> 保持自信和微笑

Flink之容错机制chekpoint

检查点（checkpoint）

Flink具体如何保证exactly-once呢? 它**使用一种被称为"检查点"（checkpoint）的特性**，在出现故障时将系统重置回正确状态。下面通过简单的类比来解释检查点的作用。

> 假设你和两位朋友正在数项链上有多少颗珠子，如下图所示。你捏住珠子，边数边拨，每拨过一颗珠子就给总数加一。你的朋友也这样数他们手中的珠子。当你分神忘记数到哪里时，怎么办呢? 如果项链上有很多珠子，你显然不想从头再数一遍，尤其是当三人的速度不一样却又试图合作的时候，更是如此(比如想记录前一分钟三人一共数了多少颗珠子，回想一下一分钟滚动窗口)。

于是，你想了一个更好的办法: **在项链上每隔一段就松松地系上一根有色皮筋，将珠子分隔开; 当珠子被拨动的时候，皮筋也可以被拨动;** 然后，你安排一个助手，让他在你和朋友拨到皮筋时记录总数。用这种方法，当有人数错时，就不必从头开始数。相反，你向其他人发出错误警示，然后你们都从上一根皮筋处开始重数，助手则会告诉每个人重数时的起始数值，例如在粉色皮筋处的数值是多少。

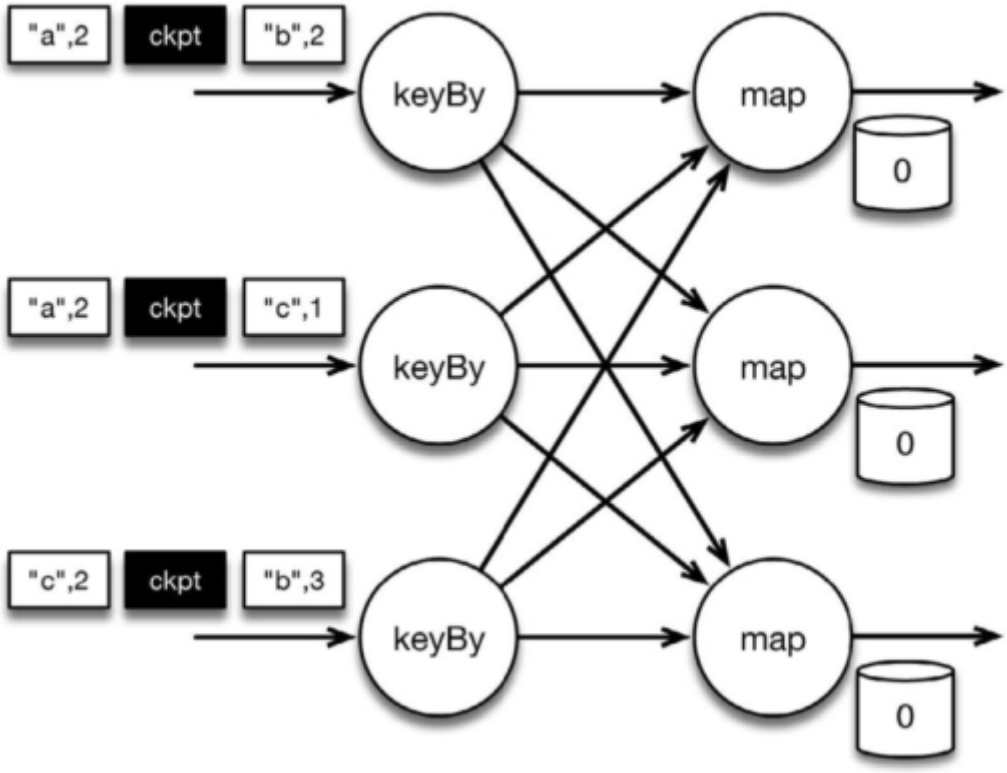
> **Flink检查点的作用就类似于皮筋标记。**数珠子这个类比的关键点是: 对于指定的皮筋而言，珠子的相对位置是确定的; 这让皮筋成为重新计数的参考点。**总状态(珠子的总数)在每颗珠子被拨动之后更新一次，助手则会保存与每根皮筋对应的检查点状态**，如当遇到粉色皮筋时一共数了多少珠子，当遇到橙色皮筋时又是多少。当问题出现时，这种方法使得重新计数变得简单。

检查点算法

> Flink检查点的核心作用是确保状态正确，即使遇到程序中断，也要正确。记住这一基本点之后，我们用一个例子来看检查点是如何运行的。**Flink为用户提供了用来定义状态的工具**。例如，以下这个Scala程序按照输入记录的第一个字段(一个字符串)进行分组并维护第二个字段的计数状态。

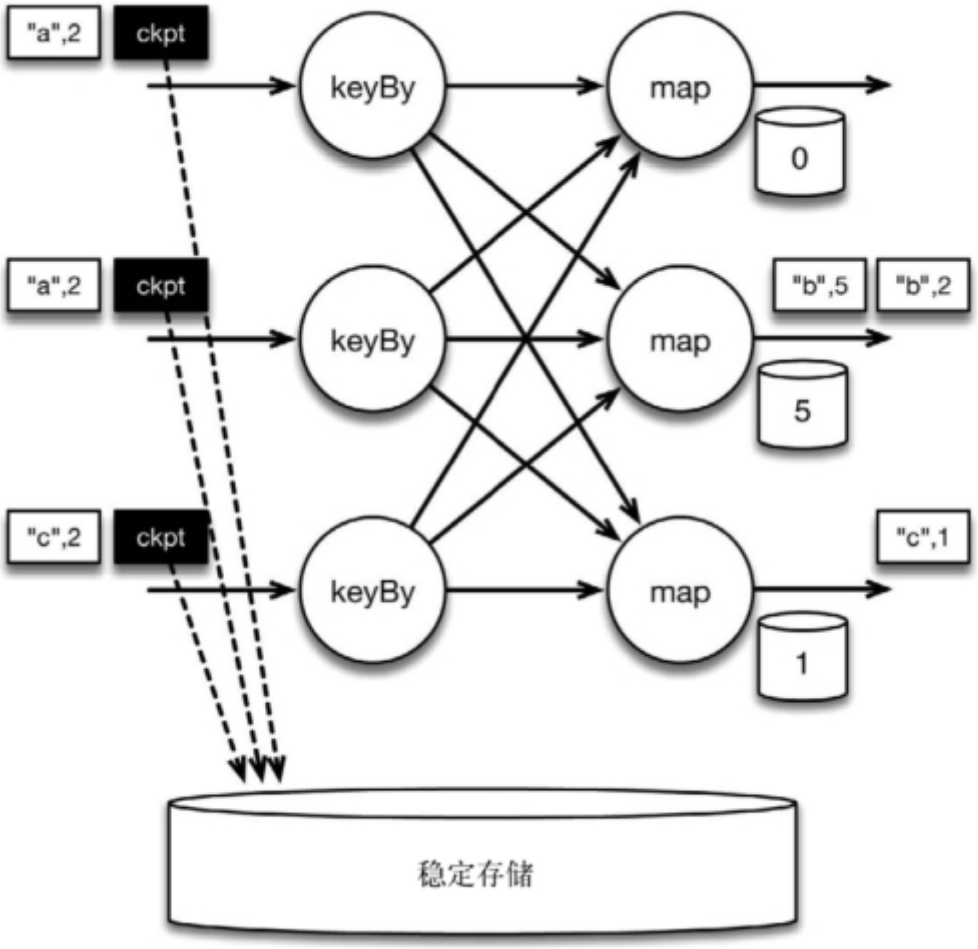
```
val stream: DataStream[(String, Int)] = ...
val counts: DataStream[(String, Int)] = stream
.keyBy(record => record._1)
.mapWithState(  (in: (String, Int), state: Option[Int])  =>
state match {
    case Some(c) => ( (in._1, c + in._2), Some(c + in._2) )
    case None => ( (in._1, in._2), Some(in._2) )
})
```

该程序有两个算子: **keyBy算子用来将记录按照第一个元素(一个字符串)进行分组，根据该key将数据进行重新分区，**然后将记录再发送给下一个算子: 有状态的map算子(mapWithState)。map算子在接收到每个元素后，将输入记录的第二个字段的数据加到现有总数中，再将更新过的元素发射出去。下图表示程序的初始状态:**输入流中的6条记录被检查点分割线(checkpoint barrier)隔开**，所有的map算子状态均为0(计数还未开始)。所有key为a的记录将被顶层的map算子处理，所有key为b的记录将被中间层的map算子处理，所有key为c的记录则将被底层的map算子处理。

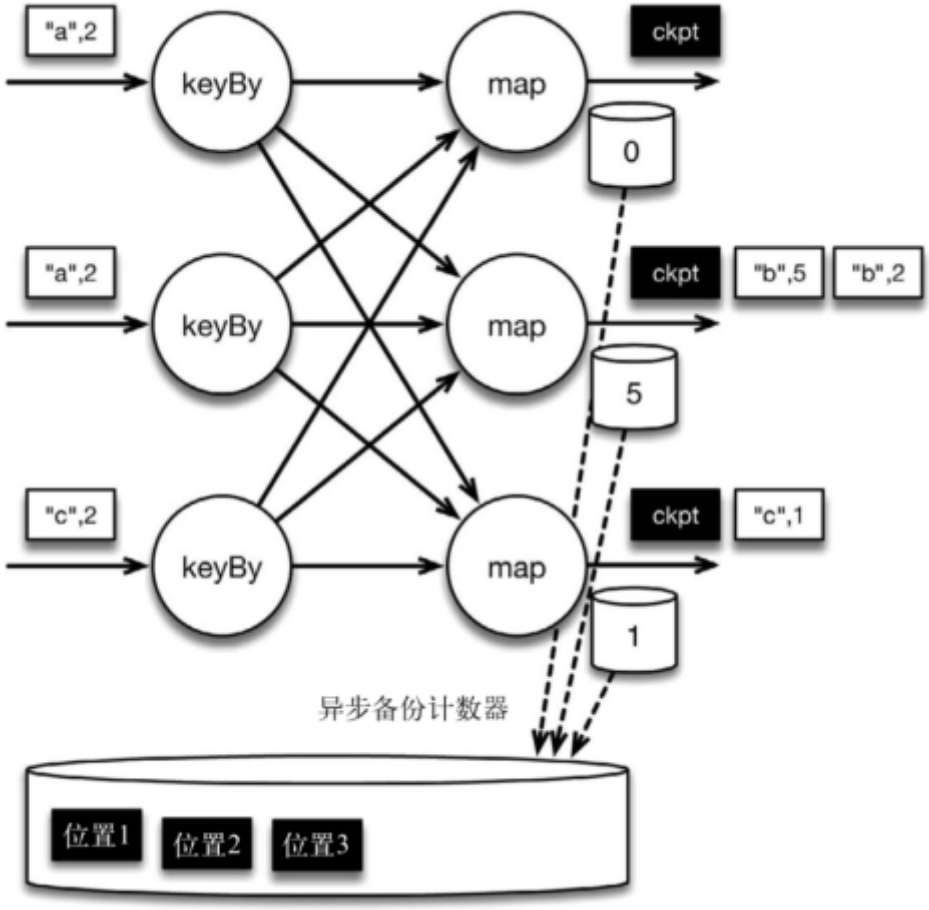


上图是程序的初始状态。注意，a、b、c三组的初始计数状态都是0，即三个圆柱上的值。ckpt表示检查点分割线（checkpoint barriers）。每条记录在处理顺序上严格地遵守在检查点之前或之后的规定，例如["b",2]在检查点之前被处理，["a",2]则在检查点之后被处理。当该程序处理输入流中的6条记录时，涉及的操作遍布3个并行实例(节点、CPU内核等)。那么，检查点该如何保证exactly-once呢?【有问题都可以私聊我WX：focusbigdata，或者关注我的公众号：FocusBigData，注意大小写】

> 检查点分割线和普通数据记录类似。**它们由算子处理，但并不参与计算，而是会触发与检查点相关的行为。**当读取输入流的数据源(在本例中与keyBy算子内联)遇到检查点屏障时，它将其在输入流中的位置保存到持久化存储中。如果输入流来自消息传输系统(Kafka)，这个位置就是偏移量。Flink的存储机制是插件化的，持久化存储可以是分布式文件系统，如HDFS。下图展示了这个过程。

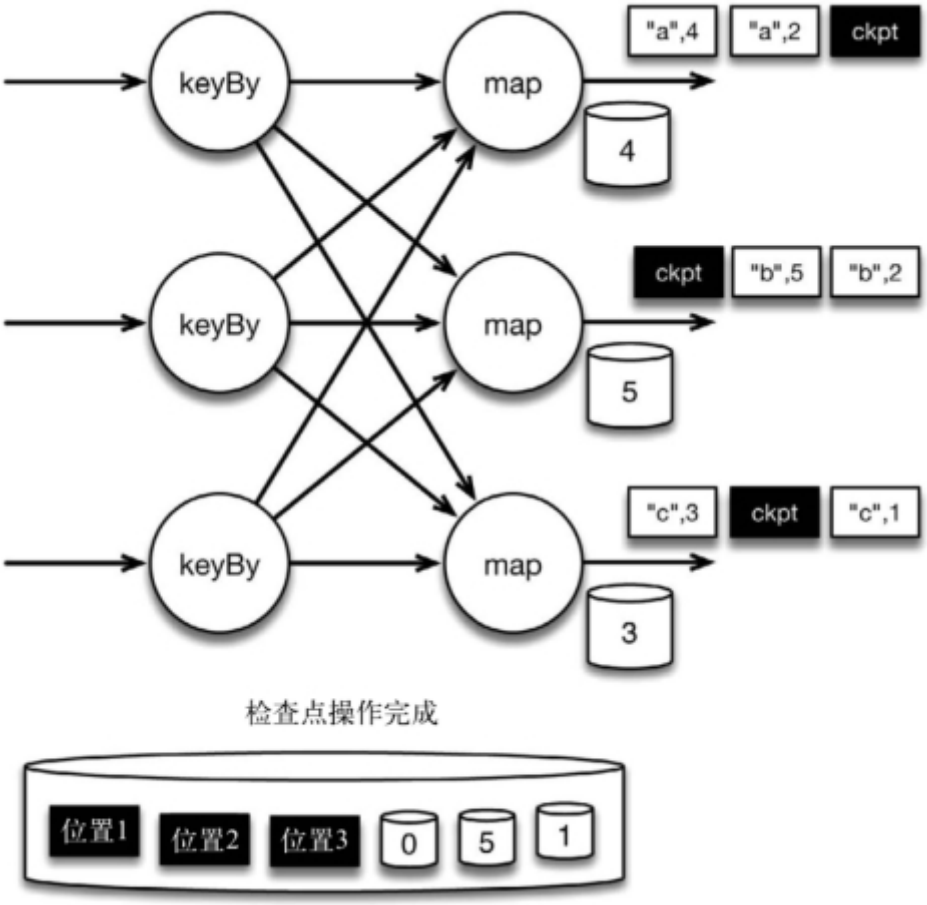


> 当Flink数据源(在本例中与keyBy算子内联)遇到检查点分界线（barrier）时，**它会将其在输入流中的位置保存到持久化存储中。**这让 Flink可以根据该位置重启。检查点像普通数据记录一样在算子之间流动。当map算子处理完前3条数据并收到检查点分界线时，它们会将状态以异步的方式写入持久化存储，如下图所示。



位于检查点之前的所有记录(["b",2]、["b",3]和["c",1])被map算子处理之后的情况。此时，**持久化存储已经备份了检查点分界线在输入流中的位置(备份操作发生在barrier被输入算子处理的时候)。**map算子接着开始处理检查点分界线，并触发将状态异步备份到稳定存储中这个动作。

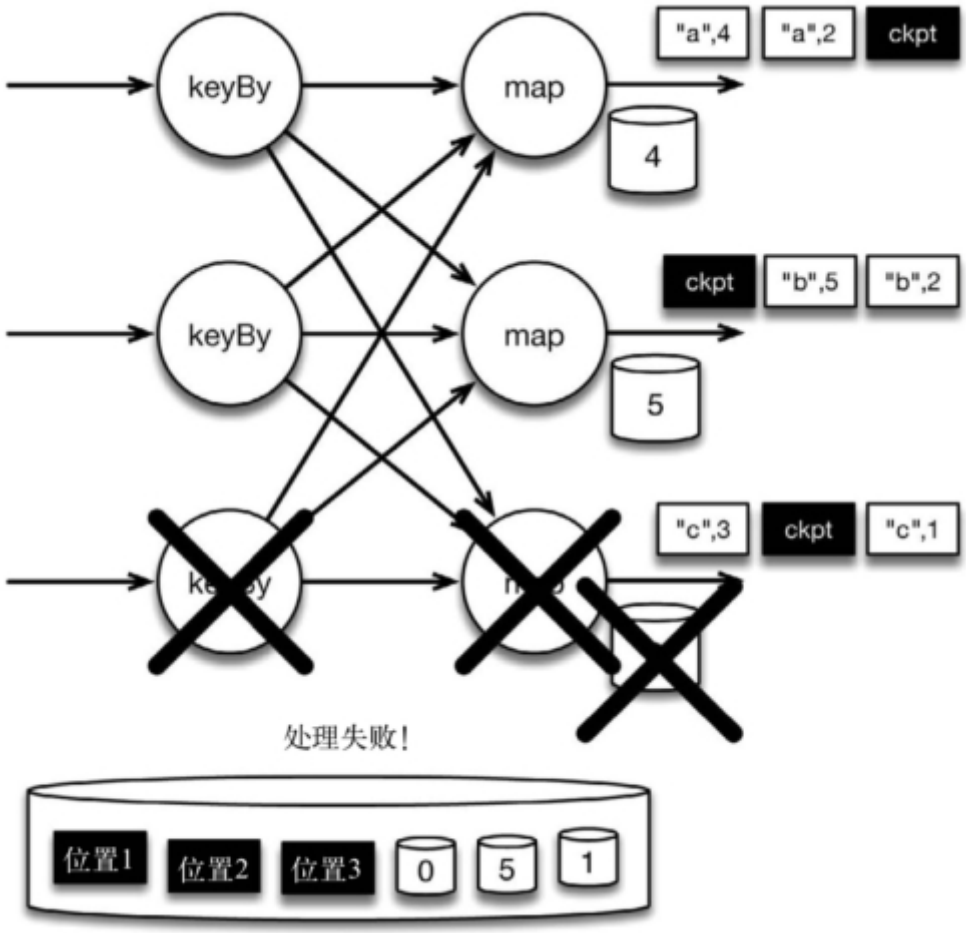
> **当map算子的状态备份和检查点分界线的位置备份被确认之后，该检查点操作就可以被标记为完成，**如下图所示。我们在无须停止或者阻断计算的条件下，在一个逻辑时间点(对应检查点屏障在输入流中的位置)为计算状态拍了快照。**通过确保备份的状态和位置指向同一个逻辑时间点，**后文将解释如何基于备份恢复计算，从而保证exactly-once。值得注意的是，当没有出现故障时，Flink检查点的开销极小，检查点操作的速度由持久化存储的可用带宽决定。回顾数珠子的例子: 除了因为数错而需要用到皮筋之外，皮筋会被很快地拨过。



检查点操作完成，状态和位置均已备份到稳定存储中。输入流中的所有数据记录都已处理完成。值得注意的是，备份的状态值与实际的状态值是不同的。**备份反映的是检查点的状态。**

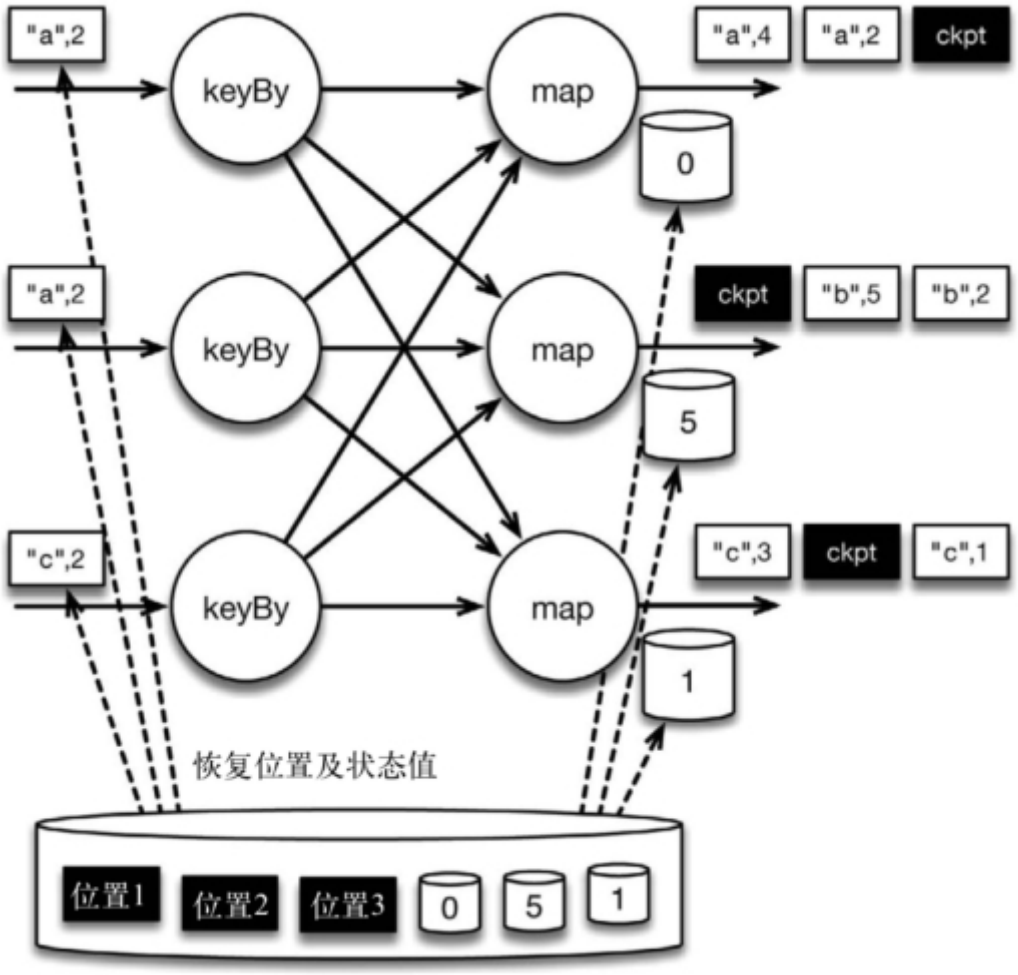
> **如果检查点操作失败，Flink可以丢弃该检查点并继续正常执行**，因为之后的某一个检查点可能会成功。虽然恢复时间可能更长，但是对于状态的保证依旧很有力。**只有在一系列连续的检查点操作失败之后，Flink才会抛出错误，因为这通常预示着发生了严重且持久的错误。**

现在来看看下图所示的情况: 检查点操作已经完成，但故障紧随其后。



> 在这种情况下，Flink会重新拓扑(可能会获取新的执行资源)，**将输入流倒回到上一个检查点**，然后恢复状态值并从该处开始继续计算。在本例中，["a",2]、["a",2]和["c",2]这几条记录将被重播。

下图展示了这一重新处理过程。从上一个检查点开始重新计算，可以保证在剩下的记录被处理之后，得到的map算子的状态值与没有发生故障时的状态值一致。



Flink将输入流倒回到上一个检查点屏障的位置，同时恢复map算子的状态值。然后，Flink从此处开始重新处理。这样做保证了在记录被处理之后，map算子的状态值与没有发生故障时的一致。

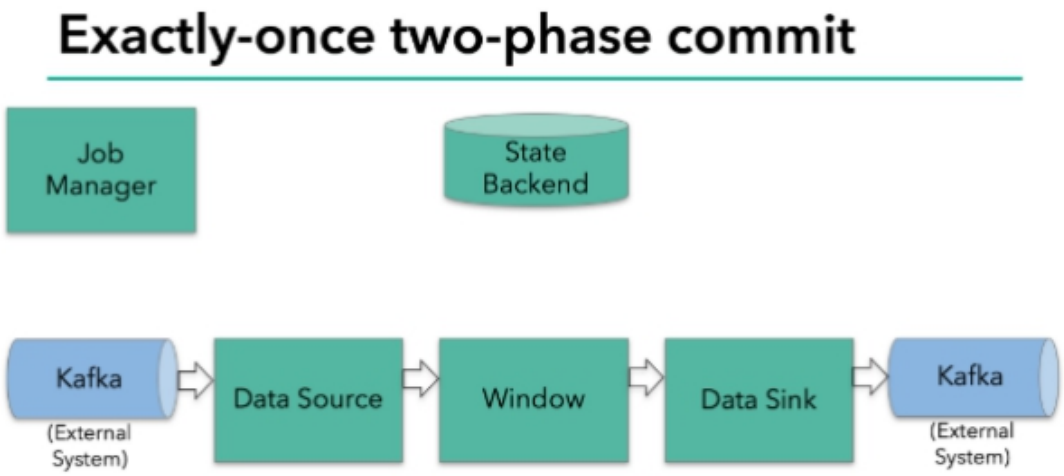
> Flink检查点算法的正式名称是异步分界线快照(asynchronous barrier snapshotting)。该算法大致基于Chandy-Lamport分布式快照算法。**检查点是Flink最有价值的创新之一，因为它使Flink可以保证exactly-once，并且不需要牺牲性能。**

Flink+Kafka如何实现端到端的exactly-once语义

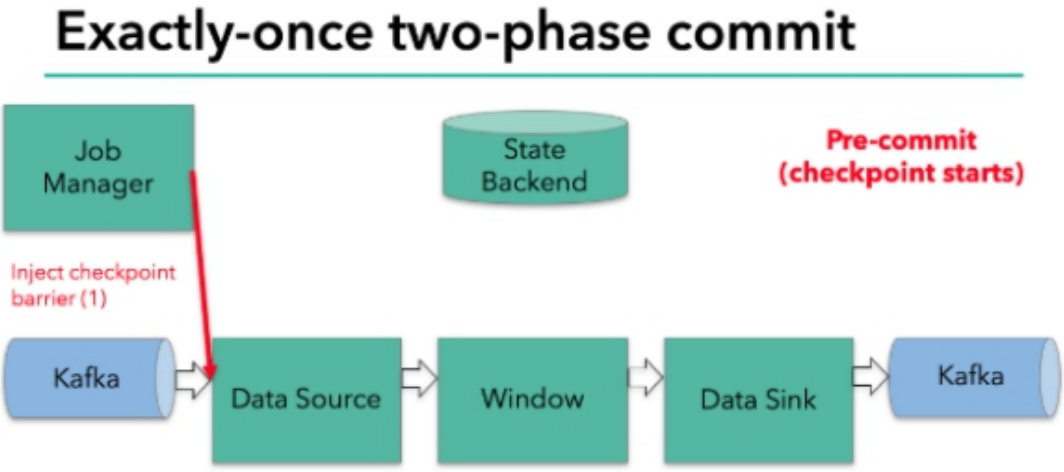
端到端的状态一致性的实现，需要每一个组件都实现，对于Flink + Kafka的数据管道系统（Kafka进、Kafka出）而言，各组件怎样保证exactly-once语义呢?

- 内部 —— 利用checkpoint机制，把状态存盘，发生故障的时候可以恢复，保证内部的状态一致性
- source —— kafka consumer作为source，可以将偏移量保存下来，如果后续任务出现了故障，恢复的时候可以由连接器重置偏移量，重新消费数据，保证一致性
- sink —— kafka producer作为sink，采用两阶段提交 sink，需要实现一个 TwoPhaseCommitSinkFunction

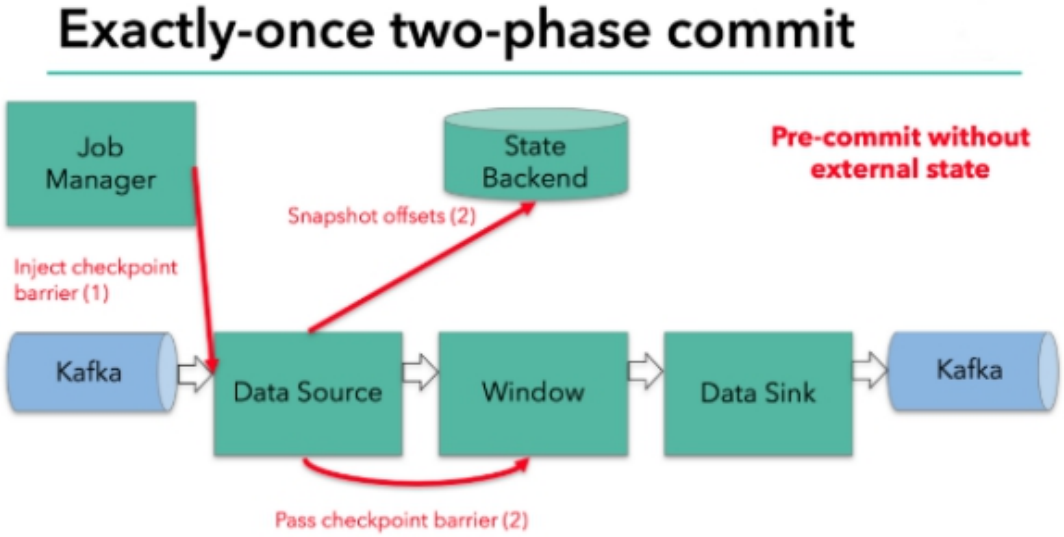
> 内部的checkpoint机制我们已经有了了解，那source和sink具体又是怎样运行的呢？接下来我们逐步做一个分析。我们知道Flink由JobManager协调各个TaskManager进行checkpoint存储，checkpoint保存在 StateBackend中，**默认StateBackend是内存级的，也可以改为文件级的进行持久化保存。**



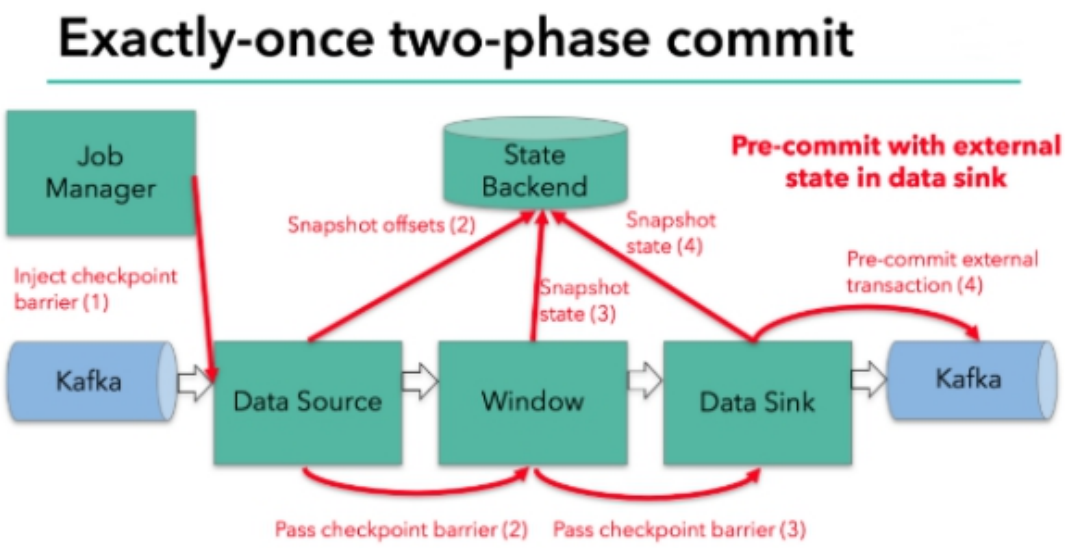
当 checkpoint 启动时，JobManager 会将检查点分界线（barrier）注入数据流；barrier会在算子间传递下去。



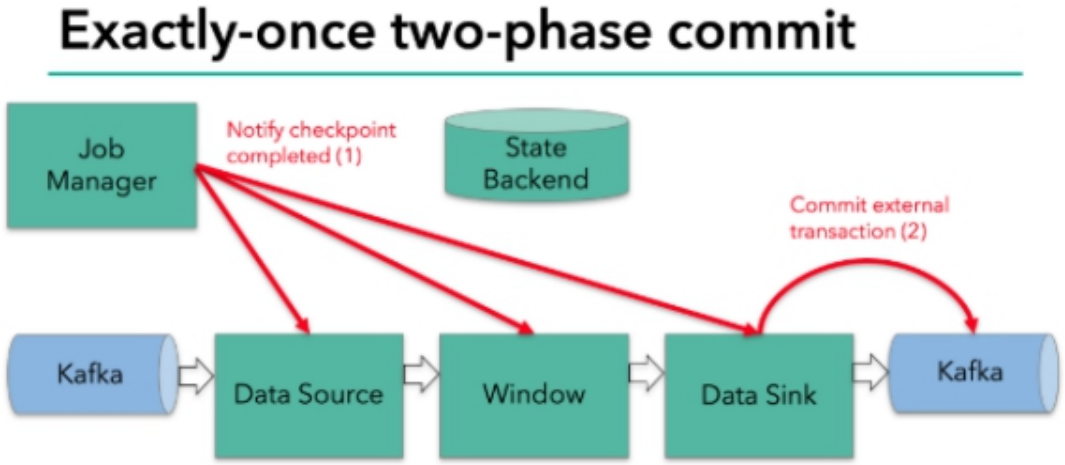
> **每个算子会对当前的状态做个快照**，保存到状态后端。对于source任务而言，就会把当前的offset作为状态保存起来。下次从checkpoint恢复时，source任务可以重新提交偏移量，从上次保存的位置开始重新消费数据。



> 每个内部的 transform 任务遇到 barrier 时，都会把状态存到 checkpoint 里。sink 任务首先把数据写入外部 kafka，这些数据都属于预提交的事务（还不能被消费）；当遇到 barrier 时，把状态保存到状态后端，并开启新的预提交事务。



> **当所有算子任务的快照完成，也就是这次的 checkpoint 完成时**，JobManager 会向所有任务发通知，确认这次 checkpoint 完成。当sink 任务收到确认通知，就会正式提交之前的事务，kafka 中未确认的数据就改为“已确认”，**数据就真正可以被消费了。**



> 所以我们看到，执行过程实际上是一个两段式提交，每个算子执行完成，会进行“预提交”，直到执行完sink操作，会发起“确认提交”，如果执行失败，预提交会放弃掉。

具体的两阶段提交步骤总结如下：

- 第一条数据来了之后，开启一个 kafka 的事务（transaction），正常写入 kafka 分区日志但标记为未提交，这就是“预提交”
- jobmanager 触发 checkpoint 操作，barrier 从 source 开始向下传递，遇到 barrier 的算子将状态存入状态后端，并通知 jobmanager
- sink 连接器收到 barrier，保存当前状态，存入 checkpoint，通知 jobmanager，并开启下一阶段的事务，用于提交下个检查点的数据
- jobmanager 收到所有任务的通知，发出确认信息，表示 checkpoint 完成
- sink 任务收到 jobmanager 的确认信息，正式提交这段时间的数据
- 外部kafka关闭事务，提交的数据可以正常消费了。

所以我们也可以看到，如果宕机需要通过StateBackend进行恢复，**只能恢复所有确认提交的操作。**

状态后端(state backend)

- MemoryStateBackend

> 内存级的状态后端，会将键控状态作为内存中的对象进行管理，**将它们存储在TaskManager的JVM堆上；而将checkpoint存储在JobManager的内存中。**

- FsStateBackend

> 将checkpoint**存到远程的持久化文件系统（FileSystem）上**。而对于本地状态，跟MemoryStateBackend一样，也会存在TaskManager的JVM堆上。

- RocksDBStateBackend

> 将所有状态序列化后，**存入本地的RocksDB中存储。**

注意：RocksDB的支持并不直接包含在flink中，需要引入依赖：

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-statebackend-rocksdb_2.11</artifactId>
    <version>1.10.0</version>
</dependency>
```

设置状态后端代码

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val checkpointPath: String = ???
val backend = new RocksDBStateBackend(checkpointPath)

env.setStateBackend(backend)
env.setStateBackend(new FsStateBackend("file:///tmp/checkpoints"))
env.enableCheckpointing(1000)
// 配置重启策略
env.setRestartStrategy(RestartStrategies.fixedDelayRestart(60, Time.of(10,
TimeUnit.SECONDS)))
```