

# 零基础入门： 从0到1学会 Apache Flink

30天成长为Flink大神的官方教程

- 
- | 多位 PMC 及核心贡献者出品
  - | 建立系统框架体系
  - | 最详细的免费教程



扫一扫二维码图案，关注我吧



开发者社区



阿里云实时计算



实时计算交流钉钉群



Flink 社区微信公众号

## 目录

Apache Flink 进阶 (一): Runtime 核心机制剖析	4
Apache Flink 进阶 (二): 时间属性深度解析	18
Apache Flink 进阶 (三): Checkpoint 原理剖析与应用实践	30
Apache Flink 进阶 (四): Flink on Yarn/K8s 原理剖析及实践	41
Apache Flink 进阶 (五): 数据类型和序列化	60
Apache Flink 进阶 (六): Flink 作业执行深度解析	71
Apache Flink 进阶 (七): 网络流控及反压剖析	88
Apache Flink 进阶 (八): 详解 Metrics 原理与实战	112
Apache Flink 进阶 (九): Flink Connector 开发	125
Apache Flink 进阶 (十): Flink State 最佳实践	141
Apache Flink 进阶 (十一): TensorFlow On Flink	149
Apache Flink 进阶 (十二): 深度探索 Flink SQL	159
Apache Flink 进阶 (十三): Python API 应用实践	181

# Apache Flink 进阶（一）：Runtime 核心机制剖析

作者：高攀（云骞）

阿里巴巴技术专家

简介：Flink 的整体架构如图 1 所示。Flink 是可以运行在多种不同的环境中的，例如，它可以通过单进程多线程的方式直接运行，从而提供调试的能力。它也可以运行在 Yarn 或者 K8S 这种资源管理系统上面，也可以在各种云环境中执行。

## 1. 综述

本文主要介绍 Flink Runtime 的作业执行的核心机制。首先介绍 Flink Runtime 的整体架构以及 Job 的基本执行流程，然后介绍在这个过程，Flink 是怎么进行资源管理、作业调度以及错误恢复的。最后，本文还将简要介绍 Flink Runtime 层当前正在进行的一些工作。

## 2. Flink Runtime 整体架构

Flink 的整体架构如图 1 所示。Flink 是可以运行在多种不同的环境中的，例如，它可以通过单进程多线程的方式直接运行，从而提供调试的能力。它也可以运行在 Yarn 或者 K8S 这种资源管理系统上面，也可以在各种云环境中执行。

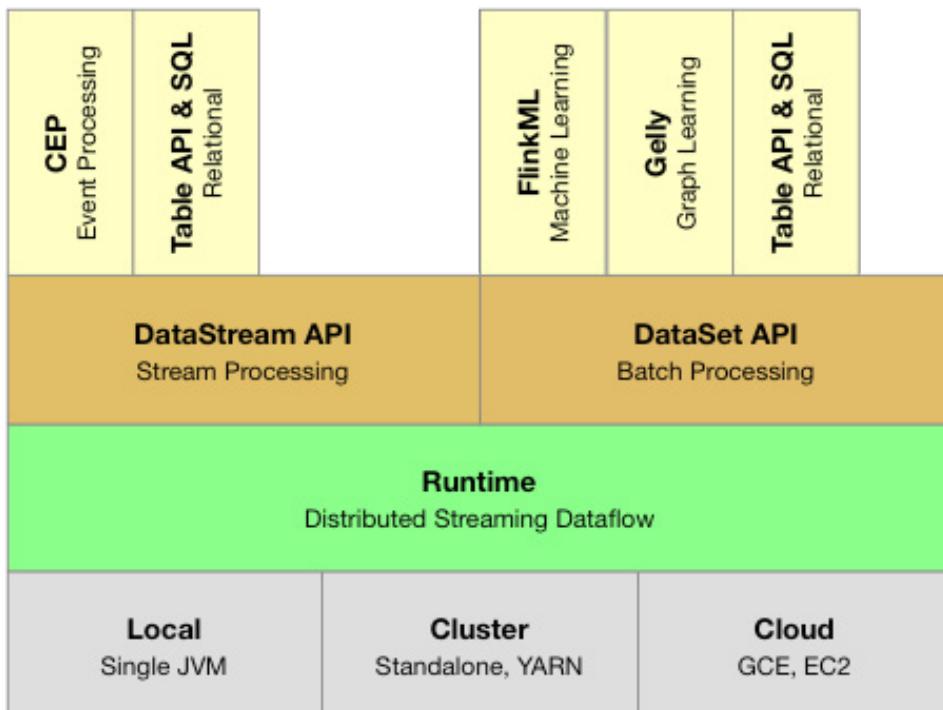


图 1 Flink 的整体架构，其中 Runtime 层对不同的执行环境提供了一套统一的分布式执行引擎

针对不同的执行环境，Flink 提供了一套统一的分布式作业执行引擎，也就是 Flink Runtime 这层。Flink 在 Runtime 层之上提供了 DataStream 和 DataSet 两套 API，分别用来编写流作业与批作业，以及一组更高级的 API 来简化特定作业的编写。本文主要介绍 Flink Runtime 层的整体架构。

Flink Runtime 层的主要架构如图 2 所示，它展示了一个 Flink 集群的基本结构。Flink Runtime 层的整个架构主要是在 FLIP-6 中实现的，整体来说，它采用了标准 master-slave 的结构，其中左侧白色圈中的部分即是 master，它负责管理整个集群中的资源和作业；而右侧的两个 TaskExecutor 则是 Slave，负责提供具体的资源并实际执行作业。

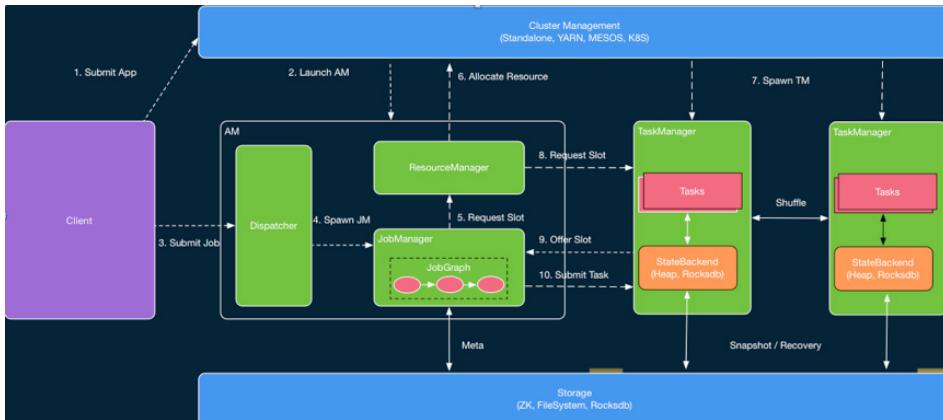


图 2 Flink 集群的基本结构。Flink Runtime 层采用了标准的 master-slave 架构

其中，Master 部分又包含了三个组件，即 Dispatcher、ResourceManager 和 JobManager。其中，Dispatcher 负责接收用户提供的作业，并且负责为这个新提交的作业拉起一个新的 JobManager 组件。ResourceManager 负责资源的管理，在整个 Flink 集群中只有一个 ResourceManager。JobManager 负责管理作业的执行，在一个 Flink 集群中可能有多个作业同时执行，每个作业都有自己的 JobManager 组件。这三个组件都包含在 AppMaster 进程中。

基于上述结构，当用户提交作业的时候，提交脚本会首先启动一个 Client 进程负责作业的编译与提交。它首先将用户编写的代码编译为一个 JobGraph，在这个过程，它还会进行一些检查或优化等工作，例如判断哪些 Operator 可以 Chain 到同一个 Task 中。然后，Client 将产生的 JobGraph 提交到集群中执行。此时有两种情况，一种是类似于 Standalone 这种 Session 模式，AM 会预先启动，此时 Client 直接与 Dispatcher 建立连接并提交作业即可。另一种是 Per-Job 模式，AM 不会预先启动，此时 Client 将首先向资源管理系统（如 Yarn、K8S）申请资源来启动 AM，然后再向 AM 中的 Dispatcher 提交作业。

当作业到 Dispatcher 后，Dispatcher 会首先启动一个 JobManager 组件，然后 JobManager 会向 ResourceManager 申请资源来启动作业中具体的任务。这

时根据 Session 和 Per-Job 模式的区别, TaskExecutor 可能已经启动或者尚未启动。如果是前者, 此时 ResourceManager 中已有记录了 TaskExecutor 注册的资源, 可以直接选取空闲资源进行分配。否则, ResourceManager 也需要首先向外部资源管理系统申请资源来启动 TaskExecutor, 然后等待 TaskExecutor 注册相应资源后再继续选择空闲资源进程分配。目前 Flink 中 TaskExecutor 的资源是通过 Slot 来描述的, 一个 Slot 一般可以执行一个具体的 Task, 但在一些情况下也可以执行多个相关联的 Task, 这部分内容将在下文进行详述。ResourceManager 选择到空闲的 Slot 之后, 就会通知相应的 TM “将该 Slot 分配给 JobManager XX”, 然后 TaskExecutor 进行相应的记录后, 会向 JobManager 进行注册。JobManager 收到 TaskExecutor 注册上来的 Slot 后, 就可以实际提交 Task 了。

TaskExecutor 收到 JobManager 提交的 Task 之后, 会启动一个新的线程来执行该 Task。Task 启动后就会开始进行预先指定的计算, 并通过数据 Shuffle 模块互相交换数据。

以上就是 Flink Runtime 层执行作业的基本流程。可以看出, Flink 支持两种不同的模式, 即 Per-job 模式与 Session 模式。如图 3 所示, Per-job 模式下整个 Flink 集群只执行单个作业, 即每个作业会独享 Dispatcher 和 ResourceManager 组件。此外, Per-job 模式下 AppMaster 和 TaskExecutor 都是按需申请的。因此, Per-job 模式更适合运行执行时间较长的大作业, 这些作业对稳定性要求较高, 并且对申请资源的时间不敏感。与之对应, 在 Session 模式下, Flink 预先启动 AppMaster 以及一组 TaskExecutor, 然后在整个集群的生命周期中会执行多个作业。可以看出, Session 模式更适合规模小, 执行时间短的作业。

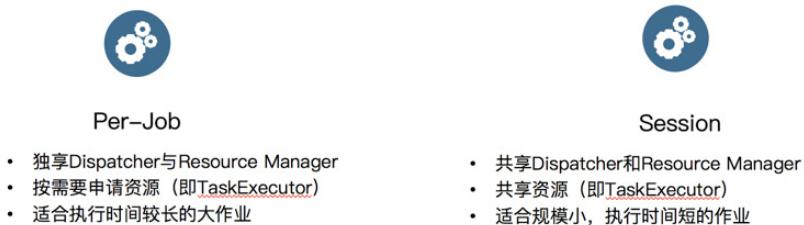


图 3 Flink Runtime 支持两种作业执行的模式

### 3. 资源管理与作业调度

本节对 Flink 中资源管理与作业调度的功能进行更深入的说明。实际上，作业调度可以看做是对资源和任务进行匹配的过程。如上节所述，在 Flink 中，资源是通过 Slot 来表示的，每个 Slot 可以用来执行不同的 Task。而在另一端，任务即 Job 中实际的 Task，它包含了待执行的用户逻辑。调度的主要目的就是为了给 Task 找到匹配的 Slot。逻辑上来说，每个 Slot 都应该有一个向量来描述它所能提供的各种资源的量，每个 Task 也需要相应的说明它所需要的各种资源的量。但是实际上在 1.9 之前，Flink 是不支持细粒度的资源描述的，而是统一的认为每个 Slot 提供的资源和 Task 需要的资源都是相同的。从 1.9 开始，Flink 开始增加对细粒度的资源匹配的支持的实现，但这部分功能目前仍在完善中。

作业调度的基础是首先提供对资源的管理，因此我们首先来看下 Flink 中资源管理的实现。如上文所述，Flink 中的资源是由 TaskExecutor 上的 Slot 来表示的。如图 4 所示，在 ResourceManager 中，有一个子组件叫做 SlotManager，它维护了当前集群中所有 TaskExecutor 上的 Slot 的信息与状态，如该 Slot 在哪个 TaskExecutor 中，该 Slot 当前是否空闲等。当 JobManger 来为特定 Task 申请资源的时候，根据当前是 Per-job 还是 Session 模式，ResourceManager 可能会去申请资源来启动新的 TaskExecutor。当 TaskExecutor 启动之后，它会通过服务发现找到当前活跃的 ResourceManager 并进行注册。在注册信息中，会包含

该 TaskExecutor 中所有 Slot 的信息。ResourceManager 收到注册信息后，其中的 SlotManager 就会记录下相应的 Slot 信息。当 JobManager 为某个 Task 来申请资源时，SlotManager 就会从当前空闲的 Slot 中按一定规则选择一个空闲的 Slot 进行分配。当分配完成后，如第 2 节所述，RM 会首先向 TaskManager 发送 RPC 要求将选定的 Slot 分配给特定的 JobManager。TaskManager 如果还没有执行过该 JobManager 的 Task 的话，它需要首先向相应的 JobManager 建立连接，然后发送提供 Slot 的 RPC 请求。在 JobManager 中，所有 Task 的请求会缓存到 SlotPool 中。当有 Slot 被提供之后，SlotPool 会从缓存的请求中选择相应的请求并结束相应的请求过程。

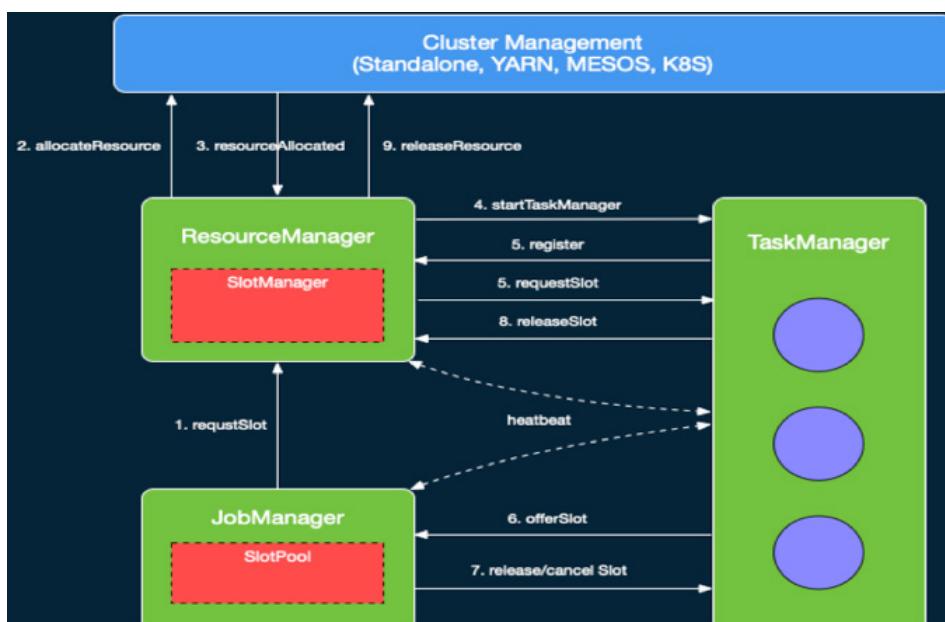


图 4 Flink 中资源管理功能各模块交互关系

当 Task 结束之后，无论是正常结束还是异常结束，都会通知 JobManager 相应的结束状态，然后在 TaskManager 端将 Slot 标记为已占用但未执行任务的状态。JobManager 会首先将相应的 Slot 缓存到 SlotPool 中，但不会立即释放。这种方式避免了如果将 Slot 直接还给 ResourceManager，在任务异常结束之后需

要重启时，需要立刻重新申请 Slot 的问题。通过延时释放，Failover 的 Task 可以尽快调度回原来的 TaskManager，从而加快 Failover 的速度。当 SlotPool 中缓存的 Slot 超过指定的时间仍未使用时，SlotPool 就会发起释放该 Slot 的过程。与申请 Slot 的过程对应，SlotPool 会首先通知 TaskManager 来释放该 Slot，然后 TaskExecutor 通知 ResourceManager 该 Slot 已经被释放，从而最终完成释放的逻辑。

除了正常的通信逻辑外，在 ResourceManager 和 TaskExecutor 之间还存在定时的心跳消息来同步 Slot 的状态。在分布式系统中，消息的丢失、错乱不可避免，这些问题会在分布式的组件中引入不一致状态，如果没有定时消息，那么组件无法从这些不一致状态中恢复。此外，当组件之间长时间未收到对方的心跳时，就会认为对应的组件已经失效，并进入到 Failover 的流程。

在 Slot 管理基础上，Flink 可以将 Task 调度到相应的 Slot 当中。如上文所述，Flink 尚未完全引入细粒度的资源匹配，默认情况下，每个 Slot 可以分配给一个 Task。但是，这种方式在某些情况下会导致资源利用率不高。如图 5 所示，假如 A、B、C 依次执行计算逻辑，那么给 A、B、C 分配单独的 Slot 就会导致资源利用率不高。为了解决这一问题，Flink 提供了 Share Slot 的机制。如图 5 所示，基于 Share Slot，每个 Slot 中可以部署来自不同 JobVertex 的多个任务，但是不能部署来自同一个 JobVertex 的 Task。如图 5 所示，每个 Slot 中最多可以部署同一个 A、B 或 C 的 Task，但是可以同时部署 A、B 和 C 的各一个 Task。当单个 Task 占用资源较少时，Share Slot 可以提高资源利用率。此外，Share Slot 也提供了一种简单的保持负载均衡的方式。

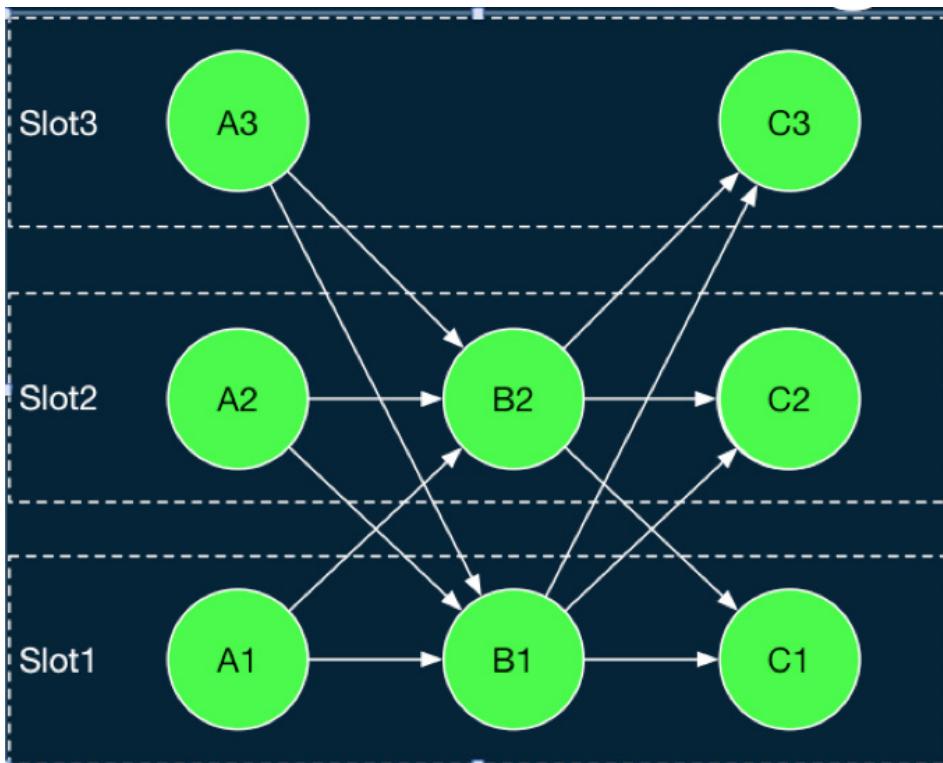


图 5 Flink Share Slot 示例。使用 Share Slot 可以在每个 Slot 中部署来自不同 JobVertex 的多个 Task

基于上述 Slot 管理和分配的逻辑，JobManager 负责维护作业中 Task 执行的状态。如上文所述，Client 端会向 JobManager 提交一个 JobGraph，它代表了作业的逻辑结构。JobManager 会根据 JobGraph 按并发展开，从而得到 JobManager 中关键的 ExecutionGraph。ExecutionGraph 的结构如图 5 所示，与 JobGraph 相比，ExecutionGraph 中对于每个 Task 与中间结果等均创建了对应的对象，从而可以维护这些实体的信息与状态。

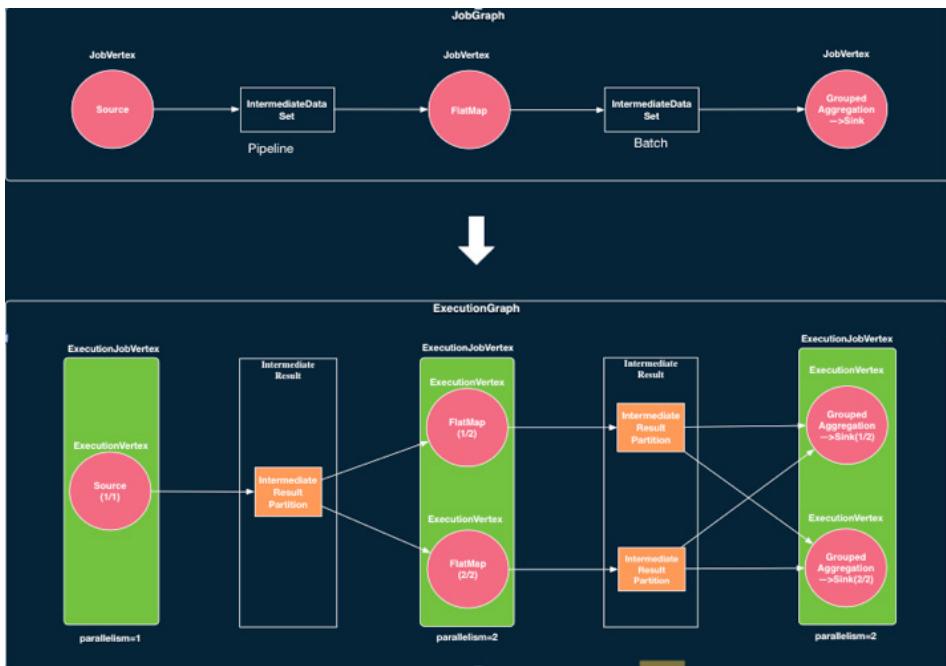


图 6 Flink 中的 JobGraph 与 ExecutionGraph。ExecutionGraph 是 JobGraph 按并发展开所形成的，它是 JobMaster 中的核心数据结构

在一个 Flink Job 中是包含多个 Task 的，因此另一个关键的问题是在 Flink 中按什么顺序来调度 Task。如图 7 所示，目前 Flink 提供了两种基本的调度逻辑，即 Eager 调度与 Lazy From Source。Eager 调度如其名所示，它会在作业启动时申请资源将所有的 Task 调度起来。这种调度算法主要用来调度可能没有终止的流作业。与之对应，Lazy From Source 则是从 Source 开始，按拓扑顺序来进行调度。简单来说，Lazy From Source 会先调度没有上游任务的 Source 任务，当这些任务执行完成时，它会将输出数据缓存到内存或者写入到磁盘中。然后，对于后续的任务，当它的前驱任务全部执行完成后，Flink 就会将这些任务调度起来。这些任务会从读取上游缓存的输出数据进行自己的计算。这一过程继续进行直到所有的任务完成计算。

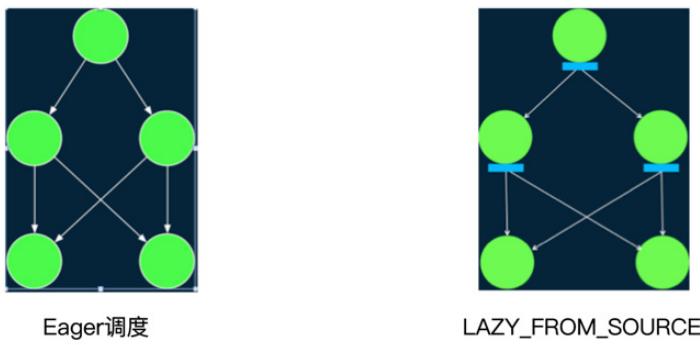


图 7 Flink 中两种基本的调度策略。其中 Eager 调度适用于流作业，而 Lazy From Source 适用于批作业

## 4. 错误恢复

在 Flink 作业的执行过程中，除正常执行的流程外，还有可能由于环境等原因导致各种类型的错误。整体上来说，错误可能分为两大类：Task 执行出现错误或 Flink 集群的 Master 出现错误。由于错误不可避免，为了提高可用性，Flink 需要提供自动错误恢复机制来进行重试。

对于第一类 Task 执行错误，Flink 提供了多种不同的错误恢复策略。如图 8 所示，第一种策略是 Restart-all，即直接重启所有的 Task。对于 Flink 的流任务，由于 Flink 提供了 Checkpoint 机制，因此当任务重启后可以直接从上次的 Checkpoint 开始继续执行。因此这种方式更适合于流作业。第二类错误恢复策略是 Restart-individual，它只适用于 Task 之间没有数据传输的情况。这种情况下，我们可以直接重启出错的任务。

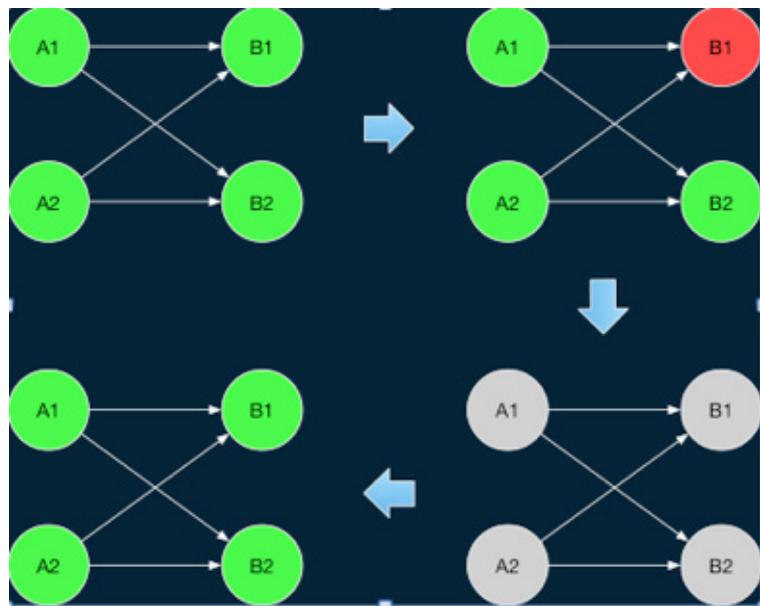


图 8 Restart-all 错误恢复策略示例。该策略会直接重启所有的 Task

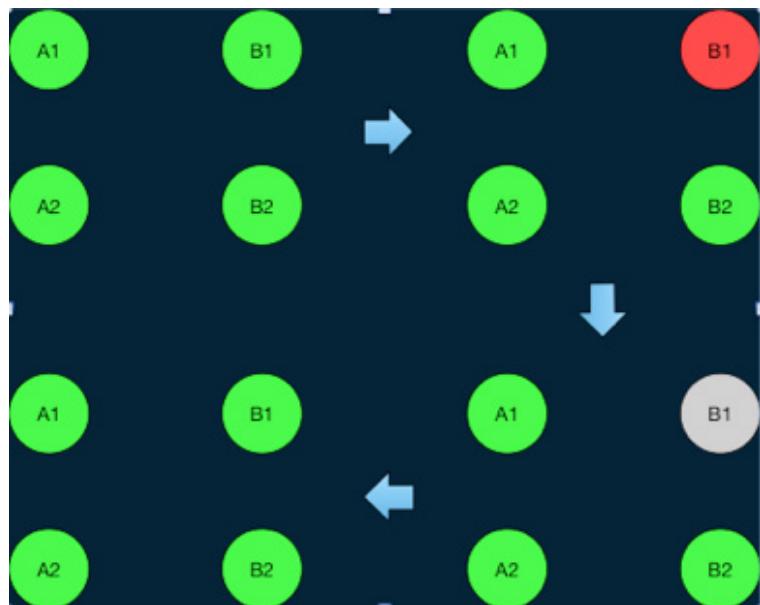


图 9 Restart-individual 错误恢复策略示例。该策略只适用于 Task 之间不需要数据传输的作业，对于这种作业可以只重启出现错误的 Task

由于 Flink 的批作业没有 Checkpoint 机制，因此对于需要数据传输的作业，直接重启所有 Task 会导致作业从头计算，从而导致一定的性能问题。为了增强对 Batch 作业，Flink 在 1.9 中引入了一种新的 Region-Based 的 Failover 策略。在一个 Flink 的 Batch 作业中 Task 之间存在两种数据传输方式，一种是 Pipeline 类型的方式，这种方式上下游 Task 之间直接通过网络传输数据，因此需要上下游同时运行；另外一种是 Blocking 类型的试，如上节所述，这种方式下，上游的 Task 会首先将数据进行缓存，因此上下游的 Task 可以单独执行。基于这两种类型的传输，Flink 将 ExecutionGraph 中使用 Pipeline 方式传输数据的 Task 的子图叫做 Region，从而将整个 ExecutionGraph 划分为多个子图。可以看出，Region 内的 Task 必须同时重启，而不同 Region 的 Task 由于在 Region 边界存在 Blocking 的边，因此，可以单独重启下游 Region 中的 Task。

基于这一思路，如果某个 Region 中的某个 Task 执行出现错误，可以分两种情况进行考虑。如图 8 所示，如果是由于 Task 本身的问题发生错误，那么可以只重启该 Task 所属的 Region 中的 Task，这些 Task 重启之后，可以直接拉取上游 Region 缓存的输出结果继续进行计算。

另一方面，如图如果错误是由于读取上游结果出现问题，如网络连接中断、缓存上游输出数据的 TaskExecutor 异常退出等，那么还需要重启上游 Region 来重新产生相应的数据。在这种情况下，如果上游 Region 输出的数据分发方式不是确定性的（如 KeyBy、Broadcast 是确定性的分发方式，而 Rebalance、Random 则不是，因为每次执行会产生不同的分发结果），为保证结果正确性，还需要同时重启上游 Region 所有的下游 Region。

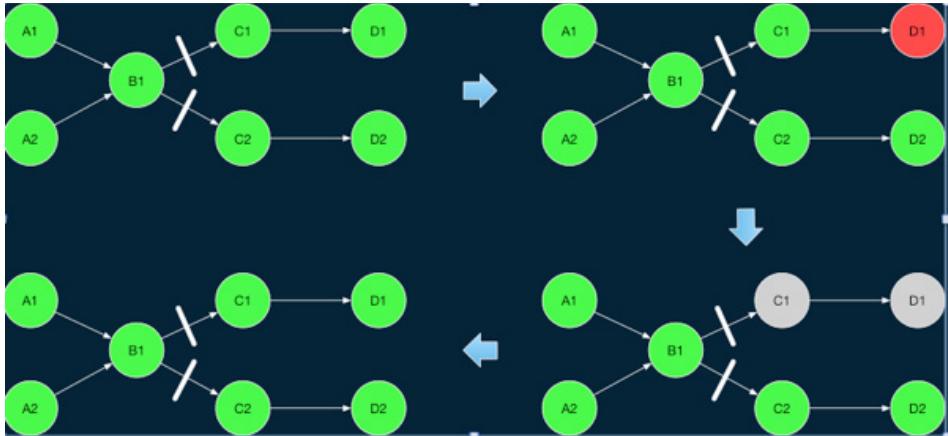


图 10 Region-based 错误恢复策略示例一。如果是由于下游任务本身导致的错误，可以只重启下游对应的 Region

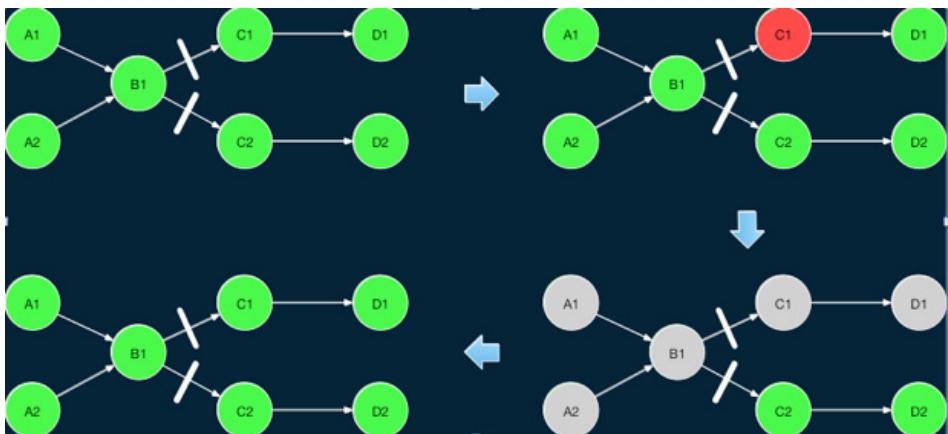


图 11 Region-based 错误恢复策略示例二。如果是由于上游失败导致的错误，那么需要同时重启上游的 Region 和下游的 Region。实际上，如果下游的输出使用了非确定的数据分割方式，为了保持数据一致性，还需要同时重启所有上游 Region 的下游 Region

除了 Task 本身执行的异常外，另一类异常是 Flink 集群的 Master 进行发生异常。目前 Flink 支持启动多个 Master 作为备份，这些 Master 可以通过 ZK 来进行选主，从而保证某一时刻只有一个 Master 在运行。当前活路的 Master 发生异常时，某个备份的 Master 可以接管协调的工作。为了保证 Master 可以准确维护作业的状态，Flink 目前采用了一种最简单的实现方式，即直接重启整个作业。实际上，

由于作业本身可能仍在正常运行，因此这种方式存在一定的改进空间。

## 5. 未来展望

Flink 目前仍然在 Runtime 部分进行不断的迭代和更新。目前来看，Flink 未来可能会在以下几个方式继续进行优化和扩展：

- **更完善的资源管理：**从 1.9 开始 Flink 开始了对细粒度资源匹配的支持。基于细粒度的资源匹配，用户可以为 TaskExecutor 和 Task 设置实际提供和使用的 CPU、内存等资源的数量，Flink 可以按照资源的使用情况进行调度。这一机制允许用户更大范围的控制作业的调度，从而为进一步提高资源利用率提供了基础。
- **统一的 Stream 与 Batch：**Flink 目前为流和批分别提供了 DataStream 和 DataSet 两套接口，在一些场景下会导致重复实现逻辑的问题。未来 Flink 会将流和批的接口都统一到 DataStream 之上。
- **更灵活的调度策略：**Flink 从 1.9 开始引入调度插件的支持，从而允许用户来扩展实现自己的调度逻辑。未来 Flink 也会提供更高性能的调度策略的实现。
- **Master Failover 的优化：**如上节所述，目前 Flink 在 Master Failover 时需要重启整个作业，而实际上重启作业并不是必须的逻辑。Flink 未来会对 Master failover 进行进一步的优化来避免不必要的作业重启。

# Apache Flink 进阶 (二): 时间属性深度解析

作者: 崔星灿, Apache Flink Committer

整理: 沙晟阳(成阳), 阿里巴巴技术专家

本文根据 Apache Flink 进阶篇系列直播课程整理而成, 由 Apache Flink Committer 崔星灿分享, 阿里巴巴技术专家沙晟阳(成阳)整理。文章将对 Flink 的时间属性及原理进行深度解析。

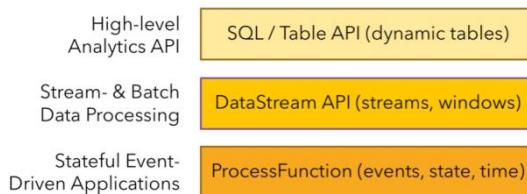
Tips: 文末可回顾全部基础篇及进阶篇系列教程。

## 1. 前言

Flink 的 API 大体上可以划分为三个层次: 处于最底层的 ProcessFunction、中间一层的 DataStream API 和最上层的 SQL/Table API, 这三层中的每一层都非常依赖于时间属性。**时间属性是流处理中最重要的一个方面, 是流处理系统的基石之一, 贯穿这三层 API**。在 DataStream API 这一层中因为封装方面的原因, 我们能够接触到时间的地方不是很多, 所以我们将重点放在底层的 ProcessFunction 和最上层的 SQL/Table API。



## 时间在Flink中的地位



## 2. Flink 时间语义

在不同的应用场景中时间语义是各不相同的，Flink 作为一个先进的分布式流处理引擎，它本身支持不同的时间语义。其核心是 Processing Time 和 Event Time (Row Time)，这两类时间主要的不同点如下表所示：

### Flink 支持的时间语义



Processing Time	Event Time (Row Time)
真实世界的时间	数据世界的时间
处理数据节点的本地时间	记录携带的Timestamp
处理简单	处理复杂
结果不确定 (无法重现)	结果确定 (可重现)

**Processing Time** 是来模拟我们真实世界的时间，其实就算是处理数据的节点本地时间也不一定是完完全全的真实世界的时间，所以说它是用来模拟真实世界的时间。而 **Event Time** 是数据世界的时间，即我们要处理的数据流世界里的时间。关于他们的获取方式，Process Time 是通过直接去调用本地机器的时间，而 Event Time 则是根据每一条处理记录所携带的时间戳来判定。

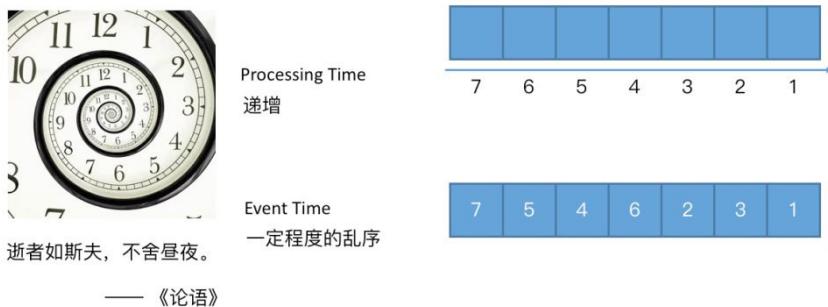
这两种时间在 Flink 内部的处理以及用户的实际使用方面，难易程度都是不同的。相对而言的 Processing Time 处理起来更加的简单，而 Event Time 要更麻烦一些。而在使用 Processing Time 的时候，我们得到的处理结果（或者说流处理应用的内部状态）是不确定的。而因为在 Flink 内部对 Event Time 做了各种保障，使用 Event Time 的情况下，无论重放数据多少次，都能得到一个相对确定可重现的结果。

因此在判断应该使用 Processing Time 还是 Event Time 的时候，可以遵循一个原则：当你的应用遇到某些问题要从上一个 checkpoint 或者 savepoint 进行

**重放，是不是希望结果完全相同。**如果希望结果完全相同，就只能用 Event Time；如果接受结果不同，则可以用 Processing Time。Processing Time 的一个常见的用途是，根据现实时间来统计整个系统的吞吐，比如要计算现实时间一个小时处理了多少条数据，这种情况只能使用 Processing Time。



## 时间的特性

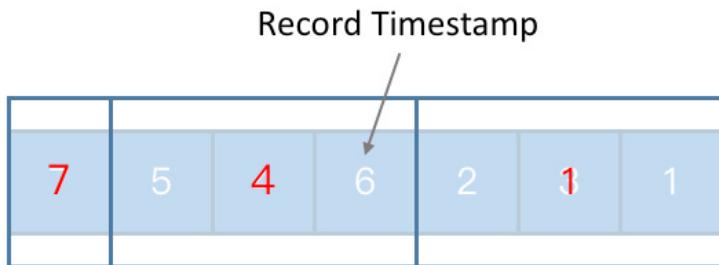


### 2.1 时间的特性

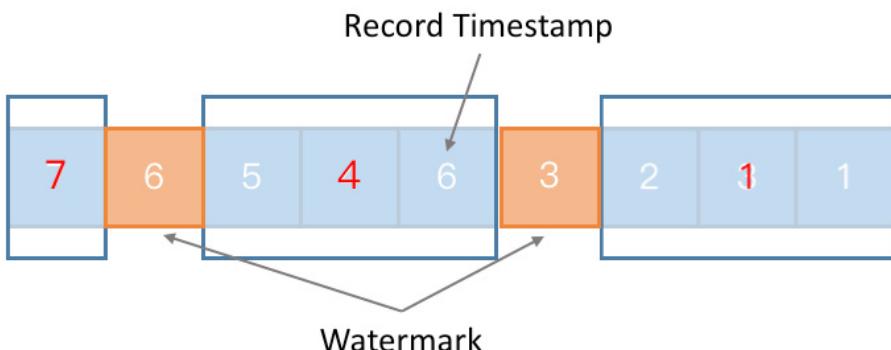
**时间的一个重要特性是：时间只能递增，不会来回穿越。**在使用时间的时候我们要充分利用这个特性。假设我们有这么一些记录，然后我们来分别看一下 Processing Time 还有 Event Time 对于时间的处理。

- 对于 Processing Time，因为我们是使用的是本地节点的时间（假设这个节点的时钟同步没有问题），我们每一次取到的 Processing Time 肯定都是递增的，递增就代表着有序，所以说我们相当于拿到的是一个有序的数据流。
- 而在用 Event Time 的时候因为时间是绑定在每一条的记录上的，由于网络延迟、程序内部逻辑、或者其他一些分布式系统的原因，数据的时间可能会存在一定程度的乱序，比如上图的例子。在 Event Time 场景下，我们把每一个记录所包含的时间称作 Record Timestamp。如果 Record Timestamp 所得到

的时间序列存在乱序，我们就需要去处理这种情况。

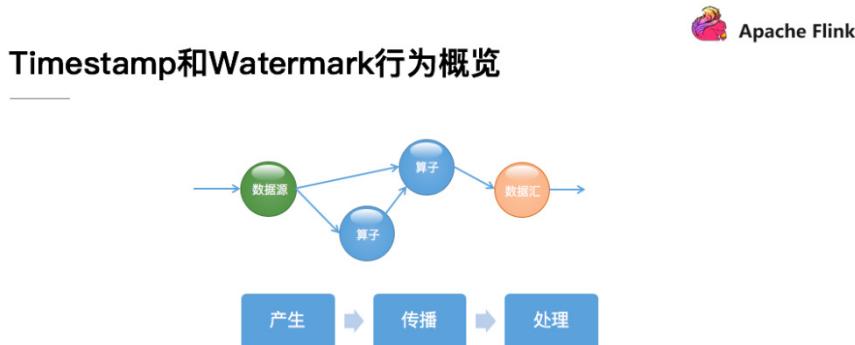


如果单条数据之间是乱序，我们就考虑对于整个序列进行更大程度的离散化。简单地讲，就是把数据按照一定的条数组成一些小批次，但这里的小批次并不是攒够多少条就要去处理，而是为了对他们进行时间上的划分。经过这种更高层次的离散化之后，我们会发现最右边方框里的时间就是一定会小于中间方框里的时间，中间框里的时间也一定会小于最左边方框里的时间。



这个时候我们在整个时间序列里插入一些类似于标志位的特殊的处理数据，这些特殊的处理数据叫做 watermark。一个 watermark 本质上就代表了这个 watermark 所包含的 timestamp 数值，表示以后到来的数据已经再也没有小于或等于这个时间的了。

### 3. Timestamp 和 Watermark 行为概览



接下来我们重点看一下 Event Time 里的 Record Timestamp (简写成 timestamp) 和 watermark 的一些基本信息。绝大多数的分布式流计算引擎对于数据都是进行了 DAG 图的抽象，它有自己的数据源，有处理算子，还有一些数据汇。数据在不同的逻辑算子之间进行流动。watermark 和 timestamp 有自己的生命周期，接下来我会从 watermark 和 timestamp 的产生、他们在不同的节点之间的传播、以及在每一个节点上的处理，这三个方面来展开介绍。

#### 3.1 Timestamp 分配和 Watermark 生成

Flink 支持两种 watermark 生成方式。第一种是在 SourceFunction 中产生，相当于把整个的 timestamp 分配和 watermark 生成的逻辑放在流处理应用的源头。我们可以在 SourceFunction 里面通过这两个方法产生 watermark：

- 通过 collectWithTimestamp 方法发送一条数据，其中第一个参数就是我们要发送的数据，第二个参数就是这个数据所对应的时间戳；也可以调用 emitWatermark 去产生一条 watermark，表示接下来不会再有时间戳小于等于这个数值记录。
- 另外，有时候我们不想在 SourceFunction 里生成 timestamp 或者

watermark，或者说使用的 SourceFunction 本身不支持，我们还可以在使用 DataStream API 的时候指定，调用的 DataStream.assignTimestampsAndWatermarks 这个方法，能够接收不同的 timestamp 和 watermark 的生成器。

**总体上而言生成器可以分为两类：**第一类是定期生成器；第二类是根据一些在流处理数据流中遇到的一些特殊记录生成的。

定期生成	根据特殊记录生成
现实时间驱动	数据驱动
每隔一段时间调用生成方法	每一次分配Timestamp都会调用生成方法
实现AssignerWithPeriodicWatermarks	实现AssignerWithPunctuatedWatermarks

两者的区别主要有三个方面，首先定期生成是现实时间驱动的，这里的“定期生成”主要是指 watermark（因为 timestamp 是每一条数据都需要有的），即定期会调用生成逻辑去产生一个 watermark。而根据特殊记录生成是数据驱动的，即是否生成 watermark 不是由现实时间来决定，而是当看到一些特殊的记录就表示接下来可能不会有符合条件的数据再发过来了，这个时候相当于每一次分配 Timestamp 之后都会调用用户实现的 watermark 生成方法，用户需要在生成方法中去实现 watermark 的生成逻辑。

大家要注意的是就是在分配 timestamp 和生成 watermark 的过程，虽然在 SourceFunction 和 DataStream 中都可以指定，但是还是建议生成的工作越靠近 DataSource 越好。这样会方便让程序逻辑里面更多的 operator 去判断某些数据是否乱序。**Flink 内部提供了很好的机制去保证这些 timestamp 和 watermark 被正确地传递到下游的节点。**

## 3.2 Watermark 传播

### Watermark传播



- Watermark以广播的形式在算子之间进行传播
- Long.MAX\_VALUE表示不会再有数据
- 单输入取其大，多输入取其小



#### 局限：

没有区分逻辑上的单流和多流，强制同步时钟

单个输入  $Wn = \max(W_{n_1}, W_{n_2}, W_{n_3} \dots)$

整个任务  $W = \min(W1, W2, W3)$

具体的传播策略基本上遵循这三点。

- **首先**，watermark 会以广播的形式在算子之间进行传播。比如说上游的算子连接了三个下游的任务，它会把自己当前的收到的 watermark 以广播的形式传到下游。
- **第二**，如果在程序里面收到了一个 Long.MAX\_VALUE 这个数值的 watermark，就表示对应的那一条流的一个部分不会再有数据发过来了，它相当于就是一个终止的标志。
- **第三**，对于单流而言，这个策略比较好理解，而对于有多个输入的算子，watermark 的计算就有讲究了，一个原则是：单输入取其大，多输入取小。

举个例子，假设这边蓝色的块代表一个算子的一个任务，然后它有三个输入，分别是 W1、W2、W3，这三个输入可以理解成任何输入，这三个输入可能是属于同一个流，也可能是属于不同的流。然后在计算 watermark 的时候，对于单个输入而言是取他们的最大值，因为我们都知道 watermark 应该遵循一个单调递增的一个原则。对于多输入，它要统计整个算子任务的 watermark 时，就会取这三个计算出来的 watermark 的最小值。即一个多个输入的任务，它的 watermark 受制于最慢的那条输入流。

这一点类似于木桶效应，整个木桶中装的水会受制于最矮的那块板。

watermark 在传播的时候有一个特点是，它的传播是幂等的。多次收到相同的 watermark，甚至收到之前的 watermark 都不会对最后的数值产生影响，因为对于单个输入永远是取最大的，而对于整个任务永远是取一个最小的。

同时我们可以注意到这种设计其实有一个局限，具体体现在它没有区分你这个输入是一条流多个 partition 还是来自于不同的逻辑上的流的 JOIN。对于同一个流的不同 partition，我们对他做这种强制的时钟同步是没有问题的，因为一开始就把一条流拆散成不同的部分，但每一个部分之间共享相同的时钟。

但是如果算子的任务是在做类似于 JOIN 操作，那么要求两个输入的时钟强制同步其实没有什么道理的，因为完全有可能是把一条离现在时间很近的数据流和一个离当前时间很远的数据流进行 JOIN，这个时候对于快的那条流，因为它要等慢的那条流，所以说它可能就要在状态中去缓存非常多的数据，这对于整个集群来说是一个很大的性能开销。

### 3.3 ProcessFunction

在正式介绍 watermark 的处理之前，先简单介绍 ProcessFunction，因为 watermark 在任务里的处理逻辑分为内部逻辑和外部逻辑。外部逻辑其实就是通过 ProcessFunction 来体现的，如果你需要使用 Flink 提供的时间相关的 API 的话就只能写在 ProcessFunction 里。

ProcessFunction 和时间相关的功能主要有三点：

- **第一点**，根据你当前系统使用的时间语义不同，你可以去获取当前你正在处理这条记录的 Record Timestamp，或者当前的 Processing Time。
- **第二点**，它可以获取当前算子的时间，可以把它理解成当前的 watermark。
- **第三点**，为了在 ProcessFunction 中去实现一些相对复杂的功能，允许注册一些 timer (定时器)。比如说在 watermark 达到某一个时间点的时候就触

发定时器，所有的这些回调逻辑也都是由用户来提供，涉及到如下三个方法，`registerEventTimeTimer`、`registerProcessingTimeTimer` 和 `onTimer`。在 `onTimer` 方法中就需要去实现自己的回调逻辑，当条件满足时回调逻辑就会被触发。

一个简单的应用是，我们在做一些时间相关的处理的时候，可能需要缓存一部分数据，但这些数据不能一直去缓存下去，所以需要有一些过期的机制，我们可以通过 `timer` 去设定这么一个时间，指定某一些数据可能在将来的某一个时间点过期，从而把它从状态里删除掉。所有的这些和时间相关的逻辑在 Flink 内部都是由自己的 Time Service (时间服务) 完成的。

## 3.4 Watermark 处理

### Watermark 处理



更新算子时间

遍历计时器队列触发回调

将Watermark发送至下游

一个算子的实例在收到 watermark 的时候，首先要更新当前的算子时间，这样的话在 `ProcessFunction` 里方法查询这个算子时间的时候，就能获取到最新的时间。第二步它会遍历计时器队列，这个计时器队列就是我们刚刚说到的 timer，你可以同时注册很多 timer，Flink 会把这些 Timer 按照触发时间放到一个优先队列中。第三步 Flink 得到一个时间之后就会遍历计时器的队列，然后逐一触发用户的回调逻辑。通过这种方式，Flink 的某一个任务就会将当前的 watermark 发送到下游的其他任务实例上，从而完成整个 watermark 的传播，从而形成一个闭环。

## 4. Table API 中的时间

下面我们来看一看 Table/SQL API 中的时间。为了让时间参与到 Table/SQL 这一层的运算中，我们需要提前把时间属性放到表的 schema 中，这样的话我们才能够在 SQL 语句或者 Table 的逻辑表达式里面使用时间去完成需求。

### 4.1 Table 中指定时间列

其实之前社区就怎么在 Table/SQL 中去使用时间这个问题做过一定的讨论，是把获取当前 Processing Time 的方法是作为一个特殊的 UDF，还是把这个列物化到整个的 schema 里面，最终采用了后者。我们这里就分开来讲一讲 Processing Time 和 Event Time 在使用的时候怎么在 Table 中指定。



对于 Processing Time，我们知道要得到一个 Table 对象（或者注册一个 Table）有两种手段：

- 可以从一个 DataStream 转化成一个 Table；
- 直接通过 TableSource 去生成这么一个 Table；

对于第一种方法而言，我们只需要在你已有的这些列中（例子中 f1 和 f2 就是两个已有的列），在最后用“列名 .proctime”这种写法就可以把最后的这一列注册为一个 Processing Time，以后在写查询的时候就可以去直接使用这一列。如果 Table 是通过 TableSource 生成的，就可以通过实现这一个 DefinedRowtimeAttributes

接口，然后就会自动根据你提供的逻辑去生成对应的 Processing Time。

相对而言，在使用 Event Time 时则有一个限制，因为 Event Time 不像 Processing Time 那样是随拿随用。如果要从 DataStream 去转化得到一个 Table，必须要提前保证原始的 DataStream 里面已经存在了 Record Timestamp 和 watermark。如果想通过 TableSource 生成的，也一定要保证要接入的数据里面存在一个类型为 long 或者 timestamp 的时间字段。

具体来说，如果要从 DataStream 去注册一个表，和 proctime 类似，只需要加上“列名 .rowtime”就可以。需要注意的是，如果要用 Processing Time，必须保证要新加的字段是整个 schema 中的最后一个字段，而 Event Time 的时候其实可以去替换某一个已有的列，然后 Flink 会自动的把这一列转化成需要的 rowtime 这个类型。

如果是通过 TableSource 生成的，只需要实现 DefinedRowtimeAttributes 接口就可以了。需要说明的一点是，在 DataStream API 这一侧其实不支持同时存在多个 Event Time (rowtime)，但是在 Table 这一层理论上可以同时存在多个 rowtime。因为 DefinedRowtimeAttributes 接口的返回值是一个对于 rowtime 描述的 List，即其实可以同时存在多个 rowtime 列，在将来可能会进行一些其他的改进，或者基于去做一些相应的优化。

## 4.2 时间列和 Table 操作

### 时间列和Table操作



- Over窗口聚合 (Over Window Aggregation)
- Group By窗口聚合 (Group By Window Aggregation)
- 时间窗口连接 (Time-WINDOWED JOIN)
- 排序 (Order By)



指定完了时间列之后，当我们要真正去查询时就会涉及到一些具体的操作。这里我列举的这些操作都是和时间列紧密相关，或者说必须在这个时间列上才能进行的。比如说“Over 窗口聚合”和“Group by 窗口聚合”这两种窗口聚合，在写 SQL 提供参数的时候只能允许你在这个时间列上进行这种聚合。第三个就是时间窗口聚合，你在写条件的时候只支持对应的时间列。最后就是排序，我们知道在一个无尽的数据流上对数据做排序几乎是不可能的事情，但因为这个数据本身到来的顺序已经是按照时间属性来进行排序，所以说如果要对一个 DataStream 转化成 Table 进行排序的话，只能是按照时间列进行排序，当然同时也可以指定一些其他的列，但是时间列这个是必须的，并且必须放在第一位。

为什么说这些操作只能在时间列上进行？

因为我们有的时候可以把到来的数据流就看成是一张按照时间排列好的一张表，而我们任何对于表的操作，其实都是必须在对它进行一次顺序扫描的前提下完成的。大家都知道数据流的特性之一就是一过性，某一条数据处理过去之后，将来其实不太好去访问它。当然因为 Flink 中内部提供了一些状态机制，我们可以在一定程度上去弱化这个特性，但是最终还是不能超越的，限制状态不能太大。所有这些操作为什么只能在时间列上进行，因为这个时间列能够保证我们内部产生的状态不会无限的增长下去，这是一个最终的前提。

# Apache Flink 进阶 (三): Checkpoint 原理剖析与应用实践

作者: 唐云(茶干)

阿里巴巴高级研发工程师

大家好，今天我将跟大家分享一下 Flink 里面的 Checkpoint，共分为四个部分。首先讲一下 Checkpoint 与 state 的关系，然后介绍什么是 state，第三部分介绍如何在 Flink 中使用 state，第四部分则介绍 Checkpoint 的执行机制。

## Checkpoint 与 state 的关系

Checkpoint 是从 source 触发到下游所有节点完成的一次全局操作。下图可以有一个对 Checkpoint 的直观感受，红框里面可以看到一共触发了 569K 次 Checkpoint，然后全部都成功完成，没有 fail 的。

Subtasks	Task Metrics	Watermarks	Accumulators	Checkpoints	Back Pressure
Overview	History	Summary	Configuration		
				<b>Checkpoint Counts</b>	
				Triggered: 569027    In Progress: 0    Completed: 569027    Failed: 0    Restored: 0	
Latest Completed Checkpoint	ID: 569027	Completion Time: 18:19:02	End to End Duration: 3ms	State Size: 9.32 KB	<a href="#">More details</a>
Latest Failed Checkpoint	None				
Latest Savepoint	None				
Latest Restore	None				

state 其实就是 Checkpoint 所做的主要持久化备份的主要数据，看下图的具体数据统计，其 state 也就 9kb 大小。

Details for Checkpoint 569116									
ID	Status	Acknowledged	Trigger Time	Latest Acknowledgement	End to End Duration	State Size	Buffered During Alignment	Discarded	Path
569116 ✓ Completed	2/2 (100%)		18:22:00	18:22:00	2ms	9.17 KB	0 B	Yes	<checkpoint-not-externally-addressable>

Operators									
Name	Acknowledged	Latest Acknowledgment	End to End Duration	State Size	Buffered During Alignment				
Source: Custom Source	1/1 (100%)	18:22:00	2ms	0 B	0 B	Show Subtasks ▾			
Flat Map -> Sink: Print to Std. Out	1/1 (100%)	18:22:00	2ms	9.17 KB	0 B	Show Subtasks ▾			

## 什么是 state

我们接下来看什么是 state。先看一个非常经典的 word count 代码，这段代码会去监控本地的 9000 端口的数据并对网络端口输入进行词频统计，我们本地行动 netcat，然后在终端输入 hello world，执行程序会输出什么？

```
env.socketTextStream("localhost", 9000)
    // split up the lines in pairs (2-tuples) containing: (word,1)
    .flatMap(new Tokenizer())
    // group by the tuple field "0" and sum up tuple field "1"
    .keyBy(0).sum(1)
    .print();
```

答案很明显，`(hello, 1)` 和 `(word, 1)`

那么问题来了，如果再次在终端输入 hello world，程序会输入什么？

答案其实也很明显，`(hello, 2)` 和 `(world, 2)`。为什么 Flink 知道之前已经处理过一次 hello world，这就是 state 发挥作用了，这里是被称为 keyed state 存储了之前需要统计的数据，所以帮助 Flink 知道 hello 和 world 分别出现过一次。

回顾一下刚才这段 word count 代码。keyby 接口的调用会创建 keyed stream 对 key 进行划分，这是使用 keyed state 的前提。在此之后，sum 方法会调用内置

的 StreamGroupedReduce 实现。

```
env.socketTextStream("localhost", 9000)
    // split up the lines in pairs (2-tuples) containing: (word,1)
    .flatMap(new Tokenizer())
    // group by the tuple field "0" and sum up tuple field "1"
    .keyBy(0).sum(1)
    .print();
```

创建  
KeyedStream  
(对key进行了划分,  
不同task上不会出  
现相同的key )

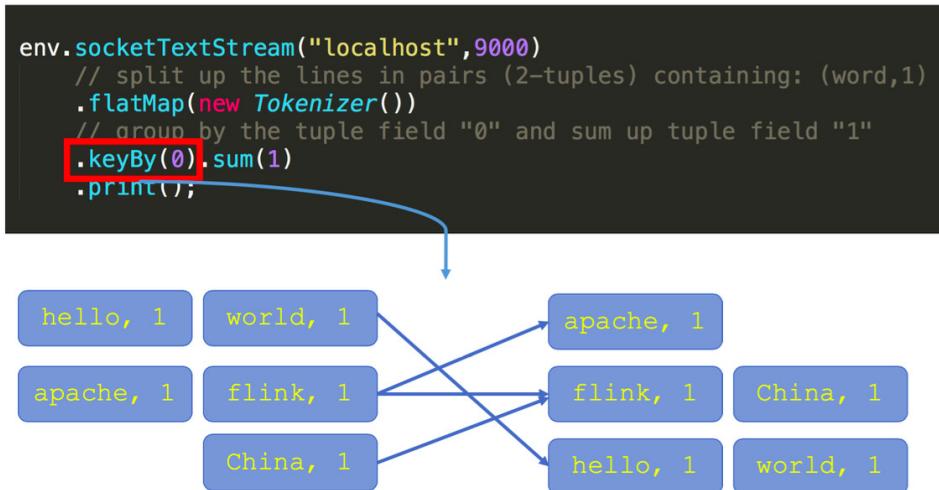
调用内置的  
StreamGroupedReduce UDF

## 什么是 keyed state

对于 keyed state, 有两个特点:

- 只能应用于 KeyedStream 的函数与操作中, 例如 Keyed UDF, window state
- keyed state 是已经分区 / 划分好的, 每一个 key 只能属于某一个 keyed state

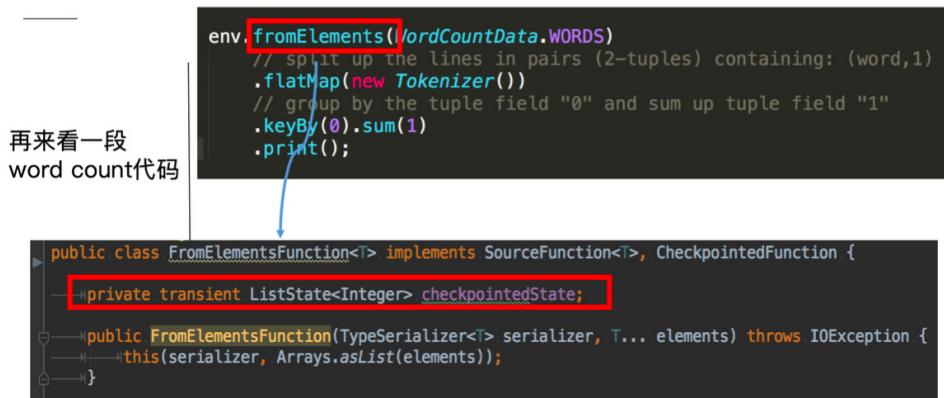
对于如何理解已经分区的概念, 我们需要看一下 keyby 的语义, 大家可以看到下图左边有三个并发, 右边也是三个并发, 左边的词进来之后, 通过 keyby 会进行相应的分发。例如对于 hello word, hello 这个词通过 hash 运算永远只会到右下方并发的 task 上面去。



## 什么是 operator state

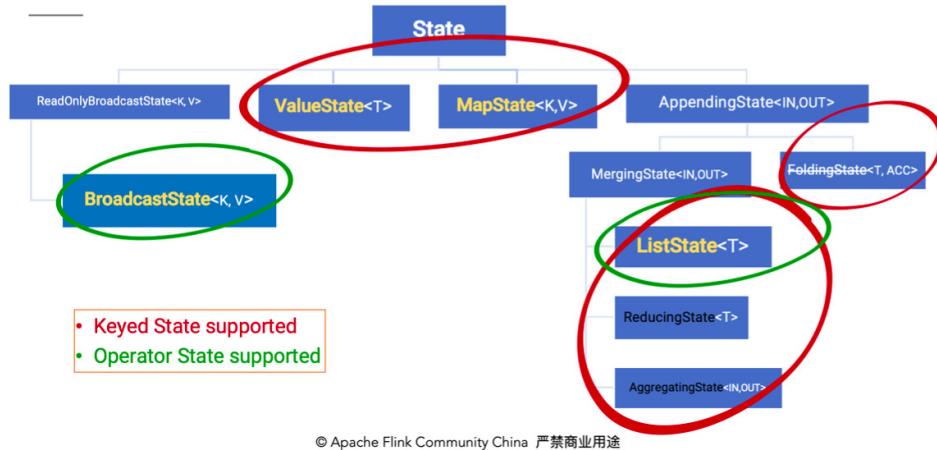
- 又称为 non-keyed state，每一个 operator state 都仅与一个 operator 的实例绑定。
- 常见的 operator state 是 source state，例如记录当前 source 的 offset

再看一段使用 operator state 的 word count 代码：



这里的 `fromElements` 会调用 `FromElementsFunction` 的类，其中就使用了

类型为 list state 的 operator state。根据 state 类型做一个分类如下图：



除了从这种分类的角度，还有一种分类的角度是从 Flink 是否直接接管：

- Managed State：由 Flink 管理的 state，刚才举例的所有 state 均是 managed state
- Raw State：Flink 仅提供 stream 可以进行存储数据，对 Flink 而言 raw state 只是一些 bytes

在实际生产中，都只推荐使用 managed state，本文将围绕该话题进行讨论。

## 如何在 Flink 中使用 state

下图就前文 word count 的 sum 所使用的 StreamGroupedReduce 类为例讲解了如何在代码中使用 keyed state：

```

public class StreamGroupedReduce<IN> extends AbstractUdfStreamOperator<IN, ReduceFunction<IN>>
    implements OneInputStreamOperator<IN, IN> {
    private transient ValueState<IN> values;

    @Override
    public void open() throws Exception {
        super.open();
        ValueStateDescriptor<IN> stateId = new ValueStateDescriptor<>(STATE_NAME, serializer);
        values = getRuntimeContext().getstate(stateId);
    } 通过 RuntimeContext 访问state

    @Override
    public void processElement(StreamRecord<IN> element) throws Exception {
        IN value = element.getValue();
        IN currentValue = values.value();

        if (currentValue != null) { 访问和修改当前key对应的state数值
            IN reduced = userFunction.reduce(currentValue, value);
            values.update(reduced);
            output.collect(element.replace(reduced));
        } else {
            values.update(value);
            output.collect(element.replace(value));
        }
    }
}

```

下图则对 word count 示例中的 FromElementsFunction 类进行详解并分享如何在代码中使用 operator state：

```

public class FromElementsFunction</> implements SourceFunction</>, CheckpointedFunction</> {
    private transient ListState<Integer> checkpointedState; 通过 FunctionInitializationContext 访问state

    @Override
    public void initialize(FunctionInitializationContext context) throws Exception {
        Preconditions.checkNotNull(this.checkpointedState == null,
            "The " + getClass().getSimpleName() + " has already been initialized.");

        this.checkpointedState = context.getOperatorStateStore().getListState(
            new ListStateDescriptor<>("from-elements-state", IntSerializer.INSTANCE));
    }

    if (context.isRestored()) {
        List<Integer> retrievedStates = new ArrayList<>();
        for (Integer entry : this.checkpointedState.get()) {
            retrievedStates.add(entry);
        }

        // given that the parallelism of the function is 1, we can only have 1 state
        Preconditions.checkArgument(retrievedStates.size() == 1, getClass().getSimpleName() + " retrieved invalid state.");
        this.numElementsToSkip = retrievedStates.get(0);
    }
} 在snapshotState时将状态数据存储到state中

@Override
public void snapshotState(FunctionSnapshotContext context) throws Exception {
    Preconditions.checkNotNull(this.checkpointedState != null,
        "The " + getClass().getSimpleName() + " has not been properly initialized");

    this.checkpointedState.clear();
    this.checkpointedState.add(this.numElementsEmitted);
}

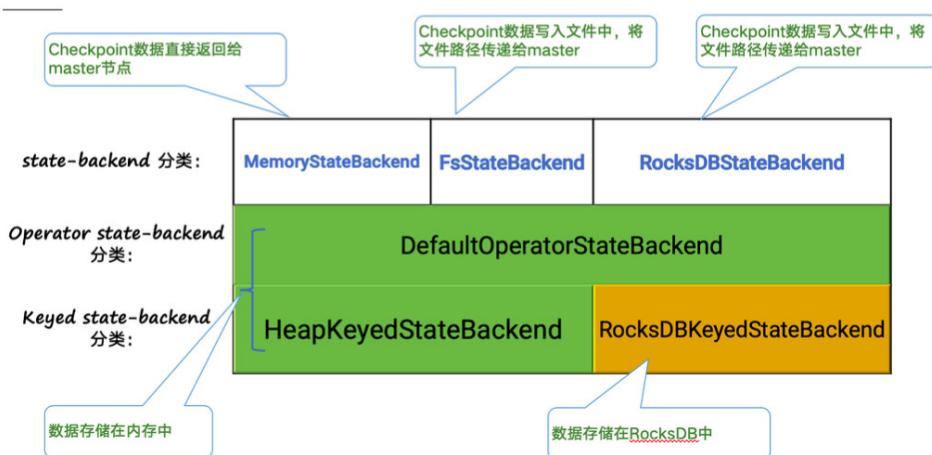
```

## Checkpoint 的执行机制

在介绍 Checkpoint 的执行机制前，我们需要了解一下 state 的存储，因为 state 是 Checkpoint 进行持久化备份的主要角色。

### Statebackend 的分类

下图阐释了目前 Flink 内置的三类 state backend，其中 `MemoryStateBackend` 和 `FsStateBackend` 在运行时都是存储在 java heap 中的，只有在执行 Checkpoint 时，`FsStateBackend` 才会将数据以文件格式持久化到远程存储上。而 `RocksDBStateBackend` 则借用了 RocksDB (内存磁盘混合的 LSM DB) 对 state 进行存储。



对于 `HeapKeyedStateBackend`，有两种实现：

- 支持异步 Checkpoint (默认): 存储格式 `CopyOnWriteStateMap`
- 仅支持同步 Checkpoint: 存储格式 `NestedStateMap`

特别在 `MemoryStateBackend` 内使用 `HeapKeyedStateBackend` 时，Checkpoint 序列化数据阶段默认有最大 5 MB 数据的限制

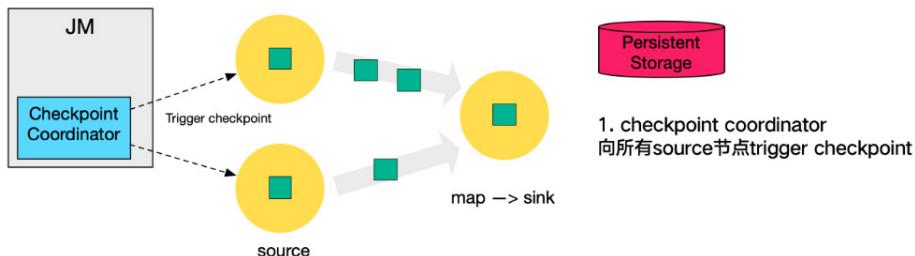
对于 `RocksDBKeyedStateBackend`, 每个 state 都存储在一个单独的 column family 内, 其中 keyGroup, Key 和 Namespace 进行序列化存储在 DB 作为 key。

State1		State2	
<u>KeyGroup + Key + Namespace</u>	value	<u>KeyGroup + Key + Namespace</u>	value
(1, K1, Window(10, 20))	v1	(2, K2, Window(10, 20))	v2
(1, K3, Window(10, 20))	v3	(2, K4, Window(10, 25))	v4
...	...	...	...

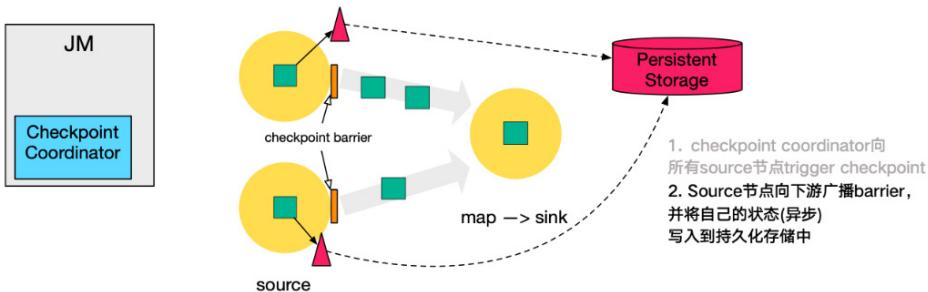
## Checkpoint 执行机制详解

本小节将对 Checkpoint 的执行流程逐步拆解进行讲解, 下图左侧是 Checkpoint Coordinator, 是整个 Checkpoint 的发起者, 中间是由两个 source, 一个 sink 组成的 Flink 作业, 最右侧的是持久化存储, 在大部分用户场景中对应 HDFS。

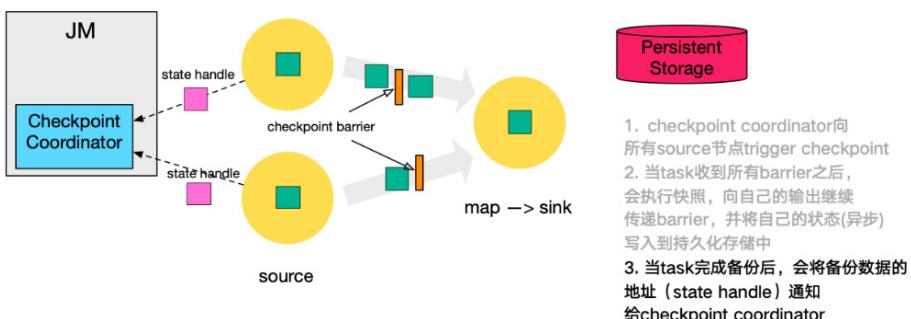
1. 第一步, Checkpoint Coordinator 向所有 source 节点 trigger Checkpoint;。



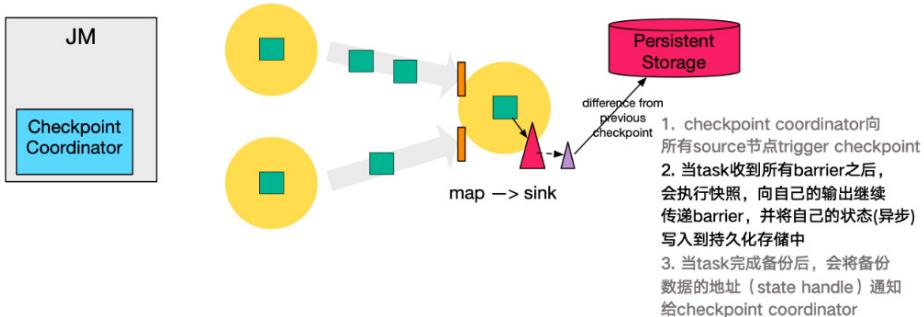
2. 第二步, source 节点向下游广播 barrier, 这个 barrier 就是实现 Chandy-Lampert 分布式快照算法的核心, 下游的 task 只有收到所有 input 的 barrier 才会执行相应的 Checkpoint。



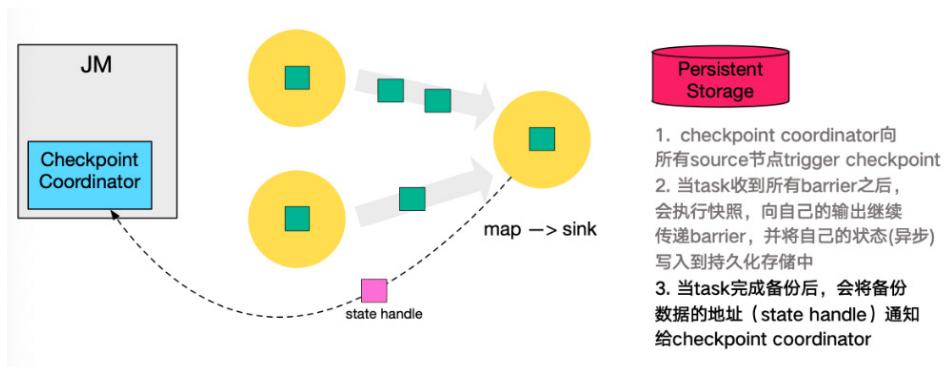
3. 第三步，当 task 完成 state 备份后，会将备份数据的地址 (state handle) 通知给 Checkpoint coordinator。



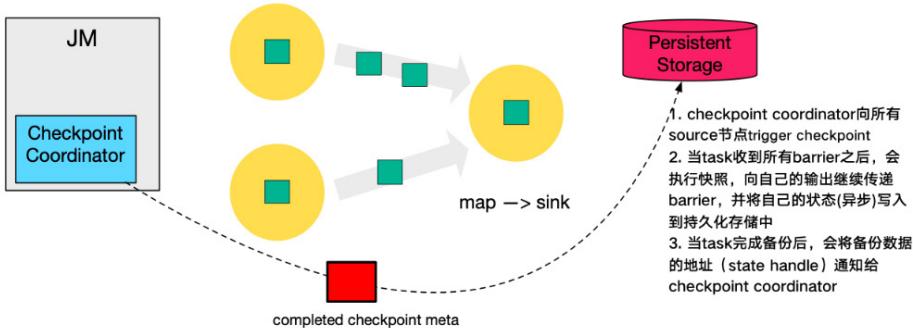
4. 第四步，下游的 sink 节点收集齐上游两个 input 的 barrier 之后，会执行本地快照，这里特地展示了 RocksDB incremental Checkpoint 的流程，首先 RocksDB 会全量刷数据到磁盘上 (红色大三角表示)，然后 Flink 框架会从中选择没有上传的文件进行持久化备份 (紫色小三角)。



5. 同样的，sink 节点在完成自己的 Checkpoint 之后，会将 state handle 返回通知 Coordinator。



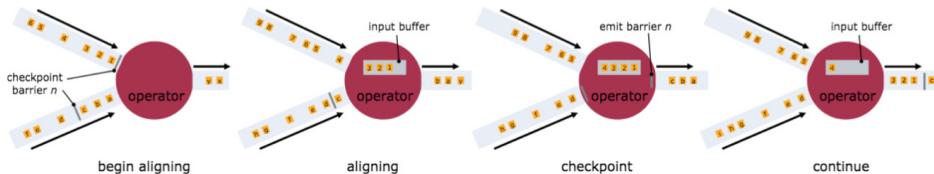
6. 最后，当 Checkpoint coordinator 收集齐所有 task 的 state handle，就认为这一次的 Checkpoint 全局完成了，向持久化存储中再备份一个 Checkpoint meta 文件。



## Checkpoint 的 EXACTLY\_ONCE 语义

为了实现 EXACTLY ONCE 语义，Flink 通过一个 input buffer 将在对齐阶段收到的数据缓存起来，等对齐完成之后再进行处理。而对于 AT LEAST ONCE 语

义，无需缓存收集到的数据，会对后续直接处理，所以导致 restore 时，数据可能会被多次处理。下图是官网文档里面就 Checkpoint align 的示意图：



需要特别注意的是，Flink 的 Checkpoint 机制只能保证 Flink 的计算过程可以做到 EXACTLY ONCE，端到端的 EXACTLY ONCE 需要 source 和 sink 支持。

## Savepoint 与 Checkpoint 的区别

作业恢复时，二者均可以使用，主要区别如下：

Savepoint	Externalized Checkpoint
用户通过命令触发，由用户管理其创建与删除	Checkpoint 完成时，在用户给定的外部持久化存储保存
标准化格式存储，允许作业升级或者配置变更	当作业 FAILED (或者 CANCELED) 时，外部存储的 Checkpoint 会保留下来
用户在恢复时需要提供用于恢复作业状态的 savepoint 路径	用户在恢复时需要提供用于恢复的作业状态的 Checkpoint 路径

# Apache Flink 进阶(四): Flink on Yarn/K8s 原理剖析及实践

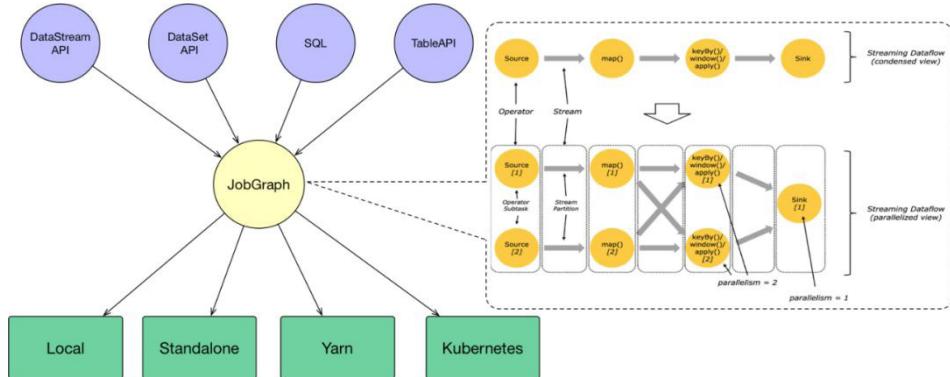
作者: 周凯波(宝牛)

阿里巴巴技术专家

本文根据 Apache Flink 进阶篇系列直播课程整理而成,由阿里巴巴技术专家周凯波(宝牛)分享,主要介绍 Flink on Yarn / K8s 的原理及应用实践,文章将从 Flink 架构、Flink on Yarn 原理及实践、Flink on Kubernetes 原理剖析三部分内容进行分享并对 Flink on Yarn/Kubernetes 中存在的部分问题进行了解答。

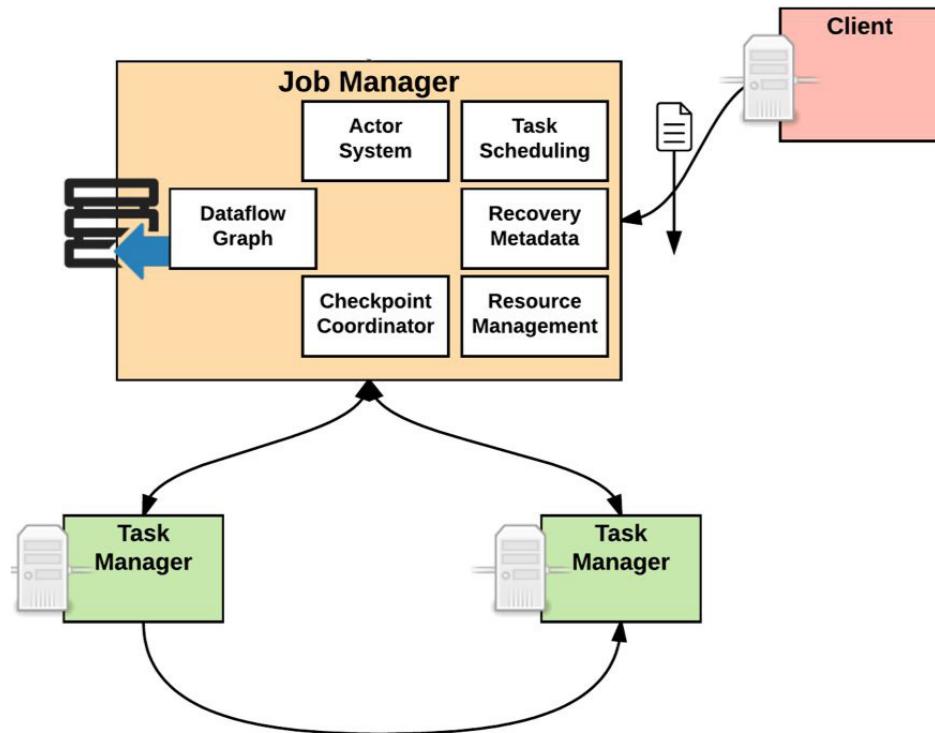
## Flink 架构概览

### Flink 架构概览 - Job



用户通过 DataStream API、DataSet API、SQL 和 Table API 编写 Flink 任务, 它会生成一个 JobGraph。JobGraph 是由 source、map()、keyBy()/window() / apply() 和 Sink 等算子组成的。当 JobGraph 提交给 Flink 集群后, 能够以 Local、Standalone、Yarn 和 Kubernetes 四种模式运行。

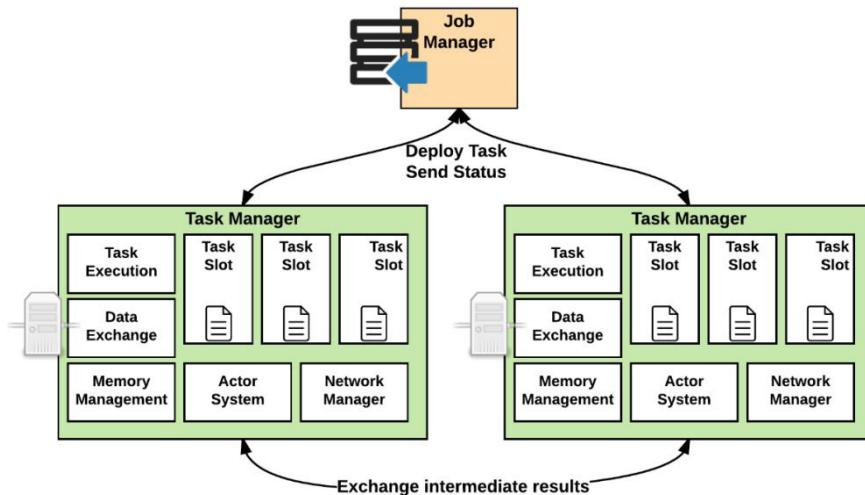
## Flink 架构概览 –JobManager



JobManager 的功能主要有：

- 将 JobGraph 转换成 Execution Graph，最终将 Execution Graph 拿来运行；
- Scheduler 组件负责 Task 的调度；
- Checkpoint Coordinator 组件负责协调整个任务的 Checkpoint，包括 Checkpoint 的开始和完成；
- 通过 Actor System 与 TaskManager 进行通信；
- 其它的一些功能，例如 Recovery Metadata，用于进行故障恢复时，可以从 Metadata 里面读取数据。

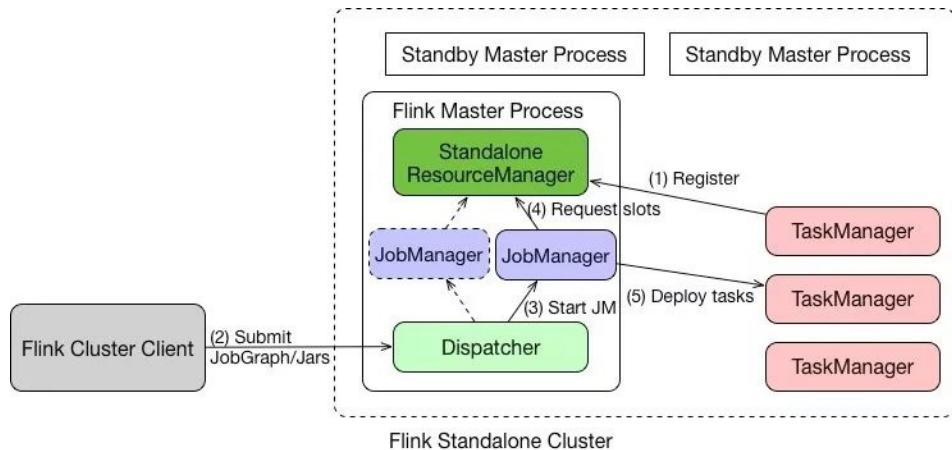
## Flink 架构概览 – TaskManager



TaskManager 是负责具体任务的执行过程，在 JobManager 申请到资源之后开始启动。TaskManager 里面的主要组件有：

- Memory & I/O Manager，即内存 I/O 的管理；
- Network Manager，用来对网络方面进行管理；
- Actor system，用来负责网络的通信；

TaskManager 被分成很多个 TaskSlot，每个任务都要运行在一个 TaskSlot 里面，TaskSlot 是调度资源里的最小单位。



在介绍 Yarn 之前先简单的介绍一下 Flink Standalone 模式，这样有助于更好地了解 Yarn 和 Kubernetes 架构。

- 在 **Standalone** 模式下，Master 和 TaskManager 可以运行在同一台机器上，也可以运行在不同的机器上。
- 在 **Master** 进程中，Standalone ResourceManager 的作用是对资源进行管理。当用户通过 Flink Cluster Client 将 JobGraph 提交给 Master 时，JobGraph 先经过 Dispatcher。
- 当 **Dispatcher** 收到客户端的请求之后，生成一个 JobManager。接着 JobManager 进程向 Standalone ResourceManager 申请资源，最终再启动 TaskManager。
- **TaskManager** 启动之后，会有一个注册的过程，注册之后 JobManager 再将具体的 Task 任务分发给这个 TaskManager 去执行。

以上就是一个 Standalone 任务的运行过程。

## Flink 运行时相关组件

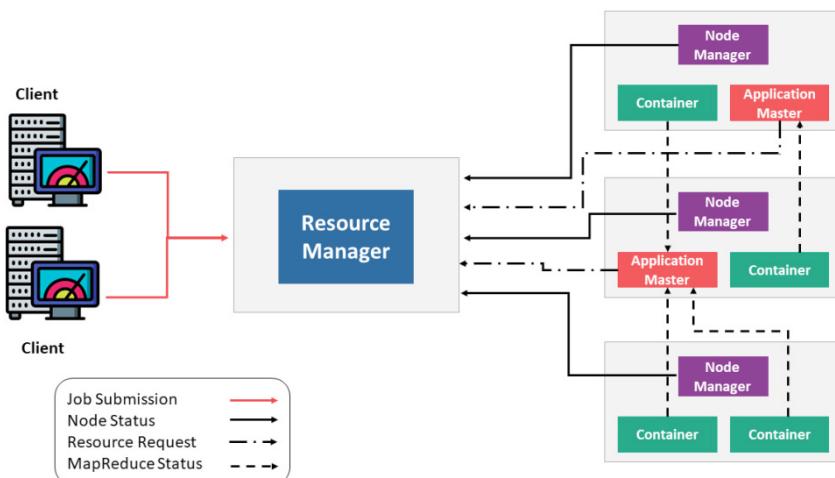
接下来总结一下 Flink 的基本架构和它在运行时的一些组件，具体如下：

- **Client:** 用户通过 SQL 或者 API 的方式进行任务的提交，提交后会生成一个 JobGraph。
- **JobManager:** JobManager 接受到用户的请求之后，会对任务进行调度，并且申请资源启动 TaskManager。
- **TaskManager:** 它负责一个具体 Task 的执行。TaskManager 向 JobManager 进行注册，当 TaskManager 接收到 JobManager 分配的任务之后，开始执行具体的任务。

## Flink on Yarn 原理及实践

### Yarn 架构原理 – 总览

Yarn 模式在国内使用比较广泛，基本上大多数公司在生产环境中都使用过 Yarn 模式。首先介绍一下 Yarn 的架构原理，因为只有足够了解 Yarn 的架构原理，才能更好的知道 Flink 是如何在 Yarn 上运行的。



Yarn 的架构原理如上图所示，最重要的角色是 ResourceManager，主要用来负责整个资源的管理，Client 端是负责向 ResourceManager 提交任务。

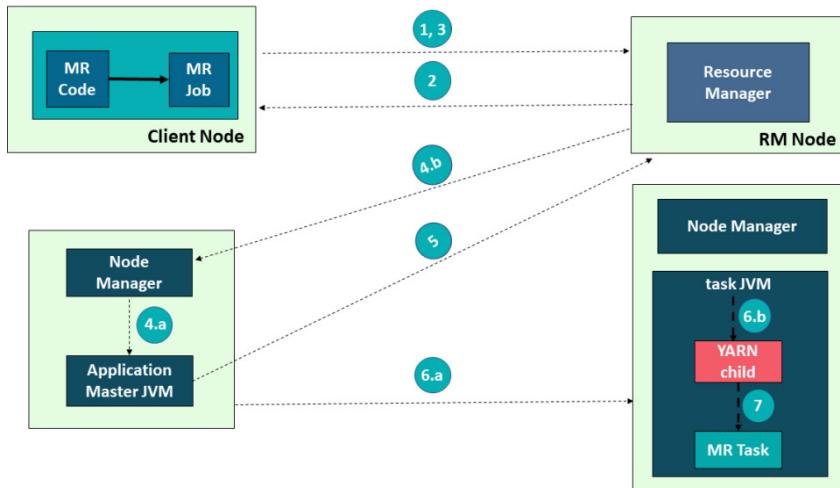
用户在 Client 端提交任务后会先给到 Resource Manager。Resource Manager 会启动 Container，接着进一步启动 Application Master，即对 Master 节点的启动。当 Master 节点启动之后，会向 Resource Manager 再重新申请资源，当 Resource Manager 将资源分配给 Application Master 之后，Application Master 再将具体的 Task 调度起来去执行。

## Yarn 架构原理 – 组件

Yarn 集群中的组件包括：

- **ResourceManager (RM)**: ResourceManager (RM) 负责处理客户端请求、启动 / 监控 ApplicationMaster、监控 NodeManager、资源的分配与调度，包含 Scheduler 和 Applications Manager。
- **ApplicationMaster (AM)**: ApplicationMaster (AM) 运行在 Slave 上，负责数据切分、申请资源和分配、任务监控和容错。
- **NodeManager (NM)**: NodeManager (NM) 运行在 Slave 上，用于单节点资源管理、AM/RM 通信以及汇报状态。
- **Container**: Container 负责对资源进行抽象，包括内存、CPU、磁盘，网络等资源。

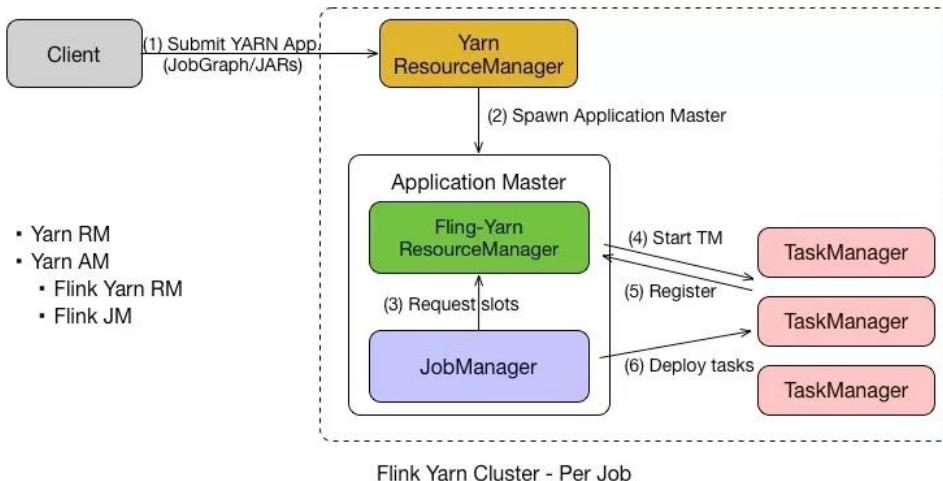
## Yarn 架构原理 – 交互



以在 Yarn 上运行 MapReduce 任务为例来讲解下 Yarn 架构的交互原理：

- 首先，用户编写 MapReduce 代码后，通过 Client 端进行任务提交。
- ResourceManager 在接收到客户端的请求后，会分配一个 Container 用来启动 ApplicationMaster，并通知 NodeManager 在这个 Container 下启动 ApplicationMaster。
- ApplicationMaster 启动后，向 ResourceManager 发起注册请求。接着 ApplicationMaster 向 ResourceManager 申请资源。根据获取到的资源，和相关的 NodeManager 通信，要求其启动程序。
- 一个或者多个 NodeManager 启动 Map/Reduce Task。
- NodeManager 不断汇报 Map/Reduce Task 状态和进展给 ApplicationMaster。
- 当所有 Map/Reduce Task 都完成时，ApplicationMaster 向 ResourceManager 汇报任务完成，并注销自己。

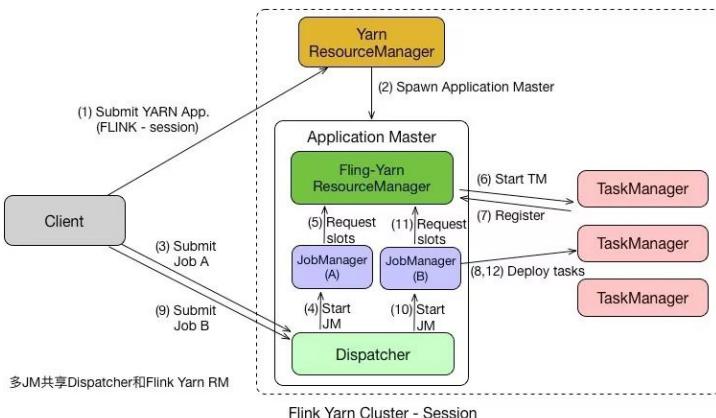
## Flink on Yarn–Per Job



Flink on Yarn 中的 Per Job 模式是指每次提交一个任务，然后任务运行完成之后资源就会被释放。在了解了 Yarn 的原理之后，Per Job 的流程也就比较容易理解了，具体如下：

- 首先 Client 提交 Yarn App，比如 JobGraph 或者 JARs。
- 接下来 Yarn 的 ResourceManager 会申请第一个 Container。这个 Container 通过 Application Master 启动进程，Application Master 里面运行的是 Flink 程序，即 Flink–Yarn ResourceManager 和 JobManager。
- 最后 Flink–Yarn ResourceManager 向 Yarn ResourceManager 申请资源。当分配到资源后，启动 TaskManager。TaskManager 启动后向 Flink–Yarn ResourceManager 进行注册，注册成功后 JobManager 就会分配具体的任务给 TaskManager 开始执行。

## Flink on Yarn-Session



在 Per Job 模式中，执行完任务后整个资源就会释放，包括 JobManager、TaskManager 都全部退出。而 Session 模式则不一样，它的 Dispatcher 和 ResourceManager 是可以复用的。Session 模式下，当 Dispatcher 在收到请求之后，会启动 JobManager(A)，让 JobManager(A) 来完成启动 TaskManager，接着会启动 JobManager(B) 和对应的 TaskManager 的运行。当 A、B 任务运行完成后，资源并不会释放。Session 模式也称为多线程模式，其特点是资源会一直存在不会释放，多个 JobManager 共享一个 Dispatcher，而且还共享 Flink-YARN ResourceManager。

Session 模式和 Per Job 模式的应用场景不一样。Per Job 模式比较适合那种对启动时间不敏感，运行时间较长的任务。Session 模式适合短时间运行的任务，一般是批处理任务。若用 Per Job 模式去运行短时间的任务，那就需要频繁的申请资源，运行结束后，还需要资源释放，下次还需再重新申请资源才能运行。显然，这种任务会频繁启停的情况不适用于 Per Job 模式，更适合用 Session 模式。

## Yarn 模式特点

Yarn 模式的优点有：

- **资源的统一管理和调度。**Yarn 集群中所有节点的资源(内存、CPU、磁盘、网络等)被抽象为 Container。计算框架需要资源进行运算任务时需要向 Resource Manager 申请 Container, Yarn 按照特定的策略对资源进行调度和进行 Container 的分配。Yarn 模式能通过多种任务调度策略来利用提高集群资源利用率。例如 FIFO Scheduler、Capacity Scheduler、Fair Scheduler，并能设置任务优先级。
- **资源隔离。**Yarn 使用了轻量级资源隔离机制 Cgroups 进行资源隔离以避免相互干扰，一旦 Container 使用的资源量超过事先定义的上限值，就将其杀死。
- **自动 failover 处理。**例如 Yarn NodeManager 监控、Yarn ApplicationManager 异常恢复。

Yarn 模式虽然有不少优点，但是也有诸多缺点，例如运维部署成本较高，灵活性不够。

## Flink on Yarn 实践

关于 Flink on Yarn 的实践在社区官网上面有很多课程，例如：《[Flink 安装部署、环境配置及运行应用程序](#)》和《[客户端操作](#)》都是基于 Yarn 进行讲解的，这里就不再赘述。

社区官网：

<https://ververica.cn/developers/flink-training-course1/>

## Flink on Kubernetes 原理剖析

Kubernetes 是 Google 开源的容器集群管理系统，其提供应用部署、维护、扩展机制等功能，利用 Kubernetes 能方便地管理跨机器运行容器化的应用。Kubernetes 和 Yarn 相比，相当于下一代的资源管理系统，但是它的能力远远不止这些。

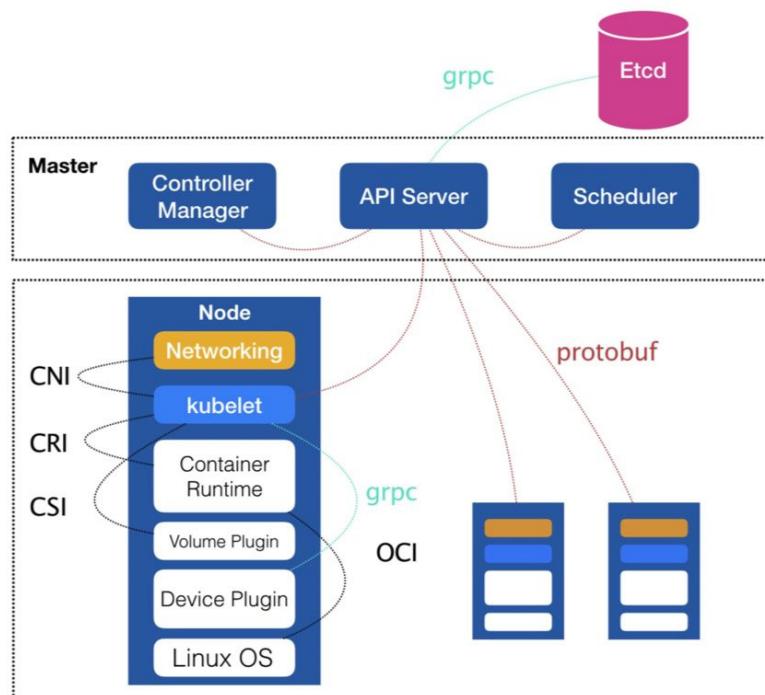
## Kubernetes- 基本概念

Kubernetes (k8s) 中的 Master 节点，负责管理整个集群，含有一个集群的资源数据访问入口，还包含一个 Etcd 高可用键值存储服务。Master 中运行着 API Server, Controller Manager 及 Scheduler 服务。

Node 为集群的一个操作单元，是 Pod 运行的宿主机。Node 节点里包含一个 agent 进程，能够维护和管理该 Node 上的所有容器的创建、启停等。Node 还含有一个服务端 kube-proxy，用于服务发现、反向代理和负载均衡。Node 底层含有 docker engine，docker 引擎主要负责本机容器的创建和管理工作。

Pod 运行于 Node 节点上，是若干相关容器的组合。在 K8s 里面 Pod 是创建、调度和管理的最小单位。

## Kubernetes- 架构图



Kubernetes 的架构如图所示，从这个图里面能看出 Kubernetes 的整个运行过程。

- API Server 相当于用户的一个请求入口，用户可以提交命令给 Etcd，这时会将这些请求存储到 Etcd 里面去。
- Etcd 是一个键值存储，负责将任务分配给具体的机器，在每个节点上的 Kubelet 会找到对应的 container 在本机上运行。
- 用户可以提交一个 Replication Controller 资源描述，Replication Controller 会监视集群中的容器并保持数量；用户也可以提交 service 描述文件，并由 kube proxy 负责具体工作的流量转发。

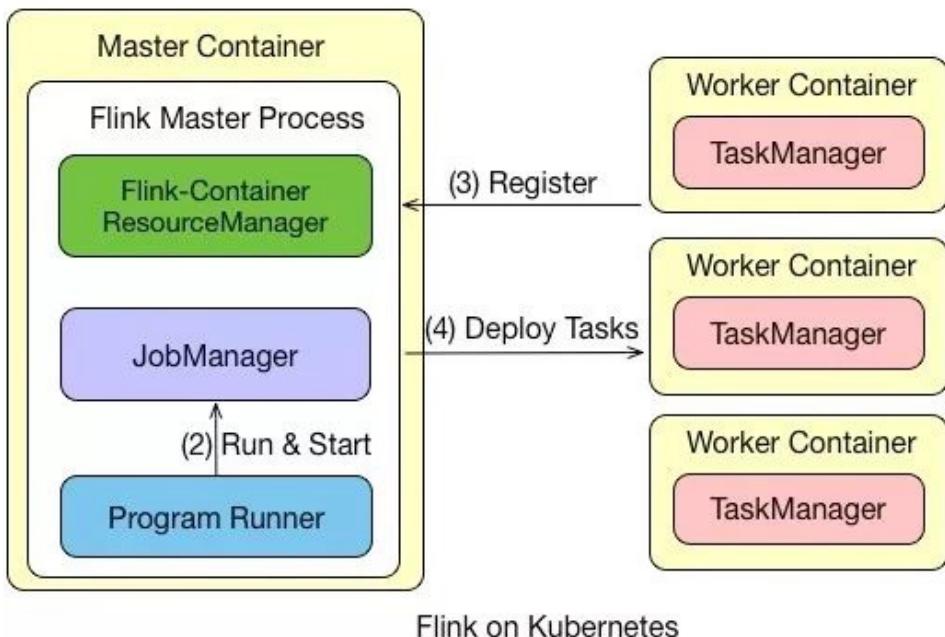
## Kubernetes- 核心概念

Kubernetes 中比较重要的概念有：

- Replication Controller (RC) 用来管理 Pod 的副本。RC 确保任何时候 Kubernetes 集群中有指定数量的 pod 副本 (replicas) 在运行，如果少于指定数量的 pod 副本，RC 会启动新的 Container，反之会杀死多余的以保证数量不变。
- Service 提供了一个统一的服务访问入口以及服务代理和发现机制
- Persistent Volume(PV) 和 Persistent Volume Claim(PVC) 用于数据的持久化存储。
- ConfigMap 是指存储用户程序的配置文件，其后端存储是基于 Etcd。

## Flink on Kubernetes – 架构

### (1) Container framework starts Master & Worker Containers



Flink on Kubernetes 的架构如图所示，Flink 任务在 Kubernetes 上运行的步骤有：

- 首先往 Kubernetes 集群提交了资源描述文件后，会启动 Master 和 Worker 的 container。
- Master Container 中会启动 Flink Master Process，包含 Flink-Container ResourceManager、JobManager 和 Program Runner。
- Worker Container 会启动 TaskManager，并向负责资源管理的 ResourceManager 进行注册，注册完成之后，由 JobManager 将具体的任务分给 Container，再由 Container 去执行。
- 需要说明的是，在 Flink 里的 Master 和 Worker 都是一个镜像，只是脚本的

命令不一样，通过参数来选择启动 master 还是启动 Worker。

## Flink on Kubernetes–JobManager

JobManager 的执行过程分为两步：

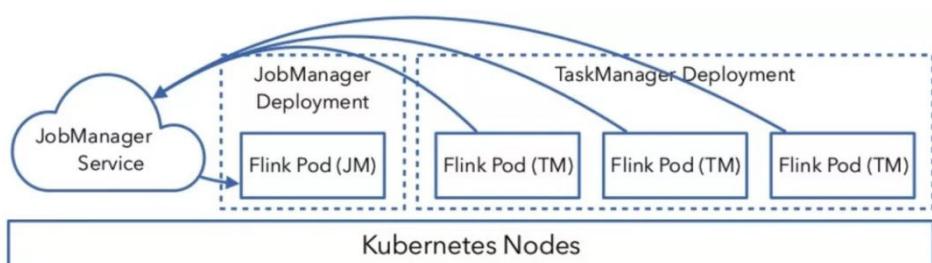
- 首先，JobManager 通过 Deployment 进行描述，保证 1 个副本的 Container 运行 JobManager，可以定义一个标签，例如 flink-jobmanager。
- 其次，还需要定义一个 JobManager Service，通过 service name 和 port 暴露 JobManager 服务，通过标签选择对应的 pods。

## Flink on Kubernetes–TaskManager

TaskManager 也是通过 Deployment 来进行描述，保证 n 个副本的 Container 运行 TaskManager，同时也需要定义一个标签，例如 flink-taskmanager。

对于 JobManager 和 TaskManager 运行过程中需要的一些配置文件，如：flink-conf.yaml、hdfs-site.xml、core-site.xml，可以通过将它们定义为 ConfigMap 来实现配置的传递和读取。

## Flink on Kubernetes– 交互



整个交互的流程比较简单，用户往 Kubernetes 集群提交定义好的资源描述文件即可，例如 deployment、configmap、service 等描述。后续的事情就交给 Kubernetes 集群自动完成。Kubernetes 集群会按照定义好的描述来启动 pod，运

行用户程序。各个组件的具体工作如下：

- Service: 通过标签 (label selector) 找到 job manager 的 pod 暴露服务。
- Deployment: 保证 n 个副本的 container 运行 JM/TM，应用升级策略。
- ConfigMap: 在每个 pod 上通过挂载 /etc/flink 目录，包含 flink-conf.yaml 内容。

## Flink on Kubernetes- 实践

接下来就讲一下 Flink on Kubernetes 的实践篇，即 K8s 上是怎么运行任务的。

- Session Cluster

```
•Session Cluster
  •启动
    •kubectl create -f jobmanager-service.yaml
    •kubectl create -f jobmanager-deployment.yaml
    •kubectl create -f taskmanager-deployment.yaml
  •Submit job
    •kubectl port-forward service/flink-jobmanager 8081:8081
    •bin/flink run -d -m localhost:8081 ./examples/streaming/
      TopSpeedWindowing.jar
  •停止
    •kubectl delete -f jobmanager-deployment.yaml
    •kubectl delete -f taskmanager-deployment.yaml
    •kubectl delete -f jobmanager-service.yaml
```

首先启动 Session Cluster，执行上述三条启动命令就可以将 Flink 的 Job-Manager-service、jobmanager-deployment、taskmanager-deployment 启动起来。启动完成之后用户可以通过接口进行访问，然后通过端口进行提交任务。若想销毁集群，直接用 kubectl delete 即可，整个资源就可以销毁。

<pre> jobmanager-deployment.yaml apiVersion: extensions/v1beta1 kind: Deployment metadata:   name: flink-jobmanager spec:   replicas: 1   template:     metadata:       labels:         app: flink         component: jobmanager     spec:       containers:         - name: jobmanager           image: flink:latest           args:             - jobmanager           ports:             - containerPort: 6123               name: rpc             - containerPort: 6124               name: blob             - containerPort: 6125               name: query             - containerPort: 8081               name: ui           env:             - name: JOB_MANAGER_RPC_ADDRESS               value: flink-jobmanager </pre>	<pre> taskmanager-deployment.yaml apiVersion: extensions/v1beta1 kind: Deployment metadata:   name: flink-taskmanager spec:   replicas: 2   template:     metadata:       labels:         app: flink         component: taskmanager     spec:       containers:         - name: taskmanager           image: flink:latest           args:             - taskmanager           ports:             - containerPort: 6121               name: data             - containerPort: 6122               name: rpc             - containerPort: 6125               name: query           env:             - name: JOB_MANAGER_RPC_ADDRESS               value: flink-jobmanager </pre>
---	---

Flink 官方提供的例子如图所示，图中左侧为 jobmanager-deployment.yaml 配置，右侧为 taskmanager-deployment.yaml 配置。

在 jobmanager-deployment.yaml 配置中，代码的第一行为 apiVersion，apiVersion 是 API 的一个版本号，版本号用的是 extensions/v1beta1 版本。资源类型为 Deployment，元数据 metadata 的名为 flink-jobmanager，spec 中含有副本数为 1 的 replicas，labels 标签用于 pod 的选取。containers 的镜像名为 jobmanager，containers 包含从公共 docker 仓库下载的 image，当然也可以使用公司内部的私有仓库。args 启动参数用于决定启动的是 jobmanager 还是 taskmanager；ports 是服务端口，常见的服务端口为 8081 端口；env 是定义的环境变量，会传递给具体的启动脚本。

右图为 taskmanager-deployment.yaml 配置，taskmanager-deployment.yaml 配置与 jobmanager-deployment.yaml 相似，但 taskmanager-deployment.yaml 的副本数是 2 个。

**jobmanager-service.yaml**

```

apiVersion: v1
kind: Service          资源类型为 Service
metadata:
  name: flink-jobmanager
spec:
  ports:
    - name: rpc
      port: 6123
    - name: blob
      port: 6124
    - name: query           要暴露的服务端口
      port: 6125
    - name: ui
      port: 8081
  selector:
    app: flink
    component: jobmanager   通过标签选择 jobmanager 的 pod

```

接下来是 jobmanager-service.yaml 的配置，jobmanager-service.yaml 的资源类型为 Service，在 Service 中的配置相对少一些，spec 中配置需要暴露的服务端口的 port，在 selector 中，通过标签选取 jobmanager 的 pod。

- **Job Cluster**

除了 Session 模式，还有一种 Per Job 模式。在 Per Job 模式下，需要将用户代码都打到镜像里面，这样如果业务逻辑的变动涉及到 Jar 包的修改，都需要重新生成镜像，整个过程比较繁琐，因此在生产环境中使用的比较少。

以使用公用 docker 仓库为例，Job Cluster 的运行步骤如下：

- **build 镜像：** 在 flink/flink-container/docker 目录下执行 build.sh 脚本，指定从哪个版本开始去构建镜像，成功后会输出“Successfully tagged”

topspeed:latest”的提示。

```
sh build.sh --from-release --flink-version 1.7.0 --hadoop-version 2.8
--scala-version 2.11 --job-jar ~/
flink/flink-1.7.1/examples/streaming/TopSpeedWindowing.jar --image-name
topspeed
```

- **上传镜像:** 在 hub.docker.com 上需要注册账号和创建仓库进行上传镜像。

```
docker tag topspeed zkb555/topspeedwindowing
docker push zkb555/topspeedwindowing
```

- **启动任务:** 在镜像上传之后，可以启动任务。

```
kubectl create -f job-cluster-service.yaml
FLINK_IMAGE_NAME=zkb555/topspeedwindowing:latest FLINK_JOB=org.apache.flink.streaming.
examples.windowing.TopSpeedWindowing FLINK_JOB_PARALLELISM=3 envsubst < job-
cluster-job.
yaml.template | kubectl create -f -
FLINK_IMAGE_NAME=zkb555/topspeedwindowing:latest FLINK_JOB_PARALLELISM=4
envsubst <
task-manager-deployment.yaml.template | kubectl create -f -
```

## Flink on Yarn/Kubernetes 问题解答

**Q: Flink 在 K8s 上可以通过 Operator 方式提交任务吗？**

目前 Flink 官方还没有提供 Operator 的方式，Lyft 公司开源了自己的 Operator 实现：<https://github.com/lyft/flinkk8soperator>。

**Q: 在 K8s 集群上如果不使用 Zookeeper 有没有其他高可用（HA）的方案？**

Etcd 是一个类似于 Zookeeper 的高可用键值服务，目前 Flink 社区正在考虑基于 Etcd 实现高可用的方案 (<https://issues.apache.org/jira/browse/FLINK-11105>) 以及直接依赖 K8s API 的方案 (<https://issues.apache.org/jira/browse/FLINK-12884>)。

**Q: Flink on K8s 在任务启动时需要指定 TaskManager 的个数，有和 Yarn 一样的动态资源申请方式吗？**

Flink on K8s 目前的实现在任务启动前就需要确定好 TaskManager 的个数，这样容易造成 TM 指定太少，任务无法启动，或者指定的太多，造成资源浪费。社区正在考虑实现和 Yarn 一样的任务启动时动态资源申请的方式。这是一种和 K8s 结合的更为 Nativey 的方式，称为 Active 模式。Active 意味着 ResourceManager 可以直接向 K8s 集群申请资源。具体设计方案和进展请关注：

[https://issues.apache.org/jira/browse/FLINK-9953。](https://issues.apache.org/jira/browse/FLINK-9953)

# Apache Flink 进阶(五): 数据类型和序列化

作者: 马庆祥

奇虎 360 数据开发高级工程师

本文根据 Apache Flink 系列直播整理而成, 由 Apache Flink Contributor、奇虎 360 数据开发高级工程师马庆祥老师分享。文章主要从如何为 Flink 量身定制序列化框架、Flink 序列化的最佳实践、Flink 通信层的序列化以及问答环节四部分分享。

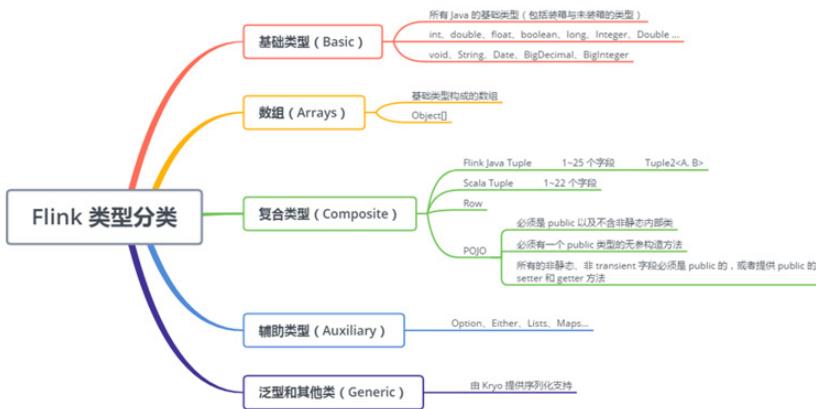
## 为 Flink 量身定制的序列化框架

### 为什么要为 Flink 量身定制序列化框架?

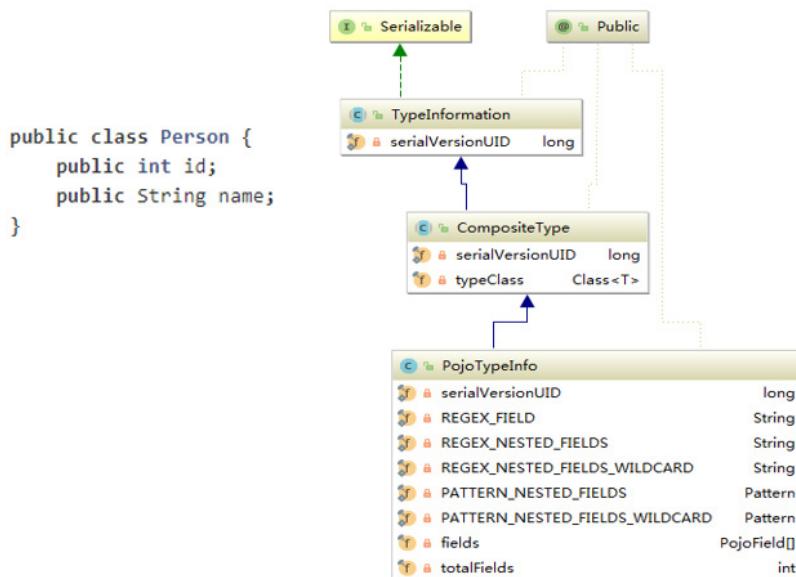
大家都知道现在大数据生态非常火, 大多数技术组件都是运行在 JVM 上的, Flink 也是运行在 JVM 上, 基于 JVM 的数据分析引擎都需要将大量的数据存储在内存中, 这就不得不面临 JVM 的一些问题, 比如 Java 对象存储密度较低等。**针对这些问题, 最常用的方法就是实现一个显式的内存管理, 也就是说用自定义的内存池来进行内存的分配回收, 接着将序列化后的对象存储到内存块中。**

现在 Java 生态圈中已经有许多序列化框架, 比如说 Java serialization, Kryo, Apache Avro 等等。但是 Flink 依然是选择了自己定制的序列化框架, 那么到底有什么意义呢? 若 Flink 选择自己定制的序列化框架, 对类型信息了解越多, 可以在早期完成类型检查, 更好的选取序列化方式, 进行数据布局, 节省数据的存储空间, 直接操作二进制数据。

## Flink 的数据类型

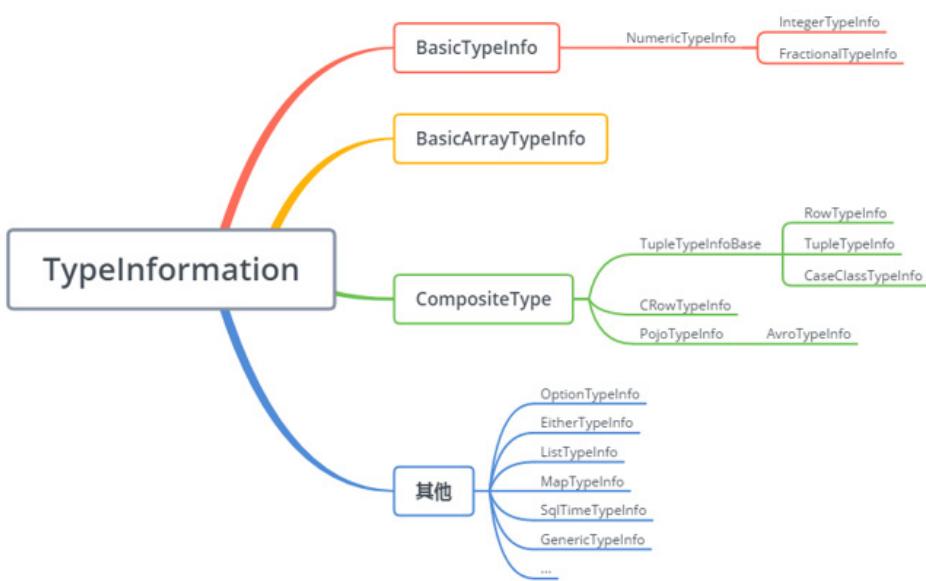


Flink 在其内部构建了一套自己的类型系统，Flink 现阶段支持的类型分类如图所示，从图中可以看到 Flink 类型可以分为基础类型 (Basic)、数组 (Arrays)、复合类型 (Composite)、辅助类型 (Auxiliary)、泛型和其它类型 (Generic)。Flink 支持任意的 Java 或是 Scala 类型。不需要像 Hadoop 一样去实现一个特定的接口 (org.apache.hadoop.io.Writable)，Flink 能够自动识别数据类型。



那这么多的数据类型，在 Flink 内部又是如何表示的呢？图示中的 Person 类，复合类型的一个 Pojo 在 Flink 中是用 PojoTypeInfo 来表示，它继承至 TypeInformation，也即在 Flink 中用 TypeInformation 作为类型描述符来表示每一种要表示的数据类型。

## TypeInformation



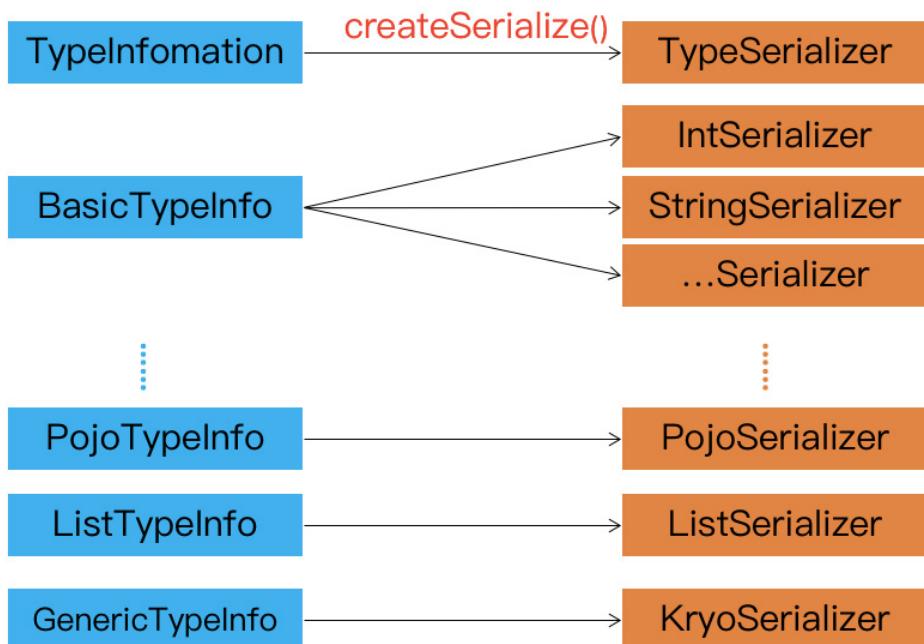
TypeInformation 的思维导图如图所示，从图中可以看出，在 Flink 中每一个具体的类型都对应了一个具体的 TypeInformation 实现类，例如 BasicTypeInfo 中的 IntegerTypeInfo 和 FractionalTypeInfo 都具体的对应了一个 TypeInformation。然后还有 BasicArrayTypeInformation、CompositeType 以及一些其它类型，也都具体对应了一个 TypeInformation。

**TypeInformation 是 Flink 类型系统的核心类。**对于用户自定义的 Function 来说，Flink 需要一个类型信息来作为该函数的输入输出类型，即 TypeInformation。该类型信息类作为一个工具来生成对应类型的序列化器 TypeSerializer，并用于执行

语义检查，比如当一些字段在作为 joing 或 grouping 的键时，检查这些字段是否在该类型中存在。

如何使用 TypeInformation？下面的实践中会为大家介绍。

### Flink 的序列化过程



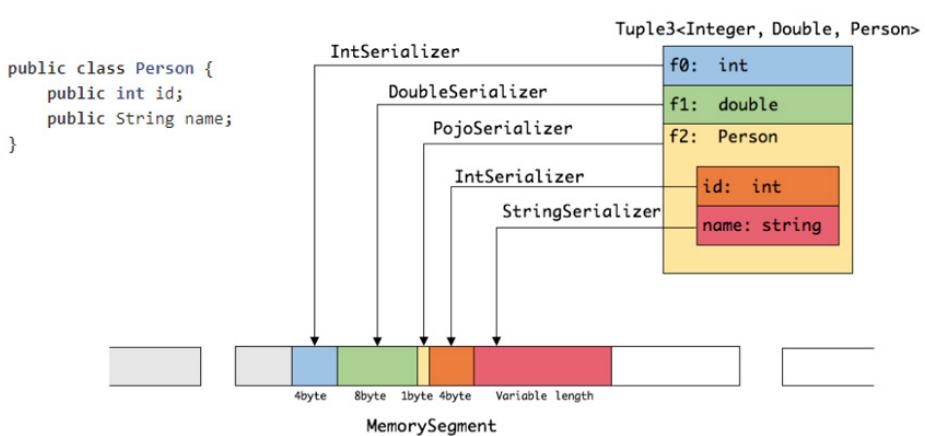
在 Flink 序列化过程中，进行序列化操作必须要有序列化器，那么序列化器从何而来？

每一个具体的数据类型都对应一个 `TypeInformation` 的具体实现，每一个 `TypeInformation` 都会为对应的具体数据类型提供一个专属的序列化器。通过 Flink 的序列化过程图可以看到 `TypeInformation` 会提供一个 `createSerialize()` 方法，通过这个方法就可以得到该类型进行数据序列化操作与反序化操作的对象 `TypeSerializer`。

对于大多数数据类型 Flink 可以自动生成对应的序列化器，能非常高效地对数据集进行序列化和反序列化，比如，BasicTypeInfo、WritableTypeInfo 等，但针对 GenericTypeInfo 类型，Flink 会使用 Kryo 进行序列化和反序列化。其中，Tuple、Pojo 和 CaseClass 类型是复合类型，它们可能嵌套一个或者多个数据类型。在这种情况下，它们的序列化器同样是复合的。它们会将内嵌类型的序列化委托给对应类型的序列化器。

简单的介绍下 Pojo 的类型规则，即在满足一些条件的情况下，才会选用 Pojo 的序列化进行相应的序列化与反序列化的一个操作。即类必须是 Public 的，且类有一个 public 的无参数构造函数，该类（以及所有超类）中的所有非静态 no-static、非瞬态 no-transient 字段都是 public 的（和非最终的 final）或者具有公共 getter 和 setter 方法，该方法遵循 getter 和 setter 的 Java bean 命名约定。当用户定义的数据类型无法识别为 POJO 类型时，必须将其作为 GenericType 处理并使用 Kryo 进行序列化。

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，如果内建的数据类型和序列化方式不能满足你的需求，Flink 的类型信息系统也支持用户拓展。若用户有一些特殊的需求，只需要实现 TypeInformation、TypeSerializer 和 TypeComparator 即可定制自己类型的序列化和比较大小方式，来提升数据类型在序列化和比较时的性能。



序列化就是将数据结构或者对象转换成一个二进制串的过程，在 Java 里面可以简单地理解成一个 byte 数组。而反序列化恰恰相反，就是将序列化过程中所生成的二进制串转换成数据结构或者对象的过程。下面就以内嵌型的 Tuple 3 这个对象为例，简述一下它的序列化过程。

Tuple 3 包含三个层面，一是 int 类型，一是 double 类型，还有一个是 Person。Person 包含两个字段，一是 int 型的 ID，另一个是 String 型的 name，它在序列化操作时，会委托相应具体序列化的序列化器进行相应的序列化操作。从图中可以看到 Tuple 3 会把 int 类型通过 IntSerializer 进行序列化操作，此时 int 只需要占用四个字节就可以了。根据 int 占用四个字节，这个能够体现出 Flink 可序列化过程中的一个优势，即在知道数据类型的前提下，可以更好的进行相应的序列化与反序列化操作。相反，如果采用 Java 的序列化，虽然能够存储更多的属性信息，但一次占据的存储空间会受到一定的损耗。

Person 类会被当成一个 Pojo 对象来进行处理，PojoSerializer 序列化器会把一些属性信息使用一个字节存储起来。同样，其字段则采取相对应的序列化器进行相应序列化，在序列化完的结果中，可以看到所有的数据都是由 MemorySegment 去支持。MemorySegment 具有什么作用呢？

MemorySegment 在 Flink 中会将对象序列化到预分配的内存块上，它代表 1 个固定长度的内存，默认大小为 32 kb。MemorySegment 代表 Flink 中的一个最小的内存分配单元，相当于是 Java 的一个 byte 数组。每条记录都会以序列化的形式存储在一个或多个 MemorySegment 中。

## Flink 序列化的最佳实践

### 最常见的场景

Flink 常见的应用场景有四种，即注册子类型、注册自定义序列化器、添加类型提示、手动创建 TypeInformation，具体介绍如下：

- **注册子类型:** 如果函数签名只描述了超类型，但是它们实际上在执行期间使用了超类型的子类型，那么让 Flink 了解这些子类型会大大提高性能。可以在 StreamExecutionEnvironment 或 ExecutionEnvironment 中调用 .registertype (clazz) 注册子类型信息。
- **注册自定义序列化:** 对于不适用于自己的序列化框架的数据类型，Flink 会使用 Kryo 来进行序列化，并不是所有的类型都与 Kryo 无缝连接，具体注册方法在下文介绍。
- **添加类型提示:** 有时，当 Flink 用尽各种手段都无法推测出泛型信息时，用户需要传入一个类型提示 TypeHint，这个通常只在 Java API 中需要。
- **手动创建一个 TypeInformation:** 在某些 API 调用中，这可能是必需的，因为 Java 的泛型类型擦除导致 Flink 无法推断数据类型。

其实在大多数情况下，用户不必担心序列化框架和注册类型，因为 Flink 已经提供了大量的序列化操作，不需要去定义自己的一些序列化器，但是在一些特殊场景下，需要去做一些相应的处理。

## 实践 – 类型声明

类型声明去创建一个类型信息的对象是通过哪种方式？通常是用 TypeInformation.of() 方法来创建一个类型信息的对象，具体说明如下：

- 对于非泛型类，直接传入 class 对象即可。

```
PojoTypeInfo<Person> typeInfo = (PojoTypeInfo<Person>) TypeInformation.  
of(Person.class);
```

- 对于泛型类，需要通过 TypeHint 来保存泛型类型信息。

```
final TypeInfomation<Tuple2<Integer, Integer>> resultType = TypeInformation.  
of(new TypeHint<Tuple2<Integer, Integer>>(){});
```

- 预定义常量。

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。而且 Flink 还提供了完全等价的 Types 类 (org.apache.flink.api.common.typeinfo.Types)。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。

- 自定义 TypeInfo 和 TypeInfoFactory。

```
@TypeInfo(MyTupleTypeInfoFactory.class)
public class MyTuple<T0, T1> {
    public T0 myfield0;
    public T1 myfield1;
}

public class MyTupleTypeInfoFactory extends TypeInfoFactory<MyTuple> {

    @Override
    public TypeInformation<MyTuple> createTypeInfo(Type t, Map<String, TypeInformation<?>> genericParameters) {
        return new MyTupleTypeInfo(genericParameters.get("T0"), genericParameters.get("T1"));
    }
}
```

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。需要注意在自定义类上使用 @TypeInfo 注解，随后创建相应的 TypeInfoFactory 并覆盖 createTypeInfo() 方法。

## 实践 – 注册子类型

Flink 认识父类，但不一定认识子类的一些独特特性，因此需要单独注册子类型。StreamExecutionEnvironment 和 ExecutionEnvironment 提供 registerType() 方法用来向 Flink 注册子类信息。

```
final ExecutionEnvironment env = ExecutionEnvironment.
getExecutionEnvironment(); Env.registerType(typeClass);
```

在 registerType() 方法内部，会使用 TypeExtractor 来提取类型信息，如上图所示，获取到的类型信息属于 PojoTypeInfo 及其子类，那么需要将其注册到一起，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)。

## 实践 -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），有两种解决方案：

- 强制使用 Avro 来代替 Kryo。

```
env.getConfig().enableForceAvro();
```

- 为 Kryo 增加自定义的 Serializer 以增强 Kryo 的功能。

```
env.getConfig().addDefaultKryoSerializer(clazz, serializer);
```

注：如果希望完全禁用 Kryo（100% 使用 Flink 的序列化机制），可以通过 Kryo-env.getConfig().disableGenericTypes() 的方式完成，但注意一切无法处理的类都将导致异常，这种对于调试非常有效。

## Flink 通信层的序列化

Flink 的 Task 之间如果需要跨网络传输数据记录，那么就需要将数据序列化之后写入 NetworkBufferPool，然后下层的 Task 读出之后再进行反序列化操作，最后进行逻辑处理。

为了使得记录以及事件能够被写入 Buffer，随后在消费时再从 Buffer 中读出，Flink 提供了数据记录序列化器 (RecordSerializer) 与反序列化器 (RecordDeserializer) 以及事件序列化器 (EventSerializer)。

Function 发送的数据被封装成 SerializationDelegate，它将任意元素公开为 IORandomAccessWritable 以进行序列化，通过 setInstance() 来传入要序列化的数据。在 Flink 通信层的序列化中，有几个问题值得关注，具体如下：

- 何时确定 Function 的输入输出类型?



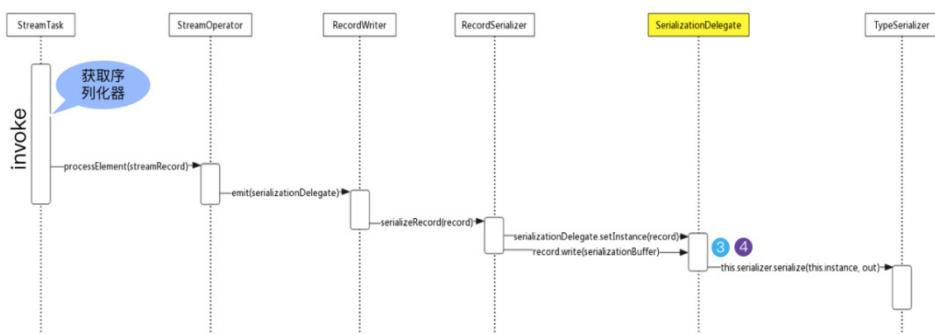
在构建 StreamTransformation 的时候通过 TypeExtractor 工具确定 Function 的输入输出类型。TypeExtractor 类可以根据方法签名、子类信息等蛛丝马迹自动提取或恢复类型信息。

- 何时确定 Function 的序列化 / 反序列化器?

构造 StreamGraph 时，通过 TypeInfomation 的 createSerializer() 方法获取对应类型的序列化器 TypeSerializer，并在 addOperator() 的过程中执行 setSerializers() 操作，设置 StreamConfig 的 TYPESERIALIZERIN1、TYPESERIALIZERIN2、TYPESERIALIZEROUT\_1 属性。

- 何时进行真正的序列化 / 反序列化操作? 这个过程与 TypeSerializer 又是怎么联系在一起的呢?

Flink task的调用过程:



大家都应该清楚 Tsk 和 StreamTask 两个概念，Task 是直接受 TaskManager 管理和调度的，而 Task 又会调用 StreamTask，而 StreamTask 中真正封装了算子的处理逻辑。在 run() 方法中，首先将反序列化后的数据封装成 StreamRe-

cord 交给算子处理；然后将处理结果通过 Collector 发动给下游（在构建 Collector 时已经确定了 SerializtionDelegate），并通过 RecordWriter 写入器将序列化后的结果写入 DataOutput；最后序列化的操作交给 SerializerDelegate 处理，实际还是通过 TypeSerializer 的 serialize() 方法完成。

# Apache Flink 进阶 (六): Flink 作业执行深度解析

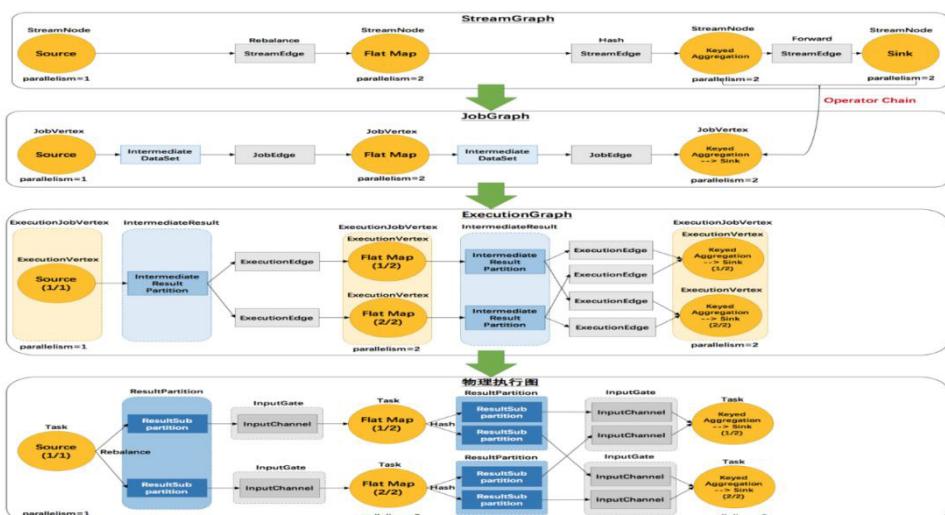
作者: 岳猛, 网易云音乐实时计算平台研发工程师

整理: 毛鹤

本文根据 Apache Flink 系列直播课程整理而成, 由 Apache Flink Contributor、网易云音乐实时计算平台研发工程师岳猛分享。主要分享内容为 Flink Job 执行作业的流程, 文章将从两个方面进行分享: 一是如何从 Program 到物理执行计划, 二是生成物理执行计划后该如何调度和执行。

## Flink 四层转化流程

Flink 有四层转换流程, 第一层为 Program 到 StreamGraph; 第二层为 StreamGraph 到 JobGraph; 第三层为 JobGraph 到 ExecutionGraph; 第四层为 ExecutionGraph 到物理执行计划。通过对 Program 的执行, 能够生成一个 DAG 执行图, 即逻辑执行图。如下:



第一部分将先讲解四层转化的流程，然后将以详细案例讲解四层的具体转化。

- 第一层 StreamGraph 从 Source 节点开始，每一次 transform 生成一个 StreamNode，两个 StreamNode 通过 StreamEdge 连接在一起，形成 StreamNode 和 StreamEdge 构成的 DAG。
- 第二层 JobGraph，依旧从 Source 节点开始，然后去遍历寻找能够嵌到一起的 operator，如果能够嵌到一起则嵌到一起，不能嵌到一起的单独生成 jobVertex，通过 JobEdge 链接上下游 JobVertex，最终形成 JobVertex 层面的 DAG。
- JobVertex DAG 提交到任务以后，从 Source 节点开始排序，根据 JobVertex 生成 ExecutionJobVertex，根据 jobVertex 的 IntermediateDataSet 构建 IntermediateResult，然后 IntermediateResult 构建上下游的依赖关系，形成 ExecutionJobVertex 层面的 DAG 即 ExecutionGraph。
- 最后通过 ExecutionGraph 层到物理执行层。

## Program 到 StreamGraph 的转化

Program 转换成 StreamGraph 具体分为三步：

- 从 StreamExecutionEnvironment.execute 开始执行程序，将 transform 添加到 StreamExecutionEnvironment 的 transformations。
- 调用 StreamGraphGenerator 的 generateInternal 方法，遍历 transformations 构建 StreamNode 及 StreamEdge。
- 通过 StreamEdge 连接 StreamNode。

```

public class WindowWordCount {
    // *****
    // PROGRAM
    // *****

    public static void main(String[] args) throws Exception {
        final ParameterTool params = ParameterTool.fromArgs(args);

        // set up the execution environment
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // get input data
        DataStream<String> text = env.readTextFile(params.get("input")).setParallelism(2);

        // make parameters available in the web interface
        env.getConfig().setGlobalJobParameters(params);

        final int windowSize = params.getInt("window", 10);
        final int slideSize = params.getInt("slide", 5);

        DataStream<Tuple2<String, Integer>> counts =
            // split up the lines in pairs (2-tuples) containing: (word,1)
            text.flatMap(new WordCount.Tokenizer()).setParallelism(1).slotSharingGroup("flatMap_sg")
                // create windows of windowSize records滑动 every slideSize records
                .keyBy()
                .countWindow(windowSize, slideSize)
                // group by the tuple field "0" and sum up tuple field "1"
                .sum(1).setParallelism(3).slotSharingGroup("sum_sg");

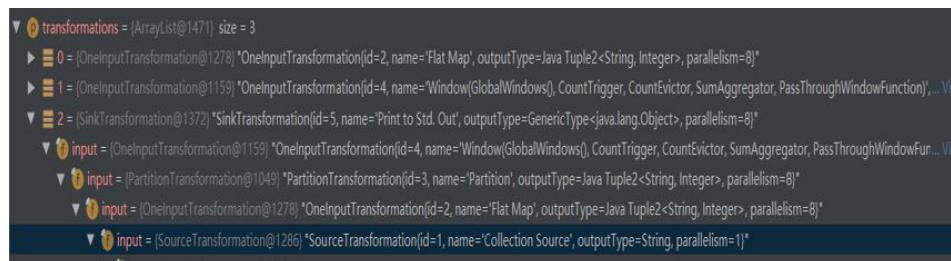
        // emit result
        counts.print().setParallelism(3);

        // execute program
        env.execute("WindowWordCount");
    }
}

```

通过 WindowWordCount 来看代码到 StreamGraph 的转化，在 flatMap transform 设置 slot 共享组为 flatMapsg，并发设置为 4，在聚合的操作中设置 slot 共享组为 sumsg，sum() 和 counts() 并发设置为 3，这样设置主要是为了演示后面如何嵌套到一起的，跟上下游节点的并发以及上游的共享组有关。

WindowWordCount 代码中可以看到，在 readTextFile() 中会生成一个 transform，且 transform 的 ID 是 1；然后到 flatMap() 会生成一个 transform，transform 的 ID 是 2；接着到 keyBy() 生成一个 transform 的 ID 是 3；再到 sum() 生成一个 transform 的 ID 是 4；最后到 counts() 生成 transform 的 ID 是 5。



transform 的结构如图所示，第一个是 flatMap 的 transform，第二个是 window

的 transform，第三个是 SinkTransform 的 transform。除此之外，还能在 transform 的结构中看到每个 transform 的 input 是什么。

接下来介绍一下 StreamNode 和 StreamEdge。

- StreamNode 是用来描述 operator 的逻辑节点，其关键成员变量有 slotSharingGroup、jobVertexClass、inEdges、outEdges 以及 transformationUID；
- StreamEdge 是用来描述两个 operator 逻辑的链接边，其关键变量有 sourceVertex、targetVertex。



WindowWordCount transform 到 StreamGraph 转化如图所示，StreamExecutionEnvironment 的 transformations 存在 3 个 transform，分别是 Flat Map (Id 2)、Window (Id 4)、Sink (Id 5)。

transform 的时候首先递归处理 transform 的 input，生成 StreamNode，然后通过 StreamEdge 链接上下游 StreamNode。需要注意的是，有些 transform 操作并不会生成 StreamNode 如 PartitionTransformation，而是生成个虚拟节点。

```

this = {StreamGraph@1178}
  jobName = "WindowWordCount"
  environment = {LocalStreamEnvironment@1044}
  executionConfig = {ExecutionConfig@1181}
  checkpointConfig = {CheckpointConfig@1179}
  chaining = true
  streamNodes = {HashMap@1182} size = 4
    0 = {HashMap$Node@1198} *1* -> *Source: Collection Source-1*
    1 = {HashMap$Node@1199} *2* -> *Flat Map-2*
      key = {Integer@1204} 2
      value = {StreamNode@1205} *Flat Map-2*
    2 = {HashMap$Node@1200} *4* -> "Window(GlobalWindows), CountTrigger, CountEvictor, SumAggregator, PassThroughWindowFunction)-4"
    3 = {HashMap$Node@1201} *5* -> *Sink: Print to Std. Out-5*

```

在转换完成后可以看到，streamNodes 有四种 transform 形式，分别为 Source、Flat Map、Window、Sink。

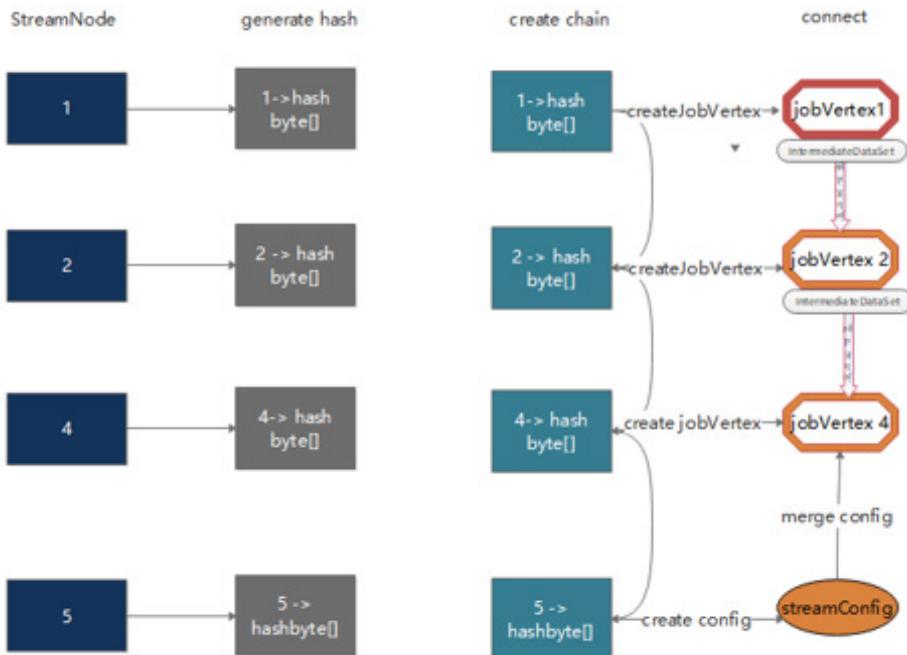
```

1 = {HashMap$Node@1199} *2* -> *Flat Map-2*
  key = {Integer@1204} 2
  value = {StreamNode@1205} *Flat Map-2*
    env = {LocalStreamEnvironment@1044}
    id = 2
    parallelism = {Integer@1215} 8
    maxParallelism = -1
    minResources = {ResourceSpec@1216} *ResourceSpec{cpuCores=0.0, heapMemoryInMB=0, directMemoryInMB=0, nativeMemoryInMB=0, stateSizeInMB=0}*
    preferredResources = {ResourceSpec@1216} *ResourceSpec{cpuCores=0.0, heapMemoryInMB=0, directMemoryInMB=0, nativeMemoryInMB=0, stateSizeInMB=0}*
    bufferTimeout = null
    operatorName = "Flat Map"
    slotSharingGroup = "flatMap"
    coLocationGroup = null
    statePartitioner = null
    statePartitioner = null
    stateKeySerializer = null
    operator = {StreamFlatMap@1219}
    outputSelectors = {ArrayList@1220} size = 0
    typeSerializerIn1 = {StringSerializer@1221}
    typeSerializerIn2 = null
    typeSerializerOut = {TupleSerializer@1222}
    inEdges = {ArrayList@1223} size = 1
    outEdges = {ArrayList@1224} size = 1
    jobVertexClass = {Class@1225} *class org.apache.flink.streaming.runtime.tasks.OnInputStreamTask* ... Navigate
    inputFormat = null

```

每个 streamNode 对象都携带并发个数、slotSharingGroup、执行类等运行信息。

## StreamGraph 到 JobGraph 的转化



StreamGraph 到 JobGraph 的转化步骤:

- 设置调度模式，Eager 所有节点立即启动。
- 广度优先遍历 StreamGraph，为每个 streamNode 生成 byte 数组类型的 hash 值。
- 从 source 节点开始递归寻找嵌到一起的 operator，不能嵌到一起的节点单独生成 jobVertex，能够嵌到一起的开始节点生成 jobVertex，其他节点以序列化的形式写入到 StreamConfig，然后 merge 到 CHAINEDTASKCONFIG，再通过 JobEdge 链接上下游 JobVertex。
- 将每个 JobVertex 的入边 (StreamEdge) 序列化到该 StreamConfig。
- 根据 group name 为每个 JobVertex 指定 SlotSharingGroup。
- 配置 checkpoint。

- 将缓存文件存文件的配置添加到 configuration 中。
- 设置 ExecutionConfig。

从 source 节点递归寻找嵌到一起的 operator 中，嵌到一起需要满足一定的条件，具体条件介绍如下：

- 下游节点只有一个输入。
- 下游节点的操作符不为 null。
- 上游节点的操作符不为 null。
- 上下游节点在一个槽位共享组内。
- 下游节点的连接策略是 ALWAYS。
- 上游节点的连接策略是 HEAD 或者 ALWAYS。
- edge 的分区函数是 ForwardPartitioner 的实例。
- 上下游节点的并行度相等。
- 可以进行节点连接操作。

```

▼ jobGraph = [JobGraph@1209] "JobGraph[jobId: 8616072e01737666bf4512f5fb0000b]"
  ▼ taskVertices = [LinkedHashMap@1685] size = 3
    ► 0 = [LinkedHashMap$Entry@1704] "e70bd798b564ea50e10e343f1ac56b" -> "Window[GlobalWindows], CountTrigger, CountEvictor, SumAggregator, PassThroughWindowFunction) -> Sink[Print to Std. Out]"
    ▼ 1 = [LinkedHashMap$Entry@1705] "0a448493b4782967b150582570326227" -> "Flat Map [org.apache.flink.streaming.runtime.tasks.OneInputStreamTask]"
      ► key = [JobVertexID@1580] "0a448493b4782967b150582570326227"
      ► value = [JobVertex@1542] "Flat Map [org.apache.flink.streaming.runtime.tasks.OneInputStreamTask]"
    ► 2 = [LinkedHashMap$Entry@1706] "bc764cd8dd7a0cff126f51c16239658" -> "Source[Collection Source [org.apache.flink.streaming.runtime.tasks.SourceStreamTask]]"

```

JobGraph 对象结构如上图所示，taskVertices 中只存在 Window、Flat Map、Source 三个 TaskVertex，Sink operator 被嵌到 window operator 中去了。

### 为什么要为每个 operator 生成 hash 值？

Flink 任务失败的时候，各个 operator 是能够从 checkpoint 中恢复到失败之前的状态的，恢复的时候是依据 JobVertexID (hash 值) 进行状态恢复的。相同的任务在恢复的时候要求 operator 的 hash 值不变，因此能够获取对应的状态。

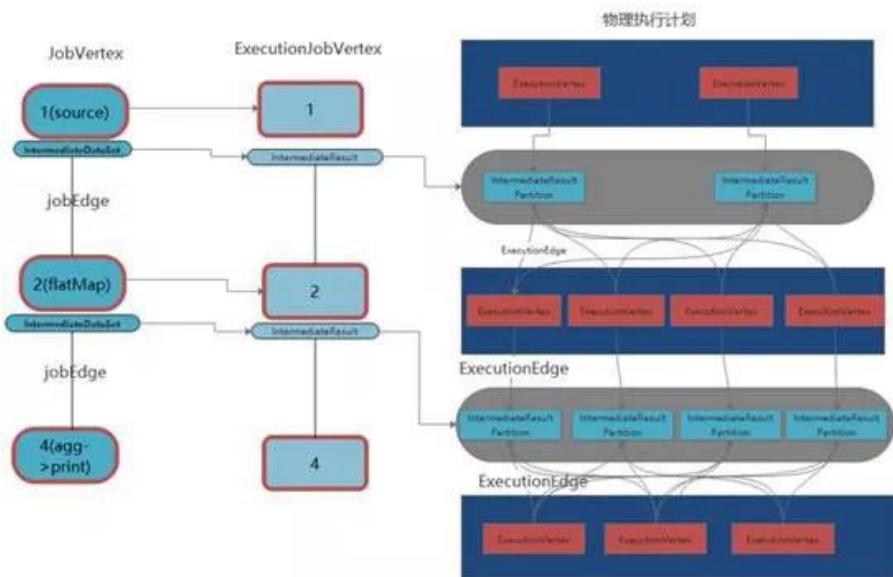
## 每个 operator 是怎样生成 hash 值的?

如果用户对节点指定了一个散列值，则基于用户指定的值能够产生一个长度为 16 的字节数组。如果用户没有指定，则根据当前节点所处的位置，产生一个散列值。

考虑的因素主要有三点：

- 一是在当前 StreamNode 之前已经处理过的节点的个数，作为当前 StreamNode 的 id，添加到 hasher 中；
- 二是遍历当前 StreamNode 输出的每个 StreamEdge，并判断当前 StreamNode 与这个 StreamEdge 的目标 StreamNode 是否可以进行链接，如果可以，则将目标 StreamNode 的 id 也放入 hasher 中，且这个目标 StreamNode 的 id 与当前 StreamNode 的 id 取相同的值；
- 三是将上述步骤后产生的字节数据，与当前 StreamNode 的所有输入 StreamNode 对应的字节数据，进行相应的位操作，最终得到的字节数据，就是当前 StreamNode 对应的长度为 16 的字节数组。

## JobGraph 到 ExecutionGraph 以及物理执行计划

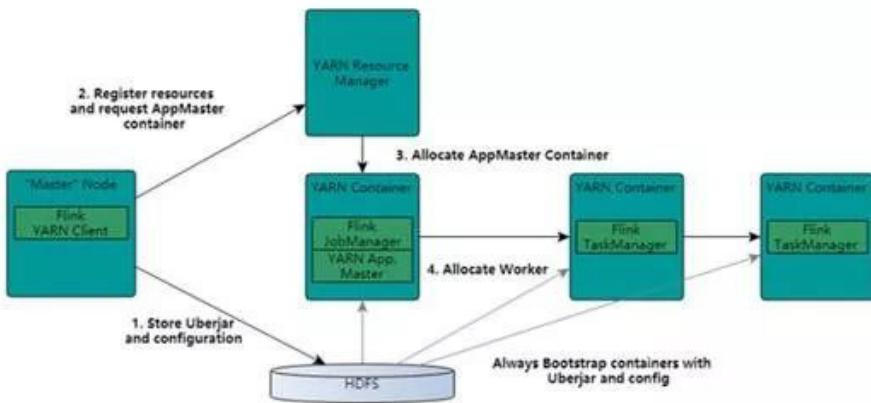


JobGraph 到 ExecutionGraph 以及物理执行计划的流程：

- 将 JobGraph 里面的 jobVertex 从 Source 节点开始排序。
- 在 executionGraph.attachJobGraph(sortedTopology) 方法里面，根据 JobVertex 生成 ExecutionJobVertex，在 ExecutionJobVertex 构造方法里面，根据 jobVertex 的 IntermediateDataSet 构建 IntermediateResult，根据 jobVertex 并发构建 ExecutionVertex，ExecutionVertex 构建的时候，构建 IntermediateResultPartition (每一个 Execution 构建 IntermediateResult 数个 IntermediateResultPartition )；将创建的 ExecutionJobVertex 与前置的 IntermediateResult 连接起来。
- 构建 ExecutionEdge，连接到前面的 IntermediateResultPartition，最终从 ExecutionGraph 到物理执行计划。

## Flink Job 执行流程

### Flink On Yarn 模式



基于 Yarn 层面的架构类似 Spark on Yarn 模式，都是由 Client 提交 App 到 RM 上面去运行，然后 RM 分配第一个 container 去运行 AM，然后由 AM 去负责资源的监督和管理。需要说明的是，Flink 的 Yarn 模式更加类似 Spark on Yarn 的

cluster 模式，在 cluster 模式中，dirver 将作为 AM 中的一个线程去运行。

Flink on Yarn 模式也是会将 JobManager 启动在 container 里面，去做个 driver 类似于的任务调度和分配，Yarn AM 与 Flink JobManager 在同一个 Container 中，这样 AM 可以知道 Flink JobManager 的地址，从而 AM 可以申请 Container 去启动 Flink TaskManager。待 Flink 成功运行在 Yarn 集群上，Flink Yarn Client 就可以提交 Flink Job 到 Flink JobManager，并进行后续的映射、调度和计算处理。

## Flink on Yarn 的缺陷

- 资源分配是静态的，一个作业需要在启动时获取所需的资源并且在它的生命周期里一直持有这些资源。这导致了作业不能随负载变化而动态调整，在负载下降时无法归还空闲的资源，在负载上升时也无法动态扩展。
- On-Yarn 模式下，所有的 container 都是固定大小的，导致无法根据作业需求来调整 container 的结构。譬如 CPU 密集的作业或许需要更多的核，但不需要太多内存，固定结构的 container 会导致内存被浪费。
- 与容器管理基础设施的交互比较笨拙，需要两个步骤来启动 Flink 作业：1. 启动 Flink 守护进程；2. 提交作业。如果作业被容器化并且将作业部署作为容器部署的一部分，那么将不再需要步骤 2。
- On-Yarn 模式下，作业管理页面会在作业完成后消失不可访问。
- Flink 推荐 per job clusters 的部署方式，但是又支持可以在一个集群上运行多个作业的 session 模式，令人疑惑。

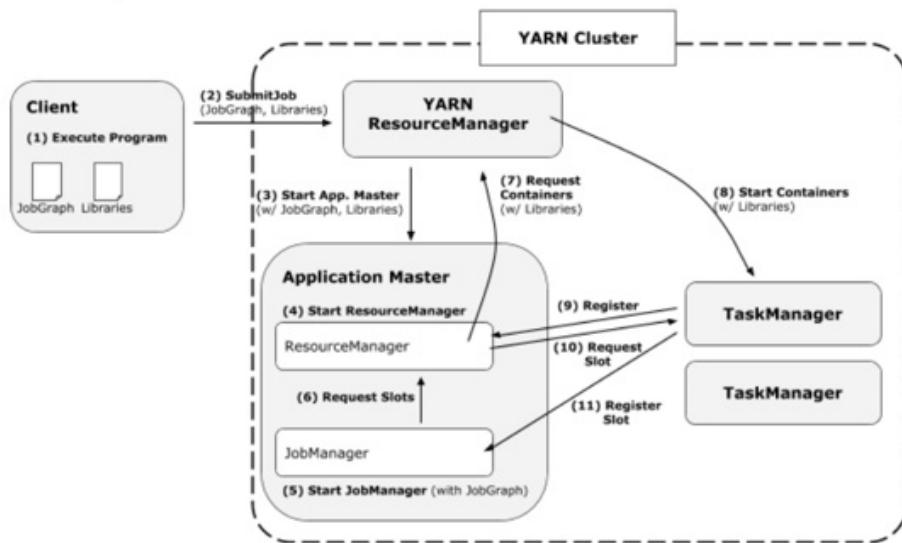
在 Flink 版本 1.5 中引入了 Dispatcher，Dispatcher 是在新设计里引入的一个新概念。Dispatcher 会从 Client 端接受作业提交请求并代表它在集群管理器上启动作业。

引入 Dispatcher 的原因主要有两点：

- 第一，一些集群管理器需要一个中心化的作业生成和监控实例；
  - 第二，能够实现 Standalone 模式下 JobManager 的角色，且等待作业提交。
- 在一些案例中，Dispatcher 是可选的 (Yarn) 或者不兼容的 (kubernetes)。

## 资源调度模型重构下的 Flink On Yarn 模式

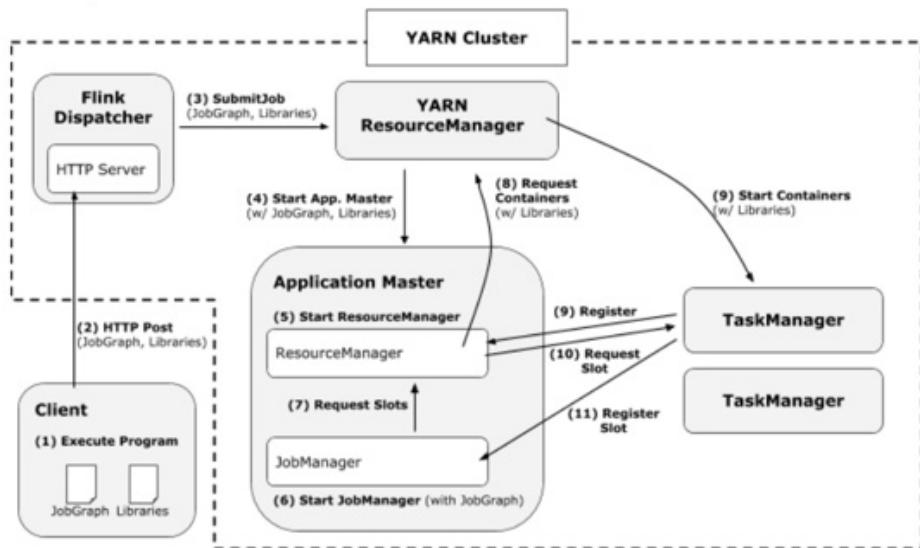
Without Dispatcher



### • 没有 Dispatcher job 运行过程

客户端提交 JobGraph 以及依赖 jar 包到 YarnResourceManager，接着 Yarn ResourceManager 分配第一个 container 以此来启动 AppMaster，Application Master 中会启动一个 FlinkResourceManager 以及 JobManager，JobManager 会根据 JobGraph 生成的 ExecutionGraph 以及物理执行计划向 FlinkResourceManager 申请 slot，FlinkResouceManager 会管理这些 slot 以及请求，如果没有可用 slot 就向 Yarn 的 ResourceManager 申请 container，container 启动以后会注册到 FlinkResourceManager，最后 JobManager 会将 subTask deploy 到对应 container 的 slot 中去。

### With Dispatcher



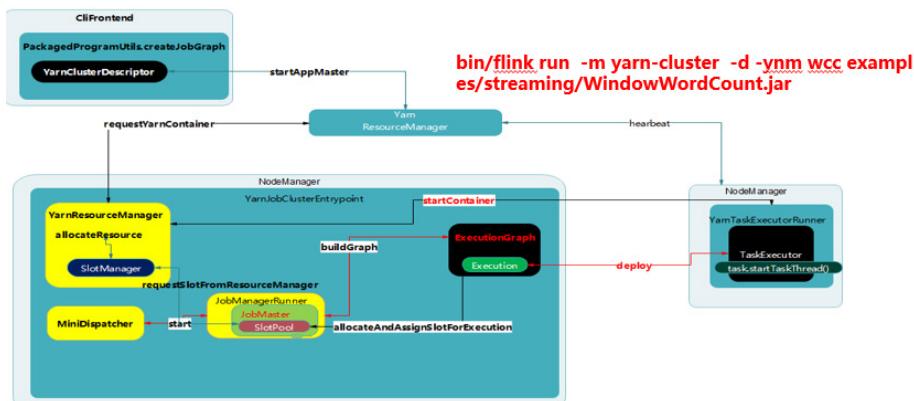
- 在有 Dispatcher 的模式下

会增加一个过程，就是 Client 会直接通过 HTTP Server 的方式，然后用 Dispatcher 将这个任务提交到 Yarn ResourceManager 中。

新框架具有四大优势，详情如下：

- client 直接在 Yarn 上启动作业，而不需要先启动一个集群然后再提交作业到集群。因此 client 再提交作业后可以马上返回。
- 所有的用户依赖库和配置文件都被直接放在应用的 classpath，而不是用动态的用户代码 classloader 去加载。
- container 在需要时才请求，不再使用时会被释放。
- “需要时申请”的 container 分配方式允许不同算子使用不同 profile (CPU 和内存结构) 的 container。

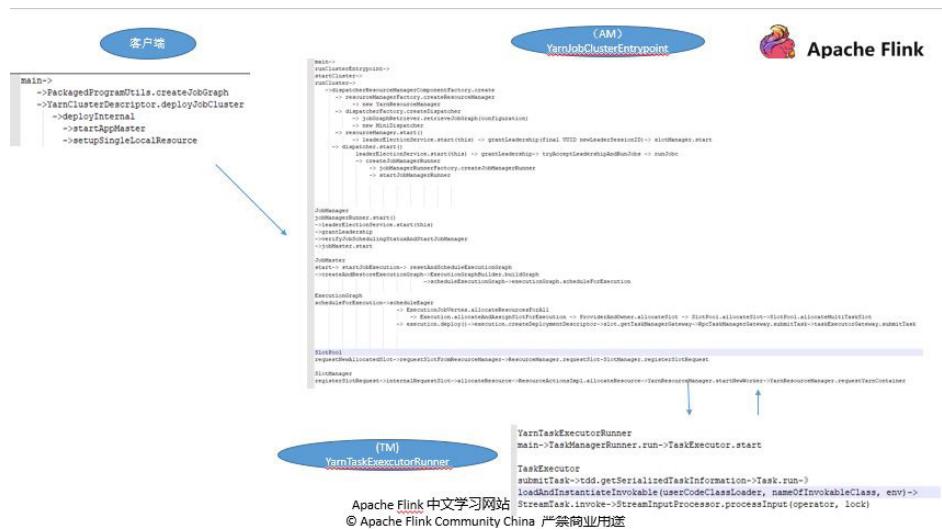
## 新的资源调度框架下 single cluster job on Yarn 流程介绍



single cluster job on Yarn 模式涉及三个实例对象：

- clifrontend
  - Invoke App code;
  - 生成 StreamGraph，然后转化为 JobGraph;
- YarnJobClusterEntryPoint (Master)
  - 依次启动 YarnResourceManager、MinDispatcher、JobManagerRunner 三者都服从分布式协同一致的策略;
  - JobManagerRunner 将 JobGraph 转化为 ExecutionGraph，然后转化为物理执行任务 Execution，然后进行 deploy，deploy 过程会向 YarnResourceManager 请求 slot，如果有直接 deploy 到对应的 YarnTaskExecutiontor 的 slot 里面，没有则向 Yarn 的 ResourceManager 申请，带 container 启动以后 deploy。
- YarnTaskExecutorRunner (slave)
  - 负责接收 subTask，并运行。

整个任务运行代码调用流程如下图：



subTask 在执行时是怎么运行的?

调用 `StreamTask` 的 `invoke` 方法, 执行步骤如下:

- \* `initializeState()` 即 `operator` 的 `initializeState()`
- \* `openAllOperators()` 即 `operator` 的 `open()` 方法
- \* 最后调用 `run` 方法来进行真正的任务处理

我们来看下 `flatMap` 对应的 `OneInputStreamTask` 的 `run` 方法具体是怎么处理的。

```

@Override protected void run() throws Exception {
    // cache processor reference on the stack, to make the code more JIT
    // friendly
    final StreamInputProcessor<IN> inputProcessor = this.inputProcessor;

    while (running && inputProcessor.processInput()) {
        // all the work happens in the "processInput" method
    }
}
  
```

最终是调用 `StreamInputProcessor` 的 `processInput()` 做数据的处理, 这里面包含用户的处理逻辑。

```

public boolean processInput() throws Exception {
  
```

```

        if (isFinished) {
            return false;
        }
        if (numRecordsIn == null) {
            try {
                numRecordsIn = ((OperatorMetricGroup) streamOperator.
getMetricGroup()).getIOMetricGroup().getNumRecordsInCounter();
            } catch (Exception e) {
                LOG.warn("An exception occurred during the metrics setup.", e);
                numRecordsIn = new SimpleCounter();
            }
        }

        while (true) {
            if (currentRecordDeserializer != null) {
                DeserializationResult result = currentRecordDeserializer.
getNextRecord(deserializationDelegate);

                if (result.isBufferConsumed()) {
                    currentRecordDeserializer.getCurrentBuffer().
recycleBuffer();
                    currentRecordDeserializer = null;
                }

                if (result.isFullRecord()) {
                    StreamElement recordOrMark = deserializationDelegate.
getInstance();
                    // 处理 watermark
                    if (recordOrMark.isWatermark()) {
                        // handle watermark
                        // watermark 处理逻辑, 这里可能引起 timer 的 trigger
                        statusWatermarkValve.inputWatermark(recordOrMark.
asWatermark(), currentChannel);
                        continue;
                    } else if (recordOrMark.isStreamStatus()) {
                        // handle stream status
                        statusWatermarkValve.inputStreamStatus(recordOrMark.
asStreamStatus(), currentChannel);
                        continue;
                    } else if (recordOrMark.isLatencyMarker()) {
                        // handle latency marker
                        synchronized (lock) {
                            streamOperator.processLatencyMarker(recordOrMark.
asLatencyMarker());
                        }
                        continue;
                    } else {

```

```

        // 用户的真正的代码逻辑
        // now we can do the actual processing
        StreamRecord<IN> record = recordOrMark.asRecord();
        synchronized (lock) {
            numRecordsIn.inc();
            streamOperator.setKeyContextElement1(record);
            // 处理数据
            streamOperator.processElement(record);
        }
        return true;
    }
}
}

// 这里会进行 checkpoint barrier 的判断和对齐，以及不同 partition 里面
checkpoint barrier 不一致时候的，数据 buffer，

final BufferOrEvent bufferOrEvent = barrierHandler.
getNextNonBlocked();
if (bufferOrEvent != null) {
    if (bufferOrEvent.isBuffer()) {
        currentChannel = bufferOrEvent.getChannelIndex();
        currentRecordDeserializer =
recordDeserializers[currentChannel];
        currentRecordDeserializer.setNextBuffer(bufferOrEvent.
getBuffer());
    }
    else {
        // Event received
        final AbstractEvent event = bufferOrEvent.getEvent();
        if (event.getClass() != EndOfPartitionEvent.class) {
            throw new IOException("Unexpected event: " + event);
        }
    }
}
else {
    isFinished = true;
    if (!barrierHandler.isEmpty()) {
        throw new IllegalStateException("Trailing data in
checkpoint barrier handler.");
    }
    return false;
}
}
}
}

```

streamOperator.processElement(record) 最终会调用用户的代码处理逻辑，假如 operator 是 StreamFlatMap 的话。

```
@Override  
    public void processElement(StreamRecord<IN> element) throws Exception {  
        collector.setTimestamp(element);  
        userFunction.flatMap(element.getValue(), collector); // 用户代码  
    }
```

如有不正确的地方，欢迎指正，关于 Flink 资源调度架构调整，网上有一篇非常不错的针对 FLIP-6 的翻译，推荐给大家。资源调度模型重构。链接如下：

<http://www.whitewood.me/2018/06/17/FLIP6- 资源调度模型重构 />

# Apache Flink 进阶(七): 网络流控及反压剖析

作者: 张俊, OPPO 大数据平台研发负责人

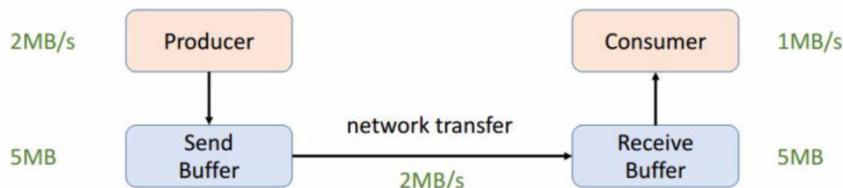
整理: 张友亮

本文根据 Apache Flink 系列直播整理而成, 由 Apache Flink Contributor、OPPO 大数据平台研发负责人张俊老师分享, 社区志愿者张友亮整理。主要内容如下:

- 网络流控的概念与背景
- TCP 的流控机制
- Flink TCP-based 反压机制 (before V1.5)
- Flink Credit-based 反压机制 (since V1.5)
- 总结与思考

## 网络流控的概念与背景

### 1. 为什么需要网络流控



5秒钟后, 将面临如下两种情况之一:

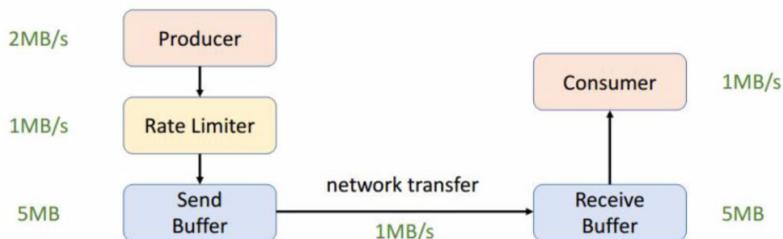
- ✓ bounded receive buffer: consumer 丢弃新到达的数据
- ✓ unbounded receive buffer: buffer 持续扩张, 耗尽 consumer 内存

首先我们可以看下这张最精简的网络流控的图, Producer 的吞吐率是 2MB/s,

Consumer 是 1MB/s，这个时候我们就会发现在网络通信的时候我们的 Producer 的速度是比 Consumer 要快的，有 1MB/s 的这样的速度差，假定我们两端都有一个 Buffer，Producer 端有一个发送用的 Send Buffer，Consumer 端有一个接收用的 Receive Buffer，在网络端的吞吐率是 2MB/s，过了 5s 后我们的 Receive Buffer 可能就撑不住了，这时候会面临两种情况：

- 如果 Receive Buffer 是有界的，这时候新到达的数据就只能被丢弃掉了。
- 如果 Receive Buffer 是无界的，Receive Buffer 会持续的扩张，最终会导致 Consumer 的内存耗尽。

## 2. 网络流控的实现：静态限速



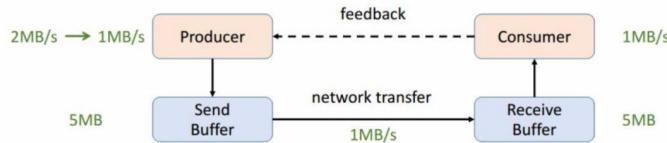
### 静态限流有两点限制：

- ✓ 通常无法事先预估 consumer 端能承受的最大速率
- ✓ consumer 承受能力通常会动态地波动

为了解决这个问题，我们就需要网络流控来解决上下游速度差的问题，传统的做法可以在 Producer 端实现一个类似 Rate Limiter 这样的静态限流，Producer 的发送速率是 2MB/s，但是经过限流这一层后，往 Send Buffer 去传数据的时候就会降到 1MB/s 了，这样的话 Producer 端的发送速率跟 Consumer 端的处理速率就可以匹配起来了，就不会导致上述问题。但是这个解决方案有两点限制：

- 事先无法预估 Consumer 到底能承受多大的速率；
- Consumer 的承受能力通常会动态地波动。

### 3. 网络流控的实现: 动态反馈 / 自动反压



动态反馈分两种，广义上的反压机制都涵盖：

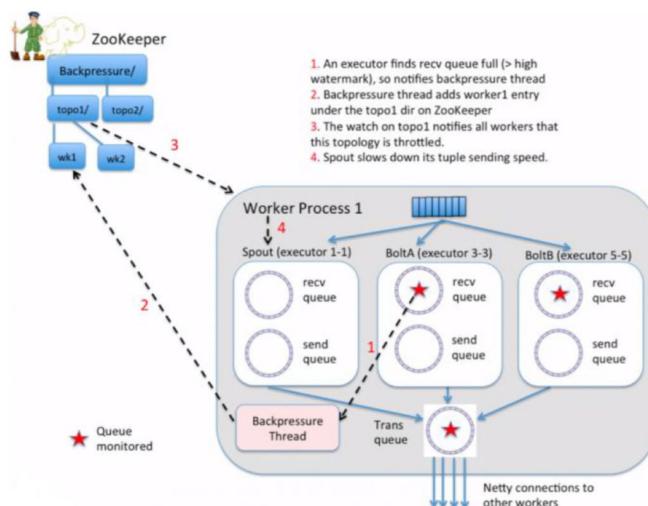
- ✓ 负反馈：接收速率小于发送速率时发生
- ✓ 正反馈：发送速率小于接收速率时发生

针对静态限速的问题我们就演进到了动态反馈（自动反压）的机制，我们需要 Consumer 能够及时的给 Producer 做一个 feedback，即告知 Producer 能够承受的速率是多少。动态反馈分为两种：

- **负反馈**: 接收速率小于发送速率时发生，告知 Producer 降低发送速率；
- **正反馈**: 发送速率小于接收速率时发生，告知 Producer 可以把发送速率提上来。

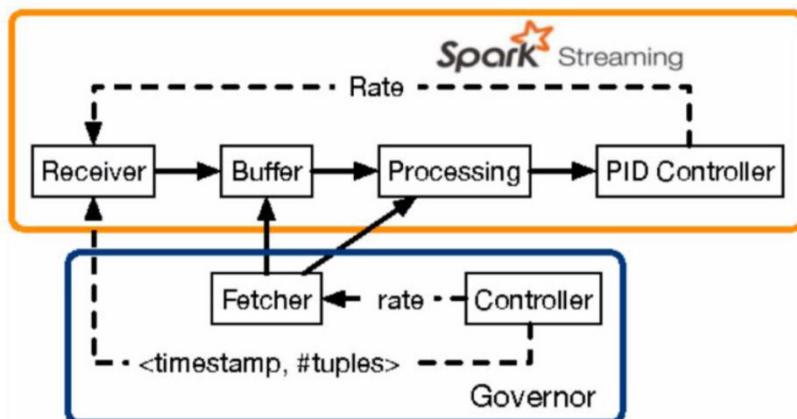
让我们来看几个经典案例：

#### 案例一：Storm 反压实现



上图就是 Storm 里实现的反压机制，可以看到 Storm 在每一个 Bolt 都会有一个监测反压的线程 (Backpressure Thread)，这个线程一但检测到 Bolt 里的接收队列 (recv queue) 出现了严重阻塞就会把这个情况写到 ZooKeeper 里，ZooKeeper 会一直被 Spout 监听，监听到有反压的情况就会停止发送，通过这样的方式匹配上下游的发送接收速率。

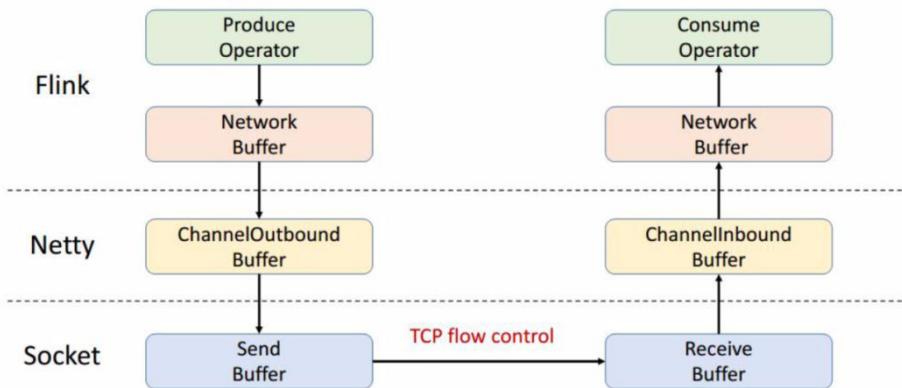
### 案例二: Spark Streaming 反压实现



Spark Streaming 里也有做类似这样的 feedback 机制，上图 Fetcher 会实时的从 Buffer、Processing 这样的节点收集一些指标然后通过 Controller 把速度接收的情况再反馈到 Receiver，实现速率的匹配。

**疑问：为什么 Flink (before V1.5) 里没有用类似的方式实现 feedback 机制？**

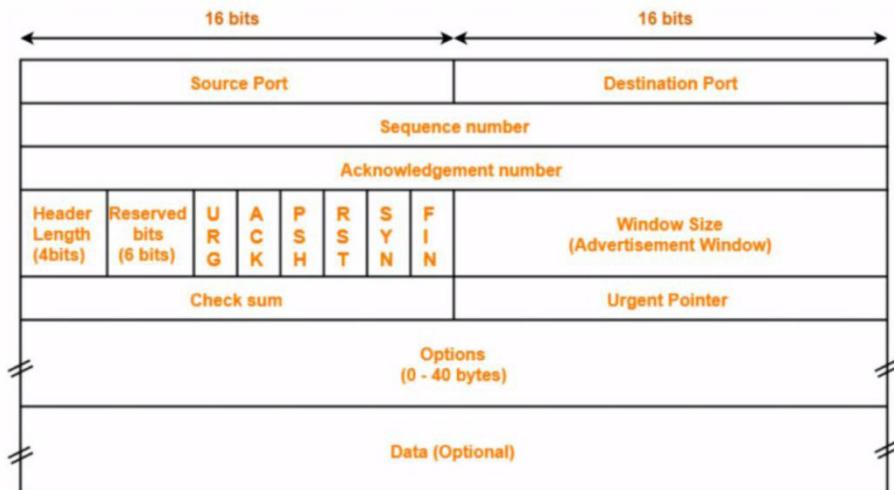
首先在解决这个疑问之前我们需要先了解一下 Flink 的网络传输是一个什么样的架构。



这张图就体现了 Flink 在做网络传输的时候基本的数据的流向，发送端在发送网络数据前要经历自己内部的一个流程，会有一个自己的 Network Buffer，在底层用 Netty 去做通信，Netty 这一层又有属于自己的 ChannelOutbound Buffer，因为最终是要通过 Socket 做网络请求的发送，所以在 Socket 也有自己的 Send Buffer，同样在接收端也有对应的三级 Buffer。学过计算机网络的时候我们应该了解到，TCP 是自带流量控制的。实际上 Flink (before V1.5) 就是通过 TCP 的流控机制来实现 feedback 的。

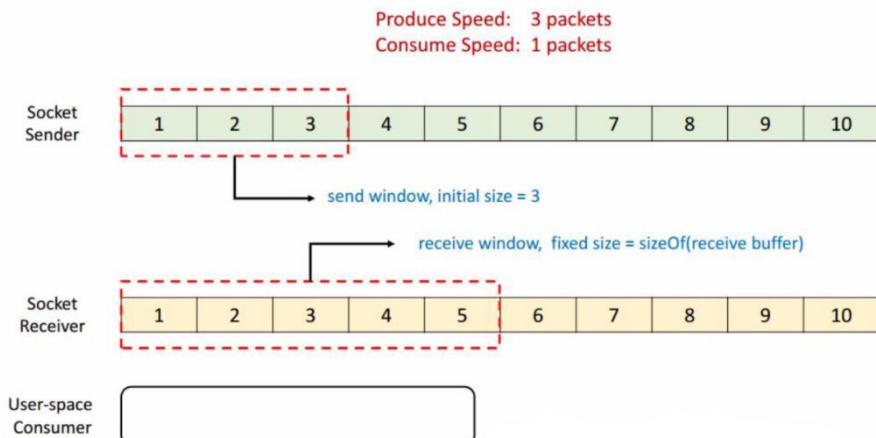
## TCP 流控机制

根据下图我们来简单的回顾一下 TCP 包的格式结构。首先，他有 Sequence number 这样一个机制给每个数据包做一个编号，还有 ACK number 这样一个机制来确保 TCP 的数据传输是可靠的，除此之外还有一个很重要的部分就是 Window Size，接收端在回复消息的时候会通过 Window Size 告诉发送端还可以发送多少数据。



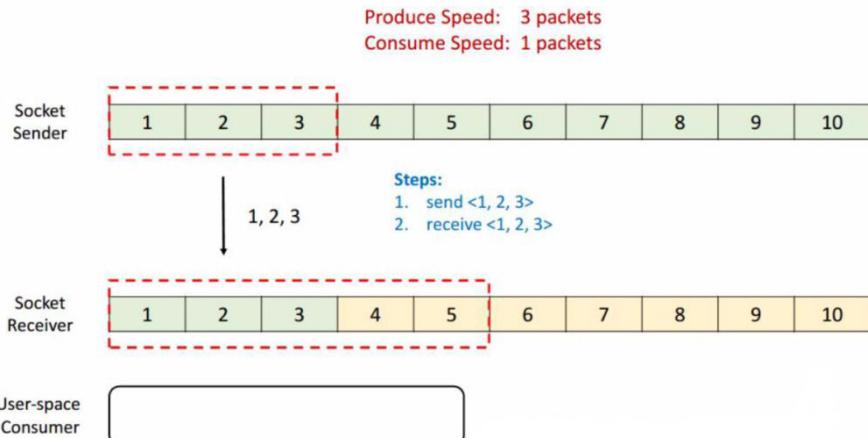
接下来我们来简单看一下这个过程。

## TCP 流控：滑动窗口

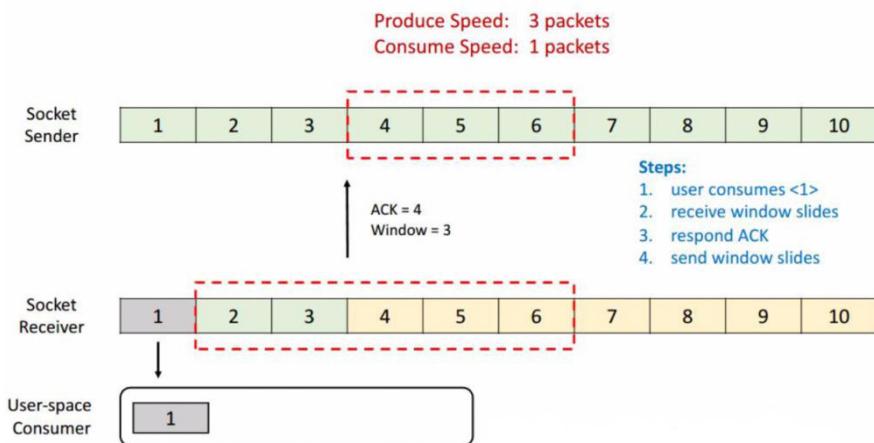


TCP 的流控就是基于滑动窗口的机制，现在我们有一个 Socket 的发送端和一个 Socket 的接收端，目前我们的发送端的速率是我们接收端的 3 倍，这样会发生什

么样的一个情况呢？假定初始的时候我们发送的 window 大小是 3，然后我们接收端的 window 大小是固定的，就是接收端的 Buffer 大小为 5。

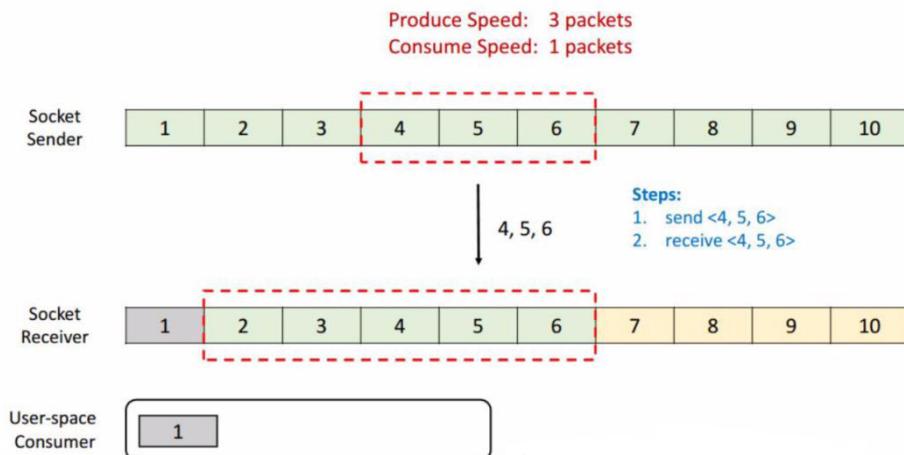


首先，发送端会一次性发 3 个 packets，将 1, 2, 3 发送给接收端，接收端接收到后会将这 3 个 packets 放到 Buffer 里去。

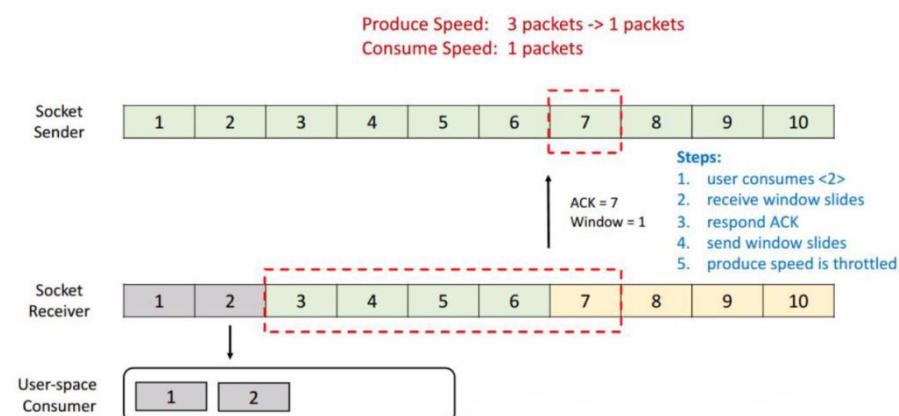


接收端一次消费 1 个 packet，这时候 1 就已经被消费了，然后我们看到接收

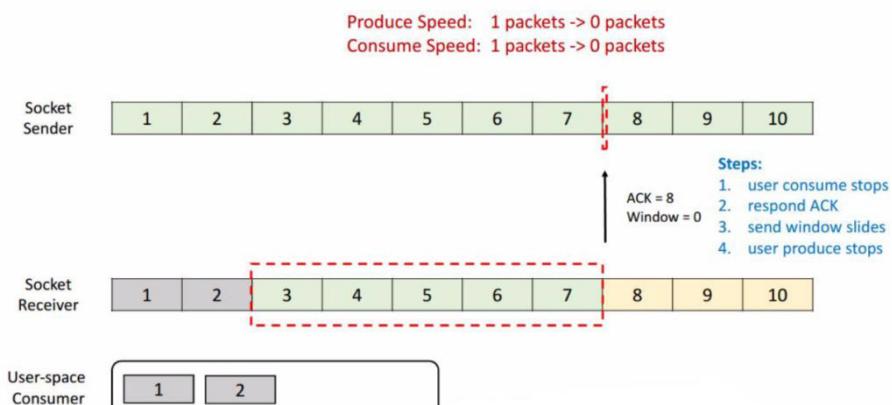
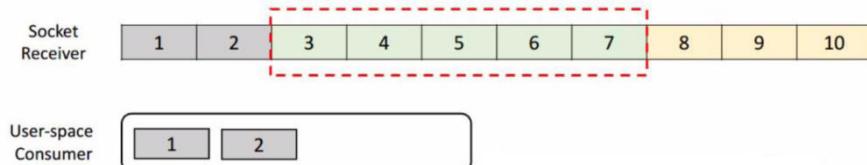
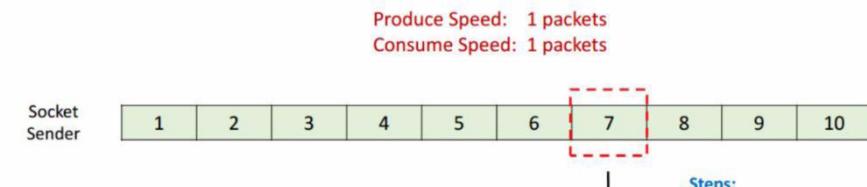
端的滑动窗口会往前滑动一格，这时候 2, 3 还在 Buffer 中而 4, 5, 6 是空出来的，所以接收端会给发送端发送  $ACK = 4$ ，代表发送端可以从 4 开始发送，同时会将 window 设置为 3 (Buffer 的大小 5 减去已经存下的 2 和 3)，发送端接收到回应后也会将他的滑动窗口向前移动到 4, 5, 6。



这时候发送端将 4, 5, 6 发送，接收端也能成功的接收到 Buffer 中去。

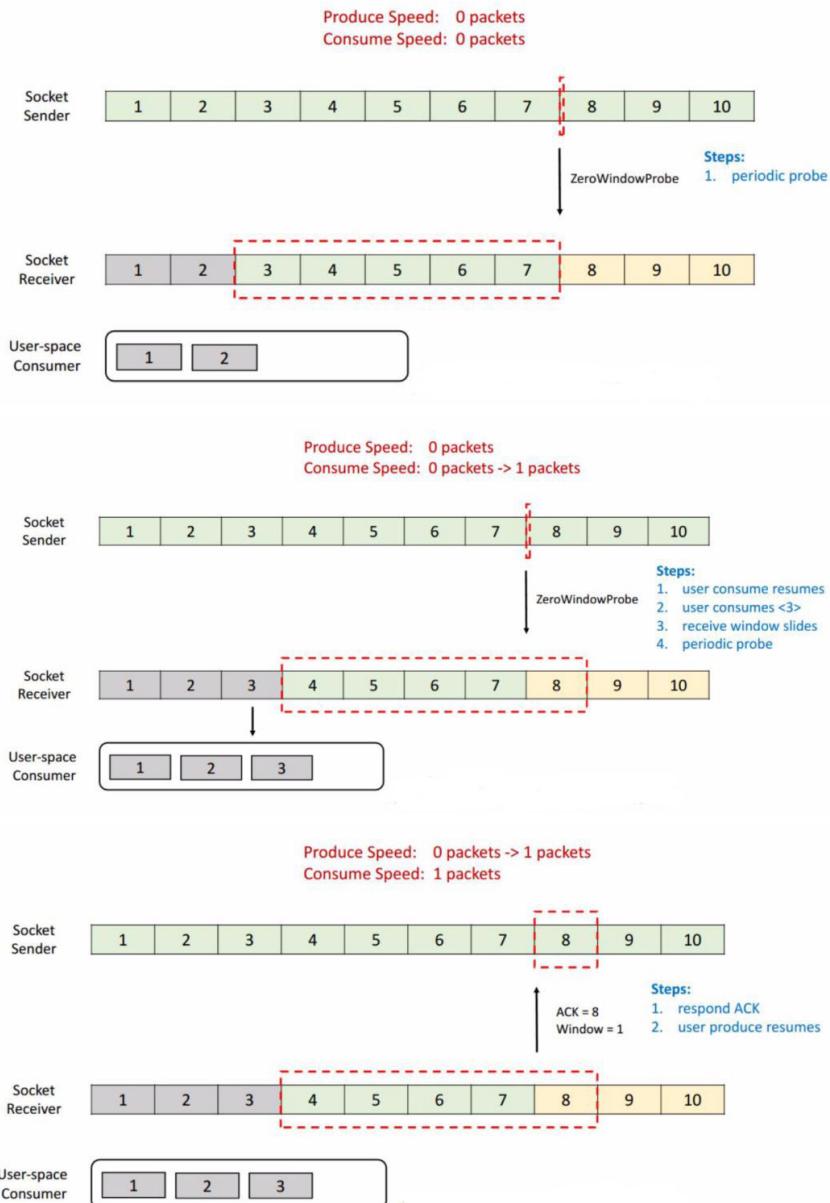


到这一阶段后，接收端就消费到 2 了，同样他的窗口也会向前滑动一个，这时候他的 Buffer 就只剩一个了，于是向发送端发送  $ACK = 7$ 、 $window = 1$ 。发送端收到之后滑动窗口也向前移，但是这个时候就不能移动 3 格了，虽然发送端的速度允许发 3 个 packets 但是 window 传值已经告知只能接收一个，所以他的滑动窗口就只能往前移一格到 7，这样就达到了限流的效果，发送端的发送速度从 3 降到 1。



我们再看一下这种情况，这时候发送端将 7 发送后，接收端接收到，但是由于接收端的消费出现问题，一直没有从 Buffer 中去取，这时候接收端向发送端发送 ACK

= 8、window = 0，由于这个时候 window = 0，发送端是不能发送任何数据，也就会使发送端的发送速度降为 0。这个时候发送端不发送任何数据了，接收端也不进行任何的反馈了，那么如何知道消费端又开始消费了呢？



TCP 当中有一个 ZeroWindowProbe 的机制，发送端会定期的发送 1 个字节的探测消息，这时候接收端就会把 window 的大小进行反馈。当接收端的消费恢复了之后，接收到探测消息就可以将 window 反馈给发送端从而恢复整个流程。TCP 就是通过这样一个滑动窗口的机制实现 feedback。

## Flink TCP-based 反压机制 (before V1.5)

### 1. 示例: WindowWordCount

```
public static void main(String[] args) throws Exception {
    StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

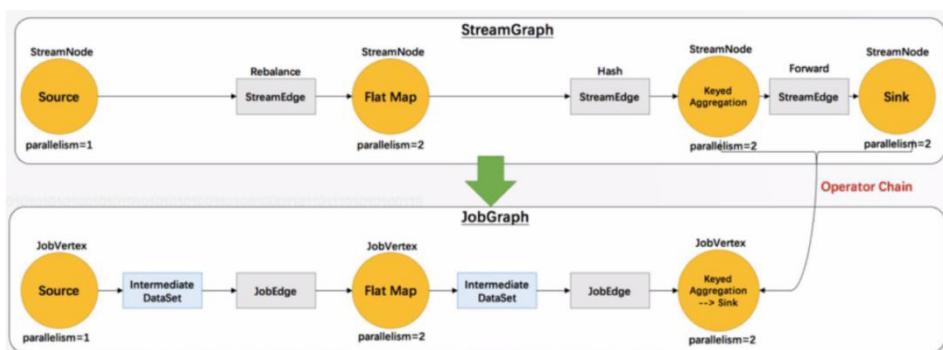
    DataStream<Tuple2<String, Integer>> dataStream = env
        .socketTextStream("localhost", 9999)
        .flatMap(new Splitter())
        .keyBy(0)
        .timeWindow(Time.seconds(5))
        .sum(1);

    dataStream.print();

    env.execute("Window WordCount");
}
```

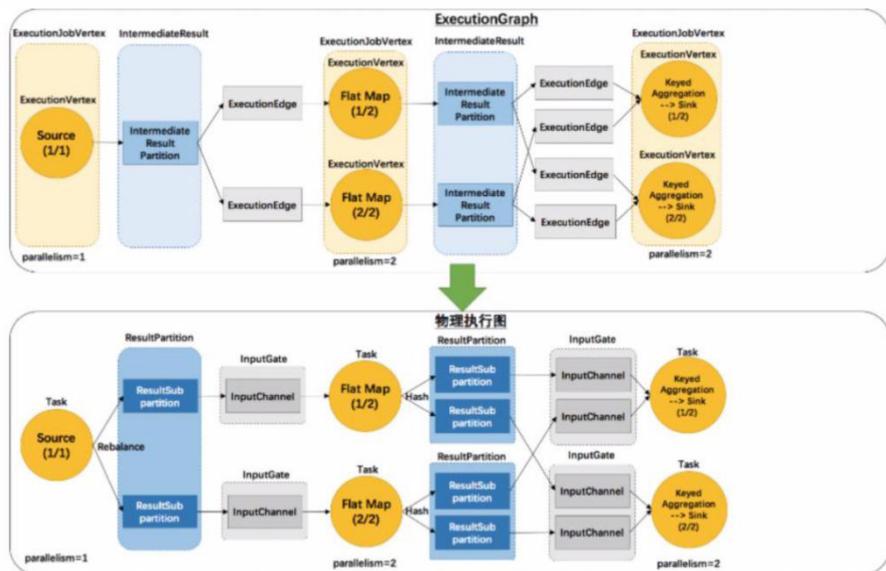
大体的逻辑就是从 Socket 里去接收数据，每 5s 去进行一次 WordCount，将这个代码提交后就进入到了编译阶段。

### 2. 编译阶段: 生成 JobGraph



这时候还没有向集群去提交任务，在 Client 端会将 StreamGraph 生成 JobGraph，JobGraph 就是做为向集群提交的最基本的单元。在生成 JobGraph 的时候会做一些优化，将一些没有 Shuffle 机制的节点进行合并。有了 JobGraph 后就会向集群进行提交，进入运行阶段。

### 3. 运行阶段: 调度 ExecutionGraph

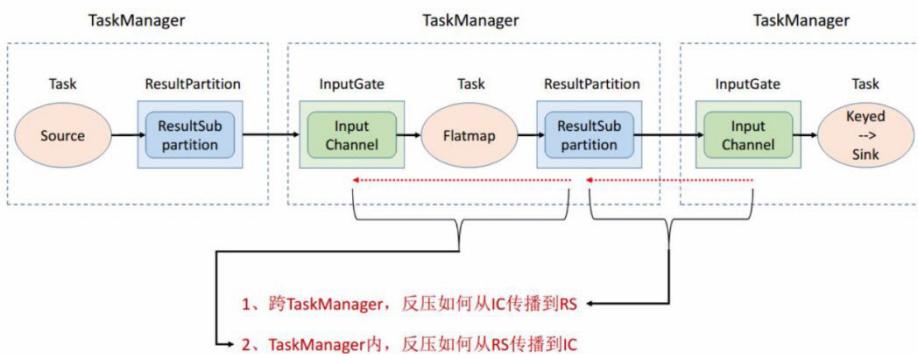


JobGraph 提交到集群后会生成 ExecutionGraph，这时候就已经具备基本的执行任务的雏形了，把每个任务拆解成了不同的 SubTask，上图 ExecutionGraph 中的 Intermediate Result Partition 就是用于发送数据的模块，最终会将 ExecutionGraph 交给 JobManager 的调度器，将整个 ExecutionGraph 调度起来。

然后我们概念化这样一张物理执行图，可以看到每个 Task 在接收数据时都会通过这样一个 InputGate 可以认为是负责接收数据的，再往前有这样一个 ResultPartition 负责发送数据，在 ResultPartition 又会去做分区跟下游的 Task 保持一致，就形成了 ResultSubPartition 和 InputChannel 的对应关系。这就是从逻辑层上来看

的网络传输的通道，基于这么一个概念我们可以将反压的问题进行拆解。

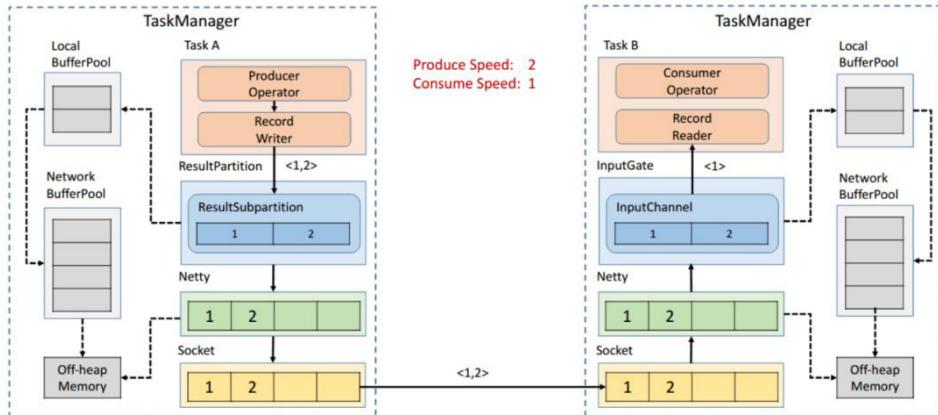
#### 4. 问题拆解：反压传播两个阶段



反压的传播实际上是分为两个阶段的，对应着上面的执行图，我们一共涉及 3 个 TaskManager，在每个 TaskManager 里面都有相应的 Task 在执行，还有负责接收数据的 InputGate，发送数据的 ResultPartition，这就是一个最基本的数据传输的通道。在这时候假设最下游的 Task (Sink) 出现了问题，处理速度降了下来这时候是如何将这个压力反向传播回去呢？这时候就分为两种情况：

- 跨 TaskManager，反压如何从 InputGate 传播到 ResultPartition。
- TaskManager 内，反压如何从 ResultPartition 传播到 InputGate。

## 5. 跨 TaskManager 数据传输

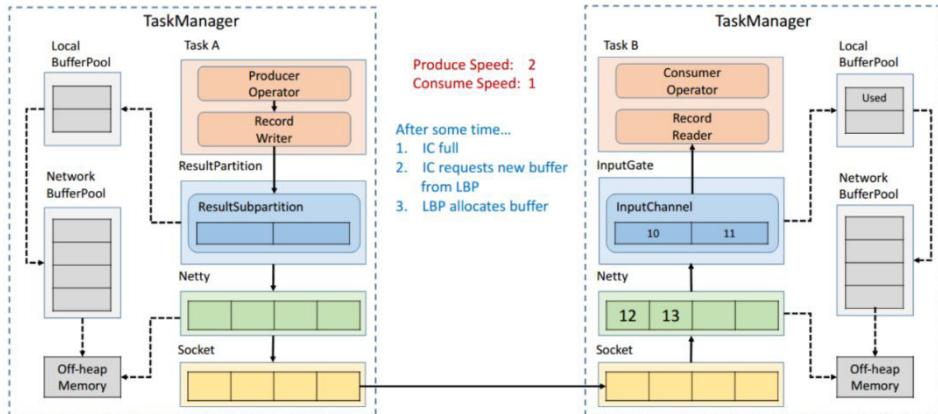


前面提到，发送数据需要 ResultPartition，在每个 ResultPartition 里面会有分区 ResultSubPartition，中间还会有一些关于内存管理的 Buffer。对于一个 TaskManager 来说会有一个统一的 Network BufferPool 被所有的 Task 共享，在初始化时会从 Off-heap Memory 中申请内存，申请到内存的后续内存管理就是同步 Network BufferPool 来进行的，不需要依赖 JVM GC 的机制去释放。有了 Network BufferPool 之后可以为每一个 ResultSubPartition 创建 Local BufferPool。

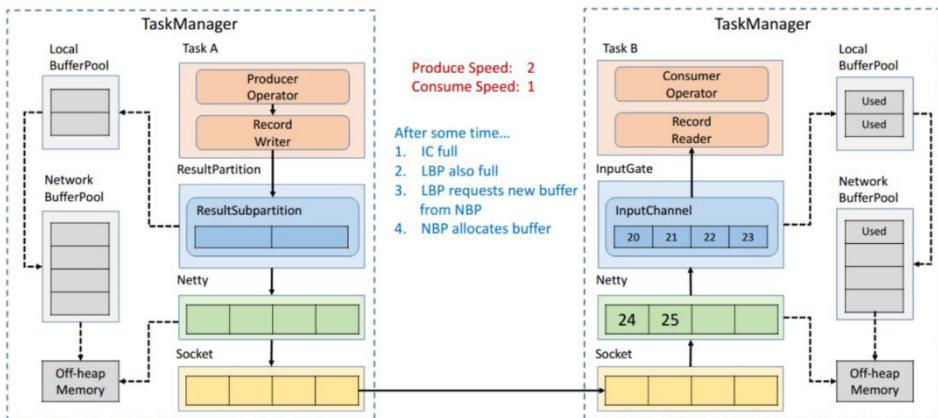
如上图左边的 TaskManager 的 Record Writer 写了 <1, 2> 这个两个数据进来，因为 ResultSubPartition 初始化的时候为空，没有 Buffer 用来接收，就会向 Local BufferPool 申请内存，这时 Local BufferPool 也没有足够的内存于是将请求转到 Network BufferPool，最终将申请到的 Buffer 按原链路返还给 ResultSubPartition，<1, 2> 这个两个数据就可以被写入了。

之后会将 ResultSubPartition 的 Buffer 拷贝到 Netty 的 Buffer 当中最终拷贝到 Socket 的 Buffer 将消息发送出去。然后接收端按照类似的机制去处理将消息消费掉。接下来我们来模拟上下游处理速度不匹配的场景，发送端的速率为 2，接收端的速率为 1，看一下反压的过程是怎样的。

## 6. 跨 TaskManager 反压过程

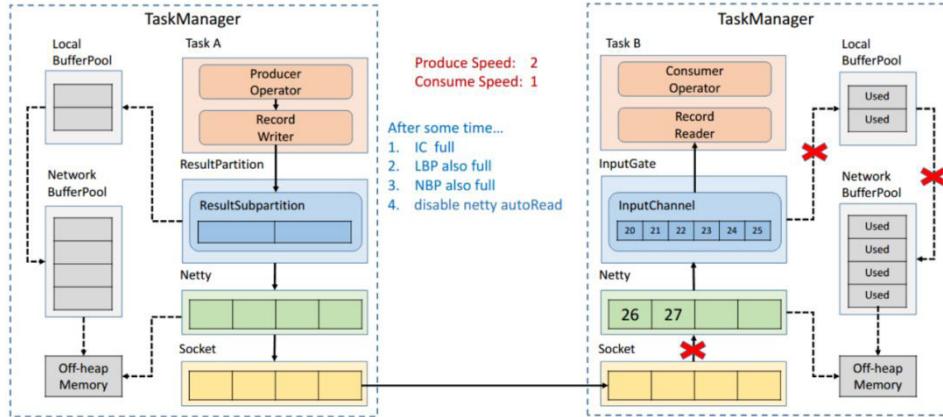


因为速度不匹配就会导致一段时间后 InputChannel 的 Buffer 被用尽，于是他会向 Local BufferPool 申请新的 Buffer，这时候可以看到 Local BufferPool 中的一个 Buffer 就会被标记为 Used。

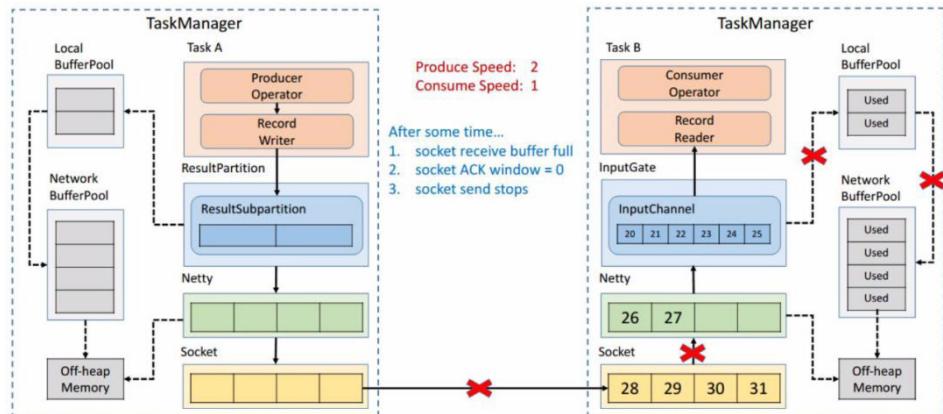


发送端还在持续以不匹配的速度发送数据，然后就会导致 InputChannel 向 Local BufferPool 申请 Buffer 的时候发现没有可用的 Buffer 了，这时候就只能向 Network BufferPool 去申请，当然每个 Local BufferPool 都有最大的可用的 Buffer，防止一个 Local BufferPool 把 Network BufferPool 耗尽。这时候看到

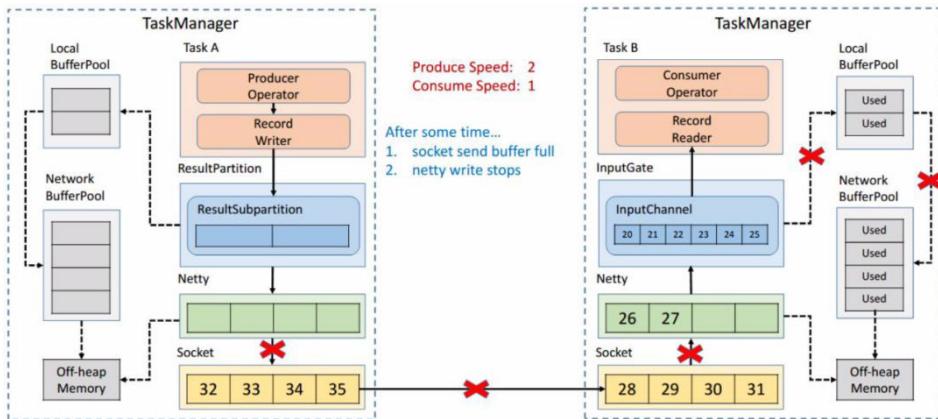
Network BufferPool 还是有可用的 Buffer 可以向其申请。



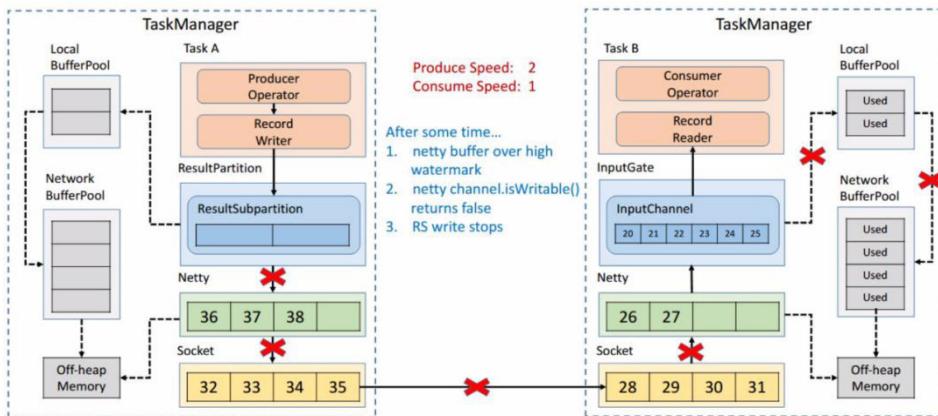
一段时间后，发现 Network BufferPool 没有可用的 Buffer，或是 Local BufferPool 的最大可用 Buffer 到了上限无法向 Network BufferPool 申请，没有办法去读取新的数据，这时 Netty AutoRead 就会被禁掉，Netty 就不会从 Socket 的 Buffer 中读取数据了。



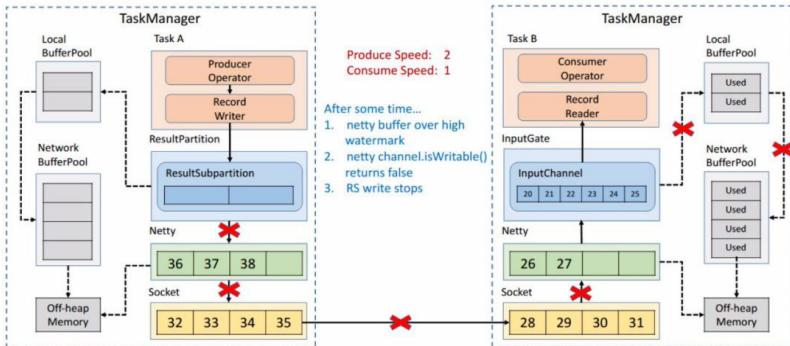
显然，再过不久 Socket 的 Buffer 也被用尽，这时就会将 Window = 0 发送给发送端（前文提到的 TCP 滑动窗口的机制）。这时发送端的 Socket 就会停止发送。



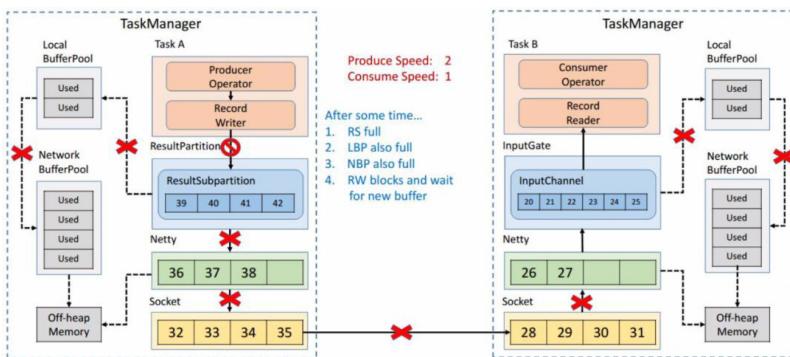
很快发送端的 Socket 的 Buffer 也被用尽, Netty 检测到 Socket 无法写了之后就会停止向 Socket 写数据。



Netty 停止写了之后, 所有的数据就会阻塞在 Netty 的 Buffer 当中了, 但是 Netty 的 Buffer 是无界的, 可以通过 Netty 的水位机制中的 high watermark 控制他的上界。当超过了 high watermark, Netty 就会将其 channel 置为不可写, ResultSubPartition 在写之前都会检测 Netty 是否可写, 发现不可写就会停止向 Netty 写数据。



这时候所有的压力都来到了 ResultSubPartition，和接收端一样他会不断的向 Local BufferPool 和 Network BufferPool 申请内存。

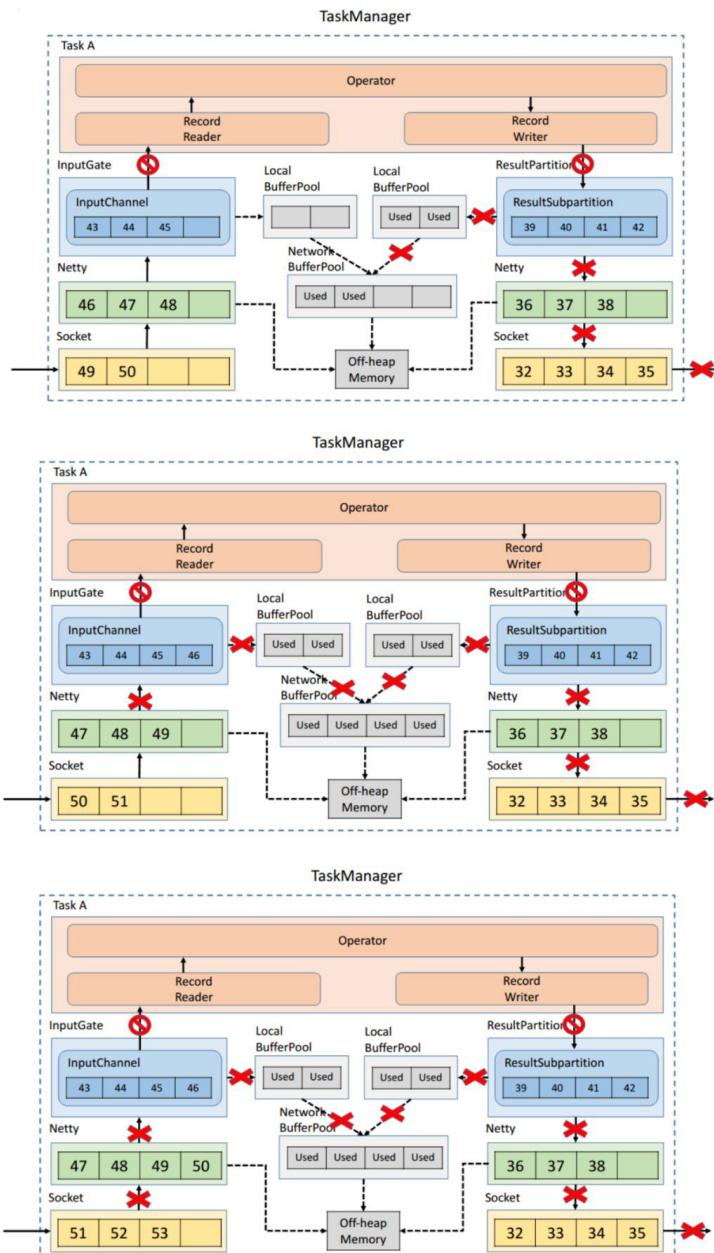


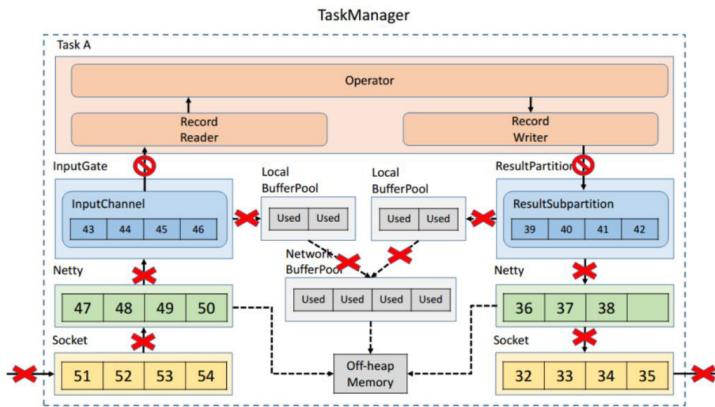
Local BufferPool 和 Network BufferPool 都用尽后整个 Operator 就会停止写数据，达到跨 TaskManager 的反压。

## 7. TaskManager 内反压过程

了解了跨 TaskManager 反压过程后再来看 TaskManager 内反压过程就更好理解了，下游的 TaskManager 反压导致本 TaskManager 的 ResultSubPartition 无法继续写入数据，于是 Record Writer 的写也被阻塞住了，因为 Operator 需要有输入才能有计算后的输出，输入跟输出都是在同一线程执行，Record Writer 阻塞了，Record Reader 也停止从 InputChannel 读数据，这时上游的 TaskManager 还在

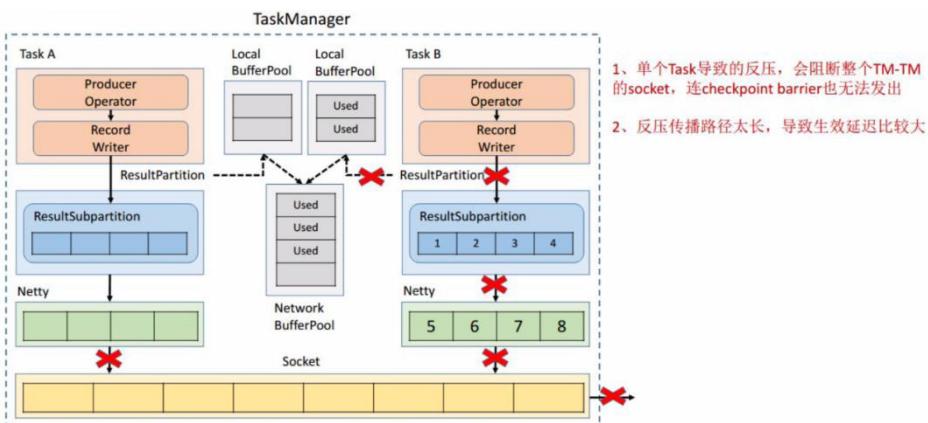
不断地发送数据，最终将这个 TaskManager 的 Buffer 耗尽。具体流程可以参考下图，这就是 TaskManager 内的反压过程。





## Flink Credit-based 反压机制 (since V1.5)

### 1. TCP-based 反压的弊端



在介绍 Credit-based 反压机制之前，先分析下 TCP 反压有哪些弊端。

- 在一个 TaskManager 中可能要执行多个 Task，如果多个 Task 的数据最终都要传输到下游的同一个 TaskManager 就会复用同一个 Socket 进行传输，这个时候如果单个 Task 产生反压，就会导致复用的 Socket 阻塞，其余的 Task 也无法使用传输，checkpoint barrier 也无法发出导致下游执行

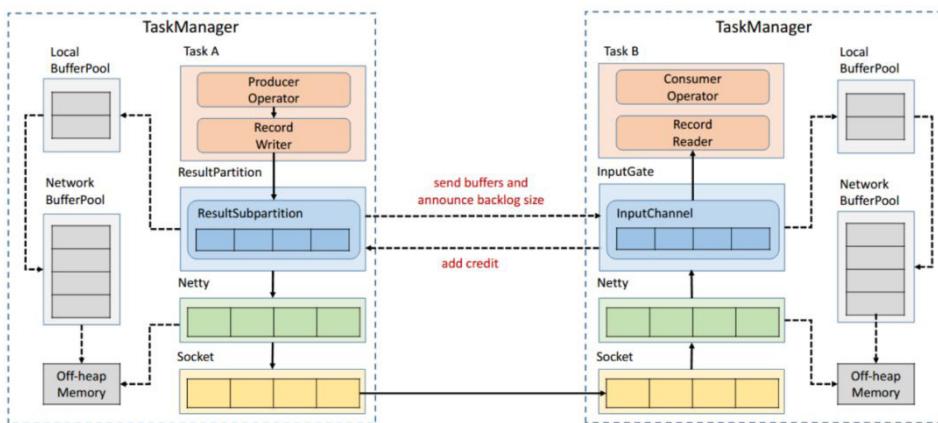
checkpoint 的延迟增大。

- 依赖最底层的 TCP 做流控，会导致反压传播路径太长，导致生效的延迟比较大。

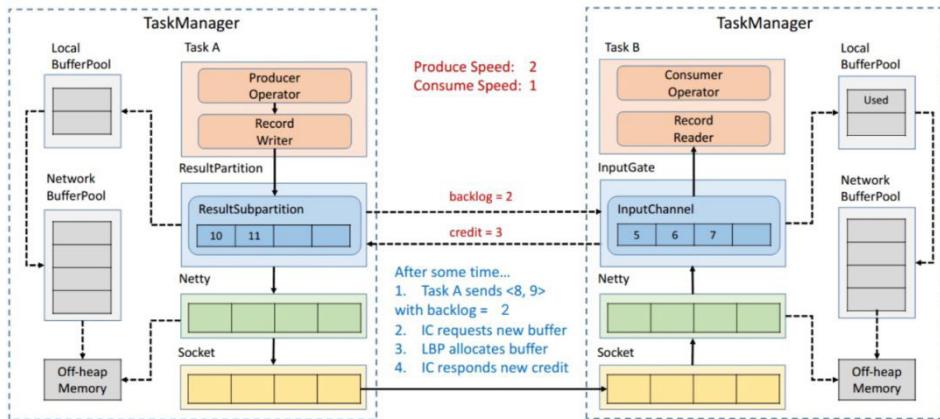
## 2. 引入 Credit-based 反压

这个机制简单的理解起来就是在 Flink 层面实现类似 TCP 流控的反压机制来解决上述的弊端，Credit 可以类比为 TCP 的 Window 机制。

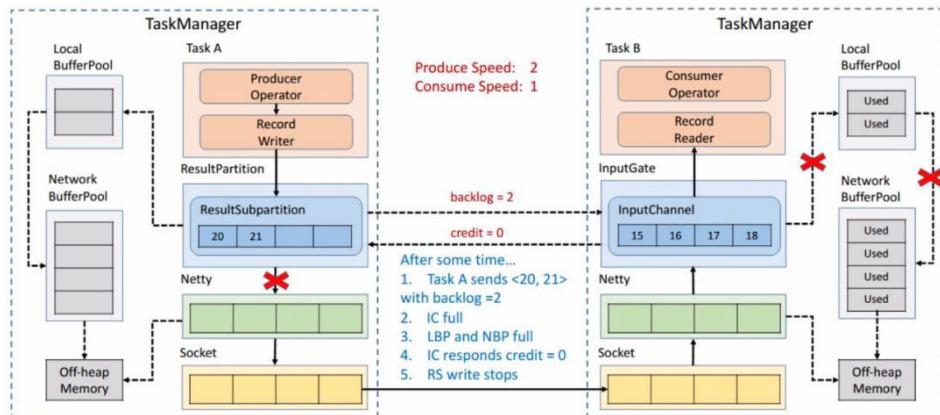
## 3. Credit-based 反压过程



如图所示在 Flink 层面实现反压机制，就是每一次 ResultSubPartition 向 InputChannel 发送消息的时候都会发送一个 backlog size 告诉下游准备发送多少消息，下游就会去计算有多少的 Buffer 去接收消息，算完之后如果有充足的 Buffer 就会返还给上游一个 Credit 告知他可以发送消息（图上两个 ResultSubPartition 和 InputChannel 之间是虚线是因为最终还是要通过 Netty 和 Socket 去通信），下面我们看一个具体示例。



假设我们上下游的速度不匹配，上游发送速率为 2，下游接收速率为 1，可以看到图上在 ResultSubPartition 中累积了两条消息，10 和 11，backlog 就为 2，这时就会将发送的数据  $<8,9>$  和 backlog = 2 一同发送给下游。下游收到了之后就会去计算是否有 2 个 Buffer 去接收，可以看到 InputChannel 中已经不足了这时就会从 Local BufferPool 和 Network BufferPool 申请，好在这个时候 Buffer 还是可以申请到的。



过了一段时间后由于上游的发送速率要大于下游的接受速率，下游的 TaskManager 的 Buffer 已经到达了申请上限，这时候下游就会向上游返回 Credit = 0，

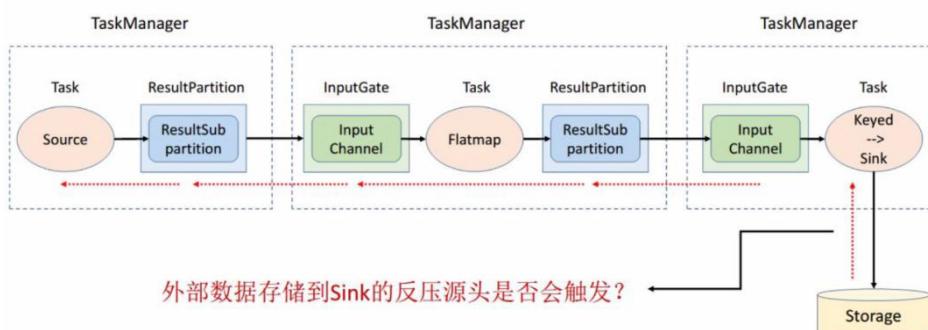
ResultSubPartition 接收到之后就不会向 Netty 去传输数据，上游 TaskManager 的 Buffer 也很快耗尽，达到反压的效果，这样在 ResultSubPartition 层就能感知到反压，不用通过 Socket 和 Netty 一层层地向上反馈，降低了反压生效的延迟。同时也不会将 Socket 去阻塞，解决了由于一个 Task 反压导致 TaskManager 和 TaskManager 之间的 Socket 阻塞的问题。

## 总结与思考

### 1. 总结：

- 网络流控是为了在上下游速度不匹配的情况下，防止下游出现过载。
- 网络流控有静态限速和动态反压两种手段。
- Flink 1.5 之前是基于 TCP 流控 + bounded buffer 实现反压。
- Flink 1.5 之后实现了自己托管的 credit – based 流控机制，在应用层模拟 TCP 的流控机制。

### 2. 思考：有了动态反压，静态限速是不是完全没有作用了？



实际上动态反压不是万能的，我们流计算的结果最终是要输出到一个外部的存储 (Storage)，外部数据存储到 Sink 端的反压是不一定会触发的，这要取决于外部存储的实现，像 Kafka 这样是实现了限流限速的消息中间件可以通过协议将反压反馈给 Sink 端，但是像 ES 无法将反压进行传播反馈给 Sink 端，这种情况下为了

防止外部存储在大的数据量下被打爆，我们就可以通过静态限速的方式在 Source 端去做限流。

所以说动态反压并不能完全替代静态限速的，需要根据合适的场景去选择处理方案。

# Apache Flink 进阶 (八): 详解 Metrics 原理与实战

作者: 刘彪

Apache Flink Contributor

本文由 Apache Flink Contributor 刘彪分享, 对什么是 Metrics、如何使用 Metrics 两大问题进行了详细的介绍, 并对 Metrics 监控实战进行解释说明。

## 什么是 Metrics ?

Flink 提供的 Metrics 可以在 Flink 内部收集一些指标, 通过这些指标让开发人员更好地理解作业或集群的状态。由于集群运行后很难发现内部的实际状况, 跑得慢或快, 是否异常等, 开发人员无法实时查看所有的 Task 日志, 比如作业很大或者有很多作业的情况下, 该如何处理? 此时 Metrics 可以很好的帮助开发人员了解作业的当前状况。

## Metric Types

Metrics 的类型如下:

1. **首先**, 常用的如 Counter, 写过 mapreduce 作业的开发人员就应该很熟悉 Counter, 其实含义都是一样的, 就是对一个计数器进行累加, 即对于多条数据和多兆数据一直往上加的过程。
2. **第二**, Gauge, Gauge 是最简单的 Metrics, 它反映一个值。比如要看现在 Java heap 内存用了多少, 就可以每次实时的暴露一个 Gauge, Gauge 当前的值就是 heap 使用的量。
3. **第三**, Meter, Meter 是指统计吞吐量和单位时间内发生“事件”的次数。它相当于求一种速率, 即事件次数除以使用的时间。
4. **第四**, Histogram, Histogram 比较复杂, 也并不常用, Histogram 用于统

计一些数据的分布，比如说 Quantile、Mean、StdDev、Max、Min 等。

## Metric Group

Metric 在 Flink 内部有多层结构，以 Group 的方式组织，它并不是一个扁平化的结构，Metric Group + Metric Name 是 Metrics 的唯一标识。

Metric Group 的层级有 TaskManagerMetricGroup 和 TaskManagerJob-MetricGroup，每个 Job 具体到某一个 task 的 group，task 又分为 TaskIOMetricGroup 和 OperatorMetricGroup。Operator 下面也有 IO 统计和一些 Metrics，整个层级大概如下图所示。Metrics 不会影响系统，它处在不同的组中，并且 Flink 支持自己去加 Group，可以有自己的层级。

```

•TaskManagerMetricGroup
  •TaskManagerJobMetricGroup
    •TaskMetricGroup
      •TaskIOMetricGroup
      •OperatorMetricGroup
        •${User-defined Group} / ${User-defined Metrics}
        •OperatorIOMetricGroup
  •JobManagerMetricGroup
    •JobManagerJobMetricGroup

```

JobManagerMetricGroup 相对简单，相当于 Master，它的层级也相对较少。

Metrics 定义还是比较简单的，即指标的信息可以自己收集，自己统计，在外部系统能够看到 Metrics 的信息，并能够对其进行聚合计算。

## 如何使用 Metrics ?

### System Metrics

System Metrics，将整个集群的状态已经涵盖得非常详细。具体包括以下方面：

- Master 级别和 Work 级别的 JVM 参数，如 load 和 time；其 Memory 划分也很详细，包括 heap 的使用情况、non-heap 的使用情况、direct 的使用情

况，以及 mapped 的使用情况；Threads 可以看到具体有多少线程；还有非常实用的 Garbage Collection。

- Network 使用比较广泛，当需要解决一些性能问题的时候，Network 非常实用。Flink 不只是网络传输，还是一个有向无环图的结构，可以看到它的每个上下游都是一种简单的生产者消费者模型。Flink 通过网络相当于标准的生产者和消费者中间通过有限长度的队列模型。如果想要评估定位性能，中间队列会迅速缩小问题的范围，能够很快的找到问题瓶颈。

```
•CPU  
•Memory  
•Threads  
•Garbage Collection  
•Network  
•Classloader  
•Cluster  
•Availability  
•Checkpointing  
•StateBackend  
•IO  
•详见 : [https://ci.apache.org/projects/flink/flink-docs-release-1.8/  
monitoring/metrics.html#system-metrics] (https://ci.apache.org/projects/flink/  
flink-docs-release-1.8/monitoring/metrics.html)
```

- 运维集群的人会比较关心 Cluster 的相关信息，如果作业太大，则需要非常关注 Checkpointing，它有可能会在一些常规的指标上无法体现出潜在问题。比如 Checkpointing 长时间没有工作，数据流看起来没有延迟，此时可能会出现作业一切正常的假象。另外，如果进行了一轮 failover 重启之后，因为 Checkpointing 长时间没有工作，有可能会回滚到很长一段时间之前的状态，整个作业可能就直接废掉了。
- RocksDB 是生产环境当中比较常用的 state backend 实现，如果数据量足够大，就需要多关注 RocksDB 的 Metrics，因为它随着数据量的增大，性能可能会下降。

## User-defined Metrics

除了系统的 Metrics 之外，Flink 支持自定义 Metrics，即 User-defined

Metrics。上文说的都是系统框架方面，对于自己的业务逻辑也可以用 Metrics 来暴露一些指标，以便进行监控。

User-defined Metrics 现在提及的都是 datastream 的 API，table、sql 可能需要 context 协助，但如果写 UDF，它们其实是大同小异的。

Datastream 的 API 是继承 RichFunction，继承 RichFunction 才可以有 Metrics 的接口。然后通过 RichFunction 会带来一个 getRuntimeContext().getMetricGroup().addGroup(…) 的方法，这里就是 User-defined Metrics 的入口。通过这种方式，可以自定义 user-defined Metric Group。如果想定义具体的 Metrics，同样需要用 getRuntimeContext().getMetricGroup().counter/gauge/meter/histogram(… ) 方法，它会有相应的构造函数，可以定义到自己的 Metrics 类型中。

```
继承 RichFunction
•Register user-defined Metric Group: getRuntimeContext() .
getMetricGroup() .addGroup(…)
•Register user-defined Metric: getRuntimeContext() .getMetricGroup() .
counter/gauge/meter/histogram(…)
```

## User-defined Metrics Example

下面通过一段简单的例子说明如何使用 Metrics。比如，定义了一个 Counter 传一个 name，Counter 默认的类型是 single counter (Flink 内置的一个实现)，可以对 Counter 进行 inc( ) 操作，并在代码里面直接获取。

Meter 也是这样，Flink 有一个内置的实现是 Meterview，因为 Meter 是多长时间内发生事件的记录，所以它是要有一个多长时间的窗口。平常用 Meter 时直接 markEvent()，相当于加一个事件不停地打点，最后用 getrate( ) 的方法直接把这一段时间发生的事件除一下给算出来。

Gauge 就比较简单了，把当前的时间打出来，用 Lambda 表达式直接把 System::currentTimeMillis 打进去就可以，相当于每次调用的时候都会去真正调一下系统当天时间进行计算。

Histogram 稍微复杂一点, Flink 中代码提供了两种实现, 在此取一其中个实现, 仍然需要一个窗口大小, 更新的时候可以给它一个值。

这些 Metrics 一般都不是线程安全的。如果想要用多线程, 就需要加同步, 更多详情请参考下面链接。

```

•Counter processedCount = getRuntimeContext().getMetricGroup().
    counter("processed_count");  processedCount.inc();
•Meter processRate = getRuntimeContext().getMetricGroup().meter("rate", new
    MeterView(60));  processRate.markEvent();
•getRuntimeContext().getMetricGroup().gauge("current_timestamp",
    System::currentTimeMillis);
•Histogram histogram = getRuntimeContext().getMetricGroup().
    histogram("histogram", new DescriptiveStatisticsHistogram(1000));  histogram.
    update(1024);
•[https://ci.apache.org/projects/flink/flink-docs-release-1.8/monitoring/metrics.html#metric-types]
```

## 获取 Metrics

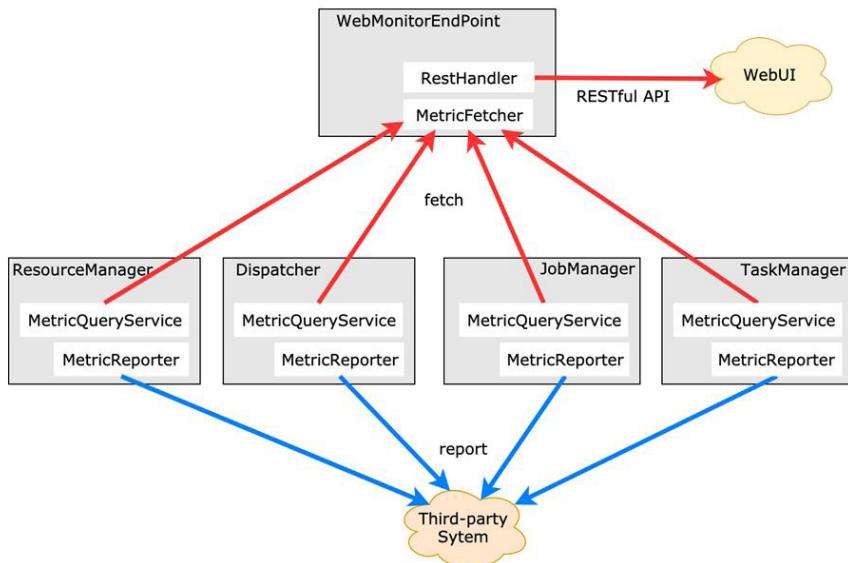
获取 Metrics 有三种方法, 首先可以在 WebUI 上看到; 其次可以通过 RESTful API 获取, RESTful API 对程序比较友好, 比如写自动化脚本或程序, 自动化运维和测试, 通过 RESTful API 解析返回的 Json 格式对程序比较友好; 最后, 还可以通过 Metric Reporter 获取, 监控主要使用 Metric Reporter 功能。

获取 Metrics 的方式在物理架构上是怎样实现的?

了解背景和原理会对使用有更深刻的理解。WebUI 和 RESTful API 是通过中心化节点定期查询把各个组件中的 Metrics 拉上来的实现方式。其中, fetch 不一定是实时更新的, 默认为 10 秒, 所以有可能在 WebUI 和 RESTful API 中刷新的数据不是实时想要得到的数据; 此外, fetch 有可能不同步, 比如两个组件, 一边在加另一边没有动, 可能是由于某种原因超时没有拉过来, 这样是无法更新相关值的, 它是 try best 的操作, 所以有时我们看到的指标有可能会延迟, 或许等待后相关值就更新了。

红色的路径通过 MetricFetcher, 会有一个中心化的节点把它们聚合在一起展

示。而 MetricReporter 不一样，每一个单独的点直接汇报，它没有中心化节点帮助做聚合。如果想要聚合，需要在第三方系统中进行，比如常见的 TSDB 系统。当然，不是中心化结构也是它的好处，它可以免去中心化节点带来的问题，比如内存放不下等，MetricReporter 把原始数据直接 Reporter 出来，用原始数据做处理会有更强大的功能。



## Metric Reporter

Flink 内置了很多 Reporter，对外部系统的技术选型可以参考，比如 JMX 是 java 自带的技术，不严格属于第三方。还有 InfluxDB、Prometheus、Slf4j（直接打 log 里）等，调试时候很好用，可以直接看 logger，Flink 本身自带日志系统，会打到 Flink 框架包里面去。详见：

- Flink 内置了很多 Reporter，对外部系统的技术选型可以参考，详见：[<https://ci.apache.org/projects/flink/flink-docs-release-1.8/monitoring/metrics.html#reporter>] (<https://ci.apache.org/projects/flink/flink-docs-release-1.8/monitoring/metrics.html>)
- Metric Reporter Configuration Example  
metrics.reporters: your\_monitor,jmx

```

metrics.reporter.jmx.class: org.apache.flink.metrics.jmx.JMXReporter
metrics.reporter.jmx.port: 1025-10000
metrics.reporter.your_monitor.class: com.your_company.YourMonitorClass
metrics.reporter.your_monitor.interval: 10 SECONDS
metrics.reporter.your_monitor.config.a: your_a_value
metrics.reporter.your_monitor.config.b: your_b_value

```

Metric Reporter 是如何配置的? 如上所示, 首先 Metrics Reporters 的名字用逗号分隔, 然后通过 metrics.reporter.jmx.class 的 classname 反射找 reporter, 还需要拿到 metrics.reporter.jmx.port 的配置, 比如像第三方系统通过网络发送的比较多。但要知道往哪里发, ip 地址、port 信息是比较常见的。此外还有 metrics.reporter.your\_monitor.class 是必须要有的, 可以自己定义间隔时间, Flink 可以解析, 不需要自行去读, 并且还可以写自己的 config。

## 实战: 利用 Metrics 监控

常用 Metrics 做自动化运维和性能分析。

### 自动化运维



自动化运维怎么做?

- 首先，收集一些关键的 Metrics 作为决策依据，利用 Metric Reporter 收集 Metrics 到存储 / 分析系统（例如 TSDB），或者直接通过 RESTful API 获取。
- 有了数据之后，可以定制监控规则，关注关键指标，Failover、Checkpoint、业务 Delay 信息。定制规则用途最广的是可以用来报警，省去很多人工的工作，并且可以定制 failover 多少次时需要人为介入。
- 当出现问题时，有钉钉报警、邮件报警、短信报警、电话报警等通知工具。
- 自动化运维的优势是可以通过大屏、报表的形式清晰的查看数据，通过大屏时刻了解作业总体信息，通过报表分析优化。

## 性能分析

性能分析一般遵循如下的流程：

## 性能分析



首先从发现问题开始，如果有 Metrics 系统，再配上监控报警，就可以很快定位问题。然后对问题进行剖析，大盘看问题会比较方便，通过具体的 System Metrics 分析，缩小范围，验证假设，找到瓶颈，进而分析原因，从业务逻辑、JVM、操作系统、State、数据分布等多维度进行分析；如果还不能找到问题原因，就只能借助 profiling 工具了。

## 实战：“我的任务慢，怎么办”

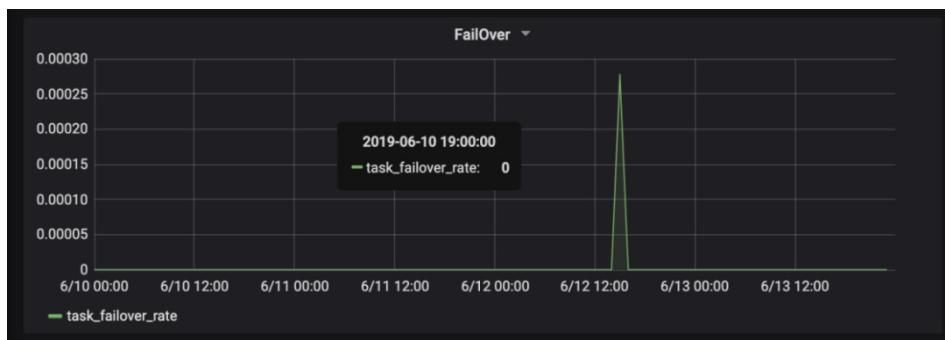
“任务慢，怎么办？”可以称之为无法解答的终极问题之一。

其原因在于这种问题是系统框架问题，比如看医生时告诉医生身体不舒服，然后就让医生下结论。而通常医生需要通过一系列的检查来缩小范围，确定问题。同理，任务慢的问题也需要经过多轮剖析才能得到明确的答案。

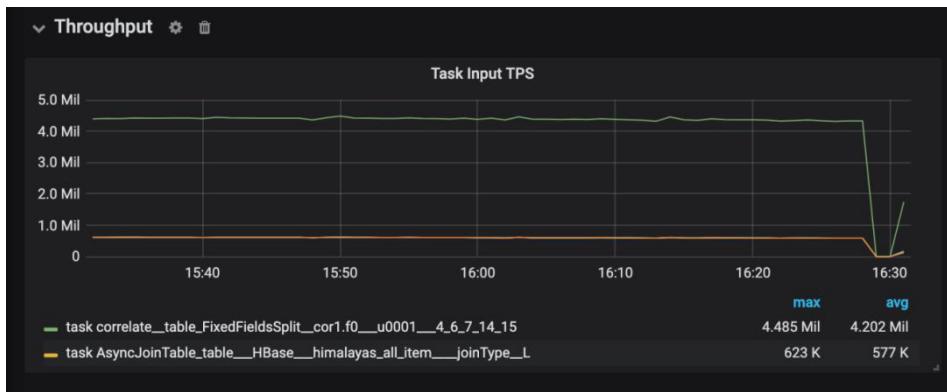
除了不熟悉 Flink 机制以外，大多数人的问题是对于整个系统跑起来是黑盒，根本不知道系统在如何运行，缺少信息，无法了解系统状态。此时，一个有效的策略是求助 Metrics 来了解系统内部的状况，下面通过一些具体的例子来说明。

- **发现问题**

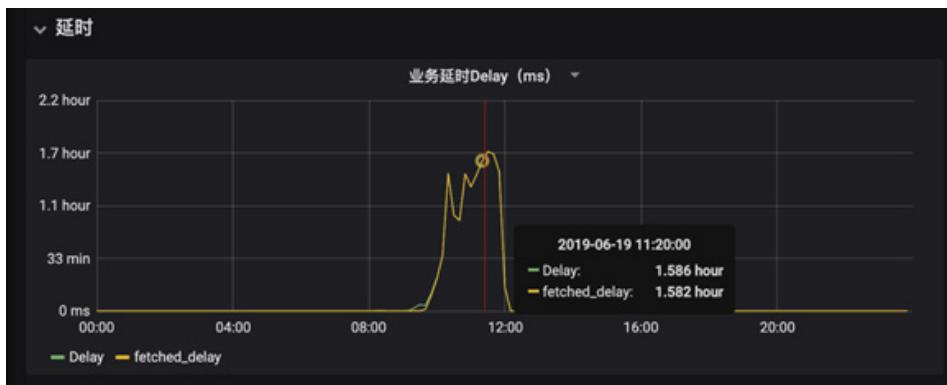
比如下图 failover 指标，线上有一个不是 0，其它都是 0，此时就发现问题了。



再比如下图 Input 指标正常都在四、五百万，突然跌成 0，这里也存在问题。

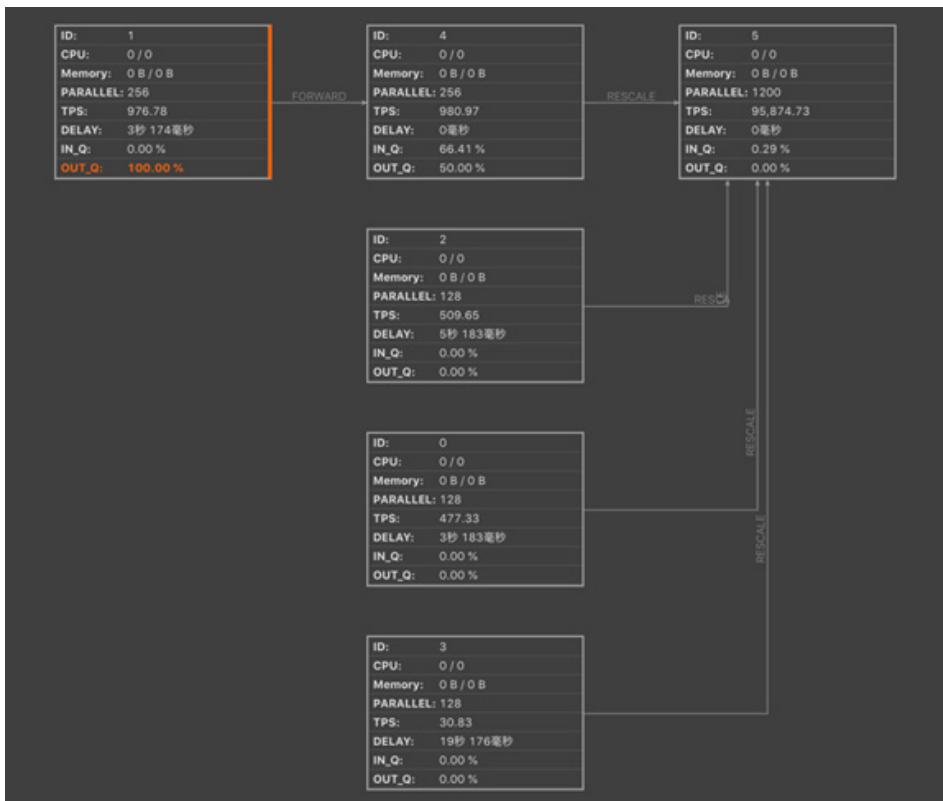


业务延时问题如下图，比如处理到的数据跟当前时间比对，发现处理的数据是一小时前的数据，平时都是处理一秒之前的数据，这也是有问题的。



- 缩小范围，定位瓶颈

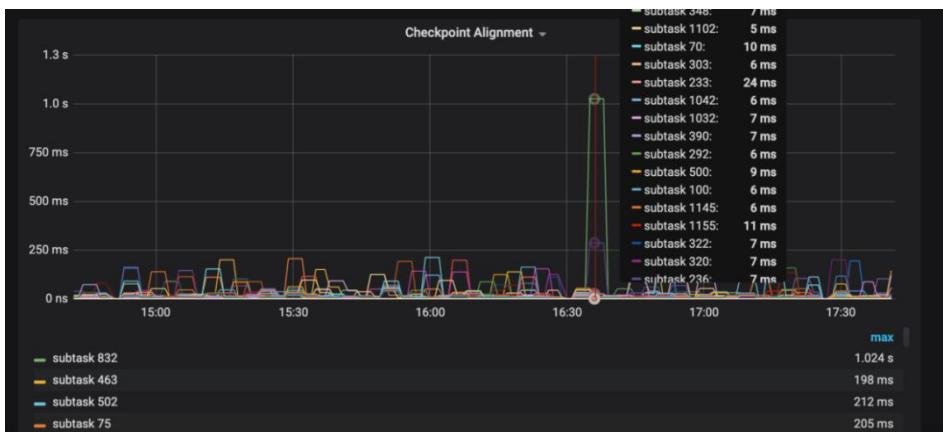
当出现一个地方比较慢，但是不知道哪里慢时，如下图红色部分，OUTQ 并发值已经达到 100% 了，其它都还比较正常，甚至优秀。到这里生产者消费者模型出现了问题，生产者 INQ 是满的，消费者 OUT\_Q 也是满的，从图中看出节点 4 已经很慢了，节点 1 产生的数据节点 4 处理不过来，而节点 5 的性能都很正常，说明节点 1 和节点 4 之间的队列已经堵了，这样我们就可以重点查看节点 1 和节点 4，缩小了问题范围。



500 个 InBps 都具有 256 个 PARALLEL，这么多个点不可能一一去看，因此需要在聚合时把 index 是第几个并发做一个标签。聚合按着标签进行划分，看哪一个并发是 100%。在图中可以划分出最高的两个线，即线 324 和线 115，这样就又进一步的缩小了范围。



利用 Metrics 缩小范围的方式如下图所示，就是用 Checkpoint Alignment 进行对齐，进而缩小范围，但这种方法用的较少。



- 多维度分析

分析任务有时候为什么特别慢呢？

当定位到某一个 Task 处理特别慢时，需要对慢的因素做出分析。分析任务慢的因素是有优先级的，可以从上向下查，由业务方面向底层系统。因为大部分问题都出现在业务维度上，比如查看业务维度的影响可以有以下几个方面，并发度是否合理、数据波峰波谷、数据倾斜；其次依次从 Garbage Collection、Checkpoint Alignment、State Backend 性能角度进行分析；最后从系统性能角度进行分析，比如 CPU、内存、Swap、Disk IO、吞吐量、容量、Network IO、带宽等。

## Q & A

**Q: Metrics 是系统内部的监控，那是否可以作为 Flink 日志分析的输出？**

可以，但是没有必要，都用 Flink 去处理其他系统的日志了，输出或报警直接当做 sink 输出就好了。因为 Metrics 是统计内部状态，你这是处理正常输入数据，直接输出就可以了。

**Q: Reporter 是有专门的线程吗?**

每个 Reporter 都有自己单独的线程。在 Flink 的内部，线程其实还是挺多的，如果跑一个作业，直接到 TaskManager 上，jstack 就能看到线程的详情。

# Apache Flink 进阶(九): Flink Connector 开发

作者: 董亭亭

快手大数据架构实时计算引擎团队负责人

本文主要分享 Flink connector 相关内容，分为以下三个部分的内容：第一部分会首先介绍一下 Flink Connector 有哪些。第二部分会重点介绍在生产环境中经常使用的 kafka connector 的基本的原理以及使用方法。第三部分答疑，对社区反馈的问题进行答疑。

## Flink Streaming Connector

Flink 是新一代流批统一的计算引擎，它需要从不同的第三方存储引擎中把数据读过来，进行处理，然后再写出到另外的存储引擎中。Connector 的作用就相当于一个连接器，连接 Flink 计算引擎跟外界存储系统。Flink 里有以下几种方式，当然也不限于这几种方式可以跟外界进行数据交换：

- 第一种 Flink 里面预定义了一些 source 和 sink。
- 第二种 Flink 内部也提供了一些 Boundled connectors。
- 第三种可以使用第三方 Apache Bahir 项目中提供的连接器。
- 第四种是通过异步 IO 方式。

下面分别简单介绍一下这四种数据读写的方式。



## Streaming Connectors

预定义的Source和Sink

Bundled Connectors

Apache Bahir中的连接器

Async I/O

Apache Flink 中文学习网站: [ververica.cn](http://ververica.cn)  
© Apache Flink Community China 严禁商业用途

## 1. 预定义的 source 和 sink

Flink 里预定义了一部分 source 和 sink。在这里分了几类。



### 预定义的Source和Sink



#### 基于文件的Source

readTextFile(path)  
readFile(fileInputFormat, path)

#### 基于文件的Sink

writeAsText  
writeAsCsv



#### 基于Socket

socketTextStream

#### 基于Socket的Sink

writeToSocket



#### 基于Collections、Iterators

fromCollection、fromElements

#### 标准输出、标准错误

Print、printToError

Apache Flink 中文学习网站: [ververica.cn](http://ververica.cn)  
© Apache Flink Community China 严禁商业用途

- 基于文件的 source 和 sink。

如果要从文本文件中读取数据，可以直接使用：

```
env.readTextFile(path)
```

就可以以文本的形式读取该文件中的内容。当然也可以使用：

```
env.readFile(fileInputFormat, path)
```

根据指定的 fileInputFormat 格式读取文件中的内容。

如果数据在 Flink 内进行了一系列的计算，想把结果写出到文件里，也可以直接使用内部预定义的一些 sink，比如将结果已文本或 csv 格式写出到文件中，可以使用 DataStream 的 writeAsText(path) 和 writeAsCsv(path)。

- 基于 Socket 的 Source 和 Sink

提供 Socket 的 host name 及 port，可以直接用 StreamExecutionEnvironment 预定的接口 socketTextStream 创建基于 Socket 的 source，从该 socket 中以文本的形式读取数据。当然如果想把结果写出到另外一个 Socket，也可以直接调用 DataStream writeToSocket。

- 基于内存 Collections、Iterators 的 Source

可以直接基于内存中的集合或者迭代器，调用 StreamExecutionEnvironment fromCollection、fromElements 构建相应的 source。结果数据也可以直接 print、printToError 的方式写出到标准输出或标准错误。

详细也可以参考 Flink 源码中提供的一些相对应的 Examples 来查看异常预定义 source 和 sink 的使用方法，例如 WordCount、SocketWindowWordCount。

## 2. Bundled Connectors

Flink 里已经提供了一些绑定的 Connector，例如 kafka source 和 sink，Es sink 等。读写 kafka、es、rabbitMQ 时可以直接使用相应 connector 的 api 即可。

第二部分会详细介绍生产环境中最常用的 kafka connector。

虽然该部分是 Flink 项目源代码里的一部分，但是真正意义上不算作 Flink 引擎相关逻辑，并且该部分没有打包在二进制的发布包里面。所以在提交 Job 时候需要注意，job 代码 jar 包中一定要将相应的 connector 相关类打包进去，否则在提交作业时就会失败，提示找不到相应的类，或初始化某些类异常。



## Bundled Connectors

---

- [Apache Kafka](#) (source/sink)
- [Apache Cassandra](#) (sink)
- [Amazon Kinesis Streams](#) (source/sink)
- [Elasticsearch](#) (sink)
- [Hadoop FileSystem](#) (sink)
- [RabbitMQ](#) (source/sink)
- [Apache NiFi](#) (source/sink)
- [Twitter Streaming API](#) (source)

➤ 注意：以上流connector是Flink 项目的一部分，但是不包括在二进制发布包中！

## 3. Apache Bahir 中的连接器

Apache Bahir 最初是从 Apache Spark 中独立出来项目提供，以提供不限于 Spark 相关的扩展 / 插件、连接器和其他可插入组件的实现。通过提供多样化的流连接器 (streaming connectors) 和 SQL 数据源扩展分析平台的覆盖面。如有需要写到 flume、redis 的需求的话，可以使用该项目提供的 connector。



## Apache Bahir中的连接器

- [Apache ActiveMQ](#)(source/sink)
- [Apache Flume](#)(sink)
- [Redis](#)(sink)
- [Akka](#)(sink)
- [Netty](#)(Source)

Apache Flink 中文学习网站: [ververica.cn](http://ververica.cn)  
© Apache Flink Community China 严禁商业用途

## 4. Async I/O

流计算中经常需要与外部存储系统交互，比如需要关联 MySQL 中的某个表。一般来说，如果用同步 I/O 的方式，会造成系统中出现大的等待时间，影响吞吐和延迟。为了解决这个问题，异步 I/O 可以并发处理多个请求，提高吞吐，减少延迟。

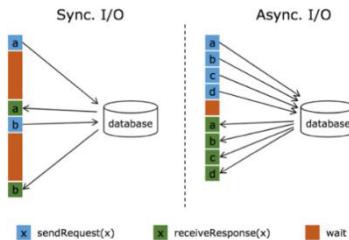
Async 的原理可参考官方文档：

<https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/stream/asyncio.html>



## Async I/O

- 使用connector并不是数据输入输出Flink的唯一方式。
- 在Map、FlatMap中使用Async I/O方式读取外部数据库等。

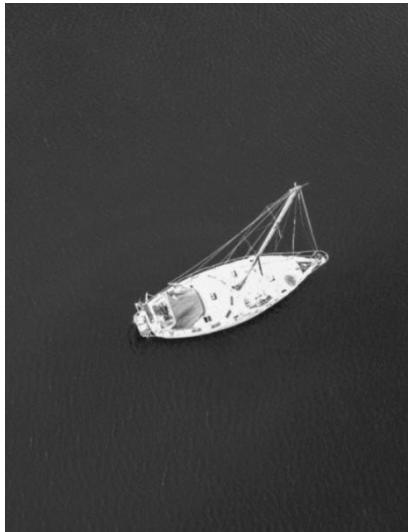


<https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/stream/asyncio.html>

Apache Flink 中文学习网站: [ververica.cn](http://ververica.cn)  
© Apache Flink Community China 严禁商业用途

## Flink Kafka Connector

本章重点介绍生产环境中最常用到的 Flink kafka connector。使用 Flink 的同学，一定会很熟悉 kafka，它是一个分布式的、分区的、多副本的、支持高吞吐的、发布订阅消息系统。生产环境环境中也经常会跟 kafka 进行一些数据的交换，比如利用 kafka consumer 读取数据，然后进行一系列的处理之后，再将结果写出到 kafka 中。这里会主要分两个部分进行介绍，一是 Flink kafka Consumer，一个是 Flink kafka Producer。



## Flink Kafka Connector

### Flink Kafka Consumer

- 反序列化数据
- 消费起始位置设置
- Topic和Partition动态发现
- Commit Offset方式
- Timestamp Extraction/Watermark 生成

### Flink Kafka Producer

- Producer分区
- 容错

Apache Flink 中文学习网站: [ververica.cn](http://ververica.cn)  
© Apache Flink Community China 严禁商业用途

首先看一个例子来串联下 Flink kafka connector。代码逻辑里主要是从 kafka 里读数据，然后做简单的处理，再写回到 kafka 中。

分别用红框框出如何构造一个 Source sink Function。Flink 提供了现成的构造 FlinkKafkaConsumer、Producer 的接口，可以直接使用。这里需要注意，因为 kafka 有多个版本，多个版本之间的接口协议会不同。Flink 针对不同版本的 kafka 有相应的版本的 Consumer 和 Producer。例如：针对 08、09、10、11 版本，Flink 对应的 consumer 分别是 FlinkKafkaConsumer 08、09、010、011，producer 也是。



## Flink Kafka Example

```

public class FlinkKafkaExample {
    public static void main(String[] args) throws Exception{
        final ParameterTool params = ParameterTool.fromArgs(args);

        // set up the execution environment
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.getConfig().setGlobalJobParameters(params);
        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
        env.enableCheckpointing(1000, CheckpointingMode.EXACTLY_ONCE);

        String sourceTopic = "topic1";
        Properties bootstrapServers = getConsumerProperties(bootStrapServers, "test-dtt-g1");
        FlinkKafkaConsumer010<String> consumer = new FlinkKafkaConsumer010<String>(sourceTopic, new SimpleStringSchema(), properties);
        consumer.setStartFromEarliest();
        consumer.setStartTimestamp(-1);
        consumer.setStartFromLineageTimestamp(156281792000L);
        consumer.setStartFromGroupOffsets();
        consumer.setStartFromDefault();

        String sinkTopic = "topic2";
        Properties prop = getProducerProperties(bootStrapServers);
        FlinkKafkaProducer010<String> producer = new FlinkKafkaProducer010<String>(sinkTopic, new SimpleStringSchema(), prop);

        env.addSource(consumer).addSink(sinkTopic)
            .map(new MapFunction<String, Tuple2<Long, String>>() {
                @Override
                public Tuple2<Long, String> map(String s) throws Exception {
                    ...
                }
            }).filter(k -> k!=null).singleOutputStreamOperator(Tuple2.ofLong, String.class)
            .assignTimestampsAndWatermarks(new BoundedOutOfOrderTimestampExtractor<Tuple2<Long, String>>(Time.seconds(5)) {
                @Override
                public long extractTimestamp(Tuple2<Long, String> element) { return element.f0; }
            }).singleOutputStreamOperator(Tuple2.ofLong, String.class)
            .map(k -> k.toString())
            .addSink(producer);

        env.execute("FlinkKafkaExample");
    }
}

```

## 1. Flink kafka Consumer

- 反序列化数据

因为 kafka 中数据都是以二进制 byte 形式存储的。读到 Flink 系统中之后，需要将二进制数据转化为具体的 java、scala 对象。具体需要实现一个 schema 类，定义如何序列化和反序列化。反序列化时需要实现 DeserializationSchema 接口，并重写 deserialize(byte[] message) 函数，如果是反序列化 kafka 中 kv 的数据时，需要实现 KeyedDeserializationSchema 接口，并重写 deserialize(byte[] messageKey, byte[] message, String topic, int partition, long offset) 函数。

另外 Flink 中也提供了一些常用的序列化反序列化的 schema 类。例如，SimpleStringSchema，按字符串方式进行序列化、反序列化。TypeInformationSerializationSchema，它可根据 Flink 的 TypeInformation 信息来推断出需要选择的 schema。JsonDeserializationSchema 使用 jackson 反序列化 json 格式消息，并返回 ObjectNode，可以使用 .get(“property”) 方法来访问相应字段。



## Flink Kafka Consumer–反序列化数据

- 将kafka中二进制数据转化为具体的java、scala 对象
- DeserializationSchema , T deserialize(byte[] message)
- KeyedDeserializationSchema , T deserialize(byte[] messageKey, byte[] message, String topic, int partition, long offset): 对于访问kafka key/value

### 常用

- SimpleStringSchema: 按字符串方式进行序列化、反序列化
- TypeInformationSerializationSchema: 基于Flink的TypeInformation来创建schema
- JsonDeserializationSchema: 使用jackson反序列化json格式消息，并返回 ObjectNode，可以使用.get("property")方法来访问字段。

Apache Flink 中文学习网站: [ververica.cn](http://ververica.cn)  
 © Apache Flink Community China 严禁商业用途

### • 消费起始位置设置

如何设置作业从 kafka 消费数据最开始的起始位置，这一部分 Flink 也提供了非常好的封装。在构造好的 FlinkKafkaConsumer 类后面调用如下相应函数，设置合适的起始位置。

- setStartFromGroupOffsets, 也是默认的策略，从 group offset 位置读取数据，group offset 指的是 kafka broker 端记录的某个 group 的最后一次的消费位置。但是 kafka broker 端没有该 group 信息，会根据 kafka 的参数 "auto.offset.reset" 的设置来决定从哪个位置开始消费。
- setStartFromEarliest, 从 kafka 最早的位置开始读取。
- setStartFromLatest, 从 kafka 最新的位置开始读取。
- setStartFromTimestamp(long), 从时间戳大于或等于指定时间戳的位置开始读取。Kafka 时戳，是指 kafka 为每条消息增加另一个时戳。该时戳可以表示消息在 producer 端生成时的时间、或进入到 kafka broker 时的时间。
- setStartFromSpecificOffsets, 从指定分区的 offset 位置开始读取，如指定的 offsets 中不存某个分区，该分区从 group offset 位置开始读取。此时需要用户给定一个具体的分区、offset 的集合。

一些具体的使用方法可以参考下图。需要注意的是，因为 Flink 框架有容错机制，如果作业故障，如果作业开启 checkpoint，会从上一次 checkpoint 状态开始恢复。或者在停止作业的时候主动做 savepoint，启动作业时从 savepoint 开始恢复。这两种情况下恢复作业时，作业消费起始位置是从之前保存的状态中恢复，与上面提到跟 kafka 这些单独的配置无关。



Apache Flink

## Flink Kafka Consumer–消费起始位置

```

public static void main(String[] args) throws Exception {
    final ParameterTool params = ParameterTool.fromArgs(args);

    // set up the execution environment
    final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
    env.getConfig().setGlobalJobParameters(params);
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
    env.enableCheckpointing(interval: 60*1000, CheckpointingMode.EXACTLY_ONCE);

    String sourceTopic = "topic1";
    String bootstrapServers = "localhost:9092";
    Properties properties = getConsumerProperties(bootstrapServers, groupid: "test-gid");
    FlinkKafkaConsumer010<String> consumer = new FlinkKafkaConsumer010<String>(sourceTopic, new SimpleStringSchema(), properties);

    consumer.setStartFromEarliest();
    consumer.setStartFromLatest();
    consumer.setStartFromTimestamp(1561281792000L);
    consumer.setStartFromGroupOffsets();

    Map<KafkaTopicPartition, Long> specificStartOffsets = new HashMap<>();
    specificStartOffsets.put(new KafkaTopicPartition(sourceTopic, partition: 0), 0L);
    specificStartOffsets.put(new KafkaTopicPartition(sourceTopic, partition: 1), 1L);
    consumer.setStartFromSpecificOffsets(specificStartOffsets);
}

```

➤ 注意：作业故障从checkpoint自动恢复，以及手动做savepoint时，消费的位置从保存状态中恢复，与该配置无关！！！

Apache Flink 中文学习网站: [ververica.cn](http://ververica.cn)  
© Apache Flink Community China 严禁商业用途

- topic 和 partition 动态发现

实际的生产环境中可能有这样一些需求，比如场景一，有一个 Flink 作业需要将五份数据聚合到一起，五份数据对应五个 kafka topic，随着业务增长，新增一类数据，同时新增了一个 kafka topic，如何在不重启作业的情况下作业自动感知新的 topic。场景二，作业从一个固定的 kafka topic 读数据，开始该 topic 有 10 个 partition，但随着业务的增长数据量变大，需要对 kafka partition 个数进行扩容，由 10 个扩容到 20。该情况下如何在不重启作业情况下动态感知新扩容的 partition？

针对上面的两种场景，首先需要在构建 FlinkKafkaConsumer 时的 properties 中设置 flink.partition-discovery.interval-millis 参数为非负值，表示开启动态发现的开关，以及设置的时间间隔。此时 FlinkKafkaConsumer 内部会启动一个单独的

线程定期去 kafka 获取最新的 meta 信息。针对场景一，还需在构建 FlinkKafka-Consumer 时，topic 的描述可以传一个正则表达式描述的 pattern。每次获取最新 kafka meta 时获取正则匹配的最新 topic 列表。针对场景二，设置前面的动态发现参数，在定期获取 kafka 最新 meta 信息时会匹配新的 partition。为了保证数据的正确性，新发现的 partition 从最早的位置开始读取。



## Flink Kafka Consumer–topic partition自动发现

原理：内部单独的线程获取kafka meta信息进行更新

flink.partition-discovery.interval-millis:发现时间间隔。默认false，设置非负值开启。



### 分区发现

- 消费的Source kafka topic进行了partition 扩容
- 新发现的分区，从earliest位置开始读取



### Topic发现

- 支持正则表达式描述topic名字

```
String sourceTopic = "topic1";
String bootStrapServers = "localhost:9092";
Properties properties = getConsumerProperties(bootStrapServers, groupId: "test-gid");
//FlinkKafkaConsumer<String> consumer = new FlinkKafkaConsumer<String>(sourceTopic, new SimpleStringSchema(), properties);
Pattern topicPattern = java.util.regex.Pattern.compile("topic[0-9]");
FlinkKafkaConsumer<String> consumer = new FlinkKafkaConsumer<String>(topicPattern, new SimpleStringSchema(), properties);
```

Apache Flink 中文学习网站: [ververica.cn](http://ververica.cn)  
 © Apache Flink Community China 严禁商业用途

### • commit offset 方式

Flink kafka consumer commit offset 方式需要区分是否开启了 checkpoint。

如果 checkpoint 关闭，commit offset 要依赖于 kafka 客户端的 auto commit。需设置 enable.auto.commit，auto.commit.interval.ms 参数到 consumer properties，就会按固定的时间间隔定期 auto commit offset 到 kafka。

如果开启 checkpoint，这个时候作业消费的 offset 是 Flink 在 state 中自己管理的。此时提交 offset 到 kafka，一般都是作为外部进度的监控，想实时知道作业消费的位置和 lag 情况。此时需要 setCommitOffsetsOnCheckpoints 为 true 来设置当 checkpoint 成功时提交 offset 到 kafka。此时 commit offset 的间隔就取决于 checkpoint 间隔。

于 checkpoint 的间隔，所以此时从 kafka 一侧看到的 lag 可能并非完全实时，如果 checkpoint 间隔比较长 lag 曲线可能会是一个锯齿状。



## Flink Kafka Consumer-commit offset方式



### Checkpoint关闭

- 依赖kafka客户端的auto commit 定期提交offset。
- 需设置enable.auto.commit, auto.commit.interval.ms 参数到consumer properties



### Checkpoint开启

- Offset自己在checkpoint state中管理和容错。提交kafka 仅作为外部监视消费进度。
- 通过setCommitOffsetsOnCheckpoints控制，Checkpoint成功之后，是否提交 offset到kafka

Apache Flink 中文学习网站: [ververica.cn](http://ververica.cn)  
© Apache Flink Community China 严禁商业用途

### • Timestamp Extraction/Watermark 生成

我们知道当 Flink 作业内使用 EventTime 属性时，需要指定从消息中提取时戳和生成水位的函数。FlinkKafkaConsumer 构造的 source 后直接调用 assignTimestampsAndWatermarks 函数设置水位生成器的好处是此时是每个 partition 一个 watermark assigner，如下图。source 生成的时戳为多个 partition 时戳对齐后的最小时戳。此时在一个 source 读取多个 partition，并且 partition 之间数据时戳有一定差距的情况下，因为在 source 端 watermark 在 partition 级别有对齐，不会导致数据读取较慢 partition 数据丢失。

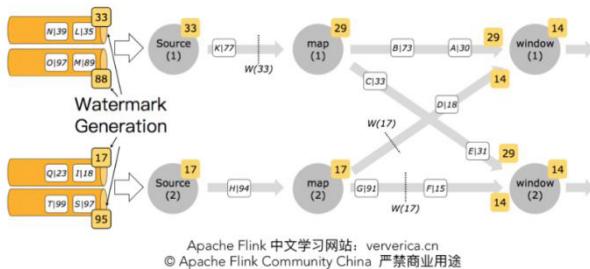


## Flink Kafka Consumer-时戳提取/水位生成



### per Kafka Partition watermark

- assignTimestampsAndWatermarks, 每个partition一个assigner, 水位为多个partition 对齐后值
- 不在 kafka source 后生成 watermark, 会出现扔掉部分数据情况。



## 2. Flink kafka Producer

### • Producer 分区

使用 FlinkKafkaProducer 往 kafka 中写数据时, 如果不单独设置 partition 策略, 会默认使用 FlinkFixedPartitioner, 该 partitioner 分区的方式是 task 所在的并发 id 对 topic 总 partition 数取余: `parallelInstanceid % partitions.length`。

- 此时如果 sink 为 4, partition 为 1, 则 4 个 task 往同一个 partition 中写数据。但当 sink task < partition 个数时会有部分 partition 没有数据写入, 例如 sink task 为 2, partition 总数为 4, 则后面两个 partition 将没有数据写入。
- 如果构建 FlinkKafkaProducer 时, partition 设置为 null, 此时会使用 kafka producer 默认分区方式, 非 key 写入的情况下, 使用 round-robin 的方式进行分区, 每个 task 都会轮循的写下游的所有 partition。该方式下游的 partition 数据会比较均衡, 但是缺点是 partition 个数过多的情况下需要维持过多的网络连接, 即每个 task 都会维持跟所有 partition 所在 broker 的连接。

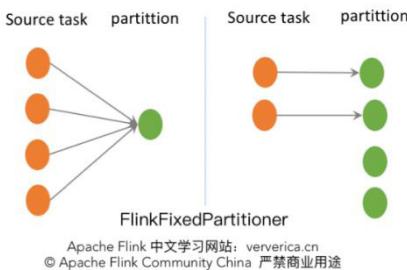


## Flink Kafka Producer-Producer分区



### Producer 分区

- FlinkFixedPartitioner (默认) : parallelInstanceid % partitions.length
- Partitioner设置为null: round-robin kafka partitioner, 维持过多链接
- custom partitioner: 自定义分区



### • 容错

Flink kafka 09、010 版本下，通过 setLogFailuresOnly 为 false, setFlushOnCheckpoint 为 true，能达到 at-least-once 语义。setLogFailuresOnly，默认为 false，是控制写 kafka 失败时，是否只打印失败的 log 不抛异常让作业停止。setFlushOnCheckpoint，默认为 true，是控制是否在 checkpoint 时 flush 数据到 kafka，保证数据已经写到 kafka。否则数据有可能还缓存在 kafka 客户端的 buffer 中，并没有真正写出到 kafka，此时作业挂掉数据即丢失，不能做到至少一次的语义。

Flink kafka 011 版本下，通过两阶段提交的 sink 结合 kafka 事务的功能，可以保证端到端精准一次。详细原理可以参考：

<https://www.ververica.com/blog/end-to-end-exactly-once-processing-apache-flink-apache-kafka>。



## Flink Kafka Producer-Producer容错



### Kafka 0.9 and 0.10

- setLogFailuresOnly: 默认false。写失败时，是否只打印失败log，不抛异常。
- setFlushOnCheckpoint: 默认true。checkpoint时保证数据写到kafka。
- at-least-once语义: setLogFailuresOnly : flase + setFlushOnCheckpoint : true



### Kafka 0.11

- FlinkKafkaProducer011，两阶段提交Sink结合kafka事务，可以保证端到端精准一次。

<https://www.ververica.com/blog/end-to-end-exactly-once-processing-apache-flink-apache-kafka>

Apache Flink 中文学习网站: ververica.cn  
© Apache Flink Community China 严禁商业用途

## 一些疑问与解答

Q: 在 Flink consumer 的并行度的设置：是对应 topic 的 partitions 个数吗？要是有多个主题数据源，并行度是设置成总体的 partitions 数吗？

A: 这个并不是绝对的，跟 topic 的数据量也有关，如果数据量不大，也可以设置小于 partitions 个数的并发数。但不要设置并发数大于 partitions 总数，因为这种情况下某些并发因为分配不到 partition 导致没有数据处理。

Q: 如果 partitioner 传 null 的时候是 round-robin 发到每一个 partition ? 如果有 key 的时候行为是 kafka 那种按照 key 分布到具体分区的行为吗？

A: 如果在构造 FlinkKafkaProducer 时，如果没有设置单独的 partitioner，则默认使用 FlinkFixedPartitioner，此时无论是带 key 的数据，还是不带 key。如果主动设置 partitioner 为 null 时，不带 key 的数据会 round-robin 的方式写出，带 key 的数据会根据 key，相同 key 数据分区的相同的 partition，如果 key 为 null，再轮询写。不带 key 的数据会轮询写各 partition。

Q: 如果 checkpoint 时间过长, offset 未提交到 kafka, 此时节点宕机了, 重启之后的重复消费如何保证呢?

A: 首先开启 checkpoint 时 offset 是 Flink 通过状态 state 管理和恢复的, 并不是从 kafka 的 offset 位置恢复。在 checkpoint 机制下, 作业从最近一次 checkpoint 恢复, 本身是会回放部分历史数据, 导致部分数据重复消费, Flink 引擎仅保证计算状态的精准一次, 要想做到端到端精准一次需要依赖一些幂等的存储系统或者事务操作。

# Apache Flink 进阶(十): Flink State 最佳实践

作者: 唐云(茶干)

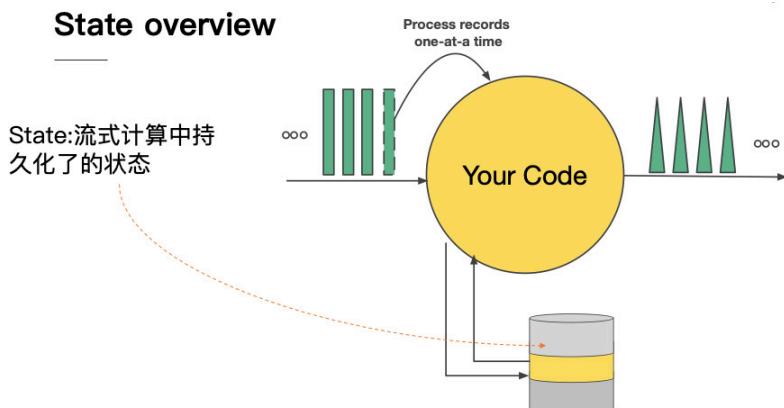
阿里巴巴高级研发工程师

本文主要分享与交流 Flink 状态使用过程中的一些经验与心得，当然标题取了“最佳实践”之名，希望文章内容能给读者带去一些干货。本文内容首先是回顾 state 相关概念，再次认识和区别不同的 state backend；之后将分别对 state 使用访问以及 checkpoint 容错相关内容进行详细讲解，分享一些经验和心得。

## State 概念回顾

我们先回顾一下到底什么是 state，流式计算的数据往往是转瞬即逝，当然，真实业务场景不可能说所有的数据都是进来之后就走掉，没有任何东西留下来，那么留下来的东西其实就是称之为 state，中文可以翻译成状态。

在下面这个图中，我们的所有的原始数据进入用户代码之后再输出到下游，如果中间涉及到 state 的读写，这些状态会存储在本地的 state backend (可以对标成嵌入式本地 kv 存储) 当中。

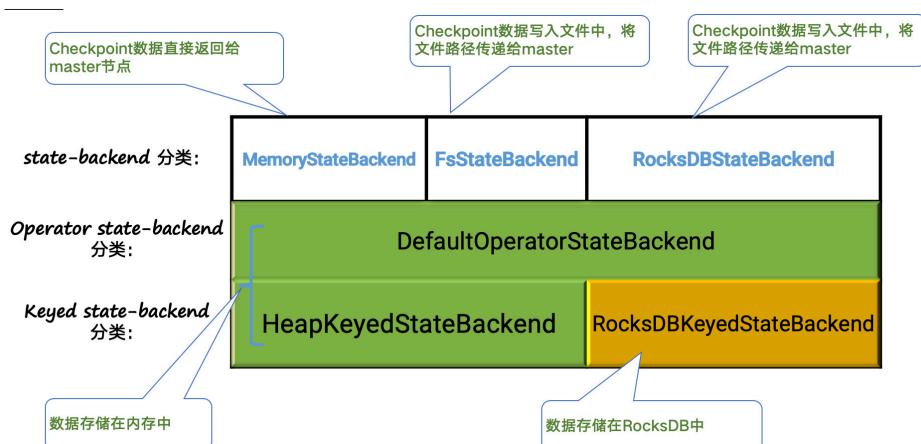


接下来我们会在四个维度来区分两种不同的 state: operator state 以及 keyed state:

- 是否存在当前处理的 key (current key): operator state 是没有当前 key 的概念, 而 keyed state 的数值总是与一个 current key 对应。
- 存储对象是否 on heap: 目前 operator state backend 仅有一种 on-heap 的实现; 而 keyed state backend 有 on-heap 和 off-heap (RocksDB) 的多种实现。
- 是否需要手动声明快照 (snapshot) 和恢复 (restore) 方法: operator state 需要手动实现 snapshot 和 restore 方法; 而 keyed state 则由 backend 自行实现, 对用户透明。
- 数据大小: 一般而言, 我们认为 operator state 的数据规模是比较小的; 认为 keyed state 规模是相对比较大的。需要注意的是, 这是一个经验判断, 不是一个绝对的判断区分标准。

## StateBackend 的分类

下面这张图对目前广泛使用的三类 state backend 做了区分, 其中绿色表示所创建的 operator/keyed state backend 是 on-heap 的, 黄色则表示是 off-heap 的。



一般而言，在生产中，我们会在 FsStateBackend 和 RocksDBStateBackend 间选择：

- FsStateBackend：性能更好；日常存储是在堆内存中，面临着 OOM 的风险，不支持增量 checkpoint。
- RocksDBStateBackend：无需担心 OOM 风险，是大部分时候的选择。

## RocksDB StateBackend 概览和相关配置讨论

RocksDB 是 Facebook 开源的 LSM 的键值存储数据库，被广泛应用于大数据系统的单机组件中。Flink 的 keyed state 本质上来说就是一个键值对，所以与 RocksDB 的数据模型是吻合的。下图分别是“window state”和“value state”在 RocksDB 中的存储格式，所有存储的 key, value 均被序列化成 bytes 进行存储。

Window state		Value state	
KeyGroup + Key + Namespace	value	KeyGroup + Key + Namespace	value
(1, K1, Window(10, 20))	v1	(2, K2, VoidNameSpace)	v2
(1, K3, Window(10, 20))	v3	(2, K4, VoidNameSpace)	v4
(1, K5, Window(10, 20))	v5	(2, K6, VoidNameSpace)	v6
...	...	...	...

在 RocksDB 中，每个 state 独享一个 Column Family，而每个 Column family 使用各自独享的 write buffer 和 block cache，上图中的 window state 和 value state 实际上分属不同的 column family。

下面介绍一些对 RocksDB 性能比较有影响的参数，并整理了一些相关的推荐配置，至于其他配置项，可以参阅[社区相关文档](#)。

state.backend.rocksdb.thread.num	后台 flush 和 compaction 的线程数，默认值‘1’，建议调大
state.backend.rocksdb.writebuffer.count	每个 column family 的 write buffer 数目，默认值‘2’。如果有需要可以适当调大
state.backend.rocksdb.writebuffer.size	每个 write buffer 的 size，默认值‘64MB’。对于写频繁的场景，建议调大
state.backend.rocksdb.block.cache-size	每个 column family 的 block cache 大小，默认值‘8MB’，如果存在重复读的场景，建议调大

## State best practice —— 一些使用 state 的心得

### Operator state 使用建议

- 慎重使用长 list

下图展示的是目前 task 端 operator state 在执行完 checkpoint 返回给 job master 端的 StateMetaInfo 的代码片段。

```
/*
 * Meta information about the operator state handle.
 */
class StateMetaInfo implements Serializable {

    private static final long serialVersionUID = 3593817615858941166L;

    private final long[] offsets;
    private final Mode distributionMode;

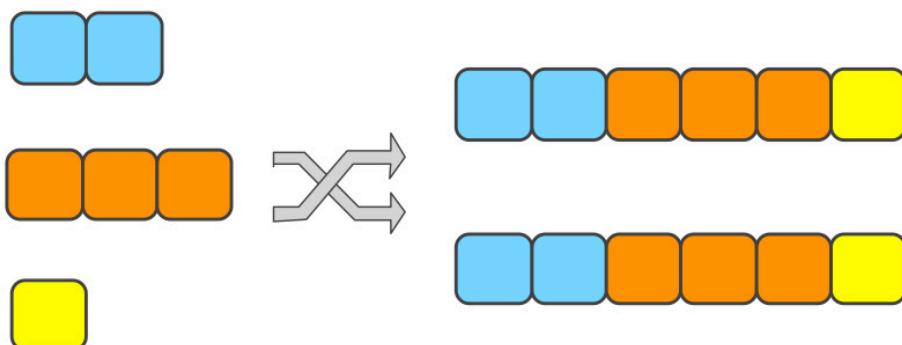
    public StateMetaInfo(long[] offsets, Mode distributionMode) {
        Preconditions.checkNotNull(offsets);
        Preconditions.checkNotNull(distributionMode);
    }
}
```

由于 operator state 没有 key group 的概念，所以为了实现改并发恢复的功能，需要对 operator state 中的每一个序列化后的元素存储一个位置偏移 offset，也就是构成了上图红框中的 offset 数组。那么如果你的 operator state 中的 list 长度达到一定规模时，这个 offset 数组就可能会有几十 MB 的规模，关键这个数组是会返回给

job master，当 operator 的并发数目很大时，很容易触发 job master 的内存超用问题。我们遇到过用户把 operator state 当做黑名单存储，结果这个黑名单规模很大，导致一旦开始执行 checkpoint，job master 就会因为收到 task 发来的“巨大”的 offset 数组，而内存不断增长直到超用无法正常响应。

- 正确使用 UnionListState

union list state 目前被广泛使用在 kafka connector 中，不过可能用户日常开发中较少遇到，他的语义是从检查点恢复之后每个并发 task 内拿到的是原先所有 operator 上的 state，如下图所示：



kafka connector 使用该功能，为的是从检查点恢复时，可以拿到之前的全局信息，如果用户需要使用该功能，需要切记恢复的 task 只取其中的一部分进行处理和用于下一次 snapshot，否则有可能随着作业不断的重启而导致 state 规模不断增长。

## Keyed state 使用建议

- 如何正确清空当前的 state

`state.clear()` 实际上只能清理当前 key 对应的 value 值，如果想要清空整个 state，需要借助于 `applyToAllKeys` 方法，具体代码片段如下：

```

    // clear state via applyToAllKeys().
    backend.applyToAllKeys(VoidNamespace.INSTANCE, VoidNamespaceSerializer.INSTANCE, listStateDescriptor,
    new KeyedStateFunction<Integer, ListState<String>>() {
        @Override
        public void process(Integer key, ListState<String> state) throws Exception {
            state.clear();
        }
    });
}

```

如果你的需求中只是对 state 有过期需求，借助于 state TTL 功能来清理会是一个性能更好的方案。

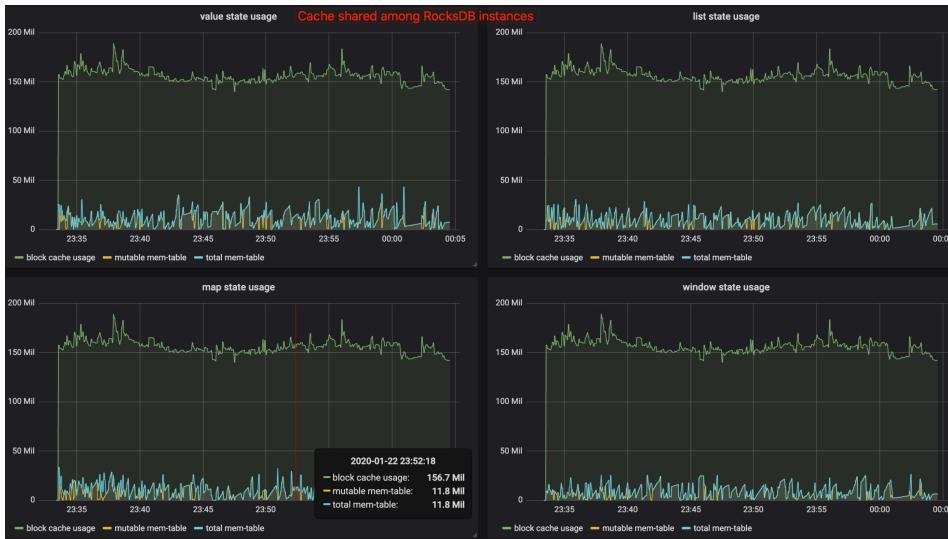
- **RocksDB 中考虑 value 值很大的极限场景**

受限于 JNI bridge API 的限制，单个 value 只支持  $2^{31}$  bytes 大小，如果存在很极限的情况，可以考虑使用 MapState 来替代 ListState 或者 ValueState，因为 RocksDB 的 map state 并不是将整个 map 作为 value 进行存储，而是将 map 中的一个条目作为键值对进行存储。

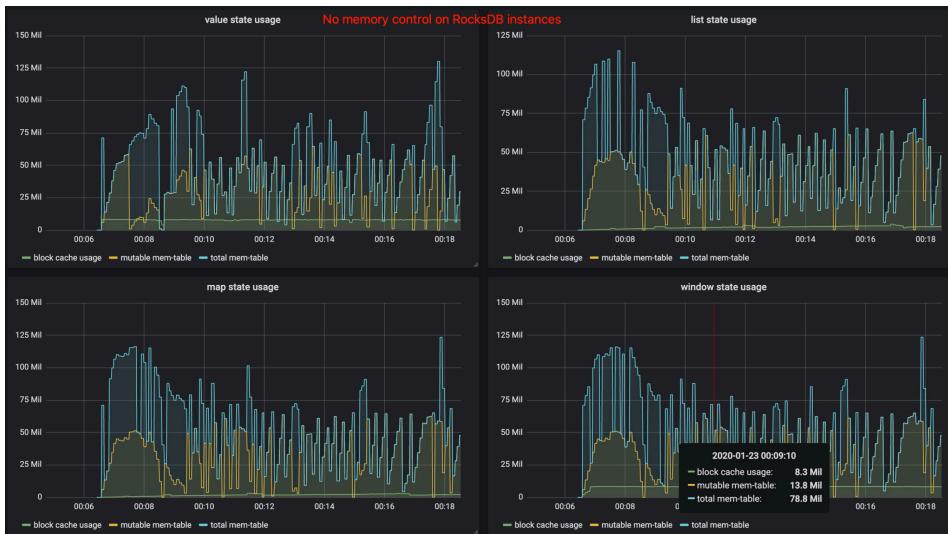
- **如何知道当前 RocksDB 的运行情况**

比较直观的方式是打开 RocksDB 的 [native metrics](#)，在默认使用 Flink managed memory 方式的情况下，`state.backend.rocksdb.metrics.block-cache-usage`, `state.backend.rocksdb.metrics.mem-table-flush-pending`, `state.backend.rocksdb.metrics.num-running-compactions` 以及 `state.backend.rocksdb.metrics.num-running-flushes` 是比较重要的相关 metrics。

下面这张图是 Flink-1.10 之后，打开相关 metrics 的示例图：



而下面这张是 Flink-1.10 之前或者关闭 state.backend.rocksdb.memory.managed 的效果：



- 容器内运行的 RocksDB 的内存超用问题

在 Flink-1.10 之前，由于一个 state 独占若干 write buffer 和一块 block cache，

所以我们会建议用户不要在一个 operator 内创建过多的 state，否则需要考虑到相应的额外内存使用量，否则容易造成在容器内运行时，相关进程被容器环境所杀。对于用户来说，需要考虑一个 slot 内有多少 RocksDB 实例在运行，一个 RocksDB 中有多少 state，整体的计算规则就很复杂，很难真得落地实施。

Flink-1.10 之后，由于引入了 RocksDB 的内存托管机制，在绝大部分情况下，RocksDB 的这一部分 native 内存是可控的，不过受限于 RocksDB 的相关 cache 实现限制（这里暂不展开，后续会有文章讨论），在某些场景下，无法做到完美控制，这时候建议打开上文提到的 native metrics，观察相关 block cache 内存使用是否存在超用情况，可以将相关内存添加到 `taskmanager.memory.task.off-heap.size` 中，使得 Flink 有更多的空间给 native 内存使用。

## 一些使用 checkpoint 的使用建议

### Checkpoint 间隔不要太短

虽然理论上 Flink 支持很短的 checkpoint 间隔，但是在实际生产中，过短的间隔对于底层分布式文件系统而言，会带来很大的压力。另一方面，由于检查点的语言，所以实际上 Flink 作业处理 record 与执行 checkpoint 存在互斥锁，过于频繁的 checkpoint，可能会影响整体的性能。当然，这个建议的出发点是底层分布式文件系统的压力考虑。

### 合理设置超时时间

默认的超时时间是 10min，如果 state 规模大，则需要合理配置。最坏情况是分布式地创建速度大于单点（job master 端）的删除速度，导致整体存储集群可用空间压力较大。建议当检查点频繁因为超时而失败时，增大超时时间。

# Apache Flink 进阶(十一): TensorFlow On Flink

作者: 陈戌超(仲卓)

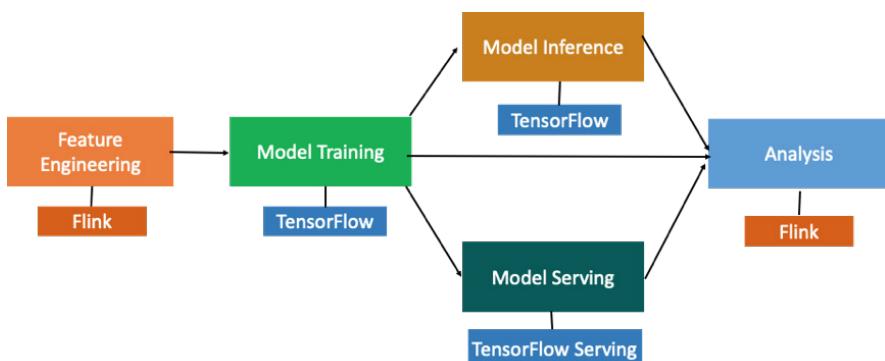
阿里巴巴技术专家

深度学习技术在当代社会发挥的作用越来越大。目前深度学习被广泛应用于个性化推荐、商品搜索、人脸识别、机器翻译、自动驾驶等多个领域，此外还在向社会各个领域迅速渗透。

## 背景

当前，深度学习的应用越来越多样化，随之涌现出诸多优秀的计算框架。其中 TensorFlow，PyTorch，MXNeT 作为广泛使用的框架更是备受瞩目。在将深度学习应用于实际业务的过程中，往往需要结合数据处理相关的计算框架如：模型训练之前需要对训练数据进行加工生成训练样本，模型预测过程中需要对处理数据的一些指标进行监控等。在这样的情况下，数据处理和模型训练分别需要使用不同的计算引擎，增加了用户使用的难度。

本文将分享如何使用一套引擎搞定机器学习全流程的解决方案。先介绍一下典型的机器学习工作流程。如图所示，整个流程包含特征工程、模型训练、离线或者是在线预测等环节。

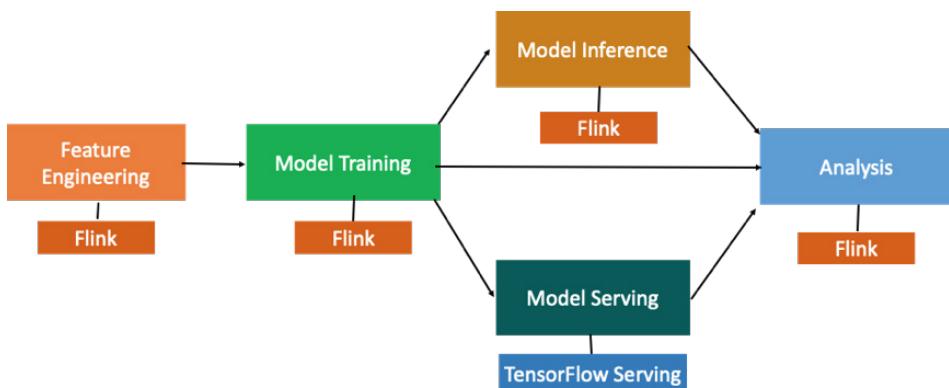


在此过程中，无论是特征工程、模型训练还是模型预测，中间都会产生日志。需要先用数据处理引擎比如 Flink 对这些日志进行分析，然后进入特征工程。再使用深度学习的计算引擎 TensorFlow 进行模型训练和模型预测。当模型训练好了以后再用 tensor serving 做在线的打分。

上述流程虽然可以跑通，但也存在一定的问题，比如：

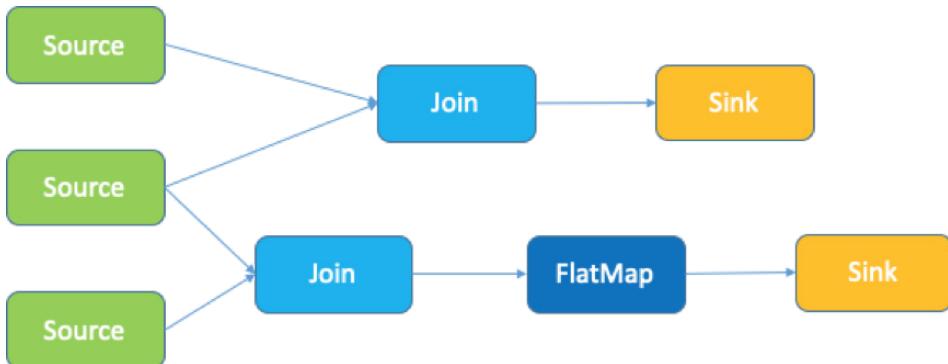
1. 同一个机器学习项目在做特征工程、模型训练、模型预测时需要用到 Flink 和 TensorFlow 两个计算引擎，部署相对而言更复杂。
2. TensorFlow 在分布式的支持上还不够友好，运行过程中需要指定机器的 IP 地址和端口号；而实际生产过程经常是运行在一个调度系统上比如 Yarn，需要动态分配 IP 地址和端口号。
3. TensorFlow 的分布式运行缺乏自动的 failover 机制。

针对以上问题，我们通过结合 Flink 和 TensorFlow，将 TensorFlow 的程序跑在 Flink 集群上的这种方式来解决，整体流程如下：



特征工程用 Flink 去执行，模型训练和模型的准实时预测目标使 TensorFlow 计算引擎可以跑在 Flink 集群上。这样就可以用 Flink 一套计算引擎去支持模型训练和模型的预测，部署上更简单的同时也节约了资源。

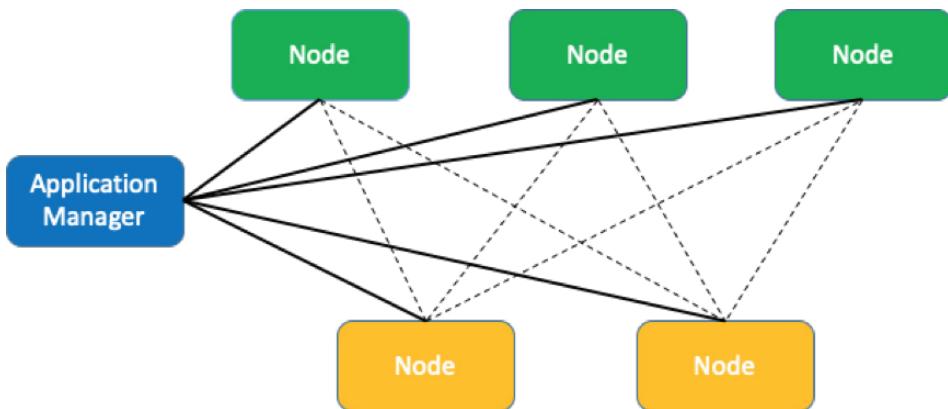
## Flink 计算简介



Flink 是一款开源大数据分布式计算引擎，在 Flink 里所有的计算都抽象成 operator，如上图所示，数据读取的节点叫 source operator，输出数据的节点叫 sink operator。source 和 sink 中间有多种多样的 Flink operator 去处理，上图的计算拓扑包含了三个 source 和两个 sink。

## 机器学习分布式拓扑

机器学习分布式运行拓扑如下图所示：



在一个机器学习的集群当中，经常会对一组节点 (node) 进行分组，如上图所

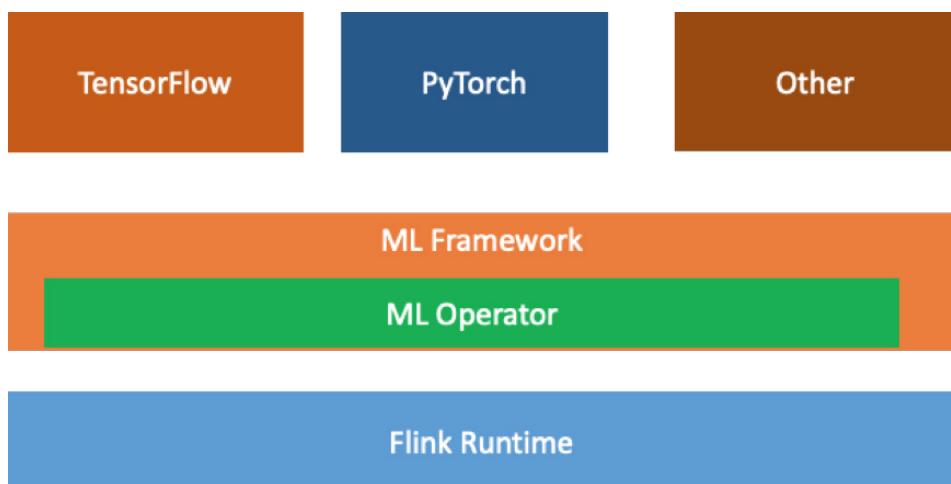
示，一组节点可以是 worker (运行算法)，也可以是 ps (更新参数)。

如何将 Flink 的 operator 结构与 Machine Learning 的 node、Application Manager 角色结合起来？下面将详细讲解 flink-ai-extended 的抽象。

## Flink-ai-extended 抽象

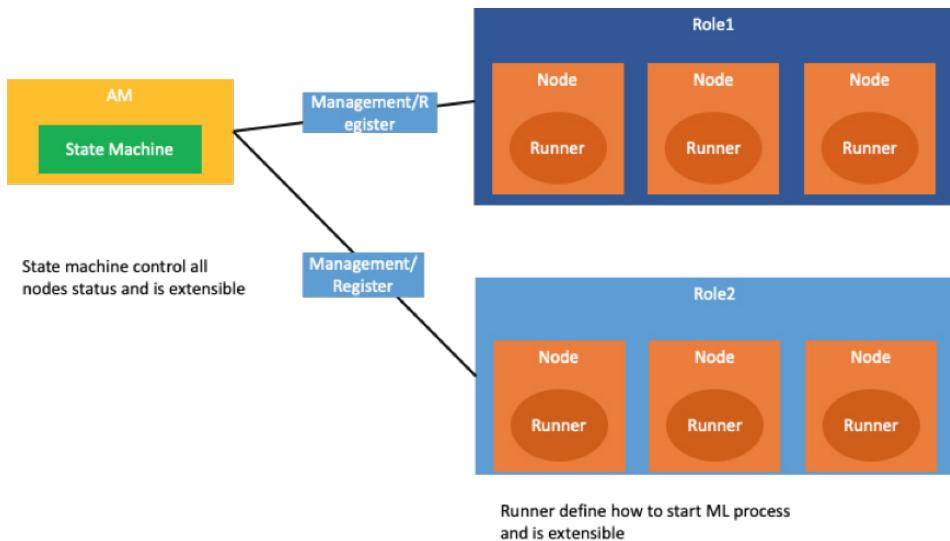
首先，对机器学习的 cluster 进行一层抽象，命名为 ML framework，同时机器学习也包含了 ML operator。通过这两个模块，可以把 Flink 和 Machine Learning Cluster 结合起来，并且可以支持不同的计算引擎，包括 TensorFlow。

如下图所示：



在 Flink 运行环境上，抽象了 ML Framework 和 ML Operator 模块，负责连接 Flink 和其他计算引擎。

## ML Framework



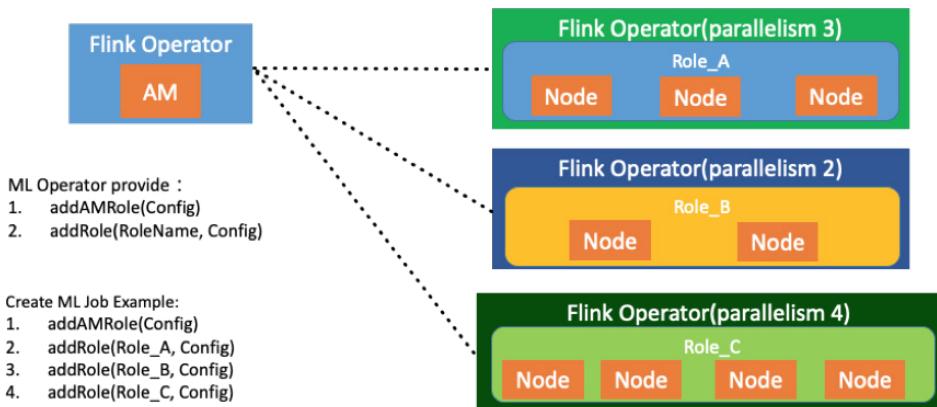
ML Framework 分为 2 个角色。

1. Application Manager (以下简称 am) 角色, 负责管理所有 node 的节点的生命周期。
2. node 角色, 负责执行机器学习的算法程序。

在上述过程中, 还可以对 Application Manager 和 node 进行进一步的抽象, Application Manager 里面我们单独把 state machine 的状态机做成可扩展的, 这样就可以支持不同类型的工作。

深度学习引擎, 可以自己定义其状态机。从 node 的节点抽象 runner 接口, 这样用户就可以根据不同的深度学习引擎去自定义运行算法程序。

## ML Operator

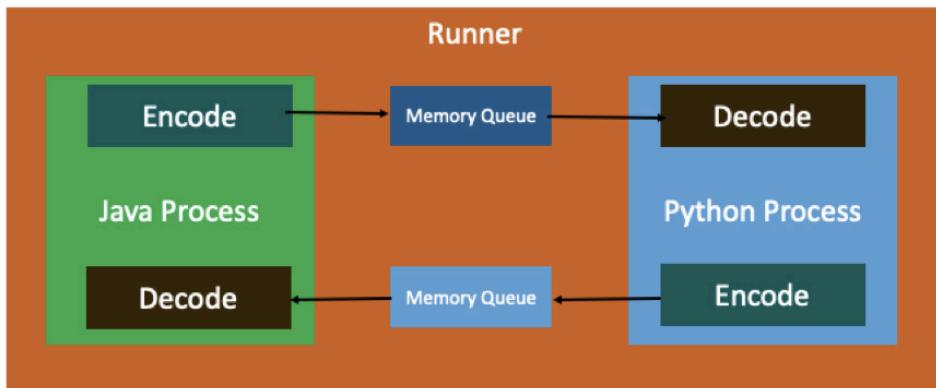


ML Operator 模块提供了两个接口：

1. addAMRole, 这个接口的作用是在 Flink 的作业里添加一个 Application Manager 的角色。Application Manager 角色如上图所示就是机器学习集群的管理节点。
2. addRole, 增加的是机器学习的一组节点。

利用 ML Operator 提供的接口，可以实现 Flink Operator 中包含一个 Application Manager 及 3 组 node 的角色，这三组 node 分别叫 role a、role b、role c，三个不同角色组成机器学习的一个 cluster。如上图代码所示。Flink 的 operator 与机器学习作业的 node 一一对应。

机器学习的 node 节点运行在 Flink 的 operator 里，需要进行数据交换，原理如下图所示：



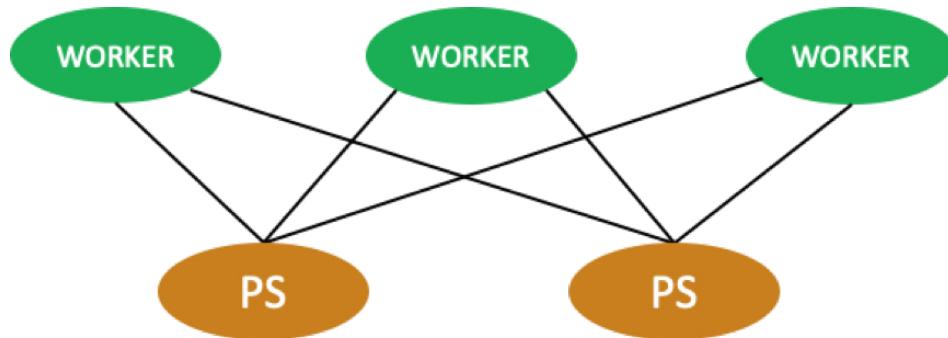
1. Encode transfer user define object to byte[]
2. Decode transfer byte[] to user define object

Encode and Decode is extensible

Flink operator 是 java 进程，机器学习的 node 节点一般是 python 进程，java 和 python 进程通过共享内存交换数据。

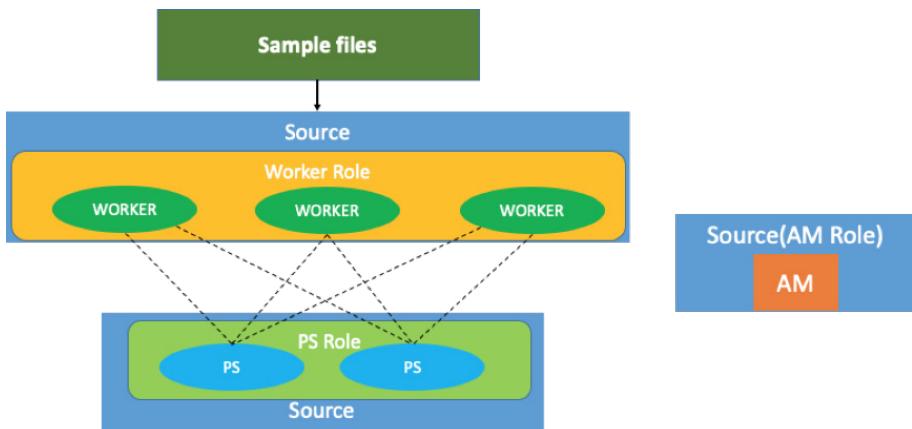
## TensorFlow On Flink

- TensorFlow 分布式运行



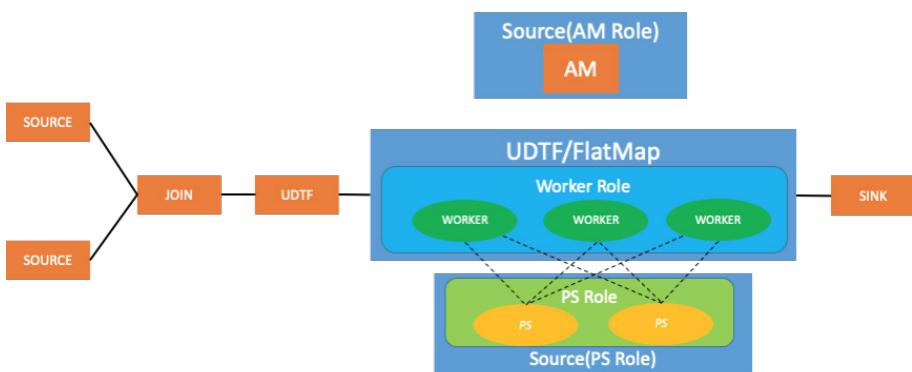
TensorFlow 分布式训练一般分为 worker 和 ps 角色。worker 负责机器学习计算，ps 负责参数更新。下面将讲解 TensorFlow 如何运行在 Flink 集群中。

- TensorFlow Batch 训练运行模式



Batch 模式下，样本数据可以是放在 HDFS 上的，对于 Flink 作业而言，它会起一个 source 的 operator，然后 TensorFlow 的 work 角色就会启动。如上图所示，如果 worker 的角色有三个节点，那么 source 的并行度就会设为 3。同理下面 ps 角色有 2 个，所以 ps source 节点就会设为 2。而 Application Manager 和别的角色并没有数据交换，所以 Application Manager 是单独的一个节点，因此它的 source 节点并行度始终为 1。这样 Flink 作业上启动了三个 worker 和两个 ps 节点，worker 和 ps 之间的通讯是通过原始的 TensorFlow 的 GRPC 通讯来实现的，并不是走 Flink 的通信机制。

- TensorFlow stream 训练运行模式

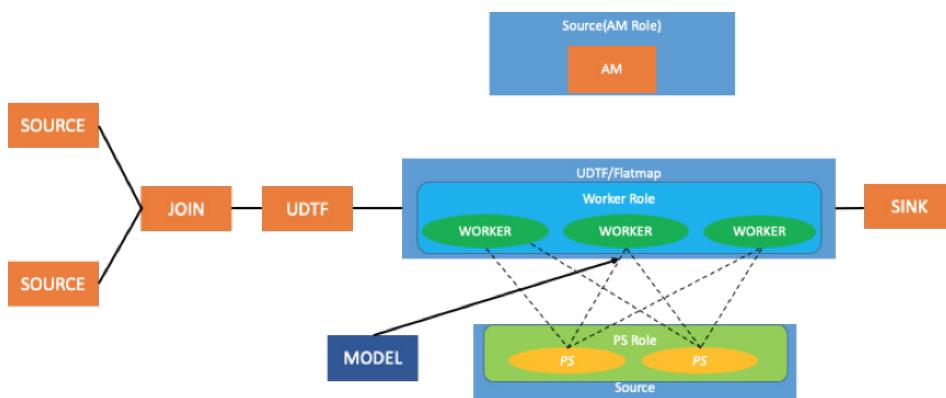


如上图所示，前面有两个 source operator，然后接 join operator，把两份数据合并为一份数据，再加自定义处理的节点，生成样本数据。在 stream 模式下，worker 的角色是通过 UDTF 或者 flatmap 来实现的。

同时，TensorFlow worker node 有 3 个，所以 flatmap 和 UDTF 相对应的 operator 的并行度也为 3，由于 ps 角色并不去读取数据，所以是通过 flink source operator 来实现的。

下面我们再讲一下，如果已经训练好的模型，如何去支持实时的预测。

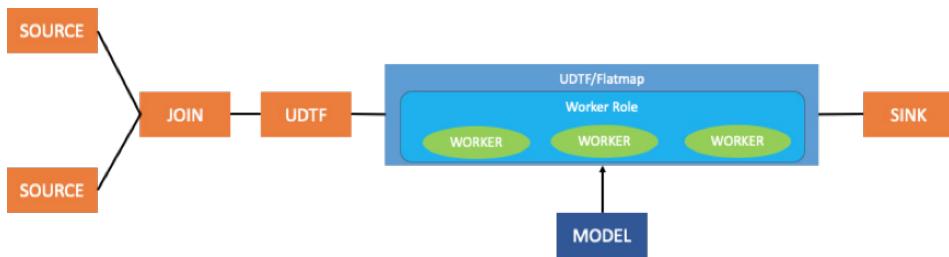
- 使用 Python 进行预测



使用 Python 进行预测流程如图所示，如果 TensorFlow 的模型是分布式训练出来的模型，并且这个模型非常大，比如说单机放不下的情况，一般出现在推荐和搜索的场景下。那么实时预测和实时训练原理相同，唯一不同的地方是多了一个加载模型的过程。

在预测的情况下，通过读取模型，将所有的参数加载到 ps 里面去，然后上游的数据还是经过和训练时候一样的处理形式，数据流入到 worker 这样一个角色中去进行处理，将预测的分数再写回到 flink operator，并且发送到下游 operator。

- 使用 Java 进行预测



如图所示，模型单机进行预测时就没必要再去起 ps 节点，单个 worker 就可以装下整个模型进行预测，尤其是使用 TensorFlow 导出 save model。同时，因为 saved model 格式包含了整个深度学习预测的全部计算逻辑和输入输出，所以不需要运行 Python 的代码就可以进行预测。

此外，还有一种方式进行预测。前面 source、join、UDTF 都是对数据进行加工处理变成预测模型可以识别的数据格式，在这种情况下，可以直接在 Java 进程里面通过 TensorFlow Java API，将训练好的模型 load 到内存里，这时会发现并不需要 ps 角色，worker 角色也都是 Java 进程，并不是 Python 的进程，所以我们可以直接在 Java 进程内进行预测，并且可以将预测结果继续发给 Flink 的下游。

## 总结

在本文中，我们讲解了 flink-ai-extended 原理，以及 Flink 结合 TensorFlow 如何进行模型训练和预测。希望通过本文大分享，大家能够使用 flink-ai-extended，通过 Flink 作业去支持模型训练和模型的预测。

# Apache Flink 进阶(十二): 深度探索 Flink SQL

作者: 贺晓令(晓令), 阿里巴巴技术专家

整理: 郑仲尼

本文根据 Apache Flink 进阶篇系列直播整理而成, 由阿里巴巴技术专家贺小令分享, 文章将从用户的角度来讲解 Flink 1.9 版本中 SQL 相关原理及部分功能变更, 希望加深大家对 Flink 1.9 新功能的理解, 在使用上能够有所帮助。主要内容:

1. 新 TableEnvironment 的设计与使用场景
2. 新 Catalog 的设计以及 DDL 实践
3. Blink Planner 的几点重要改进及优化

## 新 TableEnvironment

[FLIP-32](#) 中提出, 将 Blink 完全开源, 合并到 Flink 主分支中。合并后在 Flink 1.9 中会存在两个 Planner: Flink Planner 和 Blink Planner。

在之前的版本中, Flink Table 在整个 Flink 中是一个二等公民。而 Flink SQL 具备的易用性, 使用门槛低等特点深受用户好评, 也越来越被重视, Flink Table 模块也因此被提升为一等公民。而 Blink 在设计之初就考虑到流和批的统一, 批只是流的一种特殊形式, 所以在将 Blink 合并到 Flink 主分支的过程中, 社区也同时考虑了 Blink 的特殊设计。

## 新 TableEnvironment 整体设计

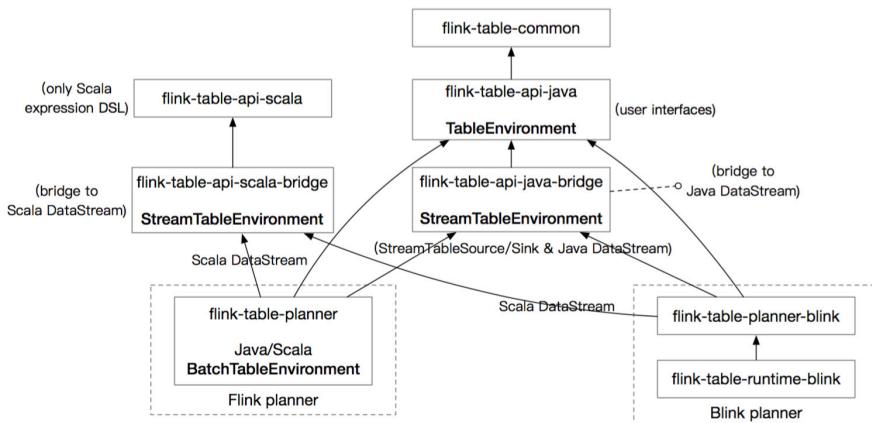


图 1 新 Table Environment 整体设计

从图 1 中，可以看出，TableEnvironment 组成部分如下：

- **flink-table-common**: 这个包中主要是包含 Flink Planner 和 Blink Planner 一些共用的代码。
- **flink-table-api-java**: 这部分是用户编程使用的 API，包含了大部分的 API。
- **flink-table-api-scala**: 这里只是非常薄的一层，仅和 Table API 的 Expression 和 DSL 相关。
- **两个 Planner**: `flink-table-planner` 和 `flink-table-planner-blink`。
- **两个 Bridge**: `flink-table-api-scala-bridge` 和 `flink-table-api-java-bridge`，从图中可以看出，Flink Planner 和 Blink Planner 都会依赖于具体的 JAVA API，也会依赖于具体的 Bridge，通过 Bridge 可以将 API 操作相应的转化为 Scala 的 DataStream、DataSet，或者转化为 JAVA 的 DataStream 或者 DataSet。

## 新旧 TableEnvironment 对比

在 Flink 1.9 之前，原来的 Flink Table 模块，有 7 个 Environment，使用和

维护上相对困难。7个Environment包括: StreamTableEnvironment, BatchTableEnvironment两类, JAVA和Scala分别2个,一共4个,加上3个父类,一共就是7个。

在新的框架之下,社区希望流和批统一,因此对原来的设计进行精简。首先,提供统一的TableEnvironment,放在flink-table-api-java这个包中。然后,在Bridge中,提供了两个用于衔接Scala DataStream和Java DataStream的StreamTableEnvironment。最后,因为Flink Planner中还残存在着toDataSet()类似的操作,所以,暂时保留BatchTableEnvironment。这样,目前一共是5个TableEnvironment。

因为未来Flink Planner将会被移除,BatchTableEnvironment就会被废弃,这样,未来就剩下3个Environment了,整个TableEnvironment的设计将更加简洁明了。

## 新TableEnvironment的应用

本节中,将介绍新的应用场景以及相关限制。

下图详细列出了新TableEnvironment的适用场景:

	Flink Stream	Flink Batch	Blink Stream	Blink Batch	From/To DataStream	From/To DataSet	UDAF/UDTF
TableEnvironment	✓	✗	✓	✓	✗	✗	✗ (Java/Scala 类型推导 还没统一)
Java/Scala StreamTableEnvironment	✓	✗	✓	✗	✓	✗	✓
Java/Scala BatchTableEnvironment	✗	✓	✗	✗	✗	✓	✓

图2 新Table Environment 适应场景

第一行，简单起见，在后续将新的 TableEnvironment 称为 UnifyTableEnvironment。在 Blink 中，Batch 被认为是 Stream 的一个特例，因此 Blink 的 Batch 可以使用 UnifyTableEnvironment。

UnifyTableEnvironment 在 1.9 中有一些限制，比如它不能够注册 UDAF 和 UDTF，当前新的 Type System 的类型推导功能还没有完成，Java、Scala 的类型推导还没统一，所以这部分的功能暂时不支持。可以肯定的是，这部分功能会在 1.10 中实现。此外，UnifyTableEnvironment 无法转化为 DataStream 和 DataSet。

第二行，Stream TableEnvironment 支持转化成 DataStream，也可以注册 UDAF 和 UDTF。如果是 JAVA 写的，就注册到 JAVA 的 TableEnvironment，如果是用 Scala 写的，就注册到 Scala 的 TableEnvironment。

注意，Blink Batch 作业是不支持 Stream TableEnvironment 的，因为目前没有 toAppendStream()，所以 toDataStream() 这样的语义暂时不支持。从图中也可以看出，目前操作只能使用 TableEnvironment。

最后一行，BatchTableEvironment 能够使用 toDataSet() 转化为 DataSet。

从上面的图 2 中，可以很清晰的看出各个 TableEnvironment 能够做什么事情，以及他们有哪些限制。

接下来，将使用示例对各种情况进行说明。

### 示例 1: Blink Batch

```
EnvironmentSettings settings = EnvironmentSettings.newInstance()
    .useBlinkPlanner().inBatchMode()
    .build();
TableEnvironment tEnv = TableEnvironment.create(settings);
tEnv...
tEnv.execute("job name");
```

从图 2 中可以看出, Blink Batch 只能使用 TableEnvironment(即 UnifyTableEnvironment), 代码中, 首先需要创建一个 EnvironmentSetting, 同时指定使用 Blink Planner, 并且指定用 Batch 模式。之所以需要指定 Blink Planner, 是因为目前 Flink 1.9 中, 将 Flink Planner 和 Blink Planner 的 jar 同时放在了 Flink 的 lib 目录下。如果不指定使用的 Planner, 整个框架并不知道需要使用哪个 Planner, 所以必须显示的指定。当然, 如果 lib 下面只有一个 Planner 的 jar, 这时不需要显示指定使用哪个 Planner。

另外, 还需要注意的是在 UnifyEnvironment 中, 用户是无法获取到 ExecutionEnvironment 的, 即用户无法在写完作业流程后, 使用 executionEnvironment.execute() 方法启动任务。需要显式的使用 tableEnvironment.execute() 方法启动任务, 这和之前的作业启动很不相同。

### 示例 2: Blink Stream

```
EnvironmentSettings settings = EnvironmentSettings.newInstance()
    .useBlinkPlanner()
    .inStreamingMode()
    .build();
StreamExecutionEnvironment execEnv = ...
StreamTableEnvironment tEnv = StreamTableEnvironment.create(execEnv,
    settings);
tEnv...
```

Blink Stream 既可以使用 UnifyTableEnvironment, 也可以使用 StreamTableEnvironment, 与 Batch 模式基本类似, 只是需要将 inBatchMode 换成 inStreamingMode。

### 示例 3: Flink Batch

```
ExecutionEnvironment execEnv = ...
BatchTableEnvironment tEnv = BatchTableEnvironment.create(execEnv);
tEnv...
```

与之前没有变化, 不做过多介绍。

### 示例 4: Flink Stream

```
EnvironmentSettings settings = EnvironmentSettings.newInstance() .
useOldPlanner().inStreamMode() .
build();
TableEnvironment tEnv = TableEnvironment.create(settings);
tEnv...
tEnv.execute("job name");
```

Flink Stream 也是同时支持 UnifyEnvironment 和 StreamTableEnvironment，只是在指定 Planner 时，需要指定为 useOldPlanner，也即 Flink Planner。因为未来 Flink Planner 会被移除，因此，特意起了一个 OlderPlanner 的名字，而且只能够使用 inStreamingMode，无法使用 inBatchMode。

## 新 Catalog 和 DDL

构建一个新的 Catalog API 主要是 [FLIP-30](#) 提出的，之前的 ExternalCatalog 将被废弃，Blink Planner 中已经不支持 ExternalCatalog 了，Flink Planner 还支持 ExternalCatalog。

## 新 Catalog 设计

下图是新 Catalog 的整体设计：

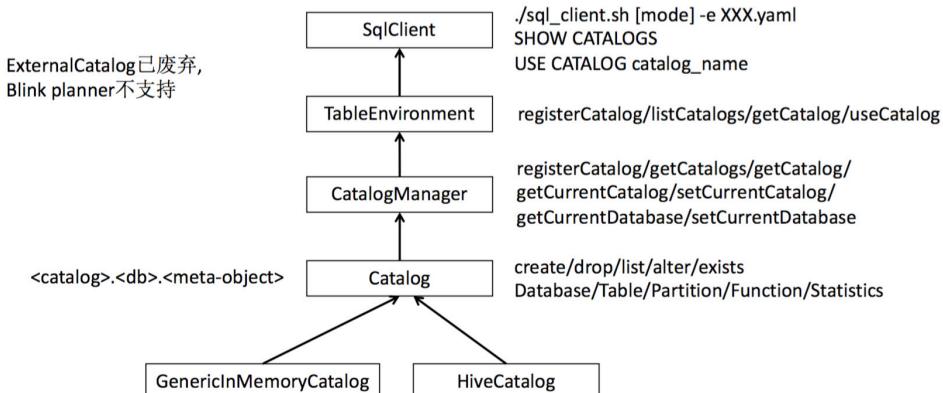


图 3 新 Catalog 设计

可以看到，新的 Catalog 有三层结构(..)，最顶层是 Catalog 的名字，中间一层是 Database，最底层是各种 MetaObject，如 Table，Partition，Function 等。当前，内置了两个 Catalog 实现：MemoryCatalog 和 HiveCatalog。当然，用户也可以实现自己的 Catalog。

Catalog 能够做什么事情呢？首先，它可以支持 Create，Drop，List，Alter，Exists 等语句，另外它也支持对 Database，Table，Partition，Function，Statistics 等的操作。基本上，常用的 SQL 语法都已经支持。

CatalogManager 正如它名字一样，主要是用来管理 Catalog，且可以同时管理多个 Catalog。也就是说，可以通过在一个相同 SQL 中，跨 Catalog 做查询或者关联操作。

例如，支持对 A Hive Catalog 和 B Hive Catalog 做相互关联，这给 Flink 的查询带来了很大的灵活性。

CatalogManager 支持的操作包括：

- 注册 Catalog (registerCatalog)
- 获取所有的 Catalog (getCatalogs)
- 获取特定的 Catalog (getCatalog)
- 获取当前的 Catalog (getCurrentCatalog)
- 设置当前的 Catalog (setCurrentCatalog)
- 获取当前的 Database(getCurrentDatabase)
- 设置当前的 Database(setCurrentDatabase)

Catalog 虽然设计了三层结构，但在使用的时候，并不需要完全指定三层结构的值，可以只写 Table Name，这时候，系统会使用 getCurrentCatalog，getCurrentDatabase 获取到默认值，自动补齐三层结构，这种设计简化了对 Catalog 的使用。如果需要切换默认的 Catalog，只需要调用 setCurrentCatalog 就可以了。

在 TableEnvironment 层，提供了操作 Catalog 的方法，例如：

- 注册 Catalog (registerCatalog)
- 列出所有的 Catalog (listCatalogs)
- 获取指定 Catalog (getCatalog)
- 使用某个 Catalog (useCatalog)

在 SQL Client 层，也做了一定的支持，但是功能有一定的限制。用户不能够使用 Create 语句直接创建 Catalog，只能通过在 yarn 文件中，通过定义 Description 的方式去描述 Catalog，然后在启动 SQL Client 的时候，通过传入 -e +file\_path 的方式，定义 Catalog。目前 SQL Client 支持列出已定义的 Catalog，使用一个已经存在的 Catalog 等操作。

## DDL 设计与使用

有了 Catalog，就可以使用 DDL 来操作 Catalog 的内容，可以使用 TableEnvironment 的 sqlUpdate() 方法执行 DDL 语句，也可以在 SQL Client 执行 DDL 语句。

sqlUpdate() 方法中，支持 Create Table, Create View, Drop Table, Drop View 四个命令。当然，insert into 这样的语句也是支持的。

下面分别对 4 个命令进行说明：

**Create Table:** 可以显示的指定 Catalog Name 或者 DB Name，如果缺省，那就按照用户设定的 Current Catalog 去补齐，然后可以指定字段名称，字段的说明，也可以支持 Partition By 语法。最后是一个 With 参数，用户可以在此处指定使用的 Connector，例如，Kafka, CSV, HBase 等。With 参数需要配置一堆的属性值，可以从各个 Connector 的 Factory 定义中找到。Factory 中会指出有哪些必选属性，哪些可选属性值。

需要注意的是，目前 DDL 中，还不支持计算列和 Watermark 的定义，后续的版本中将会继续完善这部分。

```
Create Table [[catalog_name.]db_name.]table_name(
    a int comment 'column comment',
    b bigint,
    c varchar
)comment 'table comment'
[partitioned by(b)]
With(
    update-mode='append',
    connector.type='kafka',
    ...
)
```

**Create View:** 需要指定 View 的名字，然后紧跟着的是 SQL。View 将会存储在 Catalog 中。

```
CREATE VIEW view_name AS SELECT xxx
```

**Drop Table&Drop View:** 和标准 SQL 语法差不多，支持使用 IF EXISTS 语法，如果未加 IF EXISTS，Drop 一个不存在的表，会抛出异常。

```
DROP TABLE [IF EXISTS] [[catalog_name.]db_name.]table_name
```

**SQL Client 中执行 DDL:** 大部分都只支持查看操作，仅可以使用 Create View 和 Drop View。Catalog, Database, Table , Function 这些只能做查看。用户可以在 SQL Client 中 Use 一个已经存在的 Catalog，修改一些属性，或者做 Description, Explain 这样的一些操作。

```
CREATE VIEW
DROP VIEW
SHOW CATALOGS/DATABASES/TABLES/FUNCTIONS l USE CATALOG xxx
SET xxx=yyy
DESCRIBE table_name
EXPLAIN SELECT xxx
```

DDL 部分，在 Flink 1.9 中其实基本已经成型，只是还有一些特性，在未来需要

逐渐的完善。

## Blink Planner

本节将主要从 SQL/Table API 如何转化为真正的 Job Graph 的流程开始，让大家对 Blink Planner 有一个比较清晰的认识，希望对大家阅读 Blink 代码，或者使用 Blink 方面有所帮助。然后介绍 Blink Planner 的改进及优化。

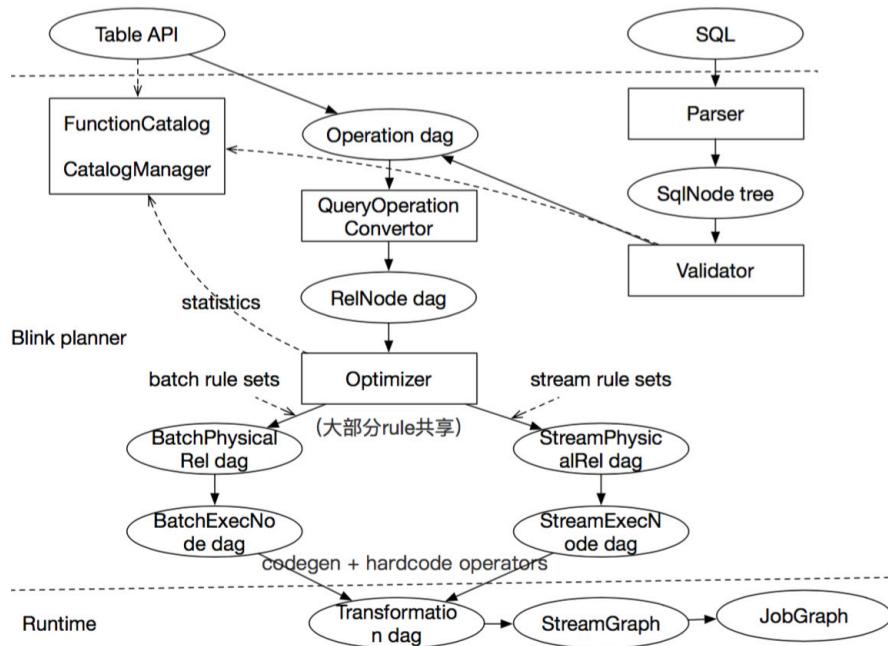


图 4 主要流程

从上图可以很清楚的看到，解析的过程涉及到了三层：Table API/SQL，Blink Planner，Runtime，下面将对主要的步骤进行讲解。

**Table API&SQL 解析验证：**在 Flink 1.9 中，Table API 进行了大量的重构，引入了一套新的 Operation，这套 Operation 主要是用来描述任务的 Logic Tree。

当 SQL 传输进来后，首先会去做 SQL 解析，SQL 解析完成之后，会得到

SqlNode Tree( 抽象语法树 ), 然后会紧接着去做 Validator ( 验证 ), 验证时会去访问 FunctionManger 和 CatalogManger, FunctionManger 主要是查询用户定义的 UDF, 以及检查 UDF 是否合法, CatalogManger 主要是检查这个 Table 或者 Database 是否存在, 如果验证都通过, 就会生成一个 Operation DAG ( 有向无环图 )。

从这一步可以看出, Table API 和 SQL 在 Flink 中最终都会转化为统一的结构, 即 Operation DAG。

**生成 RelNode:** Operation DAG 会被转化为 RelNode( 关系表达式 ) DAG。

**优化:** 优化器会对 RelNode 做各种优化, 优化器的输入是各种优化的规则, 以及各种统计信息。当前, 在 Blink Planner 里面, 绝大部分的优化规则, Stream 和 Batch 是共享的。差异在于, 对 Batch 而言, 它没有 state 的概念, 而对于 Stream 而言, 它是不支持 sort 的, 所以目前 Blink Planner 中, 还是运行了两套独立的规则集 (Rule Set), 然后定义了两套独立的 Physical Rel: BatchPhysical Rel 和 StreamPhysical Rel。优化器优化的结果, 就是具体的 Physical Rel DAG。

**转化:** 得到 Physical Rel Dag 后, 继续会转化为 ExecNode, 通过名字可以看出, ExecNode 已经属于执行层的概念了, 但是这个执行层是 Blink 的执行层, 在 ExecNode 中, 会进行大量的 CodeGen 的操作, 还有非 Code 的 Operator 操作, 最后, 将 ExecNode 转化为 Transformation DAG。

**生成可执行 Job Graph:** 得到 Transformation DAG 后, 最终会被转化成 Job Graph, 完成 SQL 或者 Table API 的解析。

## Blink Planner 改进及优化

Blink Planner 功能方面改进主要包含以下几个方面:

- **更完整的 SQL 语法支持:** 例如, IN, EXISTS, NOT EXISTS, 子查询, 完

整的 Over 语句, Group Sets 等。而且已经跑通了所有的 TPCH, TPCDS 这两个测试集, 性能还非常不错。

- 提供了更丰富, 高效的算子。
- 提供了非常完善的 cost 模型, 同时能够对接 Catalog 中的统计信息, 使 cost 根据统计信息得到更优的执行计划。
- 支持 join reorder。
- shuffle service: 对 Batch 而言, Blink Planner 还支持 shuffle service, 这对 Batch 作业的稳定性有非常大的帮助, 如果遇到 Batch 作业失败, 通过 shuffle service 能够很快的进行恢复。

性能方面, 主要包括以下部分:

- 分段优化。
- Sub-Plan Reuse。
- 更丰富的优化 Rule: 共一百多个 Rule, 并且绝大多数 Rule 是 Stream 和 Batch 共享的。
- 更高效的数据结构 BinaryRow: 能够节省序列化和反序列化的操作。
- mini-batch 支持(仅 Stream): 节省 state 的访问的操作。
- 节省多余的 Shuffle 和 Sort(Batch 模式): 两个算子之间, 如果已经按 A 做 Shuffle, 紧接着他下的下游也是需要按 A Shuffle 的数据, 那中间的这一层 Shuffle, 就可以省略, 这样就可以省很多网络的开销, Sort 的情况也是类似。Sort 和 Shuffle 如果在整个计算里面是占大头, 对整个性能是有很多提升的。

## 深入性能优化及实践

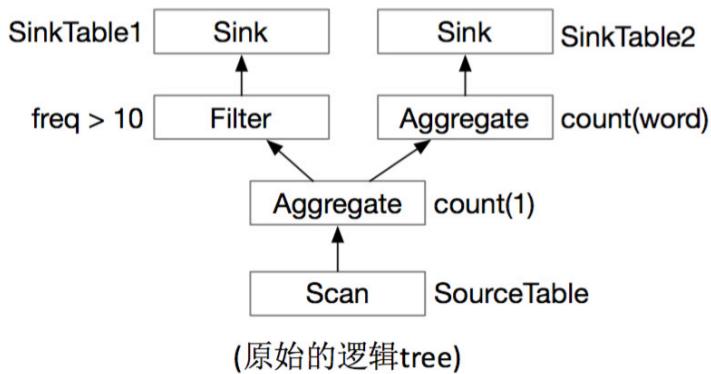
本节中, 将使用具体的示例进行讲解, 让你深入理解 Blink Planner 性能优化的设计。

## 分段优化

### 示例 5

```
create view MyView as select word, count(1) as freq from SourceTable group by word;
insert into SinkTable1 select * from MyView where freq > 10;
insert into SinkTable2 select count(word) as freq2, freq from MyView group by freq;
```

上面的这几个 SQL，转化为 RelNode DAG，大致图形如下：



(原始的逻辑tree)

图 5 示例 5 RelNode DAG

如果是使用 Flink Planner，经过优化层后，会生成如下执行层的 DAG:

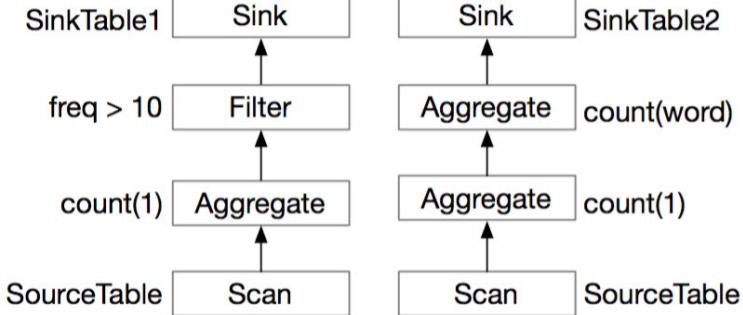


图 6 示例 5 Flink Planner DAG

可以看到，Flink Planner 只是简单的从 Sink 出发，反向的遍历到 Source，从而形成两个独立的执行链路，从上图也可以清楚的看到，Scan 和第一层 Aggregate 是有重复计算的。

在 Blink Planner 中，经过优化层之后，会生成如下执行层的 DAG：

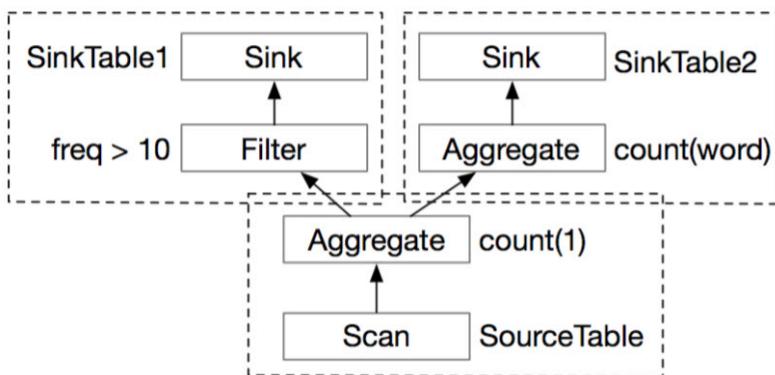


图 7 示例 5 Blink Planner DAG

Blink Planner 不是在每次调用 `insert into` 的时候就开始优化，而是先将所有的 `insert into` 操作缓存起来，等到执行前才进行优化，这样就可以看到完整的执行图，可以知道哪些部分是重复计算的。Blink Planner 通过寻找可以优化的最大公共子图，找到这些重复计算的部分。经过优化后，Blink Planner 会将最大公共子图的部分当做一个临时表，供其他部分直接使用。

这样，上面的图可以分为三部分，最大公共子图部分（临时表），临时表与 Filter 和 SinkTable1 优化，临时表与第二个 Aggregate 和 SinkTable 2 优化。

Blink Planner 其实是通过声明的 View 找到最大公共子图的，因此在开发过程中，如果需要复用某段逻辑，就将其定义为 View，这样就可以充分利用 Blink Planner 的分段优化功能，减少重复计算。

当然，当前的优化也不是最完美的，因为提前对图进行了切割，可能会导致一些

优化丢失，今后会持续地对这部分算法进行改进。

总结一下，Blink Planner 的分段优化，其实解的是多 Sink 优化问题 (DAG 优化)，单 Sink 不是分段优化关心的问题，单 Sink 可以在所有节点上优化，不需要分段。

## Sub-Plan Reuse

### 示例 6

```
insert into SinkTable
select freq from (select word, count(1) as freq from SourceTable group by
word) t where word like 'T%'
union all
select count(word) as freq2 from (select word, count(1) as freq from
SourceTable group by word) t
group by freq;
```

这个示例的 SQL 和分段优化的 SQL 其实是类似的，不同的是，没有将结果 Sink 到两个 Table 里面，而是将结果 Union 起来，Sink 到一个结果表里面。

下面看一下转化为 RelNode 的 DAG 图：

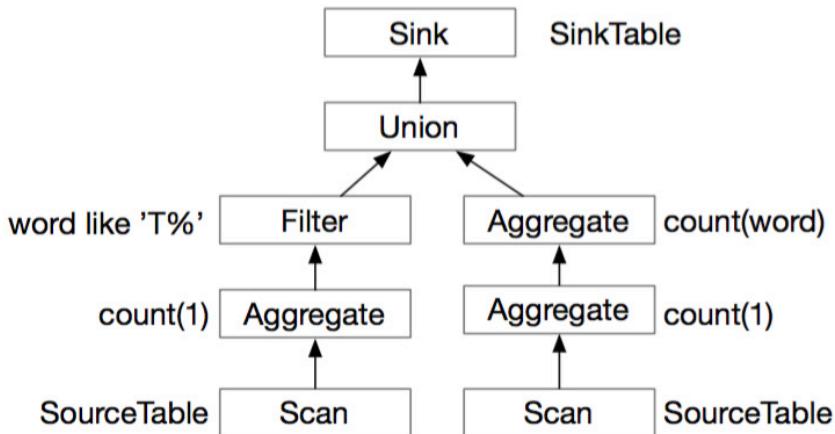


图 8 示例 6 RelNode DAG

从上图可以看出, Scan 和第一层的 Aggregate 也是有重复计算的, Blink Planner 其实也会将其找出来, 变成下面的图:

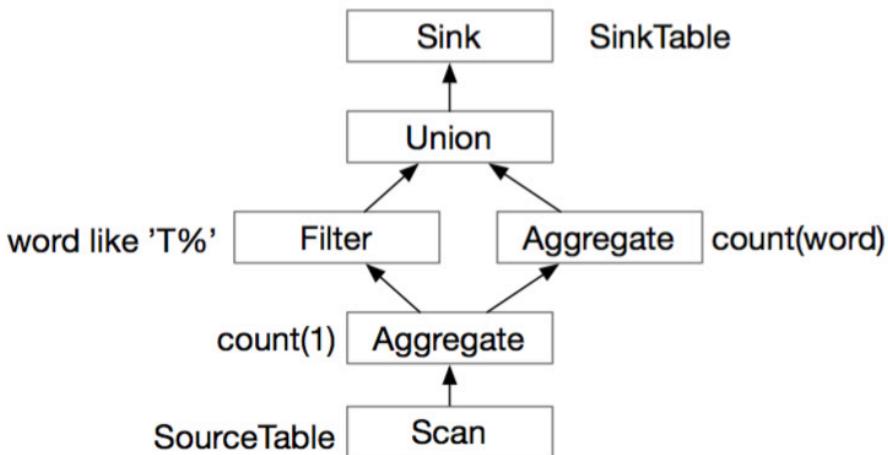


图 9 示例 6 Blink Planner DAG

Sub-Plan 优化的启用, 有两个相关的配置:

- table.optimizer.reuse-sub-plan-enabled (默认开启)
- table.optimizer.reuse-source-enabled (默认开启)

这两个配置, 默认都是开启的, 用户可以根据自己的需求进行关闭。这里主要说明一下 table.optimizer.reuse-source-enabled 这个参数。在 Batch 模式下, join 操作可能会导致死锁, 具体场景是在执行 hash-join 或者 nested-loop-join 时一定是先读 build 端, 然后再读 probe 端, 如果启用 reuse-source-enabled, 当数据源是同一个 Source 的时候, Source 的数据会同时发送给 build 和 probe 端。这时候, build 端的数据将不会被消费, 导致 join 操作无法完成, 整个 join 就被卡住了。

为了解决死锁问题, Blink Planner 会先将 probe 端的数据落盘, 这样 build 端读数据的操作才会正常, 等 build 端的数据全部读完之后, 再从磁盘中拉取 probe 端的数据, 从而解决死锁问题。但是, 落盘会有额外的开销, 会多一次写的操作;

有时候，读两次 Source 的开销，可能比一次写的操作更快，这时候，可以关闭 reuse-source，性能会更好。当然，如果读两次 Source 的开销，远大于一次落盘的开销，可以保持 reuse-source 开启。需要说明的是，Stream 模式是不存在死锁问题的，因为 Stream 模式 join 不会有选边的问题。

总结而言，sub-plan reuse 解的问题是优化结果的子图复用问题，它和分段优化类似，但他们是一个互补的过程。

注: Hash Join: 对于两张待 join 的表 t1, t2。选取其中的一张表按照 join 条件给的列建立 hash 表。然后扫描另外一张表，一行一行去建好的 hash 表判断是否有对应相等的行来完成 join 操作，这个操作称之为 probe ( 探测 )。前一张表叫做 build 表，后一张表的叫做 probe 表。

## Agg 分类优化

Blink 中的 Aggregate 操作是非常丰富的：

- group agg，例如: select count(a) from t group by b
- over agg，例如: select count(a) over (partition by b order by c) from t
- window agg， 例 如: select count(a) from t group by tumble(ts, interval '10' second), b
- table agg , 例 如: tEnv.scan('t').groupBy('a').flatAggregate(flatAggFunc('b' as ('c', 'd')))

下面主要对 Group Agg 优化进行讲解，主要是两类优化。

### 1. Local/Global Agg 优化

Local/Global Agg 主要是为了减少网络 Shuffle。要运用 Local/Global 的优化，必要条件如下：

- Aggregate 的所有 Agg Function 都是 mergeable 的，每个 Aggregate 需

要实现 merge 方法，例如 SUM, COUNT, AVG，这些都是可以分多阶段完成，最终将结果合并；但是求中位数，计算 95% 这种类似的问题，无法拆分为多阶段，因此，无法运用 Local/Global 的优化。

- table.optimizer.agg-phase-strategy 设置为 AUTO 或者 TWO\_PHASE。
- Stream 模式下，mini-batch 开启；Batch 模式下 AUTO 会根据 cost 模型加上统计数据，选择是否进行 Local/Global 优化。

### 示例 7

```
select count(*) from t group by color
```

没有优化的情况下，下面的这个 Aggregate 会产生 10 次的 Shuffle 操作。

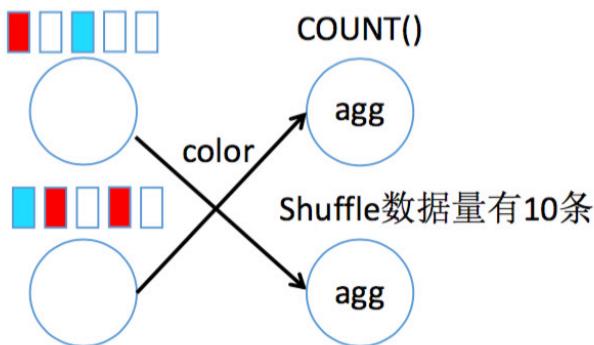


图 10 示例 7 未做优化的 Count 操作

使用 Local/Global 优化后，会转化为下面的操作，会在本地先进行聚合，然后再进行 Shuffle 操作，整个 Shuffle 的数据剩下 6 条。在 Stream 模式下，Blink 其实会以 mini-batch 的维度对结果进行预聚合，然后将结果发送给 Global Agg 进行汇总。

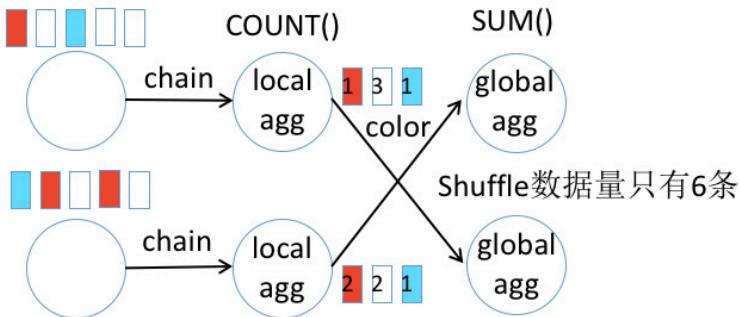


图 11 示例 7 经过 Local/Global 优化的 Count 操作

## 2. Distinct Agg 优化

Distinct Agg 进行优化，主要是对 SQL 语句进行改写，达到优化的目的。但 Batch 模式和 Stream 模式解决的问题是不同的：

- Batch 模式下的 Distinct Agg，需要先做 Distinct，再做 Agg，逻辑上需要两步才能实现，直接实现 Distinct Agg 开销太大。
- Stream 模式下，主要是解决热点问题，因为 Stream 需要将所有的输入数据放在 State 里面，如果数据有热点，State 操作会很频繁，这将影响性能。

## Batch 模式

第一层，求 distinct 的值和非 distinct agg function 的值，第二层求 distinct agg function 的值

### 示例 8

```
select color, count(distinct id), count(*) from t group by color
```

手工改写成：

```
select color, count(id), min(cnt) from (
  select color, id, count(*) filter (where $e=2) as cnt from (
```

```

select color, id, 1 as $e from t --for distinct id
union all
select color, null as id, 2 as $e from t -- for count(*)
) group by color, id, $e
) group by color

```

转化的逻辑过程, 如下图所示:

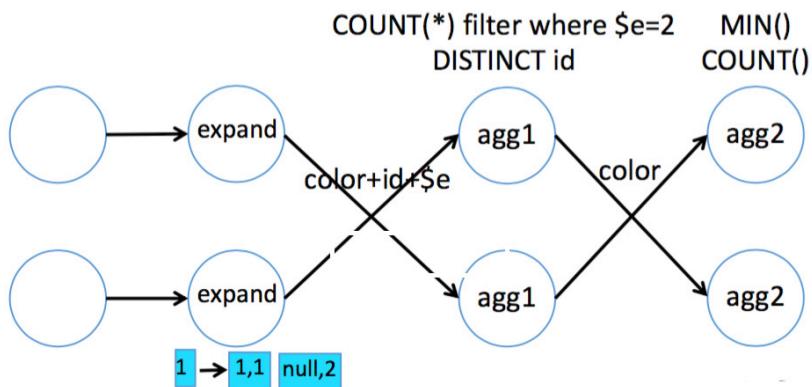


图 12 示例 8 Batch 模式 Distinct 改写逻辑

## Stream 模式

Stream 模式的启用有一些必要条件:

- 必须是支持的 agg function: avg/count/min/max/sum/first\_value(concat\_agg/single\_value);
- table.optimizer.distinct-agg.split.enabled (默认关闭)

## 示例 9

```
select color, count(distinct id), count(*) from t group by color
```

手工改写成:

```
select color, sum(dcnt), sum(cnt) from (
```

```
select color, count(distinct id) as dcnt, count(*) as cnt from t
group by color, mod(hash_code(id), 1024)
) group by color
```

改写前, 逻辑图大概如下:

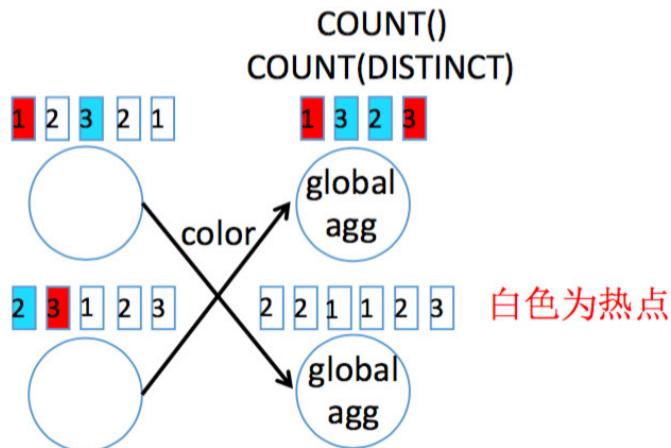


图 13 示例 9 Stream 模式未优化 Distinct

改写后, 逻辑图就会变为下面这样, 热点数据被打散到多个中间节点上。

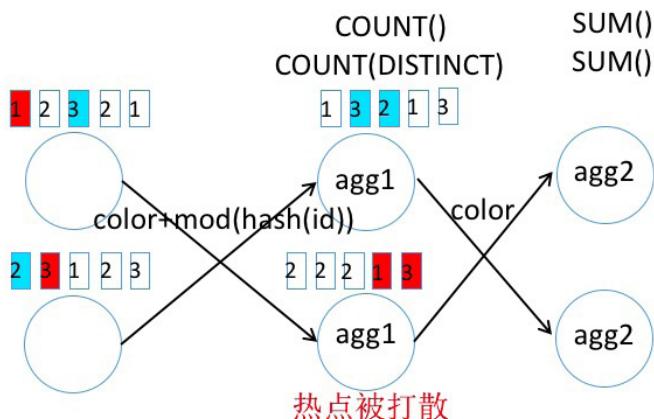


图 14 示例 9 Stream 模式优化 Distinct

需要注意的是，示例 5 的 SQL 中 `mod(hash_code(id),1024)` 中的这个 1024 为打散的维度，这个值建议设置大一些，设置太小产生的效果可能不好。

## 总结

本文首先对新的 TableEnvironment 的整体设计进行了介绍，并且列举了各种模式下 TableEnvironment 的选择，然后通过具体的示例，展示了各种模式下代码的写法，以及需要注意的事项。

在新的 Catalog 和 DDL 部分，对 Catalog 的整体设计、DDL 的使用部分也都以实例进行拆分讲解。最后，对 Blink Planner 解析 SQL/Table API 的流程、Blink Planner 的改进以及优化的原理进行了讲解，希望对大家探索和使用 Flink SQL 有所帮助。

# Apache Flink 进阶 (十三): Python API 应用实践

作者: 孙金城(金竹)

Apache Member, 阿里巴巴高级技术专家

本文重点为大家介绍 Flink Python API 的现状及未来规划, 主要内容包括: Apache Flink Python API 的前世今生和未来发展; Apache Flink Python API 架构及开发环境搭建; Apache Flink Python API 核心算子介绍及应用。

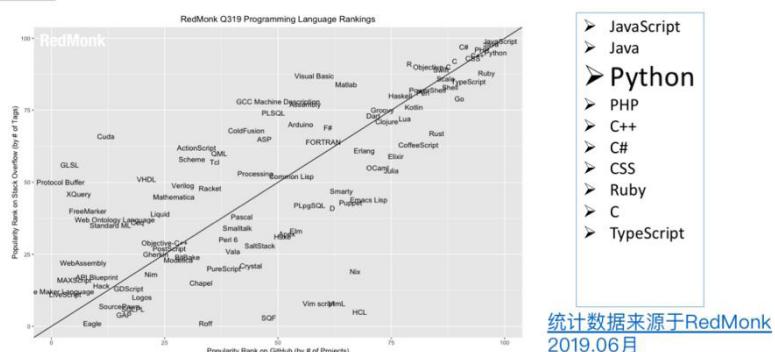
## Apache Flink Python API 的前世今生和未来发展

### 1. Flink 为什么选择支持 Python

Apache Flink 是流批统一的开源大数据计算引擎, 在 Flink 1.9.0 版本开启了新的 ML 接口和全新的 Python API 架构。那么为什么 Flink 要增加对 Python 的支持, 下文将进行详细分析。

- 最流行的开发语言

### Why Python API – 最流行的开发语言



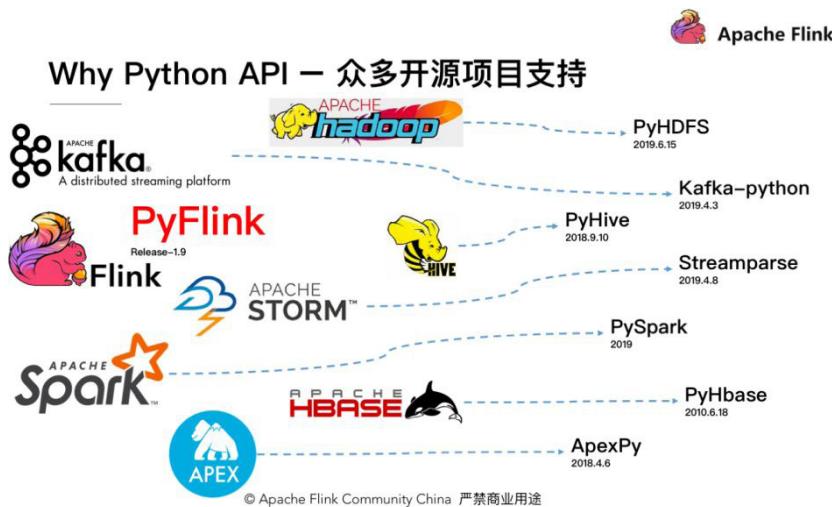
统计数据来源于 RedMonk  
2019.06月

Python 本身是非常优秀的开发语言，据 RedMonk 数据统计，除 Java 和 JavaScript 之外，受欢迎度排名第三。

RedMonk 是著名的以开发人员为中心的行业分析公司，其更详细的分析信息，大家在拿到我的 PPT 之后，可以点击链接进行详细查阅。好了，那么 Python 的火热，与我们今天向大家分享的流批统一的大数据计算引擎，Apache Flink 有什么关系呢？

带着这个问题，我们大家想想目前与大数据相关的著名的开源组件有哪些呢？比如说最早期的批处理框架 Hadoop？流计算平台 Storm，最近异常火热的 Spark？异或其他领域数仓的 Hive，KV 存储的 HBase？这些都是非常著名的开源项目，那么这些项目都无一例外的进行了 Python API 的支持。

- 众多开源项目支持



Python 的生态已相对完善，基于此，Apache Flink 在 1.9 版本中也投入了大量的精力，去推出了一个全新的 Pyflink。除大数据外，人工智能与 Python 也有十分密切的关系。

- ML 青睐的语言



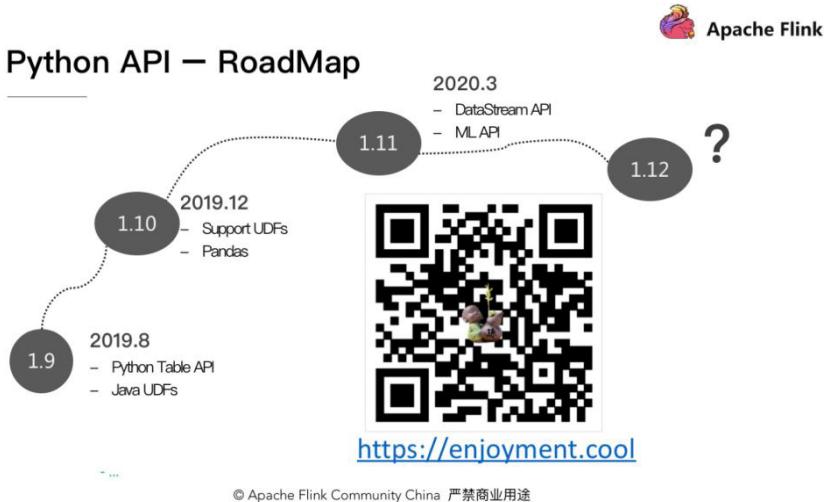
从上图统计数据可以发现，Python API 本身已经占机器学习岗位需求语言的 0.129%。相对于 R 语言，Python 语言似乎更受青睐。

Python 作为解释型语言，语法的设计哲学是”**用一种方法并且只有一种方法来做一件事**”。其简洁和易用性使其成为了世界上最受欢迎的语言，在大数据计算领域都有着很好的生态建设，同时 Python 在机器学习 在机器学习方面也有很好的前景，所以我们在近期发布的 Apache Flink 1.9 以全新的架构推出新的 Python API。

Flink 是一款流批统一的计算引擎，社区非常重视和关注 Flink 用户，除 Java 语言或者 Scala 语言，社区希望提供多种入口，多种途径，让更多的用户更方便的使用 Flink，并收获 Flink 在大数据算力上带来的价值。

因此 Flink 1.9 开始，Flink 社区以一个全新的技术体系来推出 Python API，并且已经支持了大部分常用的一些算子，比如如 JOIN, AGG, WINDOW 等。

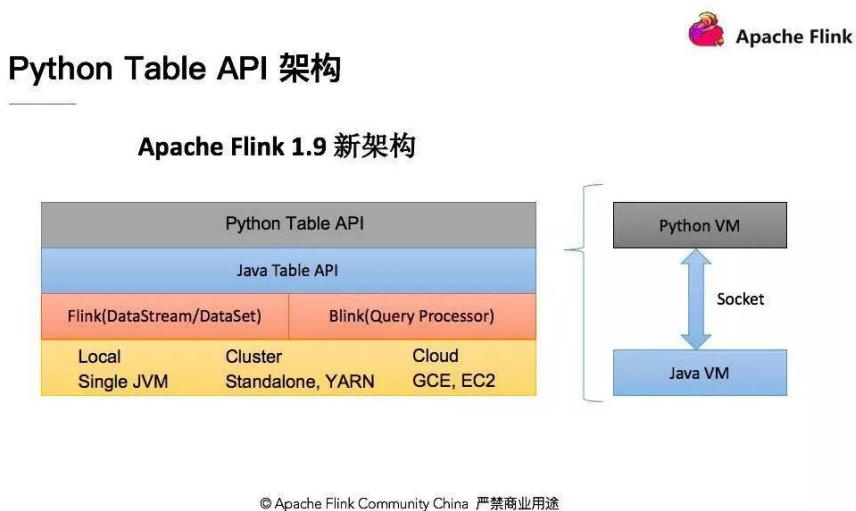
## 2. Python API – RoadMap



在 Flink 1.9 中虽然 Python 可以使用 Java 的 User-defined Function，但是还缺乏 Python native 的 User-defined function 的定义，所以我们计划在 Flink 1.10 中进行支持 Python User-defined function 的支持。并增加对数据分析工具类库 Pandas 的支持，在 Flink 1.11 增加对 DataStream API 和 ML API 的支持。

## Python API 架构及开发环境搭建

### 1. Python Table API 架构



© Apache Flink Community China 严禁商业用途

新的 Python API 架构分为用户 API 部分，PythonVM 和 Java VM 的通讯部分，和最终将作业提交到 Flink 集群进行运行的部分。那么 PythonVM 和 JavaVM 是怎样通讯的呢？我们在 Python 端会有一个 Python 的 Gateway 用于保持和 Java 通讯的链接，在 Java 部分有一个 GateWayServer 用于接收 Python 部分的调用请求。

关于 Python API 的架构部分，在 1.9 之前，Flink 的 DataSet 和 DataStream 已经有了对 Python API 的支持，但是拥有 DataSet API 和 DataStream API 两套不同的 API。

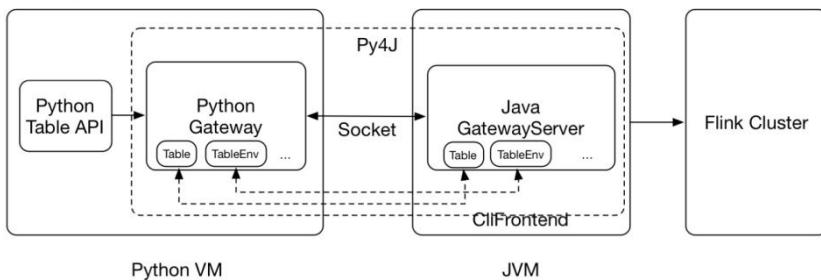
对于 Flink 这样一个流批统一的流式计算引擎来讲，统一的架构至关重要。并且对于已有的 Python DataSet API 和 DataStream API 而言，采用了 Jython 的技术体系架构，而 Jython 本身对目前 Python 的 3.X 系列无法很好的支持，所以 Flink 1.9 发布后，决定将原有的 Python API 体系架构废弃，以全新的技术架构出

现。这套全新的 Python API 基于 Table API 之上。

Table API 和 Python API 之间的通讯采用了一种简单办法，利用 Python VM 和 Java VM 进行通信。在 Python API 的书写或者调用过程中，以某种方式来与 Java API 进行通讯。操作 Python API 就像操作 Java 的 Table API 一样。新架构中可以确保以下内容：

- 不需要另外创建一套新的算子，可以轻松与 Java 的 Table API 的功能保持一致；
- 得益于现有的 Java Table API 优化模型，Python 写出来的 API，可以利用 Java API 优化模型进行优化，可以确保 Python 的 API 写出来的 Job 也能够具备极致性能。

## Python Table API 架构

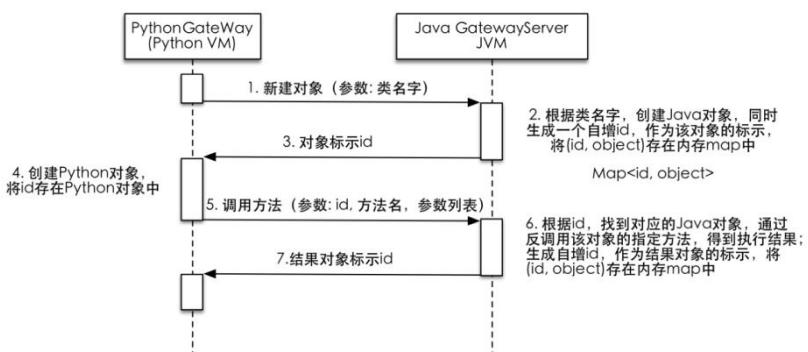


- Python object is just a wrapper of the corresponding Java object
- Leverage the QO and QE framework of Flink Table module automatically

© Apache Flink Community China 严禁商业用途

如图，当 Python 发起对 Java 的对象请求时候，在 Java 段创建对象并保存在一个存储结构中，并分配一个 ID 给 Python 端，Python 端在拿到 Java 对象的 ID 后就可以对这个对象进行操作，也就是说 Python 端可以操作任何 Java 端的对象，这也就是为什么新的架构可以保证 Python Table API 和 Java Table API 功能一致，并且能过服用现有的优化模型。

## Python Table API 架构



© Apache Flink Community China 严禁商业用途

在新的架构和通讯模型下，Python API 调用 Java API 只需要在持有 Java 对象的 ID，将调用方法的名字和参数传递给 Java VM，就能完成对 Java Table API 的调用，所以在这样的架构中开发 Python Table API 与开发 Java Table API 的方式完全一致，接下来我为大家详细介绍如何开发一个简单的 Python API 作业。

## 2. Python Table API – Job 开发



### Python Table API – Job 开发

#### Table API 开发核心部分



© Apache Flink Community China 严禁商业用途

通常来讲一个 Python Table Job 一般会分成四个部分，首先要根据目前的现状，要决定这个 Job 是以批的方式运行，还是流的方式运行。**当然后续版本用户可以不考虑，但当前 1.9 版本还是需要考虑。**

在决定第一步以怎样的方式执行 Job 后，我们需要了解数据从哪里来，如何定义 Source、结构数据类型等信息。然后需要写计算逻辑，然后就是对数据进行计算操作，但最终计算的结果需要持久化到某个系统。最后定义 Sink，与 Source 类似，我们需要定义 Sink Schema，以及每一个字段类型。

下面将详细分享如何用 Python API 写每一步？首先，我们创建一个执行环境，对于执行环境本身来讲，首先需要一个 ExecutionEnvironment，根本上我们需要一个 TableEnvironment。

那么在 TableEnvironment 中，有一个参数 Table Config，Table Config 中会有一些在执行过程中的配置参数，可以传递到 RunTime 层。除此之外，还提供了一些个性化的配置项，可以在实际业务开发中进行使用。



## Python Table API – Job 开发

### 创建执行环境

```
# 创建执行环境
exec_env = ExecutionEnvironment.get_execution_environment()

# 创建配置对象(IdleState TTL, NULL check, timezone 等)
t_config = TableConfig()

# 创建一个Table ENV
t_env = BatchTableEnvironment.create(exec_env, t_config)
```

用“逗号”分隔，用 Field 来表明这个文件中有哪些字段。那么会看到，目前里面用逗号分隔，并且只有一个字段叫 word，类型是 String。

## Python Table API – Job开发

### 创建数据源

```
# 创建数据源表
t_env.connect(FileSystem().path(source_file)) \
    # Csv/Json/Avro
    .with_format(OldCsv() \
        .line_delimiter(',')
        .field('word', DataTypes.STRING()))
# 定义字段名和类型
    .with_schema(Schema() \
        .field('word', DataTypes.STRING()))
# 注册Source
.register_table_source('mySource')
```

word
enjoyment
Apache Flink
cool
Apache Flink



© Apache Flink Community China 严禁商业用途

在定义并描述完数据源数据结构转换成 Table 数据结构后，也就是说转换到 Table API 层面之后是怎样的数据结构和数据类型？下面将通过 with\_schema 添加字段及字段类型。这里只有一个字段，数据类型也是 String，最终注册成一个表，注册到 catalog 中，就可以供后面的查询计算使用了。

## Python Table API – job开发

### 创建结果表

```
# 创建结果表
t_env.connect(FileSystem().path(sink_file)) \
    # Csv/Json/Avro
    .with_format(OldCsv() \
        .line_delimiter(',')
        .field('word', DataTypes.STRING())
        .field('count', DataTypes.BIGINT()))
# 定义字段名和类型
    .with_schema(Schema() \
        .field('word', DataTypes.STRING())
        .field('count', DataTypes.BIGINT()))
.register_table_sink('mySink')
```

word	count



© Apache Flink Community China 严禁商业用途

创建结果表，当计算完成后需要将这些结果存储到持久化系统中，以 WordCount 为例，首先存储表会有一个 word 以及它的计数两个字段，一个是 String 类型的 word，另一个是 Bigint 的计数，然后把它注册成 Sink。



## Python Table API — Job开发

### 编写业务逻辑 和执行

```
# word_count计算逻辑
# 读取数据源
t_env.scan('mySource') \
    # 按 word 进行分组
    .group_by('word') \
    # 进行count计数统计
    .select('word, count(1)') \
    # 将计算结果插入到结果表
    .insert_into('mySink')

# 执行Job
t_env.execute("wordcount")
```

word	count
Apache Flink	2
cool	1
enjoyment	1

[https://github.com/sunjincheng121/enjoyment\\_code/blob/master/myPyFlink/enjoyment/word\\_count.py](https://github.com/sunjincheng121/enjoyment_code/blob/master/myPyFlink/enjoyment/word_count.py)

© Apache Flink Community China 严禁商业用途

编写注册完 Table Sink 后，再来看如何编写逻辑。其实用 Python API 写 WordCount 和 Table API 一样非常简单。因为相对于 DataStream 而言 Python API 写一个 WordCount 只需要一行。比如 group by，先扫描 Source 表，然后 group by 一个 Word，再进行 Select word 并加上聚合统计 Count，最终将最数据结果插入到结果表里面中。

### 3. Python Table API – 环境搭建

 Apache Flink

### Python Table API – 环境搭建

---

**环境依赖检查**

- JDK 1.8+ (1.8.0\_211) `java -version`
- Maven 3.x (3.2.5) `mvn -version`
- Scala 2.11+ (2.12.0) `scala -version`
- Python 2.7+ (2.7.16) `python -V`
- Git 2.20+ (2.20.1) `git version`
- Pip 19.1+ (pip 19.1.1) `pip -V`

© Apache Flink Community China 严禁商业用途

那么 WordCount 怎样才能真正的运行起来？首先需要搭建开发环境，不同的机器上可能安装的软件版本不一样，这里列出来了一些版本的需求和要求，其中括号中是示例机器上的版本。

 Apache Flink

### Python Table API – 环境搭建

---

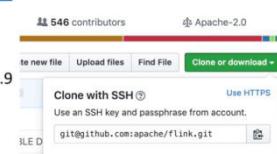
**构建Java二进制发布包**

我们要想利用Apache Flink Python API 进行业务开发，需要将PyFlink 发布包进行安装。目前PyFlink并没有发布到Python仓库，所以我们需要从源码构建。

```
# 下载源代码
git clone https://github.com/apache/flink.git

# 拉取1.9分支
cd flink; git fetch origin release-1.9
git checkout -b release-1.9 origin/release-1.9

#构建二进制发布包
mvn clean install -DskipTests -Dfast
```



© Apache Flink Community China 严禁商业用途

第二步，构建一个 Java 的二进制发布包，以从源代码进行构建，那么这一页面就是从原代码获取我们的主干代码，并且拉取 1.9 的分支。当然大家可以用 Master，但是 Master 不够稳定，还是建议大家在自己学习的过程中，最好是用 1.9 的分支去做。接下来进行实战演练环节，首先验证 PPT 的正确性。首先编译代码，示例如下：

```
// 下载源代码
git clone https://github.com/apache/flink.git/
拉取 1.9 分支
cd flink; git fetch origin release-1.9
git checkout -b release-1.9
origin/release-1.9
// 构建二进制发布包
mvn clean install -DskipTests -Dfas
```

编译完成后，需要在相应目录下找到发布包：

```
cd flink-dist/target/flink-1.9.0-bin/flink-1.9.0
tar -zcvf flink-1.9.0.tar.gz flink-1.9.0
```

在构建完 Java 的 API 之后进行检验，我们要构建一个 Python 的发布包。



## Python Table API – 环境搭建

### 构建Python 发布包

一般情况我们期望以 `pip install` 的方式安装 `python` 的类库，我们要想安装 PyFlink 的类库，也需要构建可用于 `pip install` 的发布包。执行如下命令：

```
cd flink-python; python setup.py sdist
...
copying pyflink/util/utils.py -> apache-flink-1.9.dev0/pyflink/util
Writing apache-flink-1.9.dev0/setup.cfg
creating dist
Creating tar archive
removing 'apache-flink-1.9.dev0' (and everything under it)
```

在 `dist` 目录的 `apache-flink-1.9.dev0.tar.gz` 就是我们可以用于 `pip install` 的 PyFlink 包。

库进行与本地的 Python 环境进行集成或者安装。

那么 Flink 也是一样，PyFlink 也需要打包一个 Pypip 能够识别的资源进行安装，在实际的使用中，也可以按这种命令去拷贝，在自己的环境中尝试。

```
cd flink-Python;Python setup.py sdist
```

这个过程只是将 Java 包囊括进来，再把自己 PyFlink 本身模块的一些 Java 的包和 Python 包打包成一起，它会在 dist 目录下，有一个 apache-flink-1.9.dev0.tar.gz。

```
cd dist/
```

在 dist 目录的 apache-flink-1.9.dev0.tar.gz 就是我们可以用 pip install 的 PyFlink 包。在 1.9 版本，除了 Flink Table，还有 Flink Table Blink。Flink 同时会支持两个 plan，如果大家可以尝试，我们可以自由的切换是 Flink 原有的 Planner，还是 Blink 的 Planner，大家可以去尝试。完成打包后，就可以尝试把包安装到我们的实际环境当中。



## Python Table API — 环境搭建

### 安装PyFlink

接下来我们将PyFlink安装到Python的环境中：

```
pip install dist/*.tar.gz
```

用pip命令检查是否安装成功： pip list

```
IT-C02YL16BJHD2:flink-python jincheng.sunjc$ pip list
Package           Version
apache-flink      1.9.dev0
configparser      3.7.4
entrypoints       0.3
enum34            1.1.6
flake8            3.7.8
```

接下来是一个非常简单的命令，首先检查命令的正确性，在执行之前，我们用 pip 检查一下 list，我们要看在已有的包里有没有，现在尝试把刚才打包的包再安装。在实际的使用过程中，如果升级版，也要有这个过程，要把新的包要进行安装。

```
pip install dist/*.tar.gz
pip list|grep flink
```



## Python Table API — 环境搭建

### 验证PyFlink

我们可以运行刚才开发的 word\_count.py，以验证环境的正确性：

[https://github.com/sunjincheng121/enjoyment.code/blob/master/myPyFlink/enjoyment/word\\_count.py](https://github.com/sunjincheng121/enjoyment.code/blob/master/myPyFlink/enjoyment/word_count.py)

<https://github.com/sunjincheng121/enjoyment.code/blob/master/myPyFlink/enjoyment/source.csv>

Git clone <https://github.com/sunjincheng121/enjoyment.code.git>

cd enjoyment.code; python word\_count.py

© Apache Flink Community China 严禁商业用途

安装完成后，就可以用刚才写的 WordCount 例子来验证环境是否正确。验证一下刚才的正确性，怎么验证？为了大家方便，可以直接克隆 enjoyment.code 仓库。

```
git clone https://github.com/sunjincheng121/enjoyment.code.git
cd enjoyment.code; Python word_count.py
```

接下来体验并尝试。在这个目录下，我们刚才开发的 WordCount 例子。直接用 Python 或检验环境是否 OK。这个时候 Flink Python API 会启动一个 Mini 的 Cluster，会将刚才 WordCount Job 进行执行，提交到一个 Mini Cluster 进行执行。现在 Run 的过程中其实已经在集群上进行执行了。

其实在这个代码里面是读了一个 Source 文件，把结果写到 CSV 文件，在当

前目录，是有一个 Sink CSV 的。具体的操作步骤可以查看 Flink 中文社区视频 Apache Flink Python API 现状及规划。



## Python Table API — 环境搭建

### IDE配置

Apache Flink 官网  
[https://ci.apache.org/projects/flink/flink-docs-master/flinkDev/ide\\_setup.html#pycharm](https://ci.apache.org/projects/flink/flink-docs-master/flinkDev/ide_setup.html#pycharm)

注意的细节，我们边操作，边说明。

同时我也整理到了Blog：<http://1t.click/6Nf>



© Apache Flink Community China 严禁商业用途

IDE 的配置在正常的开发过程中，其实我们大部分还是在本地进行开发的，这里推荐大家还是用 Pychram 来开发 Python 相关的逻辑或者 Job。

同时由于有大量的截图存在，也把这些内容整理到了博客当中，大家可以扫描二维码去关注和查看那么一些详细的注意事项，博客详细地址：

<https://enjoyment.cool>

这里有一个很关键的地方，大家要注意，就是可能你的环境中有很多 Python 的环境，这时候选择的环境一定是刚才 pip install 环境。具体操作详见 Apache Flink Python API 现状及规划。

## 4. Python Table API – 作业提交



### Python Table API – 作业提交

1. 启动或者配置Apache Flink 集群，我们这里启动一个本地集群：

```
./bin/start-cluster.sh
```

启动之后查看启动日志，在文件中找到

#### CLI方式提交

```
Rest endpoint listening at localhost:8081
```

的信息，证明集群启动正常，我们可以访问 Apache Flink Dashboard

<http://localhost:8081>

© Apache Flink Community China 严禁商业用途

还有哪些方式来提交 Job 呢？这是一个 CLI 的方式，也就是说真正的提交到一个现有的集群。首先启动一个集群。构建的目录一般在 target 目录下，如果要启动一个集群，直接启动就可以。

这里要说一点的是，其中一个集群外部有个 Web Port，它的端口的地址都是在 flink-conf.yaml 配置的。按照 PPT 中命令，可以去查看日志，看是否启动成功，然后从外部的网站访问。如果集群正常启动，接下来看如何提交 Job 。



## Python Table API – 作业提交

2. 提交已经写好的Job到集群运行，我们用如下命令：

```
./bin/flink run -py
~/training/0806/enjoyment.code/myPyFlink/enjoyment/word_count_cli.py
```

### CLI方式提交

详细命令说明请查阅：<https://ci.apache.org/projects/flink/flink-docs-master/ops/cli.html>

- -py 指定python文件
- -pym 指定python的module
- -pyfs 指定python依赖的资源文件
- -j 指定依赖的JAR包

© Apache Flink Community China 严禁商业用途

Flink 通过 run 提交作业，示例代码如下：

```
./bin/flink run -py ~/training/0806/enjoyment.code/myPyFlink/enjoyment/word_
count_cli.py
```

用命令行方式去执行，除了用 PY 参数，还可以指定 Python 的 module，以及其他一些依赖的资源文件、JAR 等。



## Python Table API – 作业提交

Python shell 是很方便进行研究性开发的，我们可以方便的在Python REPL 中进行开发Flink Python Table API。

详细文档参考：[https://ci.apache.org/projects/flink/flink-docs-master/ops/python\\_shell.html](https://ci.apache.org/projects/flink/flink-docs-master/ops/python_shell.html)

### Python-Shell 方式提交

接下来我们以Local和Remote两种方式，体验一下Python-Shell。

- Local  
bin/pyflink-shell.sh local (会启动一个mini Cluster)
- Remote  
bin/pyflink-shell.sh remote 127.0.0.1 4000 (需要一个已经存在的Cluster)

© Apache Flink Community China 严禁商业用途

在 1.9 版本中还为大家提供一种更便利的方式，就是以 Python Shell 交互式的方式来写 Python API 拿到结果。有两种方式可执行，第一种方式是 Local，第二种方式 Remote，其实这两种没有本质的差异。首先来看 Local，命令如下：

```
bin/pyflink-shell.sh local
```

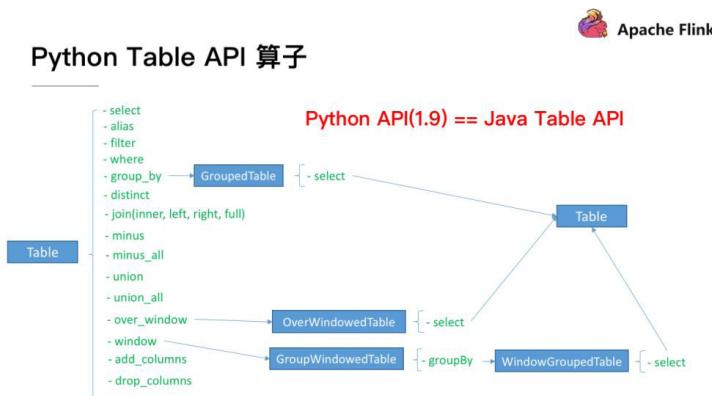
启动一个 mini Cluster，当输出后，会出来一个 Python 的 Flink CLI 同时会有一些示例程序，供大家来体验，按照上面的案例就能够达到正确的输出和提交，既可以写 Streaming，也可以写 Batch。详细步骤大家参考视频操作即可。

到目前为止，大家应该已经对 Flink 1.9 上 Python API 架构有了大概了解，同时也了解到如何搭建 Python API 环境。并且以一个简单的 WordCount 示例，体验如何在 IDE 里面去执行程序，如何以 Flink run 和交互式的方式去提交 Job。同时也体验了现有一些交互上的一种方式来使用 Flink Python API。

那么介绍完了整个 Flink 的一些环境搭建和一个简单的示例后。接下来详细介绍一下在 1.9 里面所有的核心算子。

## Flink Python API 核心算子介绍及应用

### 1. Python Table API 算子



上面分享创建一个 Job 的过程，第一要选择执行的方式是 Streaming 还是 Batch；第二个要定义使用的表，Source、Schema、数据类型；第三是开发逻辑，同时在写 WordCount 时，使用 Count 的函数。最后，在 Python API 里面内置了很多聚合函数，可以使用 count,sum, max,min 等等。

所以在目前 Flink 1.9 版本中，已经能够满足大多数常规需求。除了刚才讲到的 count。Flink Table API 算子 1.9 中也已经支持。关于 Flink Table API 算子，不论是 Python Table API 还是 Java 的 Table API，都有以下几种类型的操作。

第一单流上的操作，比如说做一些 SELECT、Filter，同时还可以在流上做一些聚合，包括开窗函数的 windows 窗口聚合以及列的一些操作，比如最下面的 add\_columns 和 drop\_columns。

除了单流，还有双流的操作，比如说双流 JOIN、双流 minus、union，这些算子在 Python Table API 里面都提供了很好的支持。Python Table API 在 Flink 1.9 中，从功能的角度看几乎完全等同于 Java Table API，下面以实际代码来看上述算子是怎么编写的以及怎么去开发 Python 算子。

## 2. Python Table API 算子 –Watermark 定义



**Apache Flink**

### Python Table API 算子–Watermark 定义

```

.with_format(
    Json()
    .fail_on_missing_field(True)
    .json_schema(
        """
        " type: 'object'
        " properties: {
        " a: {
        " type: 'string'
        }
        " time: {
        " type: 'string'
        " format: 'date-time'
        }
        "
        }
        """
    )
    .with_schema(
        Schema()
        .field("rowtime", DataTypes.TIMESTAMP())
        .rowtime(
            Rowtime()
            .timestamps_from_field("time")
            .watermarks_periodic_bounded(60000))
        .field("a", DataTypes.STRING())
    )
)

```

Watermark原理介绍，  
查阅我的博客：<http://1t.click/7dM>

细心的同学可能会注意到，我们尚未提到流的一个特质性 -> 时序。流的特性是来的顺序是可能乱序，而这种乱序又是流上客观存在的一种状态。在 Flink 中一般采用 Watermark 机制来解决这种乱序的问题。

在 Python API 中如何定义 Watermark？假设有一个 JSON 数据，a 字段 String, time 字段 datetime。这个时候定义 Watermark 就要在增加 Schema 时增加 rowtime 列。rowtime 必须是 timestamps 类型。

Watermark 有多种定义方式，上图中 watermarks\_periodic\_bounded 即会周期性的去发 Watermark，6 万单位是毫秒。如果数据是乱序的，能够处理一分钟之内的乱序，所以这个值调的越大，数据乱序接受程度越高，但是有一点数据的延迟也会越高。

关于 Watermark 原理大家可以查看我的 blog:

<http://1t.click/7dM>

### 3. Python Table API – Java UDF



## Python Table API – Java UDF

---

**Java UDF**

虽然我们在Flink-1.9中没有支持Python的UDF，但在Flink 1.9版本中我们可以使用Java UDF。

1. 创建Java项目，并配置pom依赖如下：

```
<dependency>
<groupId>org.apache.flink</groupId>
<artifactId>flink-table-common</artifactId>
<version>1.9-SNAPSHOT</version>
<scope>provided</scope>
</dependency>
```

最后，跟大家分享一下 Java UDF 在 Flink 1.9 版本中的应用，虽然在 1.9 中不支持 Python 的 UDF，但 Flink 为大家提供了可以在 Python 中使用 Java UDF。在 Flink 1.9 中，对 Table 模块进行了优化和重构，目前开发 Java UDF 只需要引入 Flink common 依赖就可以进行 Python API 开发。



## Python Table API — Java UDF

### Java UDF

2. 编写一个计算字符串长度的函数UDFLength

```
package org.apache.flink.udf;
import org.apache.flink.table.functions.ScalarFunction;

public class UDFLength extends ScalarFunction {
    public int eval(String str) {
        return str.length();
    }
}
注册和使用:
t_env.register_java_function("len", "org.apache.flink.udf.UDFLength")
...
.select("word, len(word), count(1) as count")
```

© Apache Flink Community China 严禁商业用途

接下来以一个具体的示例给大家介绍利用 Java UDF 开发 Python API UDF，假设我们开发一个求字符串长度的 UDF，在 Python 中需要用 Java 中的 register\_java\_function，function 的名字是包全路径。然后在使用时，就可以用注册的名字完成 UDF 的调用，详细可以查阅我的 Blog：

<http://1t.click/HQF>



## Python Table API — Java UDFs

### Java UDFs

开发 Python Job，并使用上面自定义的UDFLength函数：

[https://github.com/sunjincheng121/enjoyment\\_code/blob/master/myPyFlink/enjoyment/word\\_count\\_udf.py](https://github.com/sunjincheng121/enjoyment_code/blob/master/myPyFlink/enjoyment/word_count_udf.py)

提交 Python Job，并上传 UDF JAR 包：

`./bin/flink run -py word_count_udf.py -j <PATH>/flink-udf-1.0.0.jar`

© Apache Flink Community China 严禁商业用途

那怎样来执行？可以用 Flink run 命令去执行，同时需要将 UDF 的 JAR 包携  
带上去。

Java UDF 只支持 Scalar Function？其实不然，在 Java UDF 中既支持  
Scalar Function，也支持 Table Function 和 Aggregate Function。如下所示：



## Python Table API — Java UDF

### Java UDFs

- Scalar Function  
`t_env.register_java_function("len", "org.apache.flink.udf.UDFLength")  
...  
.select("word, len(word), count(1) as count")`

- Table Function  
`t_env.register_java_function("split", "com.pyflink.table.Split")  
tab.join_lateral("Split(a) as (word, length)").select("a, word, length")`

- Aggregate Function  
`t_env.register_java_function("wAvg", "com.pyflink.table.WeightedAvg")  
tab.group_by("a").select("a, wAvg(b) as d")`

© Apache Flink Community China 严禁商业用途

## 4. Python Table API 常用链接



### Python Table API 常用链接

- Python Table API 文档  
<https://ci.apache.org/projects/flink/flink-docs-master/api/python/>
- Python Table API IDE 开发环境  
<https://cwiki.apache.org/confluence/display/FLINK/Setting+up+a+Flink+development+environment>
- Python Shell  
[https://ci.apache.org/projects/flink/flink-docs-master/ops/python\\_shell.html](https://ci.apache.org/projects/flink/flink-docs-master/ops/python_shell.html)
- Python Table API Tutorial  
[https://ci.apache.org/projects/flink/flink-docs-master/tutorials/python\\_table\\_api.html](https://ci.apache.org/projects/flink/flink-docs-master/tutorials/python_table_api.html)
- 我的个人博客  
<https://enjoyment.cool/>

© Apache Flink Community China 严禁商业用途

上面所讲到的一些东西，有一些长链的文档和链接，也放在 PPT 上方便大家查阅，同时最下面我也有个人博客。希望对大家有帮助。

## 总结

简单的总结一下，本篇首先是介绍了 Apache Flink Python API 历史发展的过程，介绍了 Apache Flink Python API 架构变更的原因以及当前架构模型；任何对未来 Flink Python API 的规划与功能特性继续详细介绍，最后期望大家能在 QA 环节能给一些建议和意见，谢谢！



扫一扫二维码图案，关注我吧



开发者社区



阿里云实时计算



实时计算交流钉钉群



Flink 社区微信公众号