什么是checkpoint

- Flink Checkpoint 是一种容错恢复机制
- 这种机制保证了实时程序运行时，即使突然遇到异常也能够进行自我恢复
- Checkpoint 对于用户层面，是透明的。是 Flink 自身的系统行为，用户无法对其进行交互
- state是Checkpoint 做持久化备份的主要数据

checkpoint两个必要条件

- 需要支持重放一定时间范围内数据的数据源，比如：kafka
- 需要一个存储来保存持久化的状态，如：Hdfs，本地文件。可以在任务失败后，从存储中恢复 checkpoint 数据

checkpoint参数详解

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

// 每 60s 做一次 checkpoint
env.enableCheckpointing(60000);

// 高级配置：

// checkpoint 语义设置为 EXACTLY_ONCE，这是默认语义
env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);

// 两次 checkpoint 的间隔时间至少为 1 s，默认是 0，立即进行下一次 checkpoint
env.getCheckpointConfig().setMinPauseBetweenCheckpoints(1000);

// checkpoint 必须在 60s 内结束，否则被丢弃，默认是 10 分钟
env.getCheckpointConfig().setCheckpointTimeout(60000);

// 同一时间只能允许有一个 checkpoint
env.getCheckpointConfig().setMaxConcurrentCheckpoints(1);

// 最多允许 checkpoint 失败 3 次
env.getCheckpointConfig().setTolerableCheckpointFailureNumber(3);

// 当 Flink 任务取消时，保留外部保存的 checkpoint 信息
env.getCheckpointConfig().enableExternalizedCheckpoints(ExternalizedCheckpointCleanup.RETAIN_ON_CA
NCELLATION);

// 当有较新的 Savepoint 时，作业也会从 Checkpoint 处恢复
env.getCheckpointConfig().setPreferCheckpointForRecovery(true);

// 允许实验性的功能：非对齐的 checkpoint，以提升性能
env.getCheckpointConfig().enableUnalignedCheckpoints();
```

相关参数的文字描述：

1. env.enableCheckpointing(60000)，1 分钟触发一次 checkpoint；
2. setCheckpointTimeout，checkpoint 超时时间，默认是 10 分钟超时，超过了超时时间就会被丢弃；
3. setCheckpointingMode，设置 checkpoint 语义，可以设置为 EXACTLY_ONCE，表示既不重复消费也不丢数据；AT_LEAST_ONCE，表示至少消费一次，可能会重复消费；
4. setMinPauseBetweenCheckpoints，两次 checkpoint 之间的间隔时间。假如设置每分钟进行一次 checkpoint，两次 checkpoint 间隔时间为 30s。假设某一次 checkpoint 耗时 40s，那么理论上20s 后就要进行一次 checkpoint，但是设置了两次 checkpoint 之间的间隔时间为 30s，所以是 30s 之后才会进行 checkpoint。另外，如果配置了该参数，那么同时进行的 checkpoint 数量只能为 1；
5. enableExternalizedCheckpoints，Flink 任务取消后，外部 checkpoint 信息是否被清理。

- DELETE_ON_CANCELLATION，任务取消后，所有的 checkpoint 都将会被清理。只有在任务失败后，才会被保留；
- RETAIN_ON_CANCELLATION，任务取消后，所有的 checkpoint 都将会被保留，需要手工清理。

1. setPreferCheckpointForRecovery，恢复任务时，是否从最近一个比较新的 savepoint 处恢复，默认是 false；
2. enableUnalignedCheckpoints，是否开启试验性的非对齐的 checkpoint，可以在反压情况下极大减少 checkpoint 的次数;

checkpoint实现机制

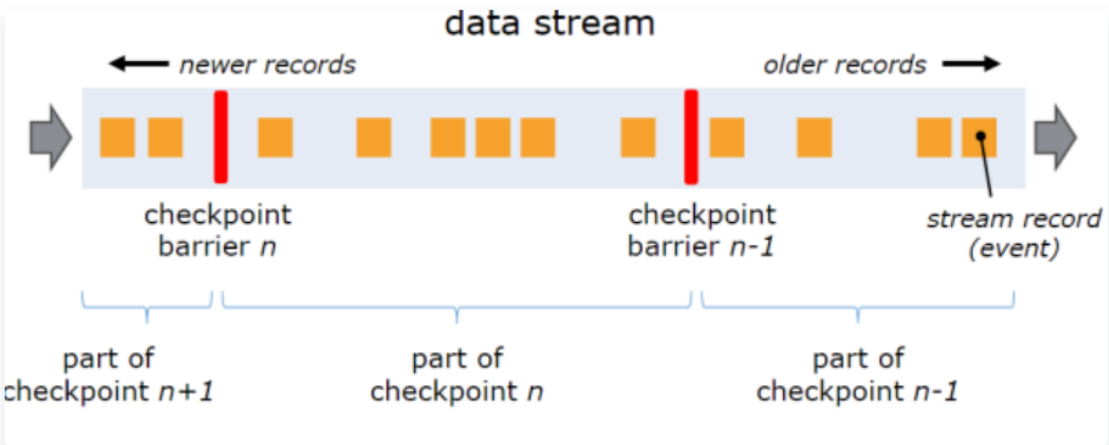
Flink 的 checkpoint 是基于 Chandy-Lamport 算法，实现了一个分布式一致性的存储快照算法。

这里我们假设一个简单的场景来描述 checkpoint 具体过程是怎样的。

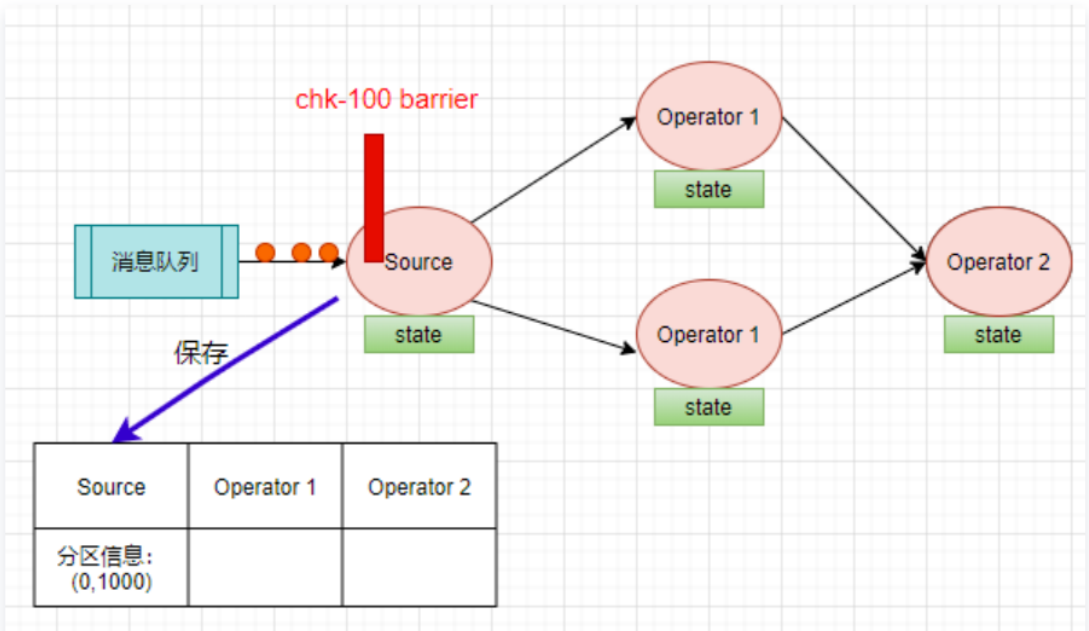
场景是：假如现在 kafka 只有一个分区，数据是每个 app 发过来的日志，我们统计每个 app 的 PV。

Flink 的 checkpoint coordinator （JobManager 的一部分）会周期性的在流事件中插入一个 barrier 事件（栅栏），用来隔离不同批次的事件，如下图红色的部分。

下图中有两个 barrier ， checkpoint barrier n-1 处的 barrier 是指 Job 从开始处理到 barrier n -1 所有的状态数据，checkpoint barrier n 处的 barrier 是指 Job 从开始处理到 barrier n 所有的状态数据。

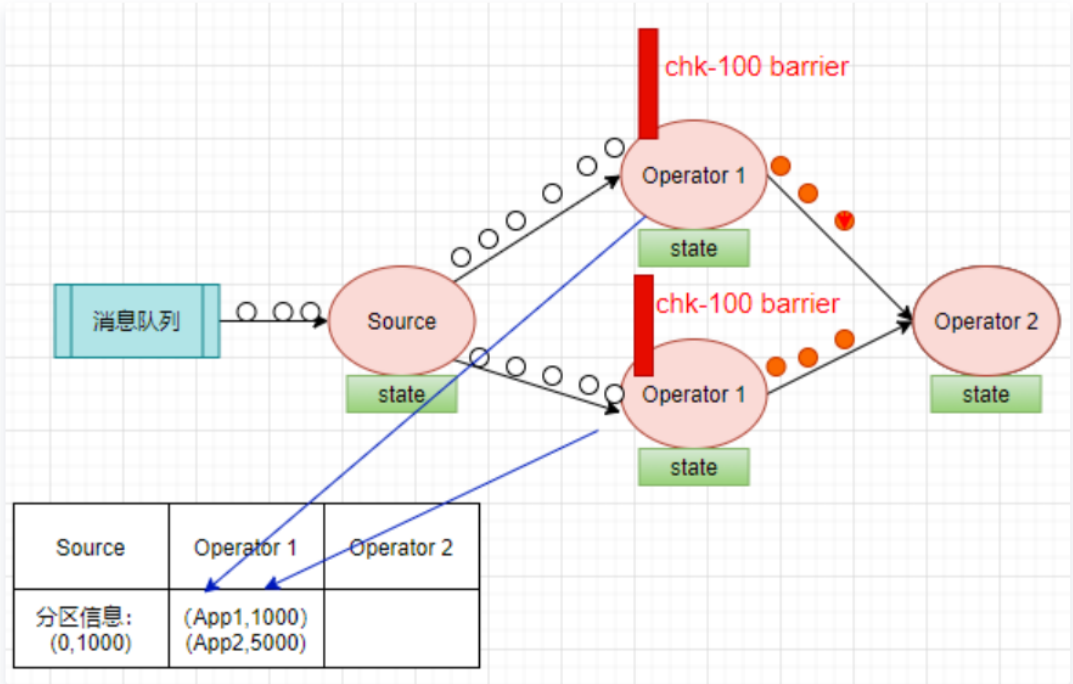


回到刚刚计算 PV 的场景，当 Source Task 接受到 JobManager 的编号为 chk-100 的 Checkpoint 触发请求后，发现自己恰好接收到了 offset 为（0，1000）【表示分区0，offset 为1000】处的数据，所以会往 offset 为（0,1000）数据之后，（0,1001）数据之前安插一个 barrier，然后自己开始做快照，把 offset （0,1000）保存到状态后端中，向 CheckpointCoordinator报告自己快照制作情况，同时向自身所有下游算子广播该barrier。如下图：

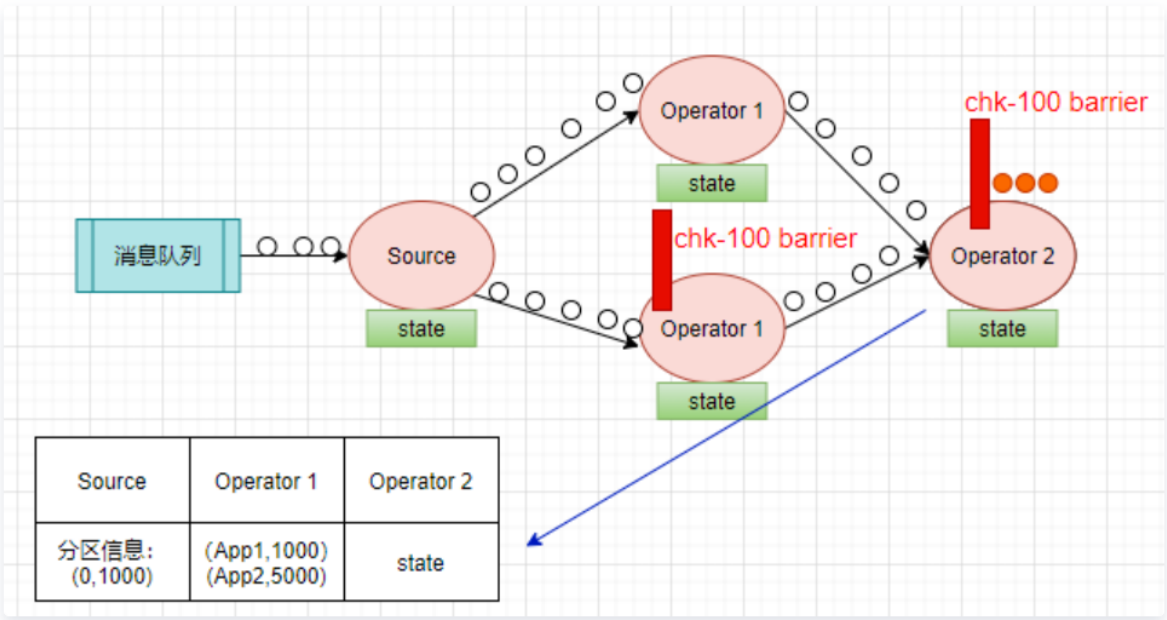


当下游计算的算子收到 barrier 后，会看是否收到了所有输入流的 barrier，我们现在只有一个分区，Source 算子只有一个实例，barrier 到了就是收到了所有的输入流的 barrier。

开始把本次的计算结果（app1,1000），（app2,5000）写到状态存储之中，向 CheckpointCoordinator 报告自己快照制作情况，同时向自身所有下游算子广播该barrier。

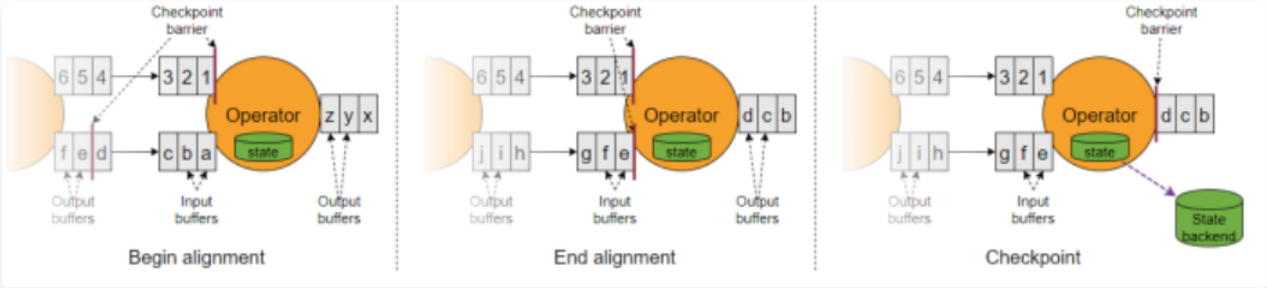


当 Operator 2 收到栅栏后，会触发自身进行快照，把自己当时的状态存储下来，向 CheckpointCoordinator 报告 自己快照制作情况。因为这是一个 sink ， 状态存储成功后，意味着本次 checkpoint 也成功了。



Barrier对齐

上面我们举的例子是 Source Task 实例只有一个的情况，在输入流的算子有多个实例的情况下，会有一个概念叫 Barrier 对齐。



可以看上面的第一张图，有两个输入流，一个是上面的数字流，一个是下面的字母流。

数字流的 barrier 在 1 后面，字母流的 barrier 在 e 后面。当上面的 barrier 到达 operator 之后，必须要等待下面的数字流的 barrier 也到达，此时数字流后面过来的数据会被缓存到缓冲区。这就是 barrier 对齐的过程。

看上面的第二张图，当数字流的 barrier 到达后，意味着输入流的所有实例的 barrier 都到达了，此时开始处理 到第三张图的时候，处理完毕，自身做快照，然后把缓冲区的 pending 数据都发出去，把 checkpoint barrier n 继续往下发送。

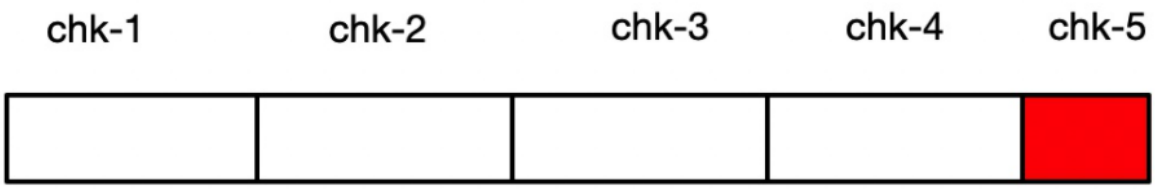
Checkpoint 语义

Flink Checkpoint 支持两种语义：**Exactly Once** 和 **At least Once**，默认的 Checkpoint 模式是 Exactly Once. Exactly Once 和 At least Once 具体是针对 Flink **状态** 而言。具体语义含义如下：

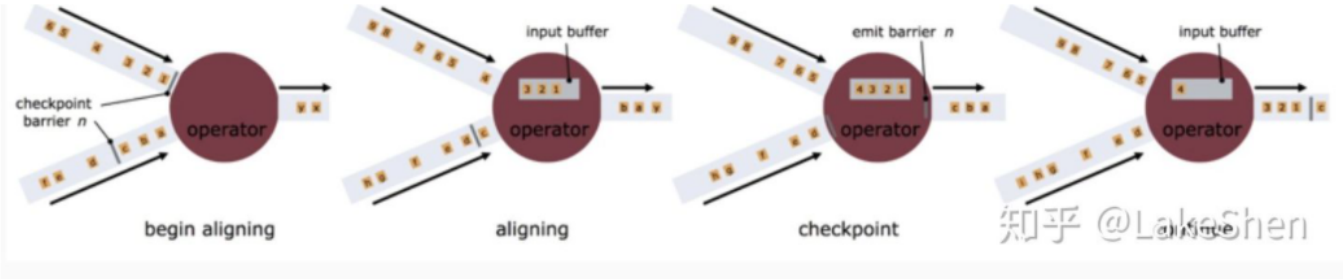
Exactly Once 含义是：保证每条数据对于 Flink 的状态结果只影响一次。打个比方，比如 WordCount程序，目前实时统计的 "hello" 这个单词数为5，同时这个结果在这次 Checkpoint 成功后，保存在了 HDFS。在下次 Checkpoint 之前， 又来2个 "hello" 单词，突然程序遇到外部异常容错自动回复，从最近的 Checkpoint 点开始恢复，那么会从单词数 5 这个状态开始恢复，Kafka 消费的数据点位还是状态 5 这个时候的点位开始计算，所以即使程序遇到外部异常自我恢复，也不会影响到 Flink 状态的结果。

At Least Once 含义是：每条数据对于 Flink 状态计算至少影响一次。比如在 WordCount 程序中，你统计到的某个单词的单词数可能会比真实的单词数要大，因为同一条消息，你可能将其计算多次。

Flink 中 Exactly Once 和 At Least Once 具体是针对 Flink 任务**状态**而言的，并不是 Flink 程序对其处理一次。举个例子，当前 Flink 任务正在做 Checkpoint，该次Checkpoint还么有完成，该次 Checkpoint 时间端的数据其实已经进入 Flink 程序处理，只是程序状态没有最终存储到远程存储。当程序突然遇到异常，进行容错恢复，那么就会从最新的 Checkpoint 进行状态恢复重启，上一部分还会进入 Flink 系统处理：



上图中表示，在进行 chk-5 Checkpoint 时，突然遇到程序异常，那么会从 chk-4 进行恢复，那么之前chk-5 处理的数据，会再次进行处理。
Exactly Once 和 At Least Once 具体在底层实现大致相同，具体差异表现在 Barrier 对齐方式处理：



如果是 Exactly Once 模式，某个算子的 Task 有多个输入通道时，当其中一个输入通道收到 Barrier 时，Flink Task 会阻塞处理该通道，其不会处理这些数据，但是会将这些数据存储到内部缓存中，一旦完成了所有输入通道的 Barrier 对齐，才会继续对这些数据进行消费处理。

对于 At least Once,同样针对某个算子的 Task 有多个输入通道的情况下，当某个输入通道接收到 Barrier 时，它不同于Exactly Once,At Least Once 会继续处理接受到的数据，即使没有完成所有输入通道 Barrier 对齐。所以使用At Least Once 不能保证数据对于状态计算只有一次影响。