详解 Flink 实时应用的确定性

发布于: 2020 年 07 月 15 日

作者：林小铂（网易游戏）

确定性（Determinism）是计算机科学中十分重要的特性，确定性的算法保证对于给定相同的输入总是产生相同的输出。在分布式实时计算领域，确定性是业界一直难以解决的课题，由此导致用离线计算修正实时计算结果的 Lambda 架构成为大数据领域过去近十年的主流架构。

而在最近几年随着 Google The Dataflow Model 的提出，实时计算和离线计算的关系逐渐清晰，在实时计算中提供与离线计算一致的确定性成为可能。本文将基于流行实时计算引擎 Apache Flink，梳理构建一个确定性的实时应用要满足什么条件。

确定性与准确性

比起确定性，准确性（Accuracy）可能是我们接触更多的近义词，大多数场景下两者可以混用，但其实它们稍有不同: 准确的东西一定是确定的，但确定性的东西未必百分百准确。在大数据领域，不少算法可以根据需求调整成本和准确性的平衡，比如 HyperLogLog 去重统计算法给出的结果是有一定误差的（因此不是准确的），但却同时是确定性的（重算可以得到相同结果）。

要分区确定性和准确性的缘故是，准确性与具体的业务逻辑紧密耦合难以评估，而确定性则是通用的需求（除去少数场景用户故意使用非确定性的算法）。当一个 Flink 实时应用提供确定性，意味着它在异常场景的自动重试或者手动重流数据的情况下，都能像离线作业一般产出相同的结果，这将很大程度上提高用户的信任度。

影响 Flink 应用确定性的因素

投递语义

常见的投递语义有 At-Most-Once、At-Least-Once 和 Exactly-Once 三种。严格来说只有 Exactly-Once 满足确定性的要求，但如果整个业务逻辑是幂等的， 基于 At-Least-Once 也可以达到结果的确定性。

实时计算的 Exactly-Once 通常指端到端的 Exactly-Once，保证输出到下游系统的数据和上游的数据是一致的，没有重复计算或者数据丢失。要达到这点，需要分别实现读取数据源（Source 端）的 Exactly-Once、计算的 Exactly-Once 和输出到下游系统（Sink 端）的 Exactly-Once。

其中前面两个都比较好保证，因为 Flink 应用出现异常会自动恢复至最近一个成功 checkpoint，Pull-Based 的 Source 的状态和 Flink 内部计算的状态都会自动回滚到快照时间点，而问题在于 Push-Based 的 Sink 端。Sink 端是否能顺利回滚依赖于外部系统的特性，通常来说需要外部系统支持事务，然而不少大数据组件对事务的支持并不是很好，即使是实时计算最常用的 Kafka 也直到 2017 年的 0.11 版本才支持事务，更多的组件需要依赖各种 trick 来达到某种场景下的 Exactly-Once。

总体来说这些 Trick 可以分为两大类:

依赖写操作的幂等性。比如 HBase 等 KV 存储虽然没有提供跨行事务，但可以通过幂等写操作配合基于主键的 Upsert 操作达到 Exactly-Once。不过由于 Upsert 不能表达 Delete 操作，这种模式不适合有 Delete 的业务场景。
预写日志（WAL，Write-Ahead-Log）。预写日志是广泛应用于事物机制的技术，包括 MySQL、PostgreSQL 等成熟关系型数据库的事物都基于预写日志。预写日志的基本原理先将变更写入缓存区，等事务提交的时候再一次全部应用。比如 HDFS/S3 等文件系统本身并不提供事务，因此实现预写日志的重担落到了它们的用户（比如 Flink）身上。通过先写临时的文件/对象，等 Flink Checkpoint 成功后再提交，Flink 的 FileSystem Connector 实现了 Exactly-Once。然而，预写日志只能保证事务的原子性和持久性，不能保证一致性和隔离性。为此 FileSystem Connector 通过将预写日志设为隐藏文件的方式提供了隔离性，至于一致性（比如临时文件的清理）则无法保证。

为了保证 Flink 应用的确定性，在选用官方 Connector，特别是 Sink Connector 时，用户应该留意官方文档关于 Connector 投递语义的说明[3]。此外，在实现定制化的 Sink Connector 时也需要明确达到何种投递语义，可以参考利用外部系统的事务、写操作的幂等性或预写日志三种方式达到 Exactly-Once 语义。

函数副作用

函数副作用是指用户函数对外界造成了计算框架意料之外的影响。比如典型的是在一个 Map 函数里将中间结果写到数据库，如果 Flink 作业异常自动重启，那么数据可能被写两遍，导致不确定性。对于这种情况，Flink 提供了基于 Checkpoint 的两阶段提交的钩子（CheckpointedFunction 和 CheckpointListener），用户可以用它来实现事务，以消除副作用的不确定性。另外还有一种常见的情况是，用户使用本地文件来保存临时数据，这些数据在 Task 重新调度的时候很可能丢失。其他的场景或许还有很多，总而言之，如果需要在用户函数里改变外部系统的状态，请确保 Flink 对这些操作是知情的（比如用 State API 记录状态，设置 Checkpoint 钩子）。

Processing Time

在算法中引入当前时间作为参数是常见的操作，但在实时计算中引入当前系统时间，即 Processing Time，是造成不确定性的最常见也最难避免的原因。对 Processing 的引用可以是很明显、有完善文档标注的，比如 Flink 的 Time Characteristic，但也可能是完全出乎用户意料的，比如来源于缓存等常用的技术。为此，笔者总结了几类常见的 Processing Time 引用:

Flink 提供的 Time Characteristic。Time Characteristic 会影响所有使用与时间相关的算子，比如 Processing Time 会让窗口聚合使用当前系统时间来分配窗口和触发计算，造成不确定性。另外，Processing Timer 也有类似的影响。
直接在函数里访问外部存储。因为这种访问是基于外部存储某个 Processing Time 时间点的状态，这个状态很可能在下次访问时就发生了变化，导致不确定性。要获得确定性的结果，比起简单查询外部存储的某个时间点的状态，我们应该获取它状态变更的历史，然后根据当前 Event Time 去查询对应的状态。这也是 Flink SQL 中 Temporary Table Join 的实现原理[1]。
对外部数据的缓存。在计算流量很大的数据时，很多情况下用户会选择用缓存来减轻外部存储的负载，但这可能会造成查询结果的不一致，而且这种不一致是不确定的。无论是使用超时阈值、LRU（Least Recently Used）等直接和系统时间相关的缓存剔除策略，还是 FIFO（First In First Out）、LFU（Less Frequently Used）等没有直接关联时间的剔除策略，访问缓存得到的结果通常和消息的到达顺序相关，而在上游经过 shuffle 的算子里面这是难以保证的（没有 shuffle 的 Embarrassingly Parallel 作业是例外）。
Flink 的 StateTTL。StateTTL 是 Flink 内置的根据时间自动清理 State 的机制，而这里的时间目前只提供 Processing Time，无论 Flink 本身使用的是 Processing Time 还是 Event Time 作为 Time Characteristic。BTW，StateTTL 对 Event Time 的支持可以关注 FLINK-12005[2]。

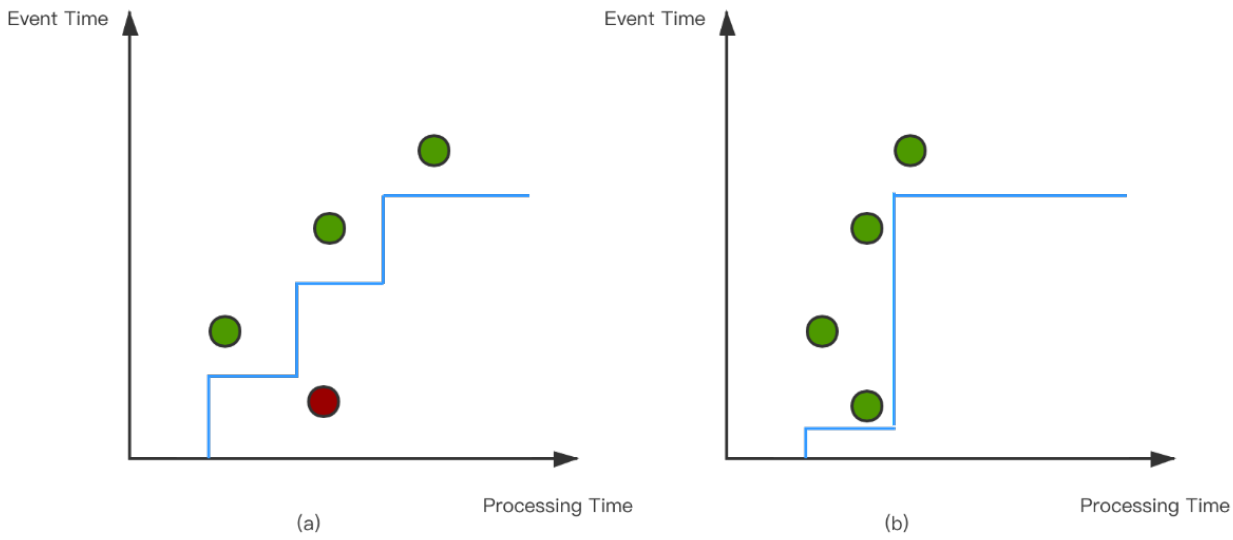
综合来讲，要完全避免 Processing Time 造成的影响是非常困难的，不过轻微的不确定性对于业务来说通常是可以接受的，我们要做的更多是提前预料到可能的影响，保证不确定性在可控范围内。

Watermark

Watermark 作为计算 Event Time 的机制，其中一个很重要的用途是决定实时计算何时要输出计算结果，类似文件结束标志符（EOF）在离线批计算中达到的效果。然而，在输出结果之后可能还会有迟到的数据到达，这称为窗口完整性问题（Window Completeness）。

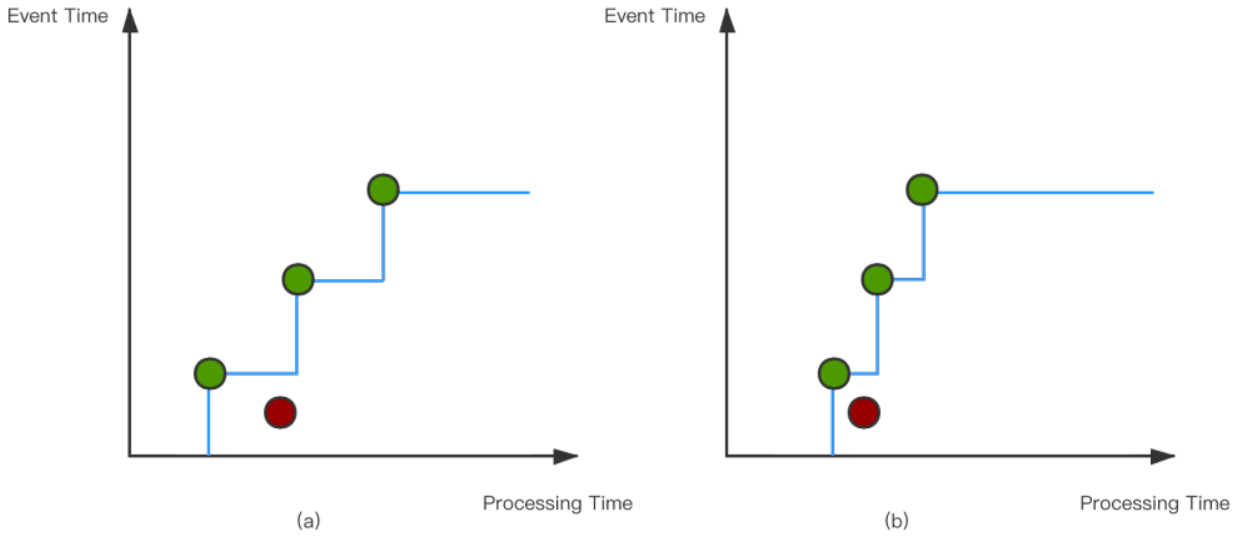
窗口完整性问题无法避免，应对办法是要么更新计算结果，要么丢弃这部分数据。因为离线场景延迟容忍度较大，离线作业可以推迟一定时间开始，尽可能地将延迟数据纳入计算。而实时场景对延迟有比较高的要求，因此一般是输出结果后让状态保存一段时间，在这段时间内根据迟到数据持续更新结果（即 Allowed Lateness），此后将数据丢弃。因为定位，实时计算天然可能出现更多被丢弃的迟到数据，这将和 Watermark 的生成算法紧密相关。

虽然 Watermark 的生成是流式的，但 Watermark 的下发是断点式的。Flink 的 Watermark 下发策略有 Periodic 和 Punctuated 两种，前者基于 Processing Time 定时触发，后者根据数据流中的特殊消息触发。



Periodic Watermark 正常状态与重放追数据状态

基于 Processing Time 的 Periodic Watermark 具有不确定。在平时流量平稳的时候 Watermark 的提升可能是阶梯式的（见图1(a)）；然而在重放历史数据的情况下，相同长度的系统时间内处理的数据量可能会大很多（见图1(b)），并且伴有 Event Time 倾斜（即有的 Source 的 Event Time 明显比其他要快或慢，导致取最小值的总体 Watermark 被慢 Watermark 拖慢），导致本来丢弃的迟到数据，现在变为 Allowed Lateness 之内的数据（见图1中红色元素）。



Punctuated Watermark 正常状态与重放追数据状态

相比之下 Punctuated Watermark 更为稳定，无论在正常情况（见图2(a)）还是在重放数据的情况（见图2(b)）下，下发的 Watermark 都是一致的，不过依然有 Event Time 倾斜的风险。对于这点，Flink 社区起草了 FLIP-27 来处理[4]。基本原理是 Source 节点会选择性地消费或阻塞某个 partition/shard，让总体的 Event Time 保持接近。

除了 Watermark 的下发有不确定之外，还有个问题是现在 Watermark 并没有被纳入 Checkpoint 快照中。这意味着在作业从 Checkpoint 恢复之后，Watermark 会重新开始算，导致 Watermark 的不确定。这个问题在 FLINK-5601[5] 有记录，但目前只体现了 Window 算子的 Watermark，而在 StateTTL 支持 Event Time 后，或许每个算子都要记录自己的 Watermark。

综上所述，Watermark 目前是很难做到非常确定的，但因为 Watermark 的不确定性是通过丢弃迟到数据导致计算结果的不确定性的，只要没有丢弃迟到数据，无论中间 Watermark 的变化如何，最终的结果都是相同的。

总结

确定性不足是阻碍实时计算在关键业务应用的主要因素，不过当前业界已经具备了解决问题的理论基础，剩下的更多是计算框架后续迭代和工程实践上的问题。就目前开发 Flink 实时应用而言，需要注意投递语义、函数副作用、Processing Time 和 Watermark 这几点造成的不确定性。

参考：

Flux capacitor, huh? Temporal Tables and Joins in Streaming SQL
https://flink.apache.org/2019/05/14/temporal-tables.html
FLINK-12005 Event time support
https://issues.apache.org/jira/browse/FLINK-12005
Fault Tolerance Guarantees of Data Sources and Sinks
https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/connectors/guarantees.html
FLIP-27: Refactor Source Interface
https://cwiki.apache.org/confluence/display/FLINK/FLIP-27%3A+Refactor+Source+Interface
[FLINK-5601] Window operator does not checkpoint watermarks
https://issues.apache.org/jira/browse/FLINK-5601