一文搞懂 Flink 的 Exactly Once 和 At Least Once

本文主要为了让你搞懂 Flink 的 Exactly Once 和 At Least Once，看完本文，你能 get 到以下知识：

- 介绍 CheckPoint 如何保障 Flink 任务的高可用
- CheckPoint 中的状态简介
- 如何实现全域一致的分布式快照?
- 什么是 barrier？什么是 barrier 对齐?
- 证明了：为什么 barrier 对齐就是 Exactly Once，为什么 barrier 不对齐就是 At Least Once。

Flink 简介

有状态函数和运算符在各个元素/事件的处理中存储数据（状态数据可以修改和查询，可以自己维护，根据自己的业务场景，保存历史数据或者中间结果到状态中）

例如：

- 当应用程序搜索某些事件模式时，状态将存储到目前为止遇到的事件序列。
- 在每分钟/小时/天聚合事件时，状态保存待处理的聚合。
- 当在数据点流上训练机器学习模型时，状态保持模型参数的当前版本。
- 当需要管理历史数据时，状态允许有效访问过去发生的事件。

什么是状态?

无状态计算的例子：

- 比如：我们只是进行一个字符串拼接，输入 a，输出 a_666,输入b，输出 b_666输出的结果跟之前的状态没关系，符合幂等性。

- 幂等性：就是用户对于同一操作发起的一次请求或者多次请求的结果是一致的，不会因为多次点击而产生了副作用

有状态计算的例子：

- 计算 pv、uv。
- 输出的结果跟之前的状态有关系，不符合幂等性，访问多次，pv 会增加。

Flink 的 CheckPoint 功能简介

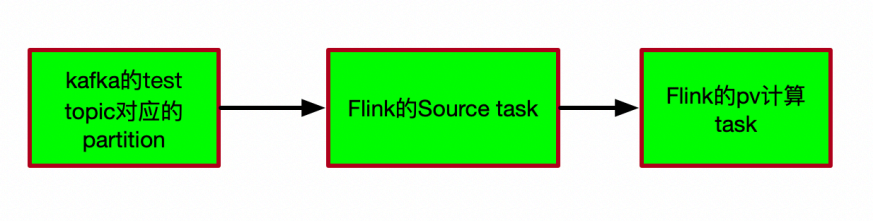
1.Flink CheckPoint 的存在就是为了解决 Flink 任务 failover 掉之后，能够正常恢复任务。那 CheckPoint 具体做了哪些功能，为什么任务挂掉之后，通过 CheckPoint 能使得任务恢复呢?

2.CheckPoint 是通过给程序快照的方式使得将历史某些时刻的状态保存下来，当任务挂掉之后，默认从最近一次保存的完整快照处进行恢复任务。问题来了，快照是什么鬼？能吃吗?

3.SnapShot 翻译为快照，指将程序中某些信息存一份，后期可以用来恢复。对于一个 Flink 任务来讲，快照里面到底保存着什么信息呢?

4.晦涩难懂的概念怎么办？当然用案例来代替咯，用案例让大家理解快照里面到底存什么信息。选一个大家都比较清楚的指标，app 的 pv，Flink 该怎么统计呢?

我们从 Kafka 读取到一条条的日志，从日志中解析出 app_id，然后将统计的结果放到内存中一个 Map 集合，app_id 作为 key，对应的 pv 做为 value，每次只需要将相应 app_id 的 pv 值 +1 后 put 到 Map 中即可。



Flink 任务 task 图

5.Flink 的 Source task 记录了当前消费到 kafka test topic 的所有 partition 的 offset，为了方便理解 CheckPoint 的作用，这里先用一个 partition 进行讲解，假设名为 “test”的 topic 只有一个 partition0。

- 例：（0，1000）

  表示 0 号 partition 目前消费到 offset 为 1000 的数据

6.Flink 的 pv task 记录了当前计算的各 app 的 pv 值，为了方便讲解，我这里有两个 app：app1、app2

- 例：（app1，50000）（app2，10000）
  - 表示 app1 当前 pv 值为 50000
  - 表示 app2 当前 pv 值为 10000
- 每来一条数据，只需要确定相应 app_id，将相应的 value 值 +1 后 put 到 map 中即可。

7.该案例中，CheckPoint 到底记录了什么信息呢?

- offset：（0，1000）
- pv：（app1，50000）（app2，10000）
- 记录的其实就是第 n 次 CheckPoint 消费的 offset 信息和各 app 的 pv 值信息，记录一下发生 CheckPoint 当前的状态信息，并将该状态信息保存到相应的状态后端。（注：状态后端是保存状态的地方，决定状态如何保存，如何保障状态高可用，我们只需要知道，我们能从状态后端拿到 offset 信息和 pv 信息即可。状态后端必须是高可用的，否则我们的状态后端经常出现故障，会导致无法通过 checkpoint 来恢复我们的应用程序）
- chk-100
- 该状态信息表示第 100 次 CheckPoint 的时候， partition 0 offset 消费到了 1000，pv 统计结果为（app1，50000）（app2，10000）。

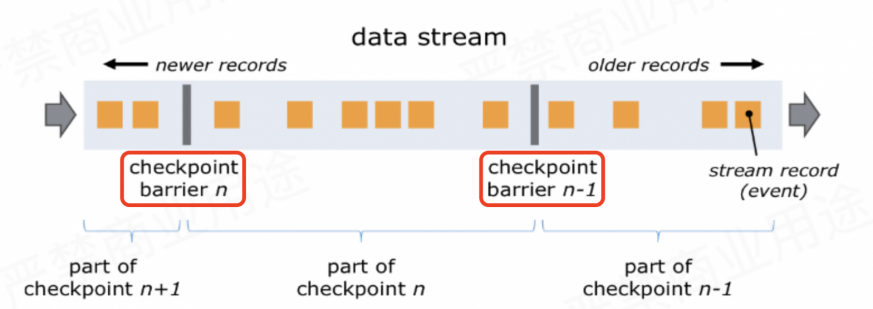
8.任务挂了，如何恢复?

- 假如我们设置了三分钟进行一次 CheckPoint，保存了上述所说的 chk-100 的 CheckPoint 状态后，过了十秒钟，offset 已经消费到（0，1100），pv 统计结果变成了（app1，50080）（app2，10020），但是突然任务挂了，怎么办?
- 莫慌，其实很简单，flink只需要从最近一次成功的 CheckPoint 保存的offset（0，1000）处接着消费即可，当然pv值也要按照状态里的 pv 值（app1，50000）（app2，10000）进行累加，不能从（app1，50080）（app2，10020）处进行累加，因为 partition 0 offset 消费到 1000 时，pv 统计结果为（app1，50000）（app2，10000）。

当然如果你想从 offset （0，1100）pv（app1，50080）（app2，10020）这个状态恢复，也是做不到的，因为那个时刻程序突然挂了，这个状态根本没有保存下来。我们能做的最高效方式就是从最近一次成功的 CheckPoint 处恢复，也就是我一直所说的 chk-100。

- 以上讲解，基本就是 CheckPoint 承担的工作，描述的场景比较简单。

9.疑问，计算 pv 的 task 在一直运行，它怎么知道什么时候去做这个快照？或者说计算 pv 的 task 怎么保障它自己计算的 pv 值（app1，50000）（app2，10000）就是 offset（0，1000）那一刻的统计结果呢？



barrier

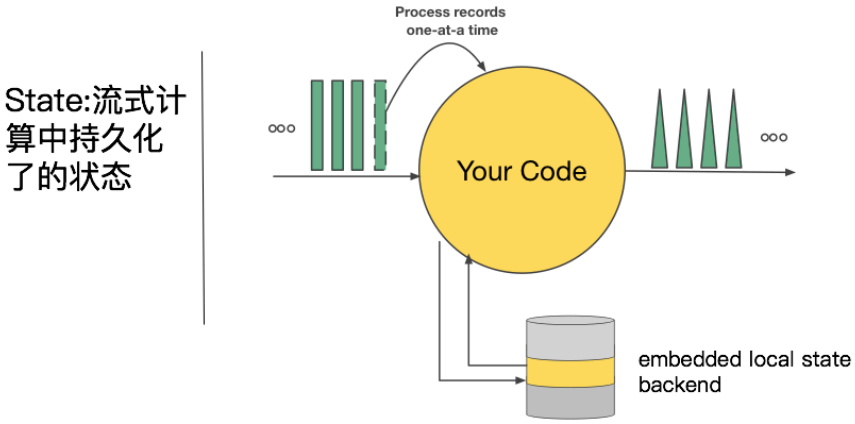
- barrier 从 Source Task 处生成，一直流到 Sink Task，期间所有的 Task 只要碰到barrier，就会触发自身进行快照。
    - CheckPoint barrier n-1 处做的快照就是指 Job 从开始处理到 barrier n-1所有的状态数据。
    - barrier n 处做的快照就是指从 Job 开始到处理到 barrier n 所有的状态数据。
- 对应到 pv 案例中就是，SourceTask 接收到 JobManager 的编号为 chk-100 的 CheckPoint 触发请求后，发现自己恰好接收到 kafka offset（0，1000）处的数据，所以会往 offset（0，1000）数据之后 offset（0，1001）数据之前安插一个 barrier，然后自己开始做快照，也就是将 offset（0，1000）保存到状态后端 chk-100 中。然后 barrier 接着往下游发送，当统计 pv 的 task 接收到 barrier 后，也会暂停处理数据，将自己内存中保存的 pv 信息（app1，50000）。（app2，10000）保存到状态后端 chk-100 中。OK，Flink 大概就是通过这个原理来保存快照的。
    - 统计 pv 的 task 接收到 barrier，就意味着 barrier 之前的数据都处理了，所以说，不会出现丢数据的情况。
- barrier 的作用就是为了把数据区分开，CheckPoint 过程中有一个同步做快照的环节不能处理 barrier 之后的数据，为什么呢？
    - 如果做快照的同时，也在处理数据，那么处理的数据可能会修改快照内容，所以先暂停处理数据，把内存中快照保存好后，再处理数据。

- 结合案例来讲就是，统计 pv 的 task 想对（app1，50000）（app2，10000）做快照，但是如果数据还在处理，可能快照还没保存下来，状态已经变成了（app1，50001）（app2，10001），快照就不准确了，就不能保障 Exactly Once 了。

- Flink 是在数据中加了一个叫做 barrier 的东西（barrier 中文翻译：栅栏），上图中红圈处就是两个 barrier。

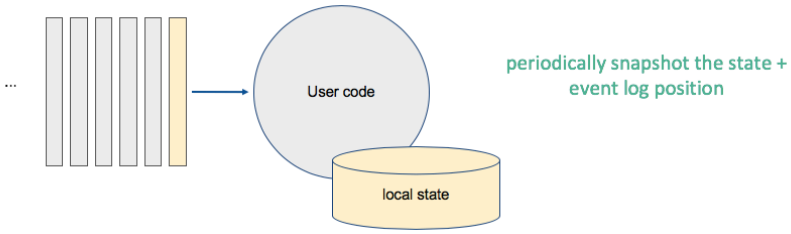
10.总结

- 流式计算中状态交互

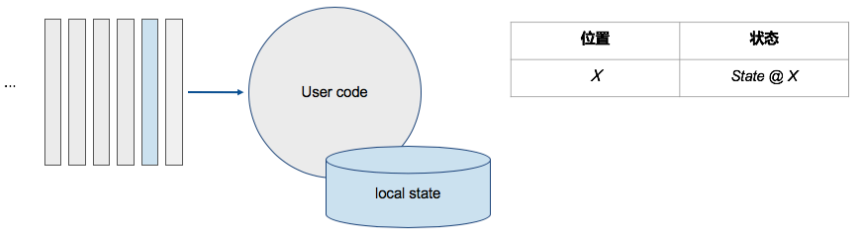


流式计算中状态交互

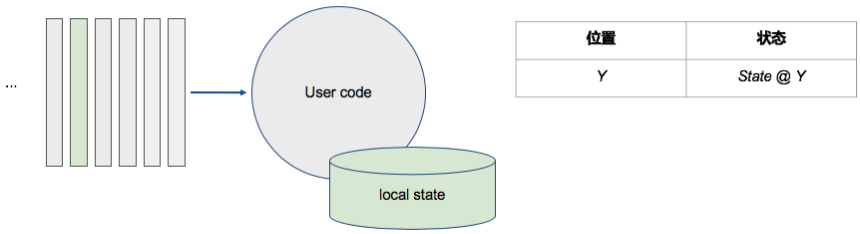
11.简易场景精确一次的容错方法



checkpoint 简介 1



checkpoint 简介 2



checkpoint 简介 3

- 消费到 Y 位置的时候，将 Y 对应的状态保存下来
- 消费到 X 位置的时候，将 X 对应的状态保存下来
- 周期性地对消费 offset 和统计的状态信息或统计结果进行快照

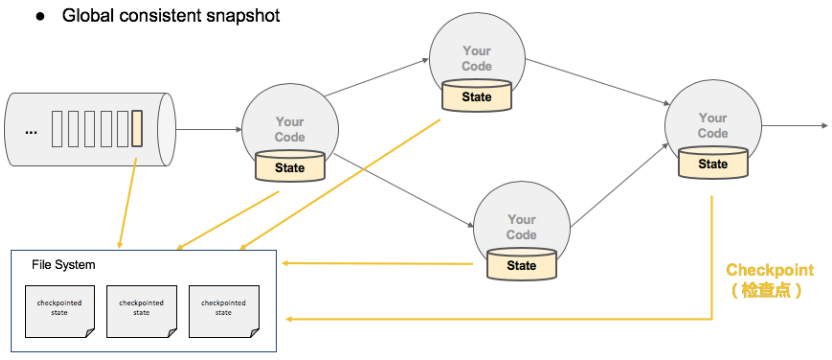
多并行度、多 Operator 情况下，CheckPoint 过程

1.分布式状态容错面临的问题与挑战

- 如何确保状态拥有精确一次的容错保证?
- 如何在分布式场景下替多个拥有本地状态的算子产生一个全域一致的快照?
- 如何在不中断运算的前提下产生快照?

2.多并行度、多 Operator 实例的情况下，如何做全域一致的快照?

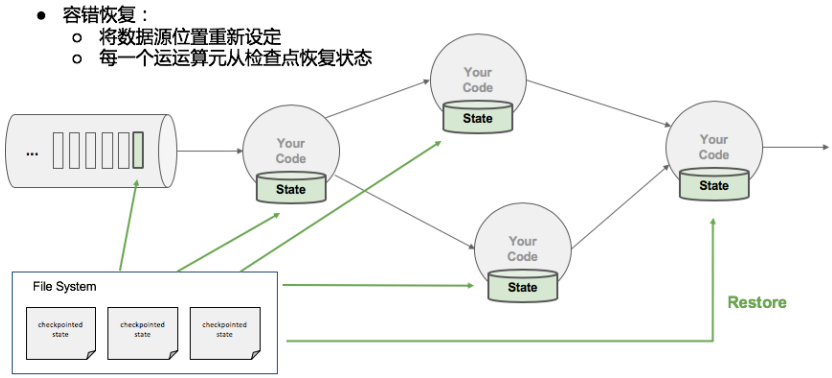
所有的 Operator 运行过程中遇到 barrier 后，都对自身的状态进行一次快照，保存到相应状态后端。



多并行度CheckPoint快照简图

对应到 pv 案例：有的 Operator 计算的 app1 的 pv，有的 Operator 计算的 app2的 pv，当他们碰到 barrier 时，都需要将目前统计的 pv 信息快照到状态后端。

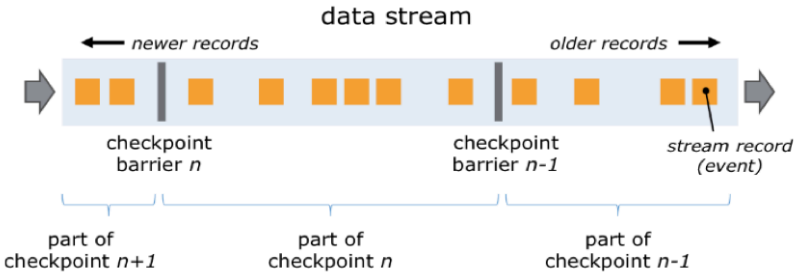
3.多 Operator 状态恢复



多并行度CheckPoint恢复

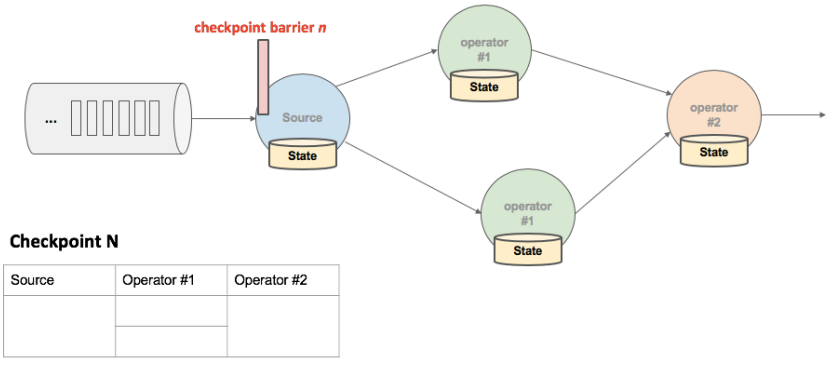
4.具体怎么做这个快照呢?

利用之前所有的 barrier 策略。



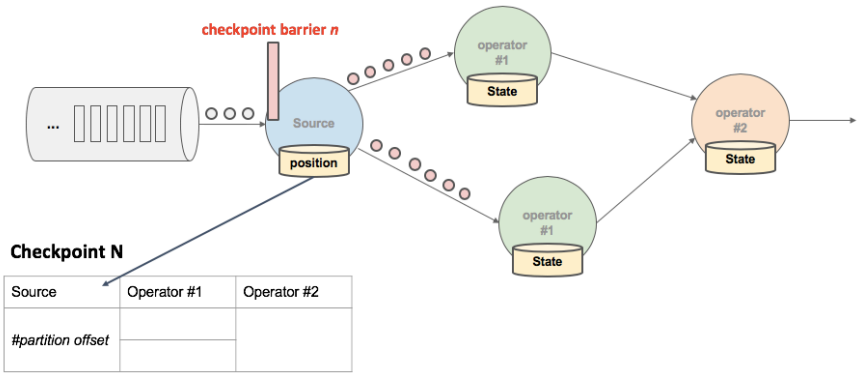
barrier

JobManager 向 SourceTask 发送 CheckPointTrigger，SourceTask 会在数据流中安插 CheckPoint barrier。



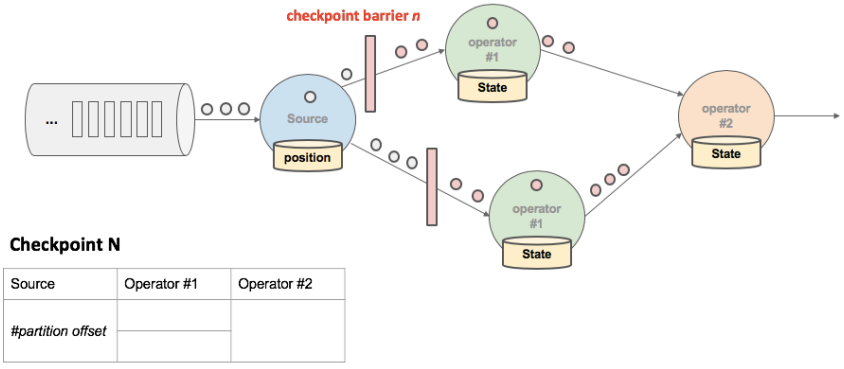
多并行度快照详图 0

Source Task 自身做快照，并保存到状态后端。



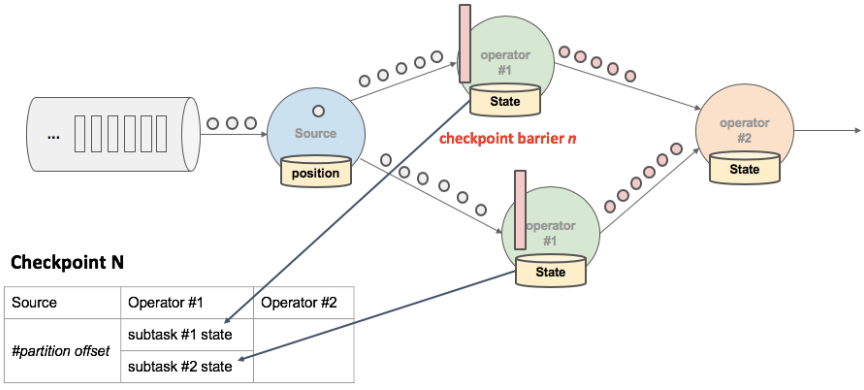
多并行度快照详图 1

Source Task 将 barrier 跟数据流一块往下游发送。

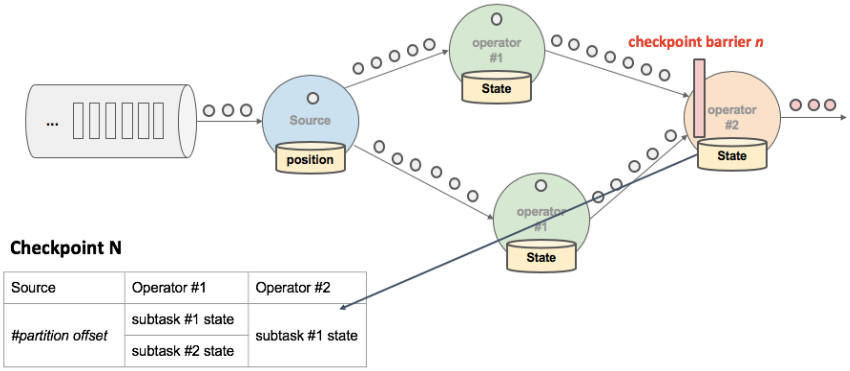


多并行度快照详图 2

当下游的 Operator 实例接收到 CheckPointbarrier 后，对自身做快照。



多并行度快照详图 3



多并行度快照详图 4

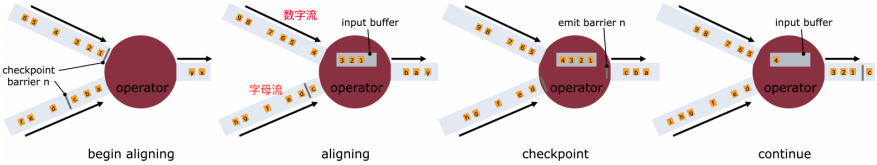
上述图中，有 4 个带状态的 Operator 实例，相应的状态后端就可以想象成填 4 个格子。整个 CheckPoint 的过程可以当做 Operator 实例填自己格子的过程，Operator 实例将自身的状态写到状态后端中相应的格子，当所有的格子填满可以简单的认为一次完整的 CheckPoint 做完了。

5.上面只是快照的过程，整个 CheckPoint 执行过程如下

- JobManager 端的 CheckPointCoordinator 向所有 SourceTask 发送 CheckPointTrigger，Source Task 会在数据流中安插 CheckPoint barrier。
- 当 task 收到所有的 barrier 后，向自己的下游继续传递 barrier，然后自身执行快照，并将自己的状态异步写入到持久化存储中。
    - 增量 CheckPoint 只是把最新的一部分更新写入到 外部存储;
    - 为了下游尽快做 CheckPoint，所以会先发送 barrier 到下游，自身再同步进行快照;
- 当 task 完成备份后，会将备份数据的地址（state handle）通知给 JobManager 的 CheckPointCoordinator。
    - 如果 CheckPoint 的持续时长超过了 CheckPoint 设定的超时时间，CheckPointCoordinator 还没有收集完所有的 State Handle，CheckPointCoordinator 就会认为本次 CheckPoint 失败，会把这次 CheckPoint 产生的所有状态数据全部删除。
- 最后 CheckPointCoordinator 会把整个 StateHandle 封装成 completed CheckPoint Meta，写入到 hdfs。

6.barrier 对齐

- 什么是 barrier 对齐?



stream_aligning

1. 一旦 Operator 从输入流接收到 CheckPointbarrier n，它就不能处理来自该流的任何数据记录，直到它从其他所有输入接收到 barrier n 为止。否则，它会混合属于快照 n 的记录和属于快照 n + 1 的记录。

2. 接收到 barrier n 的流暂时被搁置。从这些流接收的记录不会被处理，而是放入输入缓冲区。

   上图中第 2 个图，虽然数字流对应的 barrier 已经到达了，但是 barrier 之后的 1、2、3 这些数据只能放到 buffer 中，等待字母流的 barrier 到达。

3. 一旦最后所有输入流都接收到 barrier n，Operator 就会把缓冲区中 pending 的输出数据发出去，然后把 CheckPoint barrier n 接着往下游发送。

   这里还会对自身进行快照。

4. 之后，Operator 将继续处理来自所有输入流的记录，在处理来自流的记录之前先处理来自输入缓冲区的记录。

- 什么是 barrier 不对齐?

1. 上述图 2 中，当还有其他输入流的 barrier 还没有到达时，会把已到达的 barrier 之后的数据 1、2、3 搁置在缓冲区，等待其他流的 barrier 到达后才能处理。

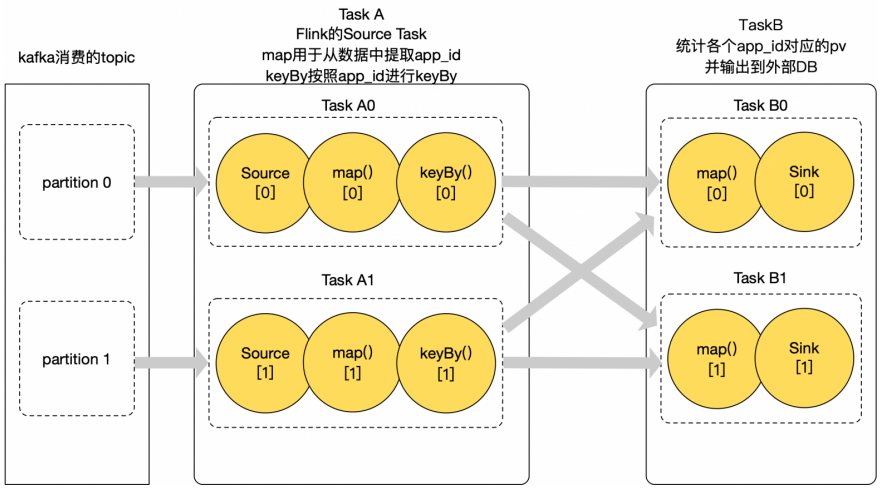
2. barrier 不对齐就是指当还有其他流的 barrier 还没到达时，为了不影响性能，也不用理会，直接处理 barrier 之后的数据。等到所有流的 barrier 的都到达后，就可以对该 Operator 做 CheckPoint 了

- 为什么要进行 barrier 对齐？不对齐到底行不行?

1. Exactly Once 时必须 barrier 对齐，如果 barrier 不对齐就变成了 At Least Once。后面的部分主要证明这句话。

2. CheckPoint 的目的就是为了保存快照，如果不对齐，那么在 chk-100 快照之前，已经处理了一些 chk-100 对应的 offset 之后的数据，当程序从 chk-100 恢复任务时，chk-100 对应的 offset 之后的数据还会被处理一次，所以就出现了重复消费。如果听不懂没关系，后面有案例让您懂。

- 结合 pv 案例来看，之前的案例为了简单，描述的 kafka 的 topic 只有 1 个 partition，这里为了讲述 barrier 对齐，所以 topic 有 2 个 partittion。



flink 消费 kafka，计算 pv 详图

1. Flink 同样会起四个 Operator 实例，我还称他们是 TaskA0、TaskA1、TaskB0、TaskB1。四个 Operator 会从状态后端读取保存的状态信息。

2. 从 offset：(0，10000)(1，10005) 开始消费，并且基于 pv：(app0，8000)(app1，12050)值进行累加统计。

3. 然后你就应该会发现这个 app1 的 pv 值 12050 实际上已经包含了 partition1 的 offset 10005~10200 的数据，所以 partition1 从 offset 10005 恢复任务时，partition1 的 offset 10005~10200 的数据被消费了两次。

4. TaskB1 设置的 barrier 不对齐，所以 CheckPoint chk-100 对应的状态中多消费了 barrier 之后的一些数据（TaskA1 发送），重启后是从 chk-100 保存的 offset 恢复，这就是所说的 At Least Once。

5. 由于上面说 TaskB0 设置的 barrier 对齐，所以 app0 不会出现重复消费，因为 app0 没有消费 offset：(0，10000)(1，10005) 之后的数据，也就是所谓的 Exactly Once。

    - chk-100
    - offset：(0，10000)(1，10005)
    - pv：(app0，8000) (app1，12050)

6. 虽然状态保存的 pv 值偏高了，但是不能说明重复处理，因为我的 TaskA1 并没有再次去消费 partition1 的 offset 10005~10200 的数据，所以相当于也没有重复消费，只是展示的结果更实时了。

7. 这里假如 TaskA0 消费的 partition0 的 offset 为 10000，TaskA1 消费的 partition1 的 offset 为 10005。那么状态中会保存 (0，10000)(1，

10005)，表示 0 号 partition 消费到了 offset 为 10000 的位置，1 号 partition 消费到了 offset 为 10005 的位置。

8. 结合业务，先介绍一下上述所有算子在业务中的功能：

    - Source 的 kafka 的 Consumer，从 kakfa 中读取数据到 Flink 应用中
    - TaskA 中的 map 将读取到的一条 kafka 日志转换为我们需要统计的 app_id
    - keyBy 按照 app_id 进行 keyBy，相同的 app_id 会分到下游 TaskB的同一个实例中
    - TaskB 的 map 在状态中查出该 app_id 对应的 pv 值，然后 +1，存储到状态中
    - 利用 Sink 将统计的 pv 值写入到外部存储介质中

9. 我们从 kafka 的两个 partition 消费数据，TaskA 和 TaskB 都有两个并行度，所以总共 Flink 有 4 个 Operator 实例，这里我们称之为 TaskA0、TaskA1、TaskB0、TaskB1。

10. 假设已经成功做了 99 次 CheckPoint，这里详细解释第 100 次 CheckPoint 过程。

    - JobManager 内部有个定时调度，假如现在 10 点 00 分 00 秒到了第 100 次 CheckPoint 的时间了，JobManager 的 CheckPointCoordinator 进程会向所有的 Source Task 发送 CheckPointTrigger，也就是向 TaskA0、TaskA1 发送 CheckPointTrigger。
    - TaskA0、TaskA1 接收到 CheckPointTrigger，会往数据流中安插 barrier，将 barrier 发送到下游，在自己的状态中记录 barrier 安插的 offset 位置，然后自身做快照，将 offset 信息保存到状态后端。

      然后 TaskA 的 map 和 keyBy 算子中并没有状态，所以不需要进行快照。
    - 接着数据和 barrier 都向下游 TaskB 发送，相同的 app_id 会发送到相同的TaskB实例上，这里假设有两个 app：app0 和 app1，经过 keyBy 后，假设 app0 分到了 TaskB0 上，app1 分到了 TaskB1 上。基于上面描述，TaskA0 和 TaskA1 中的所有 app0 的数据都发送到 TaskB0 上，所有 app1 的数据都发送到 TaskB1 上。
    - 现在我们假设 TaskB0 做 CheckPoint 的时候 barrier 对齐了，TaskB1 做 CheckPoint 的时候 barrier 不对齐，当然不能这么配置，我就是举这么个例子，带大家分析一下 barrier 对不对齐到底对统计结果有什么影响?
    - 上面说了 chk-100 的这次 CheckPoint，offset 位置为(0，10000)(1，10005)，TaskB0 使用 barrier 对齐，也就是说 TaskB0 不会处理 barrier 之后的数据，所以TaskB0 在 chk-100 快照的时候，状态后端保存的 app0 的 pv 数据是从程序开始启

动到 kafkaoffset 位置为(0，10000)(1，10005)的所有数据计算出来的 pv 值，一条不多（没处理 barrier 之后，所以不会重复），一条不少(barrier 之前的所有数据都处理了，所以不会丢失)，假如保存的状态信息为(app0，8000)表示消费到(0，10000)(1，10005)offset 的时候，app0 的 pv 值为 8000。

- TaskB1 使用的 barrier 不对齐，假如 TaskA0 由于服务器的 CPU 或者网络等其他波动，导致 TaskA0 处理数据较慢，而 TaskA1 很稳定，所以处理数据比较快。导致的结果就是 TaskB1 先接收到了 TaskA1 的 barrier，由于配置的 barrier 不对齐，所以 TaskB1 会接着处理 TaskA1 barrier 之后的数据，过了 2 秒后，TaskB1 接收到了 TaskA0 的 barrier，于是对状态中存储的 app1 的 pv 值开始做 CheckPoint 快照，保存的状态信息为(app1，12050)，但是我们知道这个(app1，12050)实际上多处理了 2 秒 TaskA1 发来的 barrier 之后的数据，也就是 kafka topic 对应的 partition1 offset 10005 之后的数据，app1 真实的 pv 数据肯定要小于这个 12050，partition1 的 offset 保存的 offset 虽然是 10005，但是我们实际上可能已经处理到了 offset 10200 的数据，假设就是处理到了 10200。

11. 分析到这里，我们先梳理一下我们的状态保存了什么：

    - chk-100
    - offset：(0，10000)(1，10005)
    - pv：(app0，8000) (app1，12050)

12. 接着程序在继续运行，过了 10 秒，由于某个服务器挂了，导致我们的四个 Operator 实例有一个 Operator 挂了，所以 Flink 会从最近一次的状态恢复，也就是我们刚刚详细讲的 chk-100 处恢复，那具体是怎么恢复的呢?

    - Flink 同样会起四个 Operator 实例，我还称他们是 TaskA0、TaskA1、TaskB0、TaskB1。四个 Operator 会从状态后端读取保存的状态信息。
    - 从 offset：(0，10000)(1，10005) 开始消费，并且基于 pv：(app0，8000) (app1，12050)值进行累加统计
    - 然后你就应该会发现这个 app1 的 pv 值 12050 实际上已经包含了 partition1 的 offset 10005~10200 的数据，所以 partition1 从 offset 10005 恢复任务时，partition1 的 offset 10005~10200 的数据被消费了两次。
    - TaskB1 设置的 barrier 不对齐，所以 CheckPoint chk-100 对应的状态中多消费了 barrier 之后的一些数据（TaskA1 发送），重启后是从 chk-100 保存的 offset 恢复，这就是所说的 At Least Once。
    - 由于上面说 TaskB0 设置的 barrier 对齐，所以 app0 不会出现重复消费，因为 app0 没有消费 offset：(0，10000)(1，10005)之后的数据，也就是所谓的 Exactly Once。

看到这里你应该已经知道了哪种情况会出现重复消费了，也应该要掌握为什么 barrier 对齐就是 Exactly Once，为什么 barrier 不对齐就是 At Least Once。

这里再补充一个问题，到底什么时候会出现 barrier 对齐?

- 首先设置了 Flink 的 CheckPoint 语义是：Exactly Once。
- Operator 实例必须有多个输入流才会出现 barrier 对齐。
  - 对齐，汉语词汇，释义为使两个以上事物配合或接触得整齐。由汉语解释可得对齐肯定需要两个以上事物，所以，必须有多个流才叫对齐。barrier 对齐其实也就是上游多个流配合使得数据对齐的过程。
  - 言外之意：如果 Operator 实例只有一个输入流，就根本不存在 barrier 对齐，自己跟自己默认永远都是对齐的。

Q & A 环节

第一种场景计算 PV，kafka 只有一个 partition，精确一次，至少一次就没有区别?
答：如果只有一个 partition，对应 Flink 任务的 Source Task 并行度只能是 1，确实没有区别，不会有至少一次的存在了，肯定是精确一次。因为只有 barrier 不对齐才会有可能重复处理，这里并行度都已经为 1，默认就是对齐的，只有当上游有多个并行度的时候，多个并行度发到下游的 barrier 才需要对齐，单并行度不会出现 barrier 不对齐，所以必然精确一次。其实还是要理解 barrier 对齐就是 Exactly Once 不会重复消费，barrier 不对齐就是 At Least Once 可能重复消费，这里只有单个并行度根本不会存在 barrier 不对齐，所以不会存在至少一次语义。
为了下游尽快做 CheckPoint，所以会先发送 barrier 到下游，自身再同步进行快照；这一步，如果向下发送 barrier 后，自己同步快照慢怎么办？下游已经同步好了，自己还没?
答: 可能会出现下游比上游快照还早的情况，但是这不影响快照结果，只是下游快照的更及时了，我只要保障下游把 barrier 之前的数据都处理了，并且不处理 barrier 之后的数据，然后做快照，那么下游也同样支持精确一次。这个问题你不要从全局思考，你单独思考上游和下游的实例，你会发现上下游的状态都是准确的，既没有丢，也没有重复计算。
这里需要注意一点，如果有一个 Operator 的 CheckPoint 失败了或者因为 CheckPoint 超时也会导致失败，那么 JobManager 会认为整个 CheckPoint 失败。失败的 CheckPoint 是不能用来恢复任务的，必须所有的算子的 CheckPoint 都成功，那么这次 CheckPoint 才能认为是成功的，才能用来恢复任务。

我程序中 Flink 的 CheckPoint 语义设置了 Exactly Once，但是我的 MySQL 中看到数据重复了？程序中设置了 1 分钟 1 次 CheckPoint，但是 5 秒向 MySQL 写一次数据，并 commit。

答：Flink 要求 end to end 的精确一次都必须实现 TwoPhaseCommitSinkFunction。如果你的 chk-100 成功了，过了 30 秒，由于 5 秒 commit 一次，所以实际上已经写入了 6 批数据进入 MySQL，但是突然程序挂了，从 chk100 处恢复，这样的话，之前提交的 6 批数据就会重复写入，所以出现了重复消费。Flink 的精确一次有两种情况，一个是 Flink 内部的精确一次，一个是端对端的精确一次，这个博客所描述的都是关于 Flink 内部去的精确一次，我后期再发一个博客详细介绍一下 Flink 端对端的精确一次如何实现。这篇文章有这么一句话 TwoPhaseCommitSinkFunction 已经把这种情况考虑在内了，并且在从 checkpoint 点恢复状态时，会优先发出一个 commit。个人感觉只要把这句话理解了，知道为什么每次恢复状态时，都需要优先发出一个 commit，那就把 Flink 的 TwoPhaseCommitSinkFunction 真正理解了。

参考内容：

1.Apache Flink 官网

- An Overview of End-to-End Exactly-OnceProcessing in Apache Flink (with Apache Kafka, too!)
- Managing Large State in Apache Flink: AnIntro to Incremental Checkpointing
- State & Fault Tolerance
- Checkpoints
- Savepoints
- State Backends
- Tuning Checkpoints and Large State
- Data Streaming Fault Tolerance

2.Flink China 社区官网系列课程

- 1.2 Flink基本概念
- 1.7 状态管理与容错机制
- 2.3 Flink Checkpoint-轻量级分布式快照
- 2.11 Flink State 最佳实践

3.谈谈流计算中的『Exactly Once』特性
4.Apache Flink结合Kafka构建端到端的Exactly-Once处理

作者：范瑞
原文地址：https://www.jianshu.com/p/8d6569361999