcessingTime：事件被处理的时间。也就是由机器的系统时间来决定。
- EventTime：事件发生的时间。一般就是数据本身携带的时间。

在本案例中，我们需要统计业务时间上的每小时的点击量，所以要基于 EventTime 来处理。那么如果让 Flink 按照我们想要的业务时间来处理呢？这里主要有两件事情要做。

第一件是告诉 Flink 我们现在按照 EventTime 模式进行处理，Flink 默认使用 ProcessingTime 处理，所以我们要显式设置下。

```
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
```

第二件事情是指定如何获得业务时间，以及生成 Watermark。Watermark 是用来追踪业务事件的概念，可以理解成 EventTime 世界中的时钟，用来指示当前处理到什么时刻的数据了。由于我们的数据源的数据已经经过整理，没有乱序，即事件的时间戳是单调递增的，所以可以将每条数据的业务时间就当做 Watermark。这里我们用 AscendingTimestampExtractor 来实现时间

戳的抽取和 Watermark 的生成。

```
java
DataStream<UserBehavior> timedData = dataSource
    .assignTimestampsAndWatermarks(new AscendingTimestampExtractor<UserBehavior>()
{
```

注：真实业务场景一般都是存在乱序的，所以一般使用 BoundedOutOfOrdernessTimestampExtractor。

```
    @Override
    public long extractAscendingTimestamp(UserBehavior userBehavior) {
        // 原始数据单位秒，将其转成毫秒
        return userBehavior.timestamp * 1000;
    }
});
```

这样我们就得到了一个带有时间标记的数据流了，后面就能做一些窗口的操作。

## 过滤出点击事件

在开始窗口操作之前，先回顾下需求“每隔 5 分钟输出过去一小时内点击量最多的前 N 个商品”。由于原始数据中存在点击、加购、购买、收藏各种行为的数据，但是我们只需要统计点击量，所以先使用 FilterFunction 将点击行为数据过滤出来。

```
java
DataStream<UserBehavior> pvData = timedData
    .filter(new FilterFunction<UserBehavior>() {
        @Override
        public boolean filter(UserBehavior userBehavior) throws Exception {
            // 过滤出只有点击的数据
            return userBehavior.behavior.equals("pv");
        }
});
```

## 窗口统计点击量

由于要每隔 5 分钟统计一次最近一小时每个商品的点击量，所以窗口大小是一小时，每隔 5 分钟滑动一次。即分别要统计 [09:00, 10:00], [09:05, 10:05], [09:10, 10:10]... 等窗口的商品点击量。是一个常见的滑动窗口需求（Sliding Window）。

```
java
DataStream<ItemViewCount> windowedData = pvData
    .keyBy("itemId")
    .timeWindow(Time.minutes(60), Time.minutes(5))
```

```
.aggregate(new CountAgg(), new WindowResultFunction());
```

我们使用`.keyBy("itemId")`对商品进行分组，使用`.timeWindow(Time size, Time slide)`对每个商品做滑动窗口（1小时窗口，5分钟滑动一次）。然后我们使用`.aggregate(AggregateFunction af, WindowFunction wf)`做增量的聚合操作，它能使用`AggregateFunction`提前聚合掉数据，减少`state`的存储压力。较之`.apply(WindowFunction wf)`会将窗口中的数据都存储下来，最后一起计算要高效地多。`aggregate()`方法的第一个参数用于

这里的`CountAgg`实现了`AggregateFunction`接口，功能是统计窗口中的条数，即遇到一条数据就加一。

```
java
/** COUNT 统计的聚合函数实现，每出现一条记录加一 */
public static class CountAgg implements AggregateFunction<UserBehavior, Long,
Long> {
    @Override
    public Long createAccumulator() {
        return 0L;
    }
    @Override
    public Long add(UserBehavior userBehavior, Long acc) {
        return acc + 1;
    }
    @Override
    public Long getResult(Long acc) {
        return acc;
    }
    @Override
    public Long merge(Long acc1, Long acc2) {
        return acc1 + acc2;
    }
}
```

`.aggregate(AggregateFunction af, WindowFunction wf)`的第二个参数`WindowFunction`将每个`key`每个窗口聚合后的结果带上其他信息进行输出。我们这里实现的`WindowResultFunction`将主键商品ID，窗口，点击量封装成了`ItemViewCount`进行输出。

```
java
/** 用于输出窗口的结果 */
public static class WindowResultFunction implements WindowFunction<Long,
ItemViewCount, Tuple, TimeWindow> {
    @Override
    public void apply(
        Tuple key, // 窗口的主键，即 itemId
        TimeWindow window, // 窗口
        Iterable<Long> aggregateResult, // 聚合函数的结果，即 count 值
        Collector<ItemViewCount> collector // 输出类型为 ItemViewCount
    ) throws Exception {
        Long itemId = ((Tuple1<Long>) key).f0;
```

```

        Long count = aggregateResult.iterator().next();
        collector.collect(ItemViewCount.of(itemId, window.getEnd(), count));
    }
}
/** 商品点击量(窗口操作的输出类型) */
public static class ItemViewCount {
    public long itemId;      // 商品 ID
    public long windowEnd;   // 窗口结束时间戳
    public long viewCount;   // 商品的点击量
    public static ItemViewCount of(long itemId, long windowEnd, long viewCount) {
        ItemViewCount result = new ItemViewCount();
        result.itemId = itemId;
        result.windowEnd = windowEnd;
        result.viewCount = viewCount;
        return result;
    }
}

```

现在我们得到了每个商品在每个窗口的点击量的数据流。

## TopN 计算最热门商品

为了统计每个窗口下最热门的商品，我们需要再次按窗口进行分组，这里根据 ItemViewCount 中的 windowEnd 进行 keyBy() 操作。然后使用 ProcessFunction 实现一个自定义的 TopN 函数 TopNHotItems 来计算点击量排名前 3 名的商品，并将排名结果格式化成字符串，便于后续输出。

```

java
DataStream<String> topItems = windowedData
    .keyBy("windowEnd")
    .process(new TopNHotItems(3)); // 求点击量前 3 名的商品

```

ProcessFunction 是 Flink 提供的一个 low-level API，用于实现更高级的功能。它主要提供了定时器 timer 的功能（支持 EventTime 或 ProcessingTime）。本案例中我们将利用 timer 来判断何时收齐了某个 window 下所有商品的点击量数据。由于 Watermark 的进度是全局的，

在 processElement 方法中，每当收到一条数据（ItemViewCount），我们就注册一个 windowEnd+1 的定时器（Flink 框架会自动忽略同一时间的重复注册）。windowEnd+1 的定时器被触发时，意味着收到了 windowEnd+1 的 Watermark，即收齐了该 windowEnd 下的所有商品窗口统计值。我们在 onTimer() 中处理将收集的所有商品及点击量进行排序，选出 TopN，并将排名信息格式化成字符串后进行输出。

这里我们还使用了 ListState<ItemViewCount> 来存储收到的每条 ItemViewCount 消息，保证在发生故障时，状态数据的不丢失和一致性。ListState 是 Flink 提供的类似 Java List 接口的

State API，它集成了框架的 checkpoint 机制，自动做到了 exactly-once 的语义保证。

```
java
/** 求某个窗口中前 N 名的热门点击商品，key 为窗口时间戳，输出为 TopN 的结果字符串 */
public static class TopNHotItems extends KeyedProcessFunction<Tuple,
ItemViewCount, String> {
    private final int topSize;
    public TopNHotItems(int topSize) {
        this.topSize = topSize;
    }
    // 用于存储商品与点击数的状态，待收齐同一个窗口的数据后，再触发 TopN 计算
    private ListState<ItemViewCount> itemState;
    @Override
    public void open(Configuration parameters) throws Exception {
        super.open(parameters);
        // 状态的注册
        ListStateDescriptor<ItemViewCount> itemsStateDesc = new
ListStateDescriptor<>(
            "itemState-state",
            ItemViewCount.class);
        itemState = getRuntimeContext().getListState(itemsStateDesc);
    }
    @Override
    public void processElement(
        ItemViewCount input,
        Context context,
        Collector<String> collector) throws Exception {
        // 每条数据都保存到状态中
        itemState.add(input);
        // 注册 windowEnd+1 的 EventTime Timer，当触发时，说明收齐了属于 windowEnd 窗口的所有商品数据
        context.timerService().registerEventTimeTimer(input.windowEnd + 1);
    }
    @Override
    public void onTimer(
        long timestamp, OnTimerContext ctx, Collector<String> out) throws
Exception {
        // 获取收到的所有商品点击量
        List<ItemViewCount> allItems = new ArrayList<>();
        for (ItemViewCount item : itemState.get()) {
            allItems.add(item);
        }
        // 提前清除状态中的数据，释放空间
        itemState.clear();
        // 按照点击量从大到小排序
        allItems.sort(new Comparator<ItemViewCount>() {
            @Override
            public int compare(ItemViewCount o1, ItemViewCount o2) {
                return (int) (o2.viewCount - o1.viewCount);
            }
        });
        // 将排名信息格式化成 String，便于打印
        StringBuilder result = new StringBuilder();
        result.append("=====\n");
        result.append("时间: ").append(new Timestamp(timestamp-1)).append("\n");
        for (int i=0;i<topSize;i++) {
            ItemViewCount currentItem = allItems.get(i);
            result.append(currentItem.viewCount).append(" ")
            .append(currentItem.itemName).append("\n");
        }
    }
}
```

```

        // No1: 商品 ID=12224 浏览量=2413
        result.append("No").append(i).append(":")
            .append(" 商品 ID=").append(currentItem.itemId)
            .append(" 浏览量=").append(currentItem.viewCount)
            .append("\n");
    }
    result.append("=====\n\n");
    out.collect(result.toString());
}
}

```

## 打印输出

最后一步我们将结果打印输出到控制台，并调用 env.execute 执行任务。

```

topItems.print();
env.execute("Hot Items Job");

```

## 运行程序

直接运行 main 函数，就能看到不断输出的每个时间点的热门商品 ID。

```

=====
时间: 2017-11-26 15:20:00.0
No0: 商品ID=3845720 浏览量=20
No1: 商品ID=812879 浏览量=20
No2: 商品ID=1871901 浏览量=18
=====

=====
时间: 2017-11-26 15:25:00.0
No0: 商品ID=812879 浏览量=20
No1: 商品ID=3845720 浏览量=19
No2: 商品ID=2453685 浏览量=16
=====

=====
时间: 2017-11-26 15:30:00.0
No0: 商品ID=812879 浏览量=19
No1: 商品ID=3845720 浏览量=19
No2: 商品ID=2331370 浏览量=17
=====

=====
时间: 2017-11-26 15:35:00.0
No0: 商品ID=812879 浏览量=21
No1: 商品ID=3845720 浏览量=20
No2: 商品ID=2364679 浏览量=18
=====

=====
时间: 2017-11-26 15:40:00.0

```

---

本文的完整代码可以通过 [GitHub](#) 访问到。本文通过实现一个“实时热门商品”的案例，学习和实践了 Flink 的多个核心概念和 API 用法。包括 EventTime、Watermark 的使用，State 的使用，Window API 的使用，以及 TopN 的实现。希望本文能加深大家对 Flink 的理解，帮助大家解决实战上遇到的问题。

# Apache Flink SQL 概览

作者 孙金城，淘宝花名 金竹

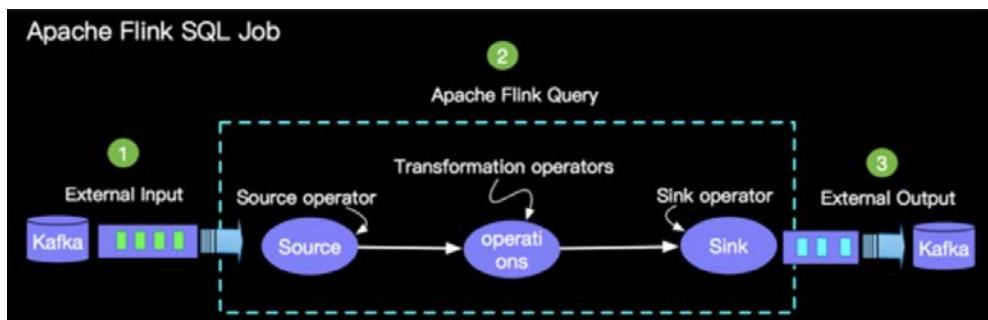
空天闊海



本篇核心目标是让大家概要了解一个完整的 Apache Flink SQL Job 的组成部分，以及 Apache Flink SQL 所提供的核心算子的语义，最后会应用 Tumble Window 编写一个 End-to-End 的页面访问的统计示例。

## Apache Flink SQL Job 的组成

我们做任何数据计算都离不开读取原始数据，计算逻辑和写入计算结果数据三部分，当然基于 Apache Flink SQL 编写的计算 Job 也离不开这个三部分，如下所示：



如上所示，一个完整的 Apache Flink SQL Job 由如下三部分：

- Source Operator - Source operator 是对外部数据源的抽象，目前 Apache Flink 内置了很多常用的数据源实现，比如上图提到的 Kafka。
- Query Operators - 查询算子主要完成如图的 Query Logic，目前支持了 Union，Join，Projection, Difference, Intersection 以及 window 等大多数传统数据库支持的操作。
- Sink Operator - Sink operator 是对外结果表的抽象，目前 Apache Flink 也内置了很多常用的结果表的抽象，比如上图提到的 Kafka。

## Apache Flink SQL 核心算子

SQL 是 Structured Query Language 的缩写，最初是由美国计算机科学家 Donald D. Chamberlin 和 Raymond F. Boyce 在 20 世纪 70 年代早期从 [Early History of SQL](#) 中了解关系模型后在 IBM 开发的。该版本最初称为[SEQUEL: A Structured English Query Language]（结构化英语查询语言），旨在操纵和检索存储在 IBM 原始准关系数据库管理系统 System R 中的数据。直到 1986 年，ANSI 和 ISO 标准组正式采用了标准的"数据库语言 SQL"语言定义。Apache Flink SQL 核心算子的语义设计也参考了 [1992](#)、[2011](#) 等 ANSI-SQL 标准。接下来我们将简单为大家介绍 Apache Flink SQL 每一个算子的语义。

### SELECT

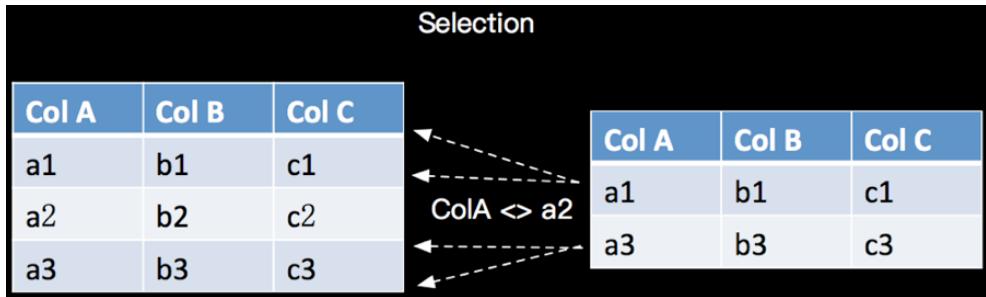
SELECT 用于从数据集/流中选择数据，语法遵循 ANSI-SQL 标准，语义是关系代数中的投影 (Projection)，对关系进行垂直分割，消去某些列。

一个使用 Select 的语句如下：

```
SELECT ColA, ColC FROM tab ;
```

### WHERE

WHERE 用于从数据集/流中过滤数据，与 SELECT 一起使用，语法遵循 ANSI-SQL 标准，语义是关系代数的 Selection，根据某些条件对关系做水平分割，即选择符合条件的记录，如下所示：



对应的 SQL 语句如下：

```
SELECT * FROM tab WHERE ColA <> 'a2' ;
```

## GROUP BY

GROUP BY 是对数据进行分组的操作，比如我需要分别计算一下一个学生表里面女生和男生的人数分别是多少，如下：



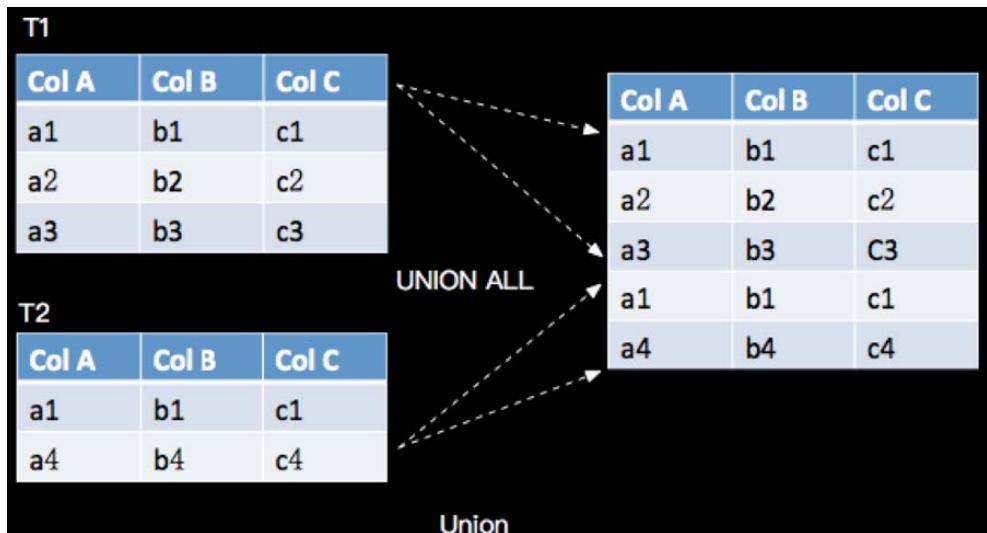
对应的 SQL 语句如下：

```
SELECT sex, COUNT(name) AS count FROM tab GROUP BY sex ;
```

## UNION ALL

UNION ALL 将两个表合并起来，要求两个表的字段完全一致，包括字段类型、字段顺序,语义对应关系代数的 Union，只是关系代数是 Set 集合操作，会有去重复操作，UNION ALL 不进行去

重, 如下所示:

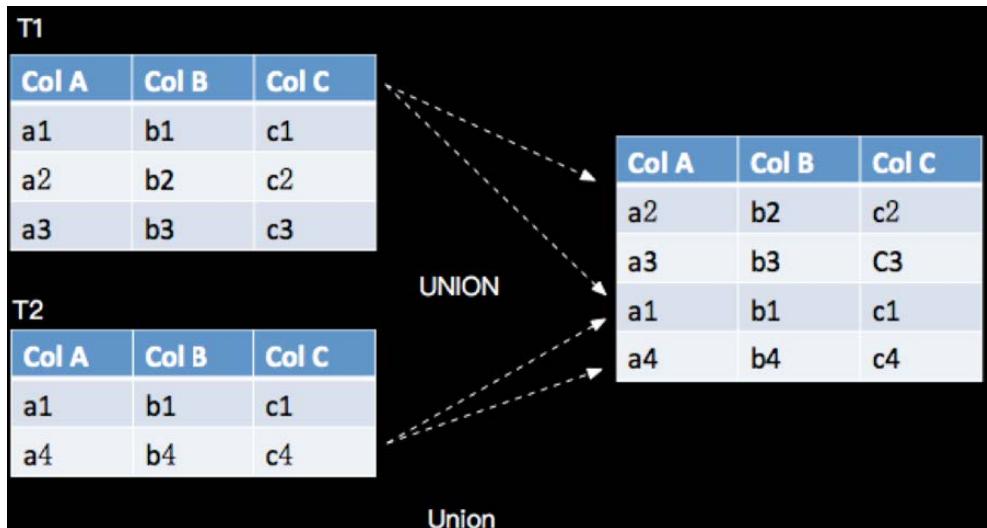


对应的 SQL 语句如下:

```
SELECT * FROM T1 UNION ALL SELECT * FROM T2
```

## UNION

UNION 将两个流给合并起来, 要求两个流的字段完全一致, 包括字段类型、字段顺序, 并且 UNION 不同于 UNION ALL, UNION 会对结果数据去重, 与关系代数的 Union 语义一致, 如下:



对应的 SQL 语句如下：

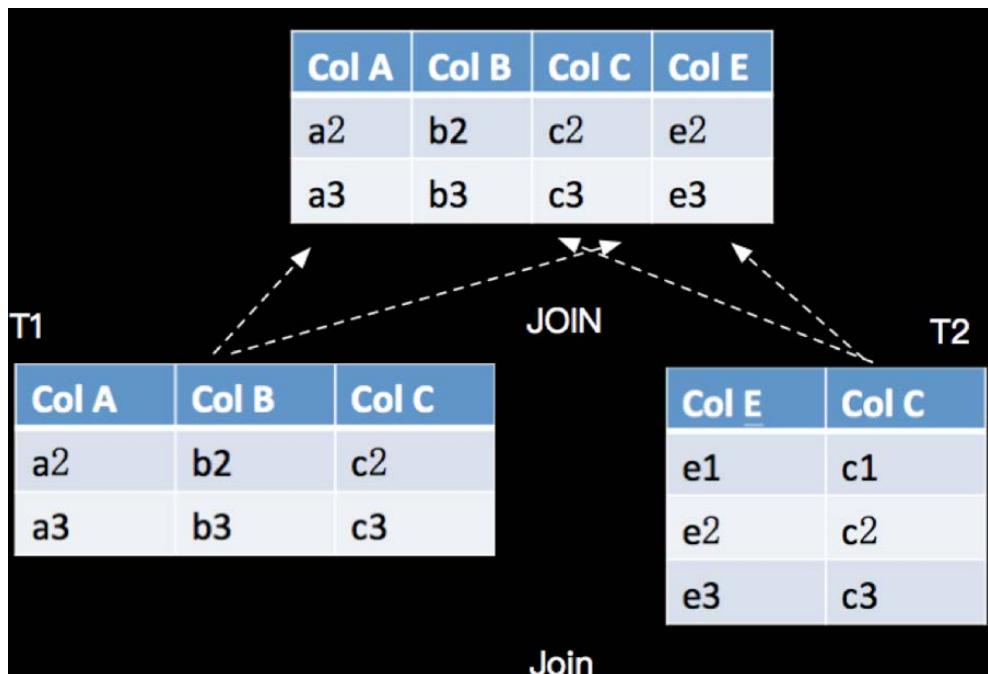
```
SELECT * FROM T1 UNION SELECT * FROM T2
```

## JOIN

JOIN 用于把来自两个表的行联合起来形成一个宽表，Apache Flink 支持的 JOIN 类型：

- JOIN - INNER JOIN
- LEFT JOIN - LEFT OUTER JOIN
- RIGHT JOIN - RIGHT OUTER JOIN
- FULL JOIN - FULL OUTER JOIN

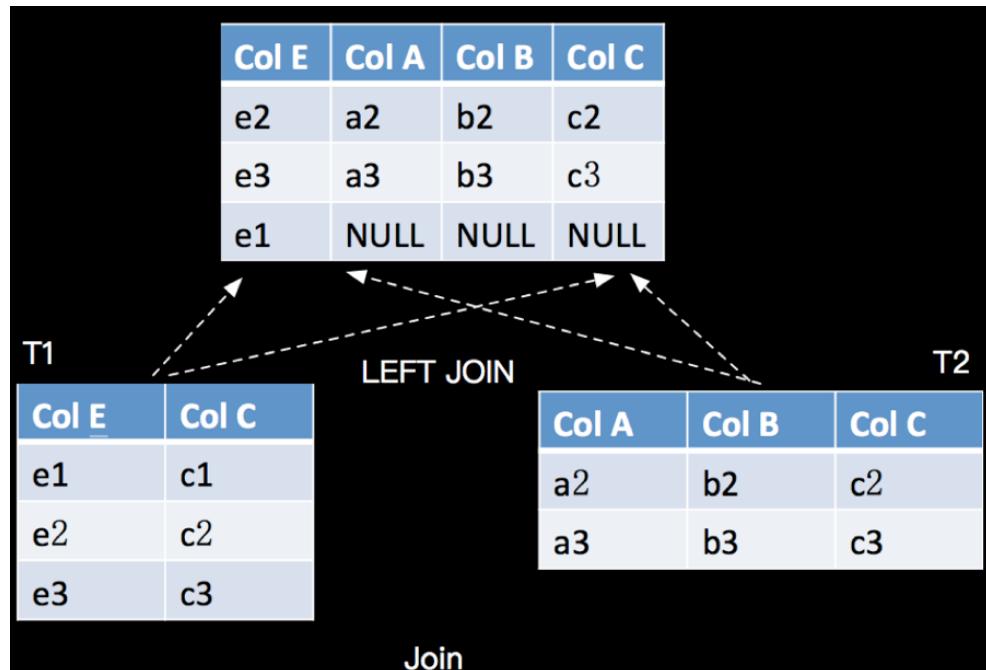
JOIN 与关系代数的 Join 语义相同，具体如下：



对应的 SQL 语句如下(INNER JOIN)：

```
SELECT ColA, ColB, T2.ColC, ColE FROM T1 JOIN T2 ON T1.ColC = T2.ColC ;
```

LEFT JOIN 与 INNER JOIN 的区别是当右表没有与左边相 JOIN 的数据时候，右边对应的字段补 NULL 输出，如下：



对应的 SQL 语句如下(LEFT JOIN)：

```
SELECT ColA, ColB, T2.ColC, ColE FROM T1 LEFT JOIN T2 ON T1.ColC = T2.ColC ;
```

说明：

- 细心的读者可能发现上面 T2.ColC 是添加了前缀 T2 了，这里需要说明一下，当两张表有字段名字一样的时候，我需要指定是从那个表里面投影的。
- RIGHT JOIN 相当于 LEFT JOIN 左右两个表交互一下位置。FULLJOIN 相当于 RIGHT JOIN 和 LEFT JOIN 之后进行 UNION ALL 操作。

## Window

在 Apache Flink 中有 2 种类型的 Window，一种是 OverWindow，即传统数据库的标准开窗，每一个元素都对应一个窗口。一种是 GroupWindow，目前在 SQL 中 GroupWindow 都是基于时间进行窗口划分的。

## OverWindow

OVER Window 目前支持由如下三个元素组合的 8 中类型：

- 时间 - Processing Time 和 EventTime
- 数据集 - Bounded 和 Unbounded
- 划分方式 - ROWS 和 RANGE 我们以的 Bounded ROWS 和 Bounded RANGE 两种常用类型，想大家介绍 Over Window 的语义

## Bounded ROWS Over Window

Bounded ROWS OVER Window 每一行元素都视为新的计算行，即，每一行都是一个新的窗口。

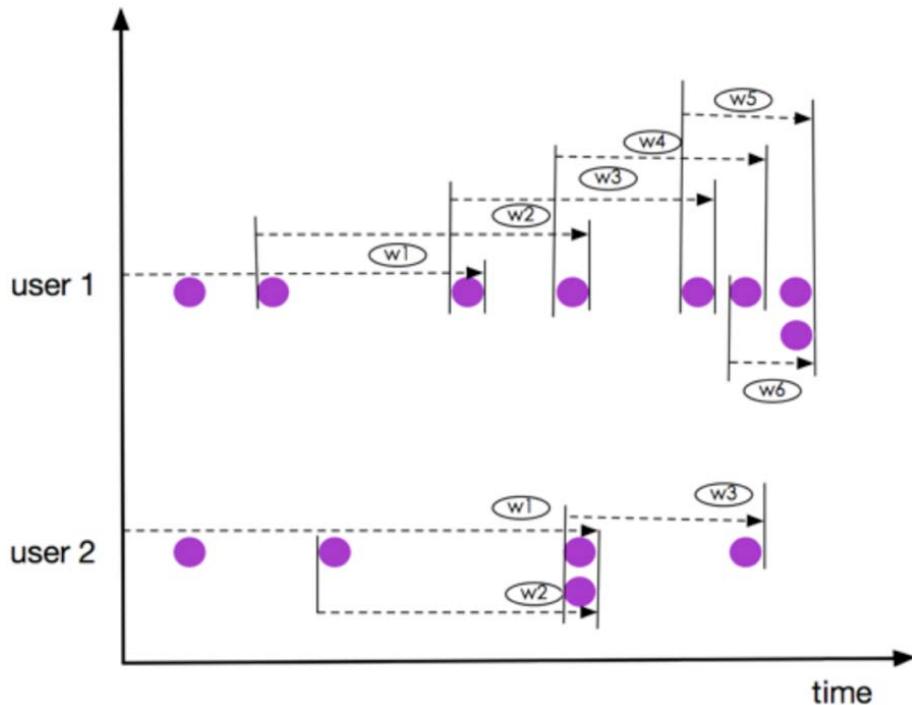
语法

```
SELECT
    agg1(col1) OVER(
        [PARTITION BY (value_expression1, ..., value_expressionN)]
        ORDER BY timeCol
        ROWS
        BETWEEN (UNBOUNDED | rowCount) PRECEDING AND CURRENT ROW) AS colName,
    ...
FROM Tab1
```

- value\_expression - 进行分区的字表达式；
- timeCol - 用于元素排序的时间字段；
- rowCount - 是定义根据当前行开始向前追溯几行元素；

语义

我们以 3 个元素(2 PRECEDING)的窗口为例，如下图：



上图所示窗口 user 1 的 w5 和 w6, user 2 的 窗口 w2 和 w3, 虽然有元素都是同一时刻到达, 但是他们仍然是在不同的窗口, 这一点有别于 RANGE OVER Window.

## Bounded RANGE Over Window

Bounded RANGE OVER Window 具有相同时间值的所有元素行视为同一计算行, 即, 具有相同时间值的所有行都是同一个窗口;

语法

Bounded RANGE OVER Window 的语法如下:

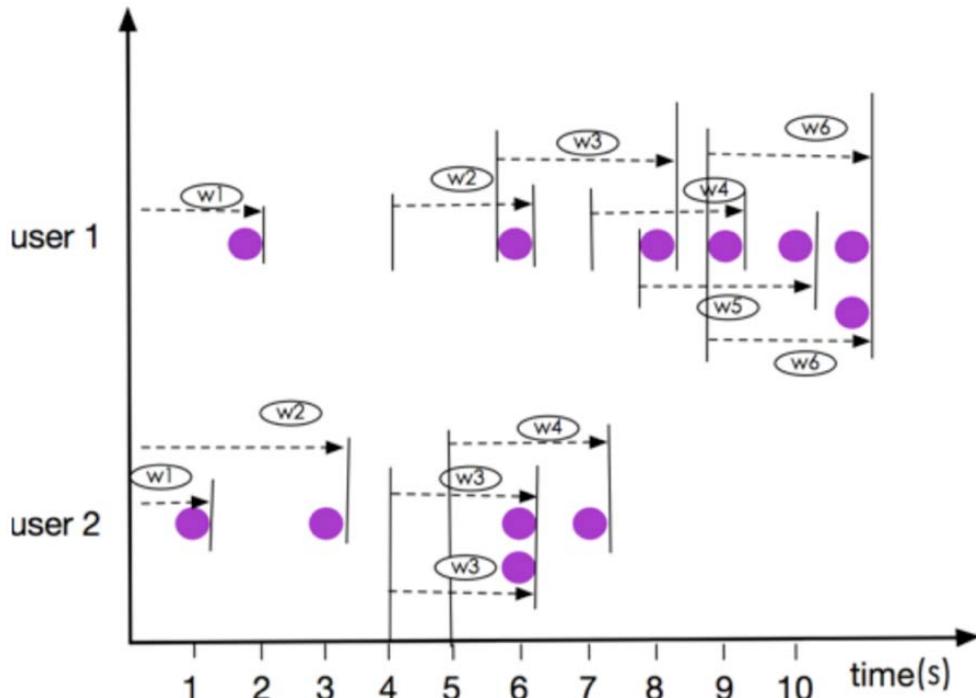
```
SELECT
  agg1(coll) OVER(
    [PARTITION BY (value_expression1, ..., value_expressionN)]
    ORDER BY timeCol
    RANGE
```

```
BETWEEN (UNBOUNDED | timeInterval) PRECEDING AND CURRENT ROW) AS colName,
...
FROM Tab1
```

- value\_expression - 进行分区的字表达式；
- timeCol - 用于元素排序的时间字段；
- timeInterval - 是定义根据当前行开始向前追溯指定时间的元素行；

语义

我们以 3 秒中数据(INTERVAL '2' SECOND)的窗口为例，如下图：



注意：上图所示窗口 user 1 的 w6， user 2 的 窗口 w3， 元素都是同一时刻到达,他们是在同一个窗口，这一点有别于 ROWS OVER Window.

## GroupWindow

根据窗口数据划分的不同，目前 Apache Flink 有如下 3 种 Bounded Window:

- Tumble - 滚动窗口，窗口数据有固定的大小，窗口数据无叠加；
- Hop - 滑动窗口，窗口数据有固定大小，并且有固定的窗口重建频率，窗口数据有叠加；
- Session - 会话窗口，窗口数据没有固定的大小，根据窗口数据活跃程度划分窗口，窗口数据无叠加；

说明：Apache Flink 还支持 UnBounded 的 Group Window，也就是全局 Window，流上所有数据都在一个窗口里面，语义非常简单，这里不做详细介绍。

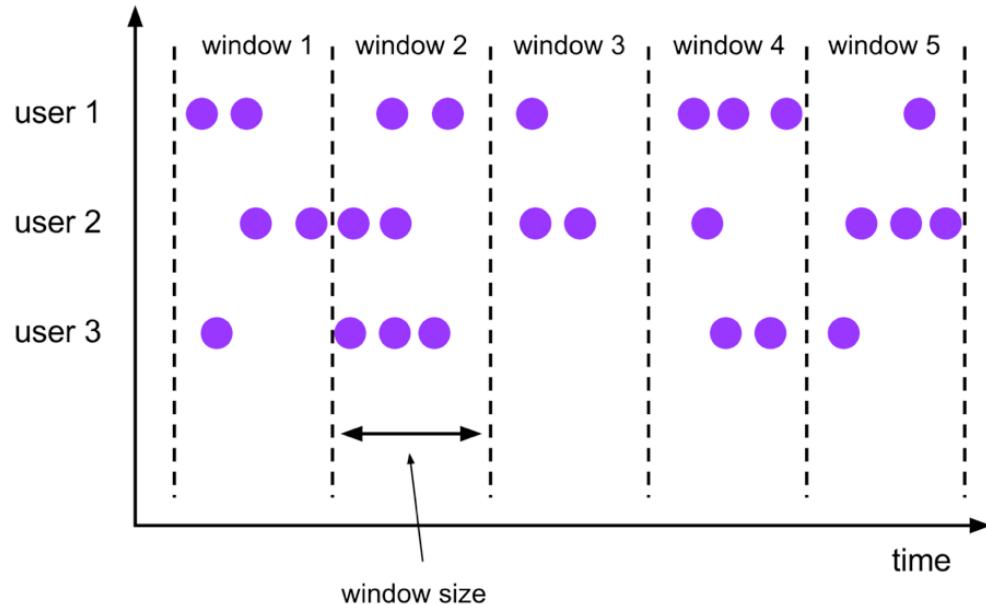
GroupWindow 的语法如下：

```
SELECT
    [gk],
    agg1(coll),
    ...
    aggN(colN)
FROM Tab1
GROUP BY [WINDOW(defination)], [gk]
```

- [WINDOW(defination)] - 在具体窗口语义介绍中介绍。

## Tumble Window

Tumble 滚动窗口有固定 size，窗口数据不重叠，具体语义如下：

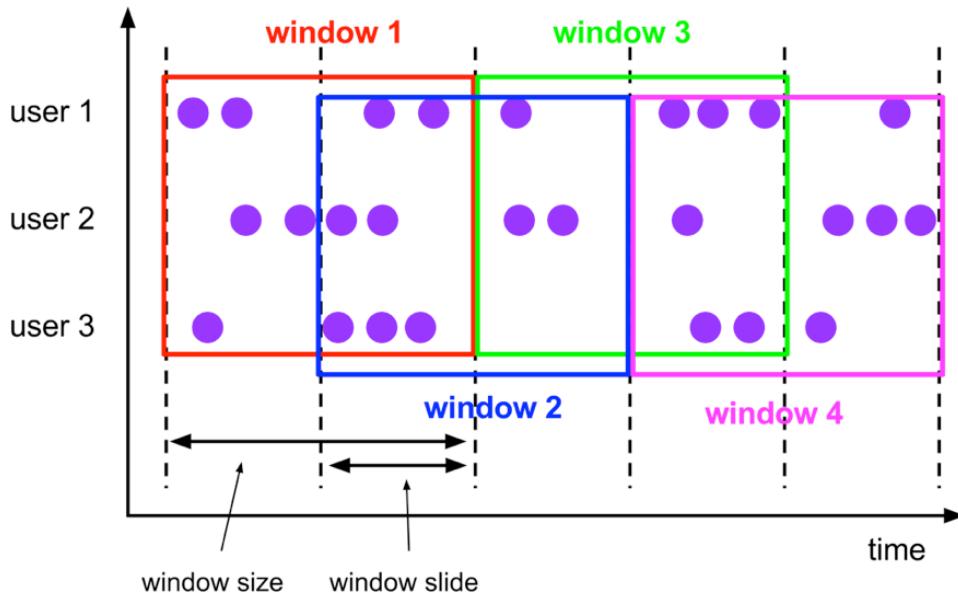


假设我们要写一个 2 分钟大小的 Tumble，示例 SQL 如下：

```
SELECT gk, COUNT(*) AS pv
FROM tab
GROUP BY TUMBLE(rowtime, INTERVAL '2' MINUTE), gk
```

## Hop Window

Hop 滑动窗口和滚动窗口类似，窗口有固定的 size，与滚动窗口不同的是滑动窗口可以通过 slide 参数控制滑动窗口的新建频率。因此当 slide 值小于窗口 size 的值的时候多个滑动窗口会重叠，具体语义如下：

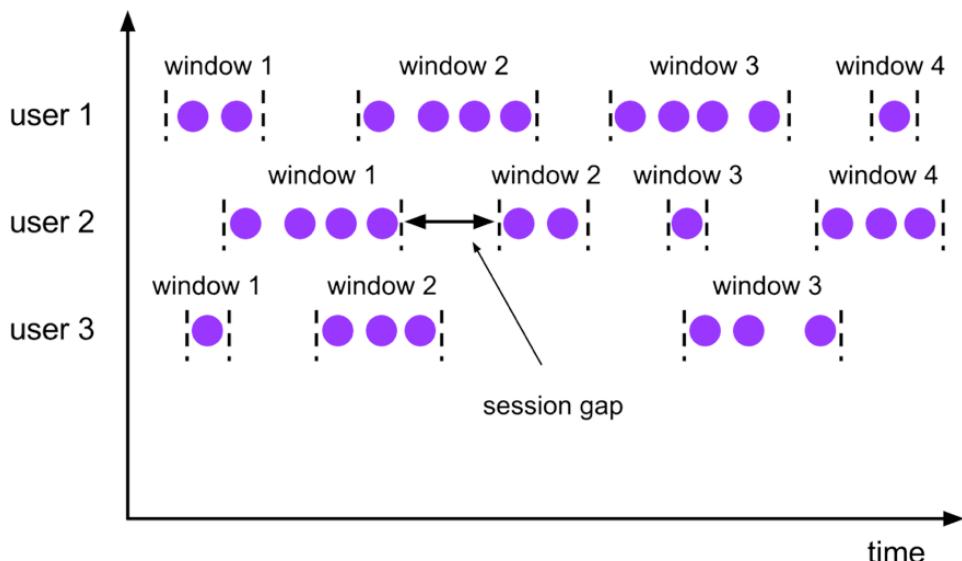


假设我们要写一个每 5 分钟统计近 10 分钟的页面访问量(PV).

```
SELECT gk, COUNT(*) AS pv
FROM tab
GROUP BY HOP(rowtime, INTERVAL '5' MINUTE, INTERVAL '10' MINUTE), gk
```

## Session Window

Session 会话窗口 是没有固定大小的窗口，通过 session 的活跃度分组元素。不同于滚动窗口和滑动窗口，会话窗口不重叠,也没有固定的起止时间。一个会话窗口在一段时间内没有接收到元素时，即当出现非活跃间隙时关闭。一个会话窗口 分配器通过配置 session gap 来指定非活跃周期的时长,具体语义如下：



假设我们要写一个统计连续的两个访问用户之间的访问时间间隔不超过 3 分钟的的页面访问量(PV).

```
SELECT gk, COUNT(*) AS pv
  FROM pageAccessSession_tab
 GROUP BY SESSION(rowtime, INTERVAL '3' MINUTE), gk
```

说明:很多场景用户需要获得 Window 的开始和结束时间，上面的 GroupWindow 的 SQL 示例中没有体现，那么窗口的开始和结束时间应该怎样获取呢? Apache Flink 我们提供了如下辅助函数:

- TUMBLE\_START/TUMBLE\_END
- HOP\_START/HOP\_END

- SESSION\_START/SESSION\_END

这些辅助函数如何使用，请参考如下完整示例的使用方式。

## 完整的 SQL Job 案例

上面我们介绍了 Apache Flink SQL 核心算子的语法及语义，这部分将选取 Bounded EventTime Tumble Window 为例为大家编写一个完整的包括 Source 和 Sink 定义的 Apache Flink SQL Job。假设有一张淘宝页面访问表(PageAccess\_tab)，有地域，用户 ID 和访问时间。我们需要按不同地域统计每 2 分钟的淘宝首页的访问量(PV)。具体数据如下：

region	userId	accessTime
ShangHai	U0010	2017-11-11 10:01:00
BeiJing	U1001	2017-11-11 10:01:00
BeiJing	U2032	2017-11-11 10:10:00
BeiJing	U1100	2017-11-11 10:11:00
ShangHai	U0011	2017-11-11 12:10:00

## Source 定义

自定义 Apache Flink Stream Source 需要实现 StreamTableSource, StreamTableSource 中通过 StreamExecutionEnvironment 的 addSource 方法获取 DataStream, 所以我们需要自定义一个 SourceFunction，并且要支持产生 WaterMark，也就是要实现 DefinedRowtimeAttributes 接口。出于代码篇幅问题，我们如下只介绍核心部分，完整代码请查看：[EventTimeTumbleWindowDemo.scala](#)

## Source Function 定义

支持接收携带 EventTime 的数据集合，Either 的数据结构 Right 是 WaterMark，Left 是元数据：

```
class MySourceFunction[T](dataList: Seq[Either[(Long, T), Long]]) extends SourceFunction[T] {
  override def run(ctx: SourceContext[T]): Unit = {
    dataList.foreach {
      case Left(t) => ctx.collectWithTimestamp(t._2, t._1)
      case Right(w) => ctx.emitWatermark(new Watermark(w)) // emit watermark
    }
  }
}
```

```
    }  
}
```

## 定义 StreamTableSource

我们自定义的 Source 要携带我们测试的数据，以及对应的 WaterMark 数据，具体如下：

```
class MyTableSource extends StreamTableSource[Row] with DefinedRowtimeAttributes {  
    // 页面访问表数据 rows with timestamps and watermarks  
    val data = Seq(  
        // Data  
        Left(1510365660000L, Row.of(new JLong(1510365660000L), "ShangHai", "U0010")),  
        // Watermark  
        Right(1510365660000L),  
        ...  
    )  
    val fieldNames = Array("accessTime", "region", "userId")  
    val schema = new TableSchema(fieldNames, Array(Types.SQL_TIMESTAMP,  
Types.STRING, Types.STRING))  
    val rowType = new RowTypeInfo(  
        Array(Types.LONG, Types.STRING,  
Types.STRING).asInstanceOf[Array[TypeInformation[_]]],  
        fieldNames)  
    override def getDataStream(execEnv: StreamExecutionEnvironment): DataStream[Row]  
= {  
    // 添加数据源实现  
    execEnv.addSource(new  
MySourceFunction[Row](data)).setParallelism(1).returns(rowType)  
    }  
    ...  
}
```

## Sink 定义

我们简单的将计算结果写入到 Apache Flink 内置支持的 CSVSink 中，定义 Sink 如下：

```
def getCsvTableSink: TableSink[Row] = {  
    val tempFile = ...  
    new CsvTableSink(tempFile.getAbsolutePath).configure(  
        Array[String]("region", "winStart", "winEnd", "pv"),  
        Array[TypeInformation[_]](Types.STRING, Types.SQL_TIMESTAMP,  
Types.SQL_TIMESTAMP, Types.LONG))  
}
```

## 构建主程序

主程序包括执行环境的定义，Source/Sink 的注册以及统计查 SQL 的执行，具体如下：

```

def main(args: Array[String]): Unit = {
    // Streaming 环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val tEnv = TableEnvironment.getTableEnvironment(env)
    // 设置 EventTime
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    //方便我们查出输出数据
    env.setParallelism(1)
    val sourceTableName = "mySource"
    // 创建自定义 source 数据结构
    val tableSource = new MyTableSource
    val sinkTableName = "csvSink"
    // 创建 CSV sink 数据结构
    val tableSink = getCSVTableSink
    // 注册 source
    tEnv.registerTableSource(sourceTableName, tableSource)
    // 注册 sink
    tEnv.registerTableSink(sinkTableName, tableSink)
    val sql =
        "SELECT  " +
        "  region, " +
        "  TUMBLE_START(accessTime, INTERVAL '2' MINUTE) AS winStart," +
        "  TUMBLE_END(accessTime, INTERVAL '2' MINUTE) AS winEnd, COUNT(region) AS pv
    " +
        " FROM mySource " +
        " GROUP BY TUMBLE(accessTime, INTERVAL '2' MINUTE), region"
    tEnv.sqlQuery(sql).insertInto(sinkTableName);
    env.execute()
}

```

## 执行并查看运行结果

执行主程序后我们会在控制台得到 Sink 的文件路径，如下：

```

Sink path :
/var/folders/88/8n406qmx2z73qvrzc_rbtv_r0000gn/T/csv_sink_8025014910735142911item

```

Cat 方式查看计算结果，如下：

```

jinchengsunjcdeMacBook-Pro:FlinkTableApiDemo jincheng.sunjc$ cat
/var/folders/88/8n406qmx2z73qvrzc_rbtv_r0000gn/T/csv_sink_8025014910735142911item
ShangHai,2017-11-11 02:00:00.0,2017-11-11 02:02:00.0,1
BeiJing,2017-11-11 02:00:00.0,2017-11-11 02:02:00.0,1
BeiJing,2017-11-11 02:10:00.0,2017-11-11 02:12:00.0,2
ShangHai,2017-11-11 04:10:00.0,2017-11-11 04:12:00.0,1

```

## 小结

本篇概要的介绍了 Apache Flink SQL 的所有核心算子，并以一个 End-to-End 的示例展示了如何编写 Apache Flink SQL 的 Job。希望对大家有所帮助，更多 Apache Flink SQL 的介绍欢迎大家访问

作者的如下专栏或专辑：

- 本篇的电子版：<http://zhuanlan.51cto.com/art/201811/586881.htm>
- 金竹 51CTO Apache Flink 漫谈 专栏 - <http://zhuanlan.51cto.com/columnlist/jinzhu>
- 金竹 云栖社区 Apache Flink 漫谈 专辑 - <https://yq.aliyun.com/album/206>



51CTO 金竹 专栏



云栖社区 金竹 专辑



因篇幅有限，测试代码和环境准备部分详情扫码查看

# Apache Flink 类型和序列化机制简介

作者 董伟柯

来自：腾讯云+社区



使用 Apache Flink（以下简称 Flink）编写处理逻辑时，新手总是容易被林林总总的概念所混淆：

为什么 Flink 有那么多的类型声明方式？

`BasicTypeInfo.STRING_TYPE_INFO`、`Types.STRING`、`Types.STRING()` 有何区别？

`TypeInfoFactory` 又是什么？

`TypeInformation.of` 和 `TypeHint` 是如何使用的呢？

接下来本文将逐步解密 Flink 的类型和序列化机制。

## Flink 的类型分类

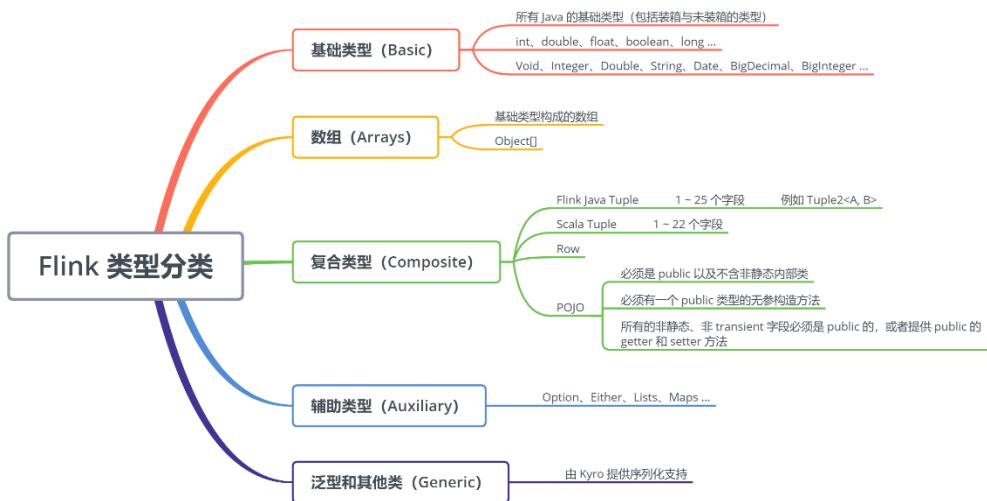


图 1: Flink 类型分类

Flink 的类型系统源码位于 `org.apache.flink.api.common.typeinfo` 包，让我们对图 1 深入追踪，看一下类的继承关系图：

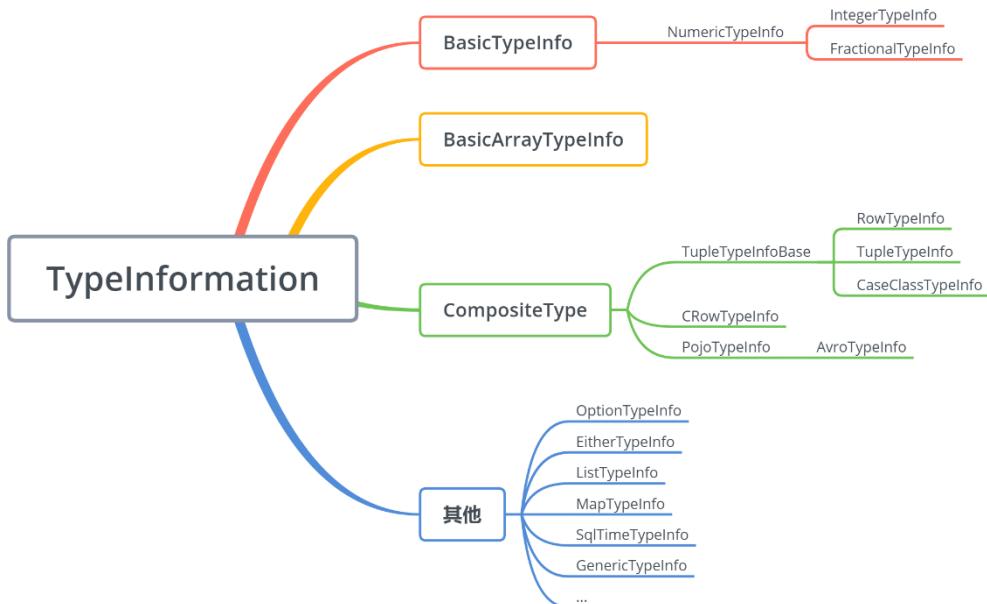


图 2: TypeInformation 类继承关系图

可以看到，图 1 和 图 2 是一一对应的，`TypeInformation` 类是描述一切类型的公共基类，它和它的所有子类必须可序列化（`Serializable`），因为类型信息将会伴随 Flink 的作业提交，被传递给每个执行节点。

由于 Flink 自己管理内存，采用了一种非常紧凑的存储格式（见[官方博文](#)），因而类型信息在整个数据处理流程中属于至关重要的元数据。

## TypeExtractoror 类型提取

Flink 内部实现了名为 `TypeExtractoror` 的类，可以利用方法签名、子类信息等蛛丝马迹，自动提取和恢复类型信息（当然也可以显式声明，即本文所介绍的内容）。

然而由于 Java 的类型擦除，自动提取并不是总是有效。因而一些情况下（例如通过 `URLClassLoader` 动态加载的类），仍需手动处理；例如下图中对 `DataSet` 变换时，使用 `.returns()` 方法声明返回类型。

这里需要说明一下，`returns()` 接受三种类型的参数：字符串描述的类名（例如 `"String"`）、`TypeHint`（接下来会讲到，用于泛型类型参数）、Java 原生 Class（例如 `String.class`）等；不过字符串形式的用法即将废弃，如果确实有必要，请使用 `Class.forName()` 等方法来解决。

```
inputDS
    .groupBy(groupKeys: _*)
    .reduce(new DistinctReduce)
    .setCombineHint(CombineHint.HASH) // use hash-combiner
    .name(newName = "distinct")
    .returns(inputDS.getType)
}
```

图 3：使用 `.returns` 方法声明返回类型

下面是 `ExecutionEnvironment` 类的 `registerType` 方法，它可以向 Flink 注册子类信息（Flink 认识父类，但不一定认识子类的一些独特特性，因而需要注册），下面是 Flink-ML 机器学习库代码的例子：

```
// Registers the different FlinkML related types for Kryo serialization
*
* @param env The Flink execution environment where the types need to be registered
*/
def registerFlinkMLTypes(env: ExecutionEnvironment): Unit = {

    // Vector types
    env.registerType(classOf[org.apache.flink.ml.math.DenseVector])
    env.registerType(classOf[org.apache.flink.ml.math.SparseVector])

    // Matrix types
    env.registerType(classOf[org.apache.flink.ml.math.DenseMatrix])
    env.registerType(classOf[org.apache.flink.ml.math.SparseMatrix])

    // Breeze Vector types
    env.registerType(classOf[breeze.linalg.DenseVector[_]])
    env.registerType(classOf[breeze.linalg.SparseVector[_]])
}
```

图 4: Flink-ML 注册子类类型信息

从下图可以看到，如果通过 `TypeExtractor.createTypeInfo(type)` 方法获取到的类型信息属于 `PojoTypeInfo` 及其子类，那么将其注册到一起；否则统一交给 Kryo 去处理，Flink 并不过问（这种情况下性能会变差）。

```
public void registerType(Class<?> type) {
    if (type == null) {
        throw new NullPointerException("Cannot register null type class.");
    }

    TypeInformation<?> typeInfo = TypeExtractor.createTypeInfo(type);

    if (typeInfo instanceof PojoTypeInfo) {
        config.registerPojoType(type);
    } else {
        config.registerKryoType(type);
    }
}
```

图 5: Flink 允许注册自定义类型

## 声明类型信息的常见手段

通过 `TypeInformation.of()` 方法，可以简单地创建类型信息对象。

- 对于非泛型的类，直接传入 `Class` 对象即可

```
Outer o = new Outer( a: 10, new Inner( x: 4L), (short) 12);
PojoTypeInfo<Outer> tpeInfo = (PojoTypeInfo<Outer>) TypeInformation.of(Outer.class);
```

图 6: class 对象作为参数

- 对于泛型类，需要借助 `TypeHint` 来保存泛型类型信息

`TypeHint` 的原理是创建匿名子类，运行时 `TypeExtractor` 可以通过 `getGenericSuperclass().getActualTypeArguments()` 方法获取保存的实际类型。

```
// serialization / deserialization schemas for writing and consuming the extra records
final TypeInformation<Tuple2<Integer, Integer>> resultType =
    TypeInformation.of(new TypeHint<Tuple2<Integer, Integer>>() {});
```

图 7: TypeHint 作为参数，保存泛型信息

- 预定义的快捷方式

例如 `BasicTypeInfo`，这个类定义了一系列常用类型的快捷方式，对于 `String`、`Boolean`、`Byte`、`Short`、`Integer`、`Long`、`Float`、`Double`、`Char` 等基本类型的类型声明，可以直接使用。

```
public class BasicTypeInfo<T> extends TypeInformation<T> implements AtomicType<T> {
    private static final long serialVersionUID = -4099520409311770L;

    public static final BasicTypeInfo<String> STRING_TYPE_INFO = new BasicTypeInfo<String>(String.class, new Class<>[][], StringSerializer.INSTANCE, StringComparator.class);
    public static final BasicTypeInfo<Boolean> BOOLEAN_TYPE_INFO = new BasicTypeInfo<Boolean>(Boolean.class, new Class<>[][], BooleanSerializer.INSTANCE, BooleanComparator.class);
    public static final BasicTypeInfo<Byte> BYTE_TYPE_INFO = new IntegerTypeInfo<Byte>(Byte.class, new Class<>[][], Integer.class, Long.class, Float.class, Double.class, Character.class);
    public static final BasicTypeInfo<Short> SHORT_TYPE_INFO = new IntegerTypeInfo<Short>(Short.class, new Class<>[][], Integer.class, Long.class, Float.class, Double.class, Character.class);
    public static final BasicTypeInfo<Integer> INT_TYPE_INFO = new IntegerTypeInfo<Integer>(Integer.class, new Class<>[][], Integer.class, Long.class, Float.class, Double.class, Character.class);
    public static final BasicTypeInfo<Long> LONG_TYPE_INFO = new IntegerTypeInfo<Long>(Long.class, new Class<>[][], Long.class, Double.class, Character.class), LongSerializer.INSTANCE, LongComparator;
    public static final BasicTypeInfo<Float> FLOAT_TYPE_INFO = new FractionalTypeInfo<Float>(Float.class, new Class<>[][], Double.class, FloatSerializer.INSTANCE, FloatComparator.class);
    public static final BasicTypeInfo<Double> DOUBLE_TYPE_INFO = new FractionalTypeInfo<Double>(Double.class, new Class<>[][], DoubleSerializer.INSTANCE, DoubleComparator.class);
    public static final BasicTypeInfo<Character> CHAR_TYPE_INFO = new BasicTypeInfo<Character>(Character.class, new Class<>[][], CharSerializer.INSTANCE, CharComparator.class);
    public static final BasicTypeInfo<Date> DATE_TYPE_INFO = new BasicTypeInfo<Date>(Date.class, new Class<>[][], DateSerializer.INSTANCE, DateComparator.class);
    public static final BasicTypeInfo<Void> VOID_TYPE_INFO = new BasicTypeInfo<Void>(Void.class, new Class<>[][], VoidSerializer.INSTANCE, ComparatorClass.NULL);
    public static final BasicTypeInfo<BigInt> BIG_INT_TYPE_INFO = new BasicTypeInfo<BigInt>(BigInt.class, new Class<>[][], BigIntSerializer.INSTANCE, BigIntComparator.class);
    public static final BasicTypeInfo<BigDecimal> BIG_DEC_TYPE_INFO = new BasicTypeInfo<BigDecimal>(BigDecimal.class, new Class<>[][], BigDecSerializer.INSTANCE, BigDecComparator.class),
}
```

图 8: BasicTypeInfo 快捷方式

例如下面是对 Row 类型各字段的类型声明，使用方法非常简明，不再需要 new XxxTypeInfo<>(很多很多参数)

```
public static final RowTypeInfo ROW_TYPE_INFO = new RowTypeInfo(
    BasicTypeInfo.INT_TYPE_INFO,
    BasicTypeInfo.STRING_TYPE_INFO,
    BasicTypeInfo.STRING_TYPE_INFO,
    BasicTypeInfo.DOUBLE_TYPE_INFO,
    BasicTypeInfo.INT_TYPE_INFO);
```

图 9: 使用 BasicTypeInfo 快捷方式来声明一行 (Row) 每个字段的类型信息

当然，如果觉得 BasicTypeInfo 还是太长，Flink 还提供了完全等价的 Types 类 (org.apache.flink.api.common.typeinfo.Types)：

```
public class Types {

    public static final BasicTypeInfo<String> STRING = BasicTypeInfo.STRING_TYPE_INFO;
    public static final BasicTypeInfo<Boolean> BOOLEAN = BasicTypeInfo.BOOLEAN_TYPE_INFO;
    public static final BasicTypeInfo<Byte> BYTE = BasicTypeInfo.BYTE_TYPE_INFO;
    public static final BasicTypeInfo<Short> SHORT = BasicTypeInfo.SHORT_TYPE_INFO;
    public static final BasicTypeInfo<Integer> INT = BasicTypeInfo.INT_TYPE_INFO;
    public static final BasicTypeInfo<Long> LONG = BasicTypeInfo.LONG_TYPE_INFO;
    public static final BasicTypeInfo<Float> FLOAT = BasicTypeInfo.FLOAT_TYPE_INFO;
    public static final BasicTypeInfo<Double> DOUBLE = BasicTypeInfo.DOUBLE_TYPE_INFO;
    public static final BasicTypeInfo<BigDecimal> DECIMAL = BasicTypeInfo.BIG_DEC_TYPE_INFO;

    public static final SqlTimeTypeInfo<Date> SQL_DATE = SqlTimeTypeInfo.DATE;
    public static final SqlTimeTypeInfo<Time> SQL_TIME = SqlTimeTypeInfo.TIME;
    public static final SqlTimeTypeInfo<Timestamp> SQL_TIMESTAMP = SqlTimeTypeInfo.TIMESTAMP;
```

图 10: Types 类

特别需要注意的是，flink-table 模块也有一个 Types 类 (org.apache.flink.table.api.Types)，用于 table 模块内部的类型定义信息，用法稍有不同。使用 IDE 的自动 import 时一定要小心：

```
RowTypeInfo rowSchema = new RowTypeInfo(
    new TypeInformation[]{Types.INT(), Types.BOOLEAN(), Types.ROW(Types.INT(), Types.DOUBLE())},
    new String[] {"f1", "f2", "f3"
});
```

图 11: flink-table 模块的 Types 类

#### 4. 自定义 TypeInfo 和 TypeInfoFactory

通过自定义 TypeInfo 为任意类提供 Flink 原生内存管理（而非 Kryo），可令存储更紧凑，运行时也更高效。

开发者在自定义类上使用 @TypeInfo 注解，随后创建相应的 TypeInfoFactory 并覆盖 createTypeInfo 方法。

注意需要继承 TypeInformation 类，为每个字段定义类型，并覆盖元数据方法，例如是否是基本类型 (isBasicType)、是否是 Tuple (isTupleType)、元数（对于一维的 Row 类型，等于字段的个数）等等，从而为 TypeExtractor 提供决策依据。

```

@TypeInformation(MyTupleTypeFactory.class)
public static class MyTuple<T0, T1> {
    // empty
}

public static class MyTupleTypeFactory extends TypeInfoFactory<MyTuple> {
    @Override
    @unchecked/
    public TypeInformation<MyTuple> createTypeInfo(Type t, Map<String, TypeInformation<?>> genericParameters) {
        return new MyTupleTypeInfo(genericParameters.get("T0"), genericParameters.get("T1"));
    }
}

public static class MyTupleTypeInfo<T0, T1> extends TypeInformation<MyTuple<T0, T1>> {
    private TypeInformation field0;
    private TypeInformation field1;

    public TypeInformation getField0() { return field0; }

    public TypeInformation getField1() { return field1; }

    public MyTupleTypeInfo(TypeInformation field0, TypeInformation field1) {
        this.field0 = field0;
        this.field1 = field1;
    }

    @Override
    public boolean isBasicType() { return false; }

    @Override
    public boolean isTupleType() { return false; }

    @Override
    public int getArity() { return 0; }

    @Override
    public int getTotalFields() { return 0; }

    @Override
    public Class<MyTuple<T0, T1>> getTypeClass() { return null; }

    @Override
    public boolean isKeyType() { return false; }
}

```

图 12：为自定义类提供类型支持（图片未展示全部字段）

更多示例，请参考 Flink 源码的 [org/apache/flink/api/java/typeutils/TypeInfoFactoryTest.java](#)

## TypeSerializer

Flink 自带了很多 TypeSerializer 子类，大多数情况下各种自定义类型都是常用类型的排列组合，因而可以直接复用：

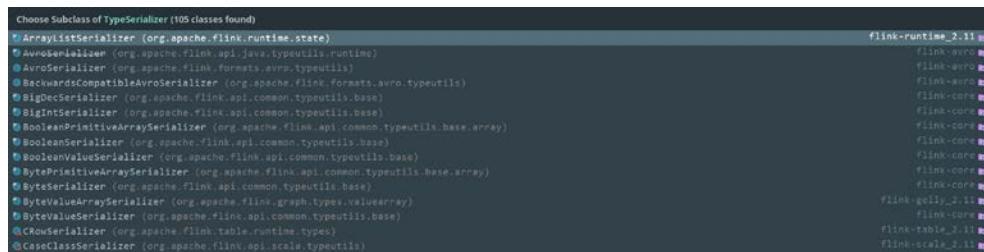


图 13：Flink 自带的 TypeSerializer 子类概览

如果不能满足，那么可以继承 TypeSerializer 及其子类以实现自己的序列化器。

## Kryo 序列化

对于 Flink 无法序列化的类型(例如用户自定义类型，没有 registerType, 也没有自定义 TypeInfo 和 TypeInfoFactory)，默认会交给 Kryo 处理。

如果 Kryo 仍然无法处理 (例如 Guava、Thrift、Protobuf 等第三方库的一些类)，有以下两种解决方案：

1. 可以强制使用 Avro 来替代 Kryo：

```
env.getConfig().enableForceAvro(); // env 代表 ExecutionEnvironment 对象，下同
```

2. 为 Kryo 增加自定义的 Serializer 以增强 Kryo 的功能：

```
env.getConfig().addDefaultKryoSerializer(Class<?> type, Class<? extends
Serializer<?>> serializerClass)
```

```
// cast because our test serializer is not typed to TestPojo
env.getExecutionConfig().addDefaultKryoSerializer(TestPojo.class, (Class) ExceptionThrowingTestSerializer.class);

TypeInformation<TestPojo> pojoType = new GenericTypeInfo<>(TestPojo.class);

// make sure that we are in fact using the KryoSerializer
assertTrue(pojoType.createSerializer(env.getExecutionConfig()) instanceof KryoSerializer);
```

图 14: 为 Kryo 增加自定义的 Serializer

以及

```
env.getConfig().registerTypeWithKryoSerializer(Class<?> type, T serializer)
```

```
env.registerTypeWithKryoSerializer(TestClass.class, new TestClassSerializer());
```

图 15: 为 Kryo 增加自定义的 Serializer

如果希望完全禁用 Kryo (100% 使用 Flink 的序列化机制), 则可以使用以下设置, 但注意一切无法处理的类都将导致异常:

```
env.getConfig().disableGenericType();
```

## 类型机制的陷阱与缺陷

金无足赤, 人无完人。Flink 内置的类型系统虽然强大而灵活, 但仍然有一些需要注意的点:

### 1. Lambda 函数的类型提取

由于 Flink 类型提取依赖于继承等机制, 而 lambda 函数比较特殊, 它是匿名的, 也没有与之相关的类, 所以其类型信息较难获取。

Eclipse 的 JDT 编译器会把 lambda 函数的泛型签名等信息写入编译后的字节码中, 而对于 javac 等常见的其他编译器, 则不会这样做, 因而 Flink 就无法获取具体类型信息了。

### 2. Kryo 的 JavaSerializer 在 Flink 下存在 Bug

推荐使用 org.apache.flink.api.java.typeutils.runtime.kryo.JavaSerializer 而非

com.esotericsoftware.kryo.serializers.JavaSerializer 以防止与 Flink 不兼容。

## 类型机制与内存管理

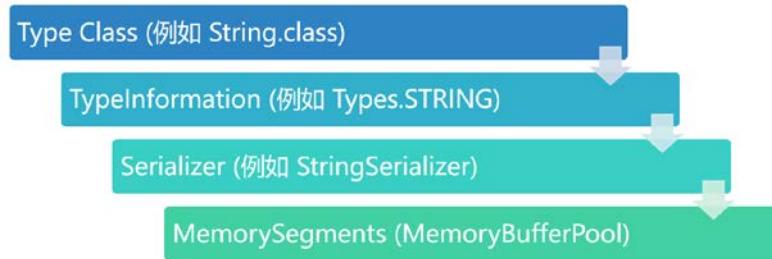


图 16: 类型信息到内存块

下面以 StringSerializer 为例，来看下 Flink 是如何紧凑管理内存的：

```
@Override  
public void serialize(String record, DataOutputView target) throws IOException {  
    StringValue.writeString(record, target);  
}
```

图 17: StringSerializer 类的 serialize() 方法

下面是具体的序列化过程：

```
public static final void writeString(CharSequence cs, DataOutput out) throws IOException {
    if (cs != null) {
        // the length we write is offset by one, because a length of zero indicates a null value
        int lenToWrite = cs.length() + 1;
        if (lenToWrite < 0) {
            throw new IllegalArgumentException("CharSequence is too long.");
        }

        // write the length, variable-length encoded
        while (lenToWrite >= HIGH_BIT) {
            out.write( b: lenToWrite | HIGH_BIT);
            lenToWrite >>>= 7;
        }
        out.write(lenToWrite);

        // write the char data, variable length encoded
        for (int i = 0; i < cs.length(); i++) {
            int c = cs.charAt(i);

            while (c >= HIGH_BIT) {
                out.write( b: c | HIGH_BIT);
                c >>>= 7;
            }
            out.write(c);
        }
    } else {
        out.write( b: 0);
    }
}
```

图 18: String 对象的序列化过程

可以看到，Flink 对于内存管理是非常细致的，层次分明，代码也容易理解。

# 深度剖析阿里巴巴对 Apache Flink 的优化与改进

本文主要从两个层面深度剖析：阿里巴巴对 Flink 究竟做了哪些优化？

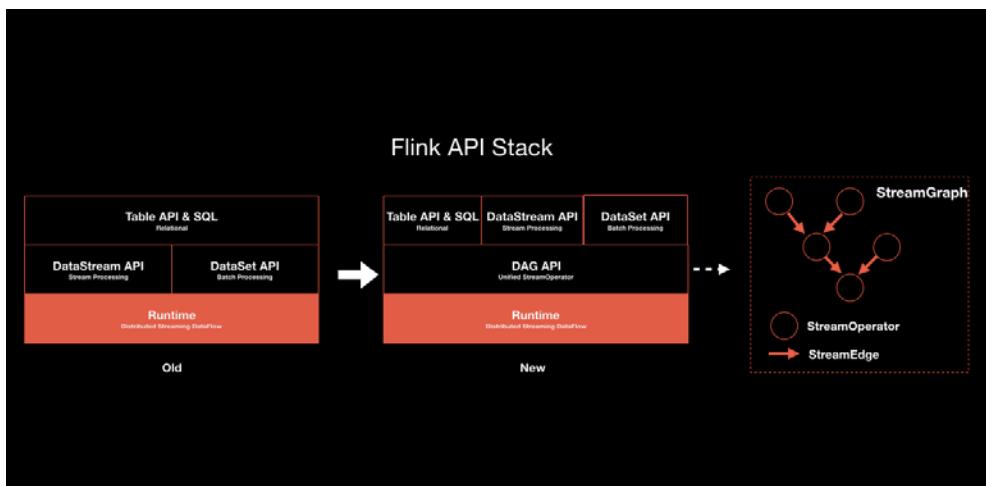
## 取之开源，用之开源

### 一、SQL 层

为了能够真正做到用户根据自己的业务逻辑开发一套代码，能够同时运行在多种不同的场景，Flink 首先需要给用户提供一个统一的 API。在经过一番调研之后，阿里巴巴实时计算认为 SQL 是一个非常适合的选择。在批处理领域，SQL 已经经历了几十年的考验，是公认的经典。在流计算领域，近年来也不断有流表二象性、流是表的 ChangeLog 等理论出现。在这些理论基础之上，阿里巴巴提出了动态表的概念，使得流计算也可以像批处理一样使用 SQL 来描述，并且逻辑等价。这样一来，用户就可以使用 SQL 来描述自己的业务逻辑，相同的查询语句在执行时可以是一个批处理任务，也可以是一个高吞吐低延迟的流计算任务，甚至是先使用批处理技术进行历史数据的计算，然后自动的转成流计算任务处理最新的实时数据。在这种声明式的 API 之下，引擎有了更多的选择和优化空间。接下来，我们将介绍其中几个比较重要的优化。

首先是对 SQL 层的技术架构进行升级和替换。调研过 Flink 或者使用过 Flink 的开发者应该知道，

Flink 有两套基础的 API，一套是 DataStream，另一套是 DataSet。DataStream API 是针对流式处理的用户提供，DataSet API 是针对批处理用户提供，但是这两套 API 的执行路径是完全不一样的，甚至需要生成不同的 Task 去执行。Flink 原生的 SQL 层在经过一系列优化之后，会根据用户希望是批处理还是流处理的不同选择，去调用 DataSet 或者是 DataStream API。这就会造成用户在日常开发和优化中，经常要面临两套几乎完全独立的技术栈，很多事情可能需要重复的去做两遍。这样也会导致在一边的技术栈上做的优化，另外一边就享受不到。因此阿里巴巴在 SQL 层提出了全新的 Quyer Processor，它主要包括一个流和批可以尽量做到复用的优化层（Query Optimizer）以及基于相同接口的算子层（Query Executor）。这样一来，80%以上的工作可以做到两边复用，比如一些公共的优化规则，基础数据结构等等。同时，流和批也会各自保留自己一些独特的优化和算子，以满足不同的作业行为。



在 SQL 层的技术架构统一之后，阿里巴巴开始寻求一种更高效的基础数据结构，以便让 Blink 在 SQL 层的执行更加高效。在原生 Flink SQL 中，都统一使用了一种叫 Row 的数据结构，它完全由 JAVA 的一些对象构成关系数据库中的一行。假如现在的一行数据由一个整型，一个浮点型以及一个字符串组成，那么 Row 当中就会包含一个 JAVA 的 Integer、Double 和 String。众所周知，这些 JAVA 的对象在堆内有不少的额外开销，同时在访问这些数据的过程中也会引入不必要的装箱拆箱操作。基于这些问题，阿里巴巴提出了一种全新的数据结构 BinaryRow，它和原来的 Row 一样也是表示一个关系数据中的一行，但与之不同的是，它完全使用二进制数据来存储这些数据。在上述例子中，三个不同类型的字段统一由 JAVA 的 byte[] 来表示。这会带来诸多好处：

- 首先在存储空间上，去掉了许多无谓的额外消耗，使得对象的存储更为紧凑；

- 其次在和网络或者状态存储打交道的时候，也可以省略掉很多不必要的序列化反序列化开销；
- 最后在去掉各种不必要的装箱拆箱操作之后，整个执行代码对 GC 也更加友好。

通过引入这样一个高效的基础数据结构，整个 SQL 层的执行效率得到了一倍以上的提升。

在算子的实现层面，阿里巴巴引入了更广范围的代码生成技术。得益于技术架构和基础数据结构的统一，很多代码生成技术得以达到更广范围的复用。同时由于 SQL 的强类型保证，用户可以预先知道算子需要处理的数据的类型，从而可以生成更有针对性更高效的执行代码。在原生 Flink SQL 中，只有类似  $a > 2$  或者  $c + d$  这样的简单表达式才会应用代码生成技术，在阿里巴巴优化之后，有一些算子会进行整体的代码生成，比如排序、聚合等。这使得用户可以更加灵活的去控制算子的逻辑，也可以直接将最终运行代码嵌入到类当中，去掉了昂贵的函数调用开销。一些应用代码生成技术的基础数据结构和算法，比如排序算法，基于二进制数据的 `HashMap` 等，也可以在流和批的算子之间进行共享和复用，让用户真正享受到了技术和架构的统一带来的好处。在针对批处理的某些场景进行数据结构或者算法的优化之后，流计算的性能也能够得到提升。接下来，我们聊聊阿里巴巴在 Runtime 层对 Flink 又大刀阔斧地进行了哪些改进。

## 二、Runtime 层

为了让 Flink 在 Alibaba 的大规模生产环境中生根发芽，实时计算团队如期遇到了各种挑战，首当其冲的就是如何让 Flink 与其他集群管理系统进行整合。Flink 原生集群管理模式尚未完善，也无法原生地使用其他其他相对成熟的集群管理系统。基于此，一系列棘手的问题接连浮现：多租户之间资源如何协调？如何动态的申请和释放资源？如何指定不同资源类型？

为了解决这个问题，实时计算团队经历大量的调研与分析，最终选择的方案是改造 Flink 资源调度系统，让 Flink 可以原生地跑在 Yarn 集群之上；并且重构 Master 架构，让一个 Job 对应一个 Master，从此 Master 不再是集群瓶颈。以此为契机，阿里巴巴和社区联手推出了全新的 Flp-6 架构，让 Flink 资源管理变成可插拔的架构，为 Flink 的可持续发展打下了坚实的基础。如今 Flink 可以无缝运行在 YARN、Mesos 和 K8s 之上，正是这个架构重要性的有力说明。

解决了 Flink 集群大规模部署问题后，接下来的就是可靠和稳定性，为了保证 Flink 在生产环境中的高可用，阿里巴巴着重改善了 Flink 的 FailOver 机制。首先是 Master 的 FailOver，Flink 原生的 Master FailOver 会重启所有的 Job，改善后 Master 任何 FailOver 都不会影响 Job 的正常运行；其次引入了 Region-based 的 Task FailOver，尽量减少任何 Task 的 FailOver 对用户造成的影响。有了这些改进的保驾护航，阿里巴巴的大量业务方开始把实时计算迁移到 Flink 上运行。

Stateful Streaming 是 Flink 的最大亮点，基于 Chandy-Lamport 算法的 Checkpoint 机制让 Flink 具备 Exactly Once 一致性的计算能力，但在早期 Flink 版本中 Checkpoint 的性能在大规模数据量下存在一定瓶颈，阿里巴巴也在 Checkpoint 上进行了大量改进，比如：

- 增量 Checkpoint 机制：阿里巴巴生产环境中遇到大 JOB 有几十 TB State 是常事，做一次全量 CP 地动山摇，成本很高，因此阿里巴巴研发了增量 Checkpoint 机制，从此之后 CP 从暴风骤雨变成了细水长流；
- Checkpoint 小文件合并：都是规模惹的祸，随着整个集群 Flink JOB 越来越多，CP 文件数也水涨船高，最后压的 HDFS NameNode 不堪重负，阿里巴巴通过把若干 CP 小文件合并成一个大文件的组织方式，最终把 NameNode 的压力减少了几十倍。

虽然说所有的数据可以放在 State 中，但由于一些历史的原因，用户依然有一些数据需要存放在像 HBase 等一些外部 KV 存储中，用户在 Flink Job 需要访问这些外部的数据，但是由于 Flink 一直都是单线程处理模型，导致访问外部数据的延迟成为整个系统的瓶颈，显然异步访问是解决这个问题的直接手段，但是让用户在 UDF 中写多线程同时还要保证 ExactlyOnce 语义，却并非易事。阿里巴巴在 Flink 中提出了 AsyncOperator，让用户在 Flink JOB 中写异步调用和写“Hello Word”一样简单，这个让 Flink Job 的吞吐有了很大的飞跃。

Flink 在设计上是一套批流统一的计算引擎，在使用过快如闪电的流计算之后，批用户也开始有兴趣入住 Flink 小区。但批计算也带来了新的挑战，首先在任务调度方面，阿里巴巴引入了更加灵活的调度机制，能够根据任务之间的依赖关系进行更加高效的调度；其次就是数据 Shuffle，Flink 原生的 Shuffle Service 和 TM 绑定，任务执行完之后要依旧保持 TM 无法释放资源；还有就是原有的 Batch shuffle 没有对文件进行合并，所以基本无法在生产中使用。阿里巴巴开发了 Yarn Shuffle Service 功能的同时解决了以上两个问题。在开发 Yarn Shuffle Service 的时候，阿里巴巴发现开发一套新的 Shuffle Service 非常不便，需要侵入 Flink 代码的很多地方，为了让其他开发者方便的扩展不同 Shuffle，阿里巴巴同时改造了 Flink Shuffle 架构，让 Flink 的 Shuffle 变成可插拔的架构。目前阿里巴巴的搜索业务已经在使用 Flink Batch Job，并且已经开始服务于生产。

经过 3 年多打磨，Blink 已经在阿里巴巴开始茁壮生长，但是对 Runtime 的优化和改进是永无止境的，一大波改进和优化正在路上。



扫描微信二维码  
关注阿里技术公众号



扫描钉钉二维码  
进入Flink专刊读者群



扫描微信二维码  
关注InfoQ公众号



扫描微信二维码  
关注AI前线公众号

# FLINK FORWARD



The Apache Flink® Conference

不仅仅是流计算 | More Than Streaming

2018年12月20日-21日 · 北京

Flink Forward 是由 Apache 官方授权，Apache Flink China 社区支持，旨在汇集大数据领域一流人才参加的国际型技术会议。通过参会不仅可以了解到 Flink 社区的最新动态和发展计划，还可以了解到国内外一线大厂围绕 Flink 生态的生产实践经验，是大数据从业人员不可错过的盛会。

Flink Forward 过去只在德国柏林、美国旧金山举办。今年将由阿里巴巴作为独家承办方将这一盛会引入中国，共建生态。

Organized by  
 Alibaba Group

Created by  
 dataArtisans



# EXPERTS RECOMMEND

## 专家推荐

蒋晓伟 / 阿里巴巴 研究员

Apache Flink 最初作为一套优秀流式处理引擎在阿里集团落地，目前已经作为阿里集团最大的实时处理引擎，支持了包括天猫、淘宝、阿里云等所有阿里集团核心业务部门。今年双十一期间 Flink 引擎完美支撑了双十一峰值高达 17 亿的流量洪峰，为整个阿里集团平稳度过双十一立下汗马功劳。未来，Apache Flink 定位是一套兼具流、批、机器学习等多种计算功能的大数据引擎，更好地服务到中国企业实现大数据升级与转型。后续我们有理由期待 Apache Flink 将成为开源大数据最闪耀的新星！

鞠大升 / 美团点评 研究员

Apache Flink 优雅的架构设计，解决了大数据流式计算领域的核心难题，包括流批统一处理、流式状态管理和精确计算保证等，为流式计算深入应用注入新的生命力。

堵俊平 / 腾讯云 大数据基础研发负责人

我很早之前就接触过 Apache Flink，它优雅的架构和相对其他流计算框架的独具特色的设计让我印象深刻。一直以来，Apache Flink 社区发展的很快，已成为流计算领域最受欢迎的项目之一。期待 Flink 技术在 Apache 社区的支持下持续发展，不断完善生态和拓展更多的应用场景。

孟文瑞 / Uber 资深软件工程师

Apache Flink 作为新一代数据处理引擎，提供了丰富的窗口机制以及简洁完善的 API，能够更好地支持不同的数据流应用场景。与其他引擎比较，我们看到 Flink 在硬件资源、处理性能以及开发周期方面的优势。祝愿 Flink 可以被更多人所使用，社区可以更加的成熟。

滕昱 / DellEMC 软件开发总监

Apache Flink 天生为流式计算而生，使用同一套 API 解决了流式 / 批处理两种处理方式的需求。其设计理念与流式存储开源项目 Pravega 完美契合。希望今后双方保持紧密合作、共同发展，完成从底层存储到上层计算的统一大数据流水线架构。

师锐 / 字节跳动 计算架构技术负责人

自 2017 年以来，Apache Flink 在字节跳动逐步取代原有的 JStorm 框架，成为公司内部流式数据处理引擎的唯一标准。Flink 兼具高吞吐，低延迟，并支持 Exactly Once 语义等特性，在公司多个亿级用户规模产品的模型训练、广告系统、实时数据分析等领域发挥了极为重要的作用。

罗李

Apache Flink 在支持高吞吐，低延迟和 Exactly Once 等特性上非常适合现代业务对计算引擎苛刻的要求，Flink 将这些支持这些高端特性的引擎包装在其简单易用的接口下，为业务应用开发提供了极大的便利。同时其对 CEP 的复杂语义定义的支持，对实时数据集上的 SQL 支持，都为满足更复杂的业务需求提供了简单直观的 API 支持。Flink 社区也非常活跃，参与建设的机构和个人也越来越多，相信 Flink 将来的发展一定会更加成功。

时金魁 / 华为 技术专家

Apache Flink 是个优秀的流处理框架，DataFlow 模型和 StreamSQL 在一开始就走在对的路上，在设计良好的 Runtime 上可以构造更多可能。我们团队（华为云实时流计算服务）从 2015 年投入 Flink 至今，开发了 Serverless 化的 Flink 服务，新增诸多 Flink 高级特性，始终相信 Flink。期待 Flink 社区走的更扎实，商业更成功。