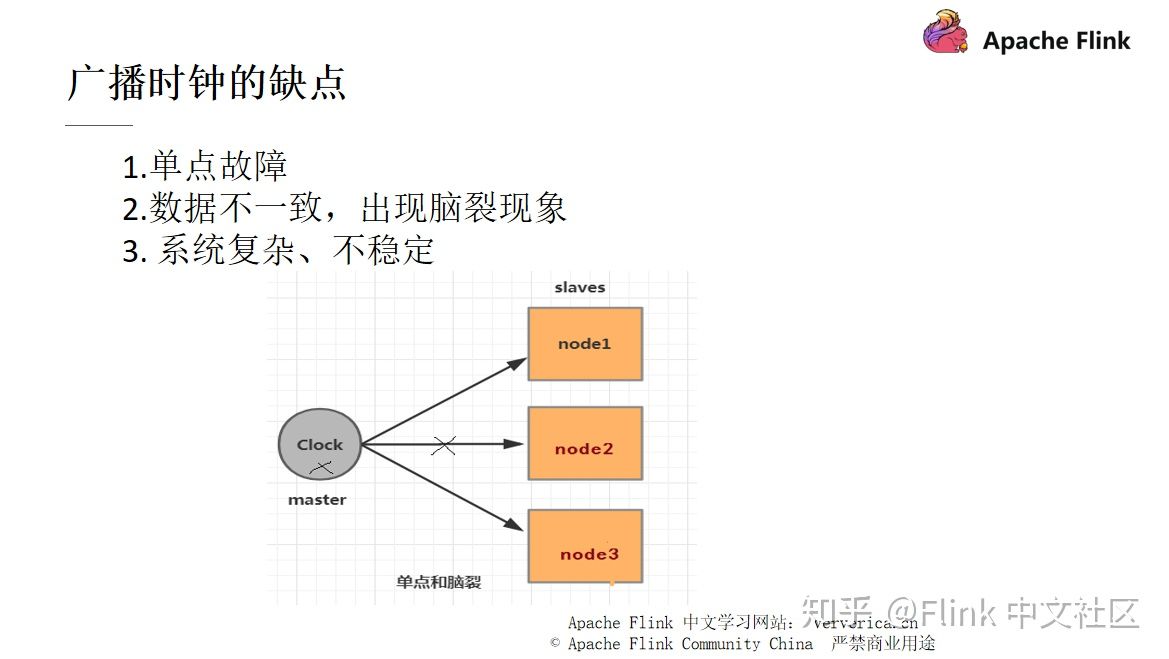
**Flink作业问题分析和调优实践**

**简介：** 本文主要分享 Flink 的 CheckPoint 机制、反压机制及 Flink 的内存模型。对这3部分内容的熟悉是调优的前提，文章主要从以下几个部分分享：原理剖析、性能定位、经典场景调优、内存调优。

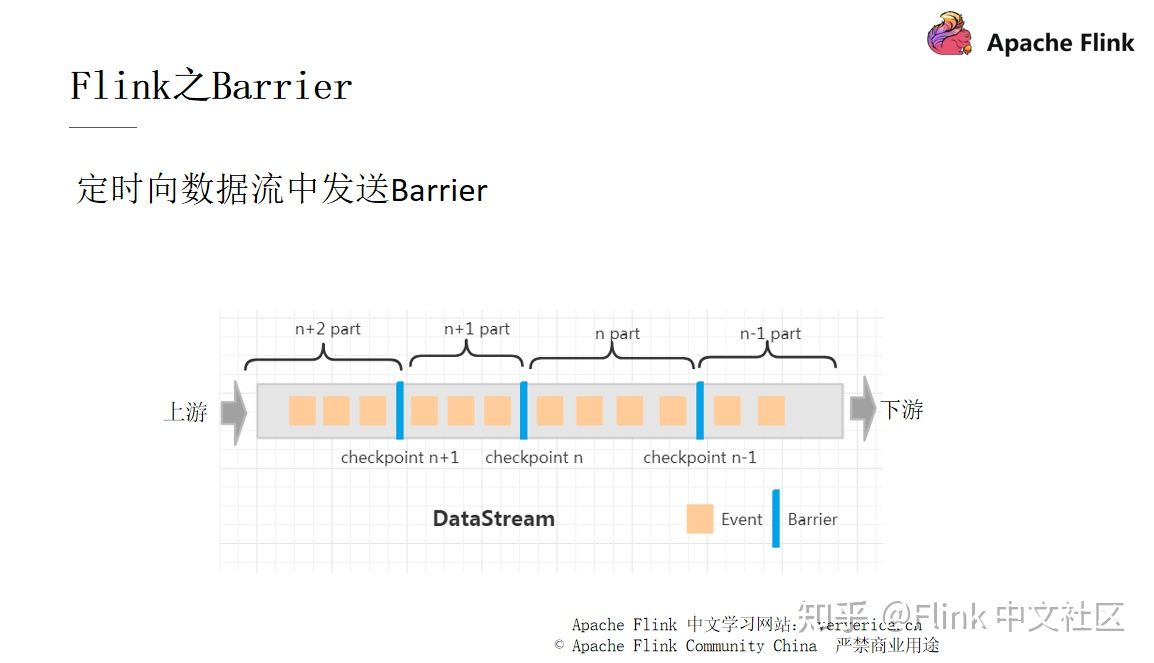
摘要：本文主要分享 Flink 的 CheckPoint 机制、反压机制及 Flink 的内存模型。对这3部分内容的熟悉是调优的前提，文章主要从以下几个部分分享：

1. 原理剖析
2. 性能定位
3. 经典场景调优
4. 内存调优

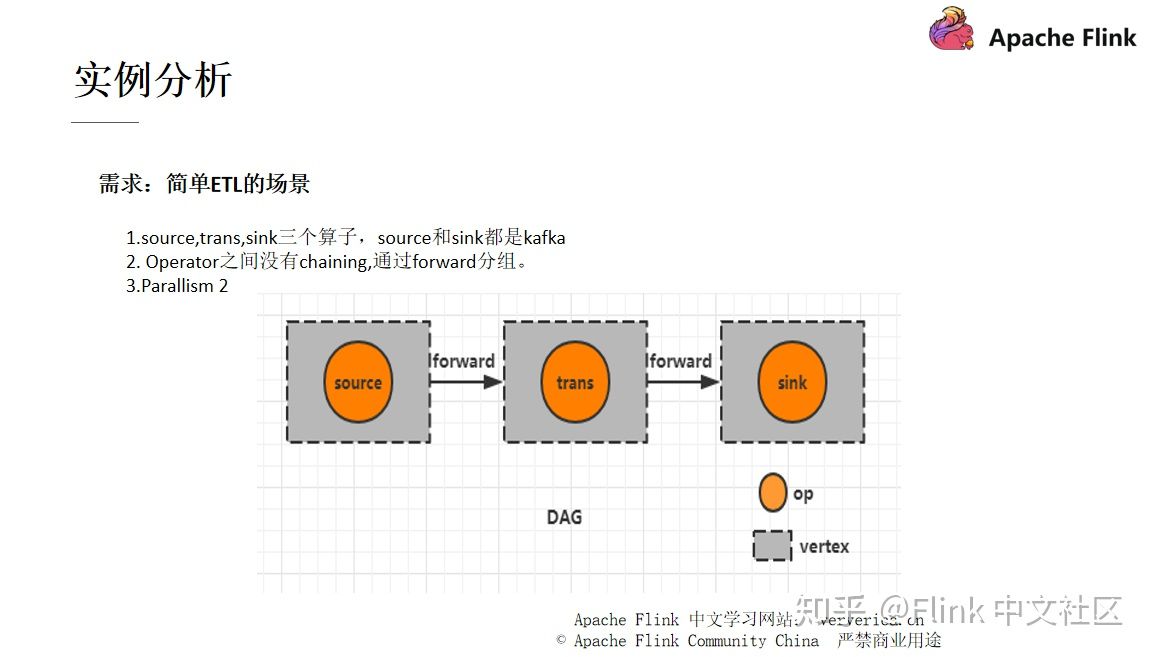
Checkpoint 机制  
1.什么是 checkpoint  
简单地说就是 Flink 为了达到容错和 exactly-once 语义的功能，定期把 state 持久化下来，而这一持久化的过程就叫做 checkpoint ，它是 Flink Job 在某一时刻全局状态的快照。  
当我们要对分布式系统实现一个全局状态保留的功能时，传统方案会引入一个统一时钟，通过分布式系统中的 master 节点广播出去给每一个 slaves 节点，当节点接收到这个统一时钟时，它们就记录下自己当前的状态即可。



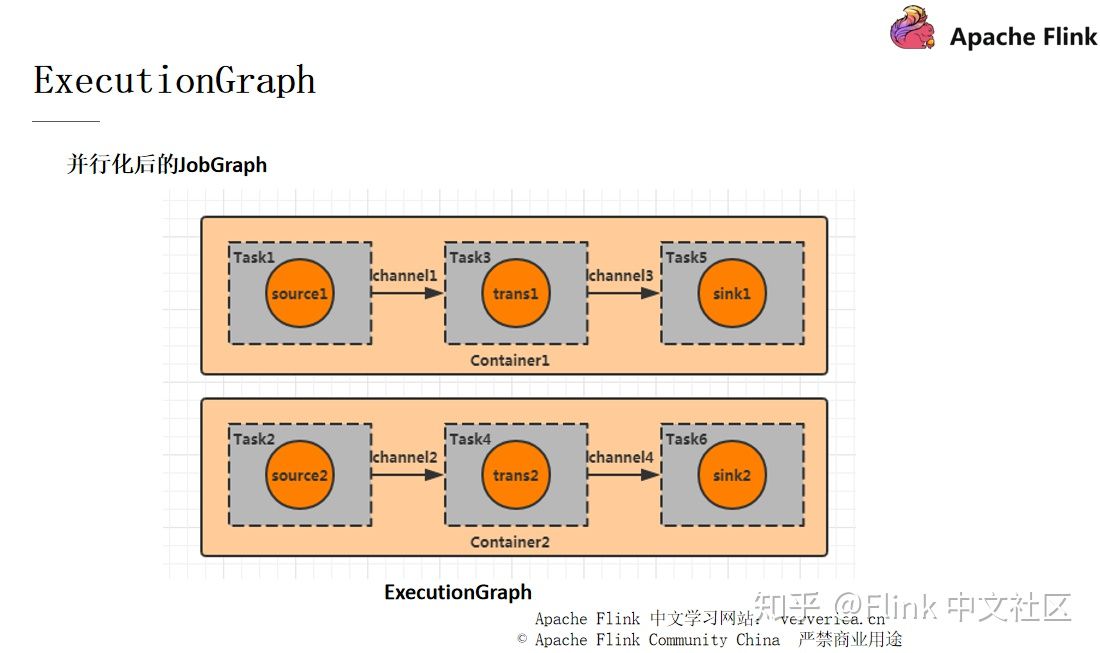
但是统一时钟的方式也存在一定的问题，某一个 node 进行的 GC 时间比较长，或者 master 与 slaves 的网络在当时存在波动而造成时钟的发送延迟或者发送失败，都会造成此 slave 和其它的机器出现数据不一致而最终导致脑裂的情况。如果我们想要解决这个问题，就需要对 master 和 slaves 做一个 HA（High Availability）。但是，一个系统越是复杂，就越不稳定且维护成本越高。  
Flink 是将 checkpoint 都放进了一个名为 Barrier 的流。



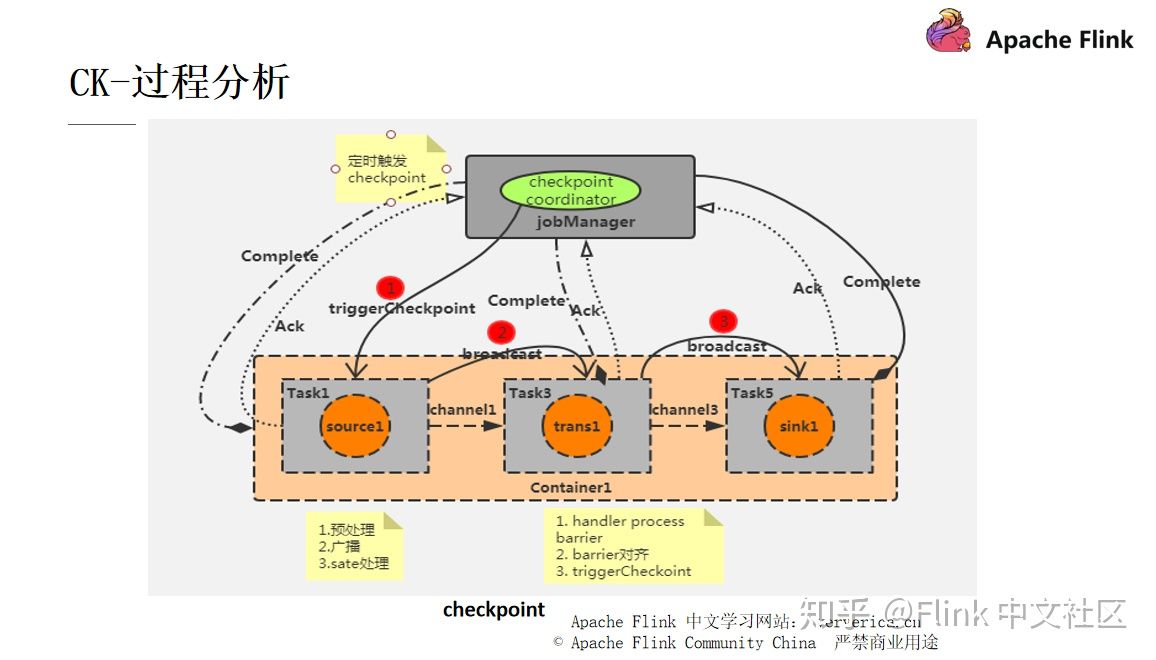
上图中就是一个 Barrier 的例子，从上游的第一个 Task 到下游的最后一个 Task，每次当 Task 经过图中蓝色的栅栏时，就会触发 save snapshot（快照）的功能。我们用一个例子来简单说明。  
2.实例分析



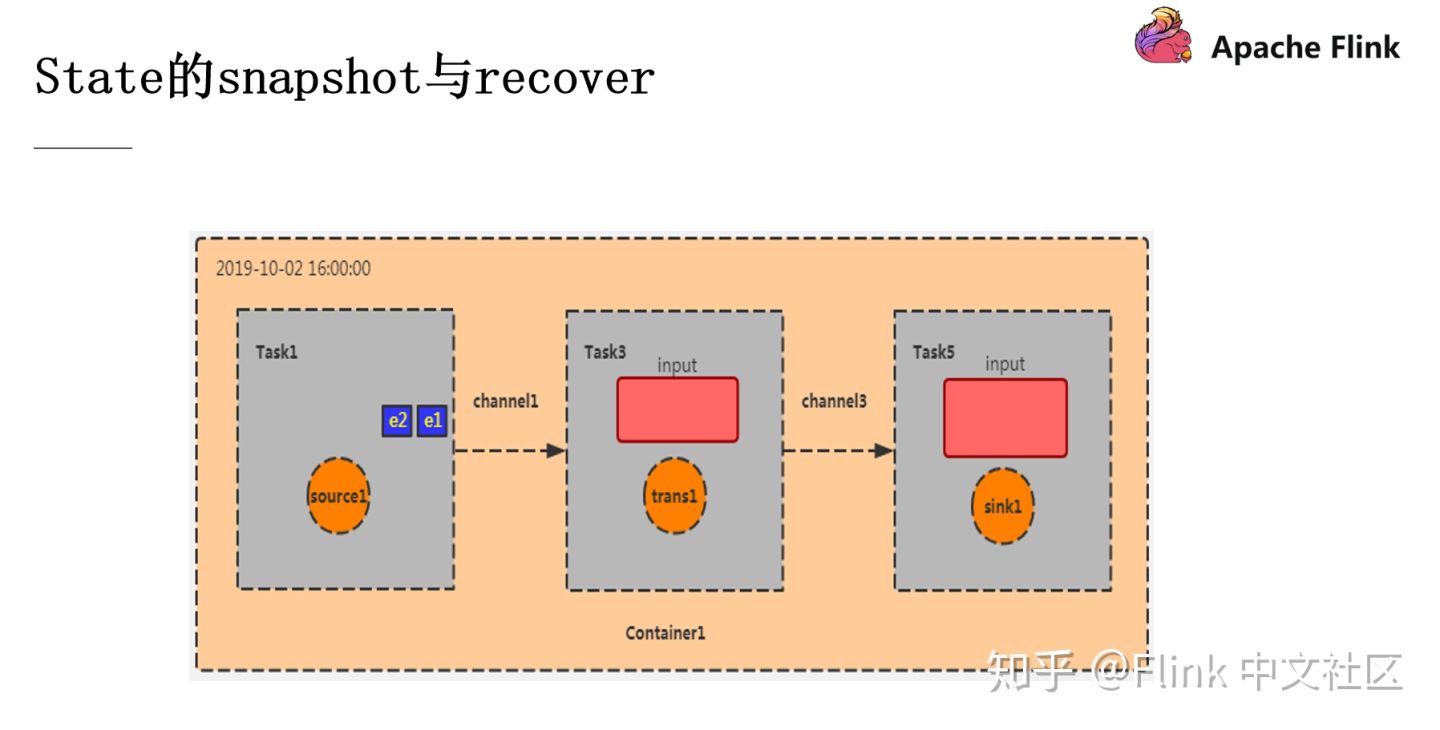
这是一个简单的 ETL 过程，首先我们把数据从 Kafka 中拿过来进行一个 trans 的转换操作，然后再发送到一个下游的 Kafka  
此时这个例子中没有进行 chaining 的调优。所以此时采用的是 forward strategy ，也就是 “一个 task 的输出只发送给一个 task 作为输入”，这样的方式，这样做也有一个好处就是如果两个 task 都在一个 JVM 中的话，那么就可以避免不必要的网络开销  
设置 Parallism 为 2，此时的 DAG 图如下：



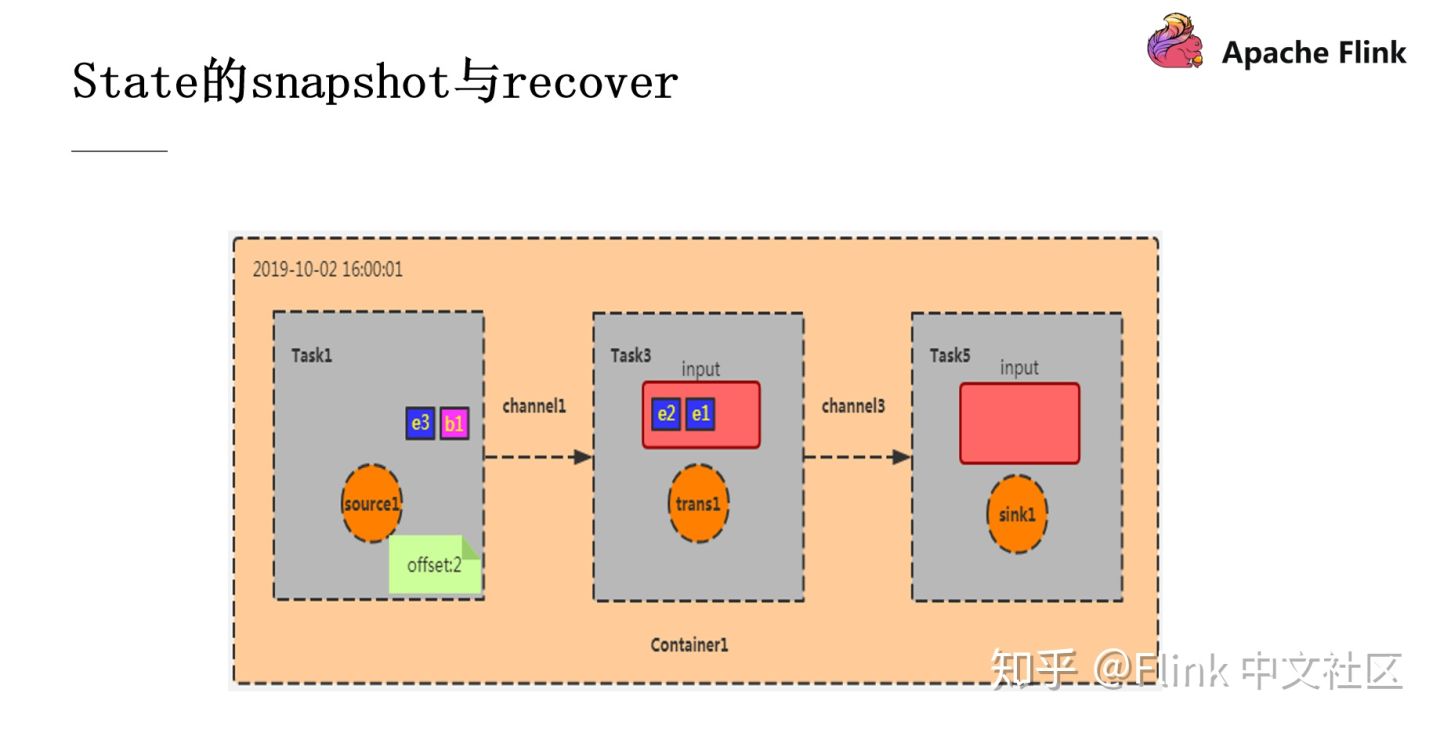
■ CK的分析过程



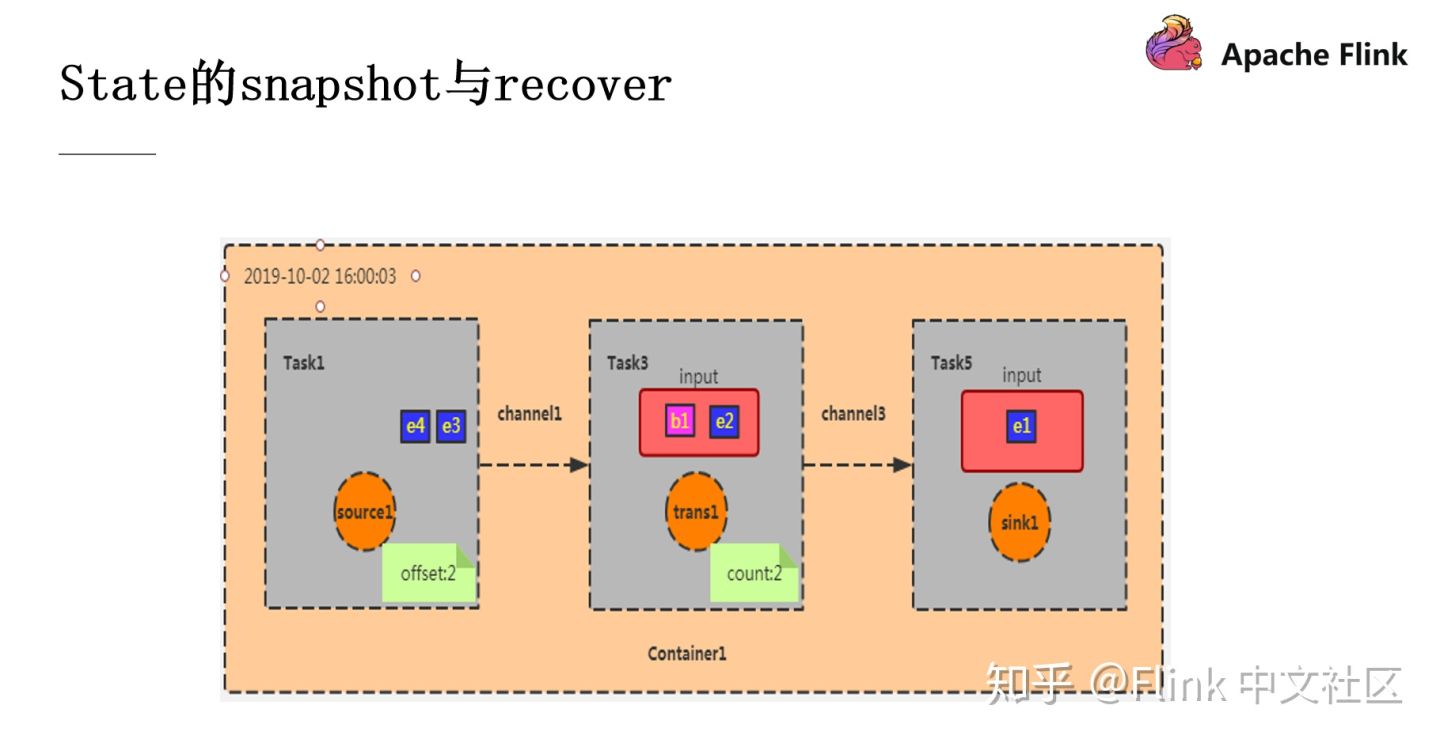
每一个 Flink 作业都会有一个 JobManager ，JobManager 里面又会有一个 checkpoint coordinator 来管理整个 checkpoint 的过程，我们可以设置一个时间间隔让 checkpoint coordinator 将一个 checkpoint 的事件发送给每一个 Container 中的 source task，也就是第一个任务（对应并行图中的 task1，task2）。  
当某个 Source 算子收到一个 Barrier 时，它会暂停自身的数据处理，然后将自己的当前 state 制作成 snapshot（快照），并保存到指定的持久化存储中，最后向 CheckpointCoordinator 异步发送一个 ack（Acknowledge character --- 确认字符），同时向自身所有下游算子广播该 Barrier 后恢复自身的数据处理。  
每个算子按照上面不断制作 snapshot 并向下游广播，直到最后 Barrier 传递到 sink 算子，此时快照便制作完成。这时候需要注意的是，上游算子可能是多个数据源，对应多个 Barrier 需要全部到齐才一次性触发 checkpoint ，所以在遇到 checkpoint 时间较长的情况时，有可能是因为数据对齐需要耗费的时间比较长所造成的。  
■ Snapshot & Recover



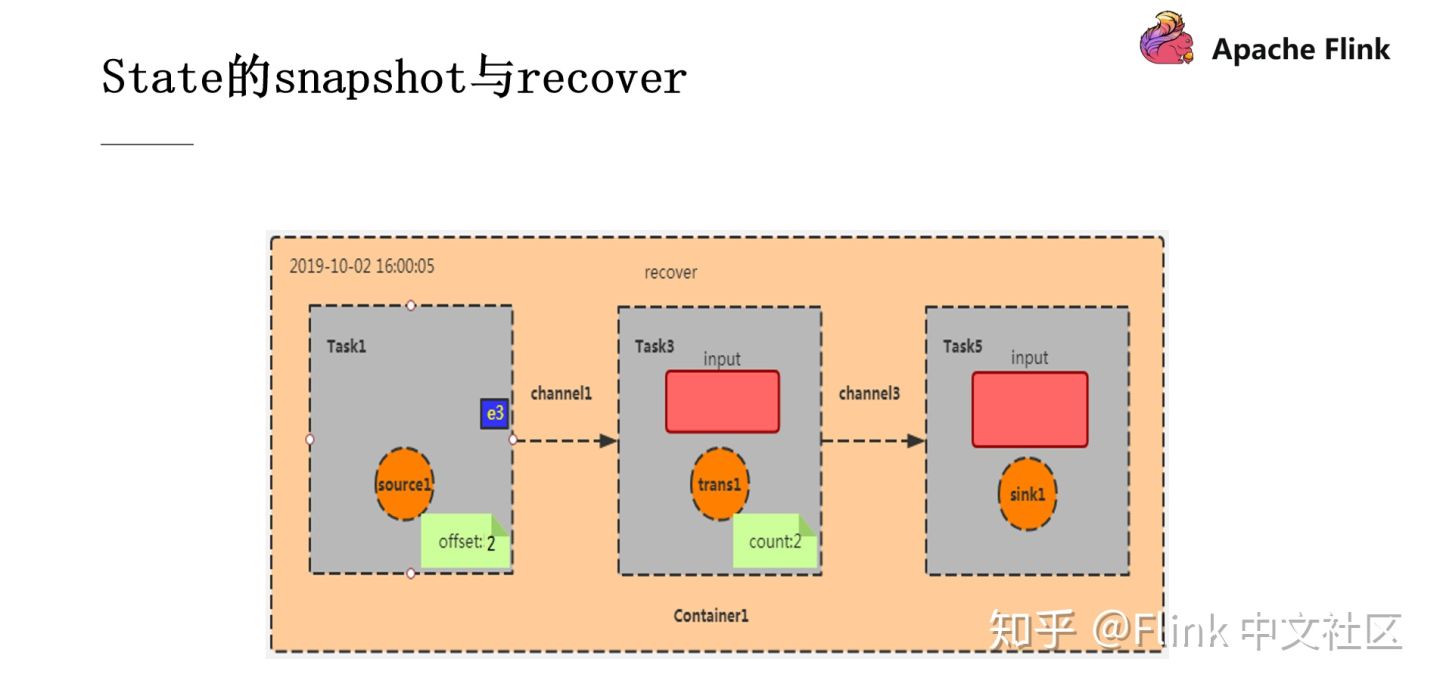
如图，这是我们的Container容器初始化的阶段，e1 和 e2 是刚从 Kafka 消费过来的数据，与此同时，CheckpointCoordinator 也往它发送了 Barrier。



此时 Task1 完成了它的 checkpoint 过程，效果就是记录下 offset 为2（e1，e2），然后把 Barrier 往下游的算子广播，Task3 的输入为 Task1 的输出，现在假设我的这个程序的功能是统计数据的条数，此时 Task3 的 checkpoint 效果就是就记录数据数为2（因为从 Task1 过来的数据就是 e1 和 e2 两条），之后再将 Barrier 往下广播，当此 Barrier 传递到 sink 算子，snapshot 就算是制作完成了。



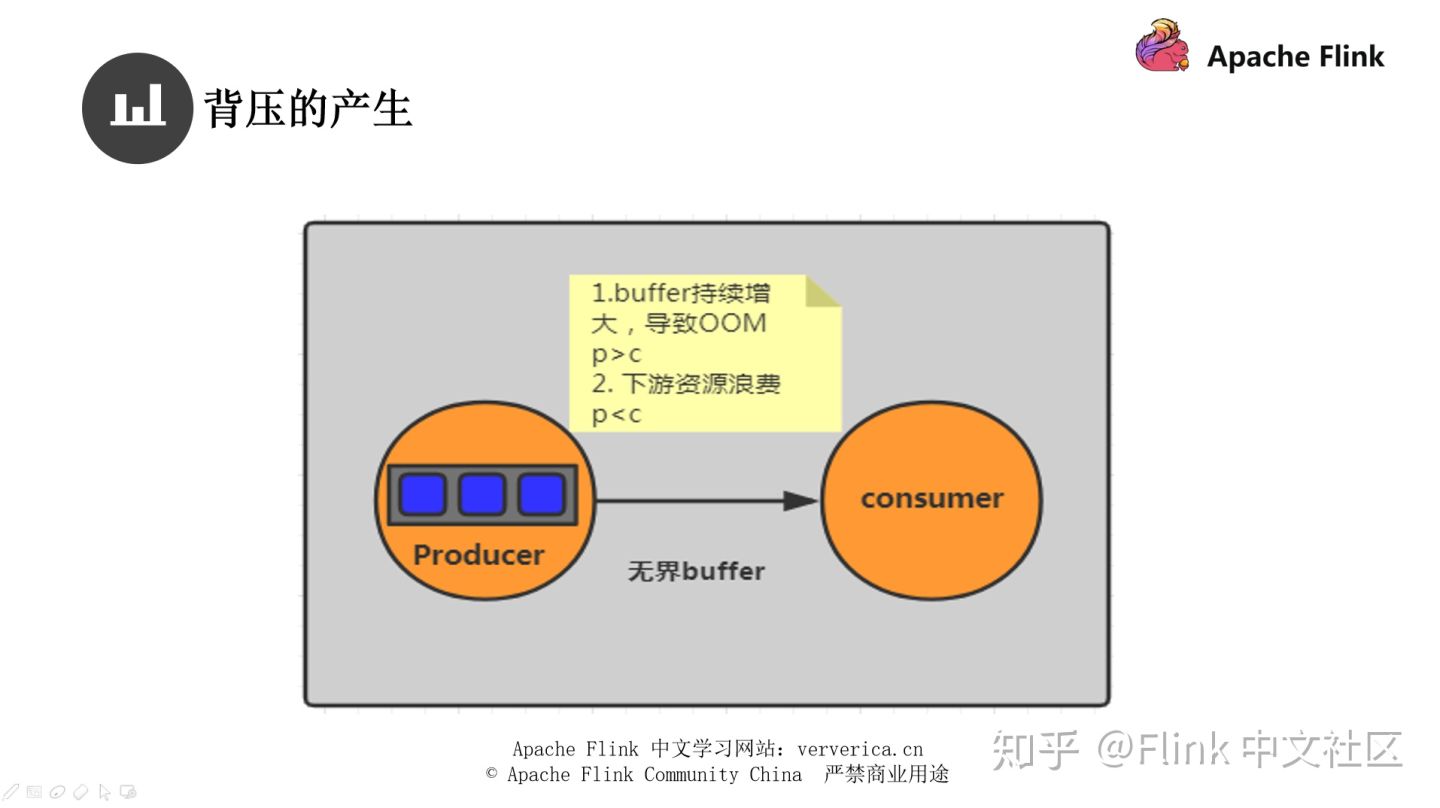
此时 source 中还会源源不断的产生数据，并产生新的 checkpoint ，但是此时如果 Container 宕机重启就需要进行数据的恢复了。刚刚完成的 checkpoint 中 offset为2，count为2，那我们就按照这个 state 进行恢复。此时 Task1 会从 e3 开始消费，这就是 Recover 操作。



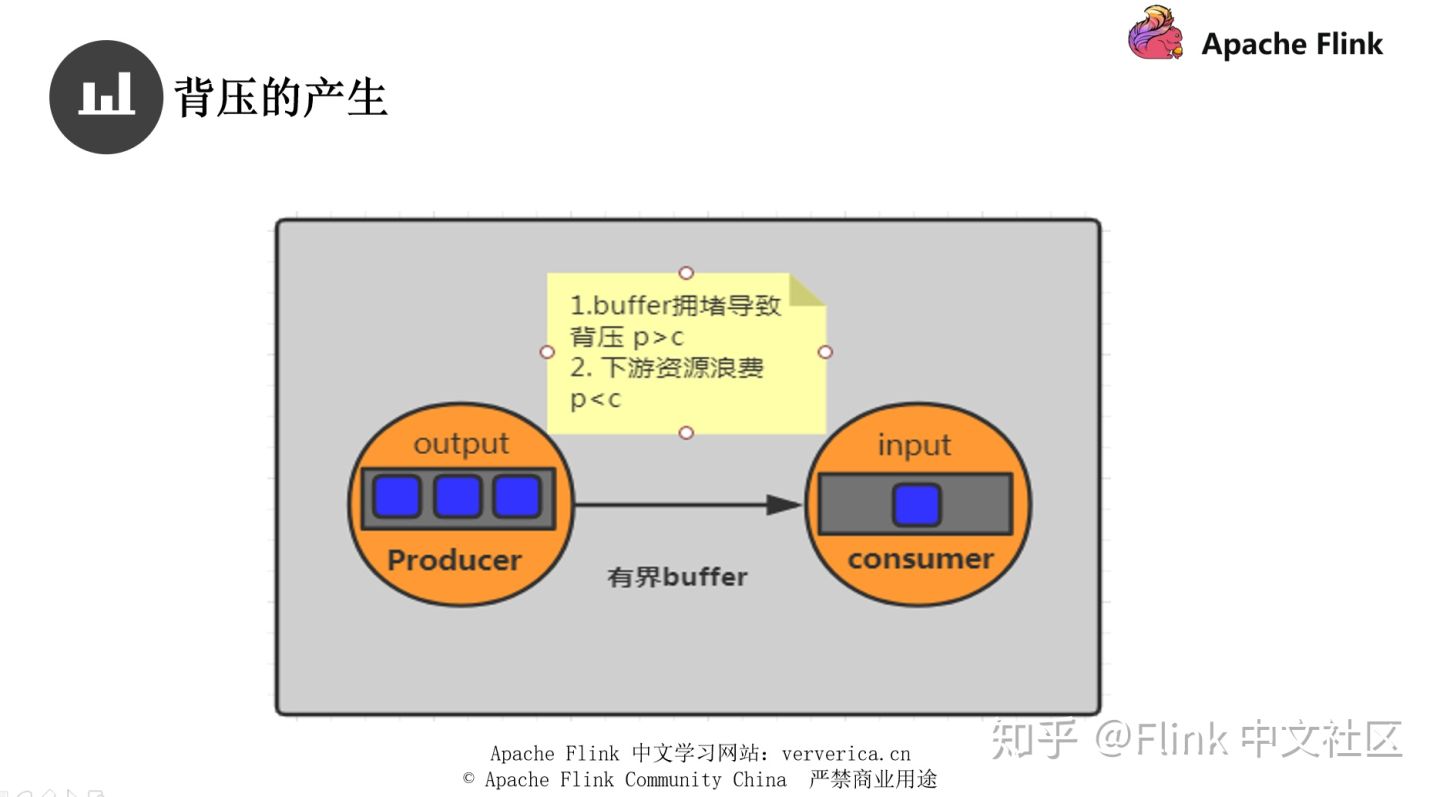
■ checkpoint 的注意事项  
下面列举的3个注意要点都会影响到系统的吞吐，在实际开发过程中需要注意：



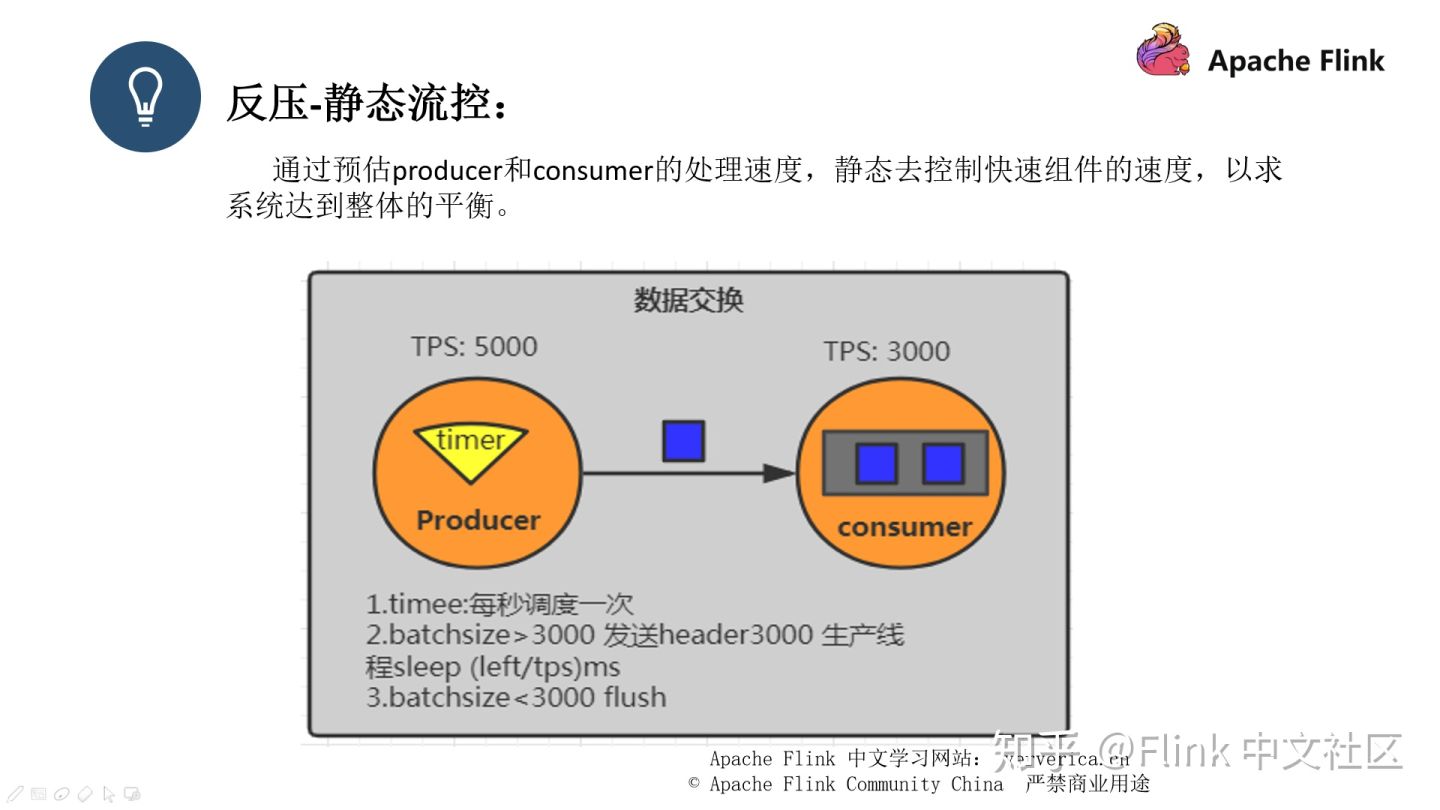
3.背压的产生及 Flink 的反压处理  
在分布式系统中经常会出现多个 Task 多个 JVM 之间可能需要做数据的交换，我们使用生产者和消费者来说明这个事情。



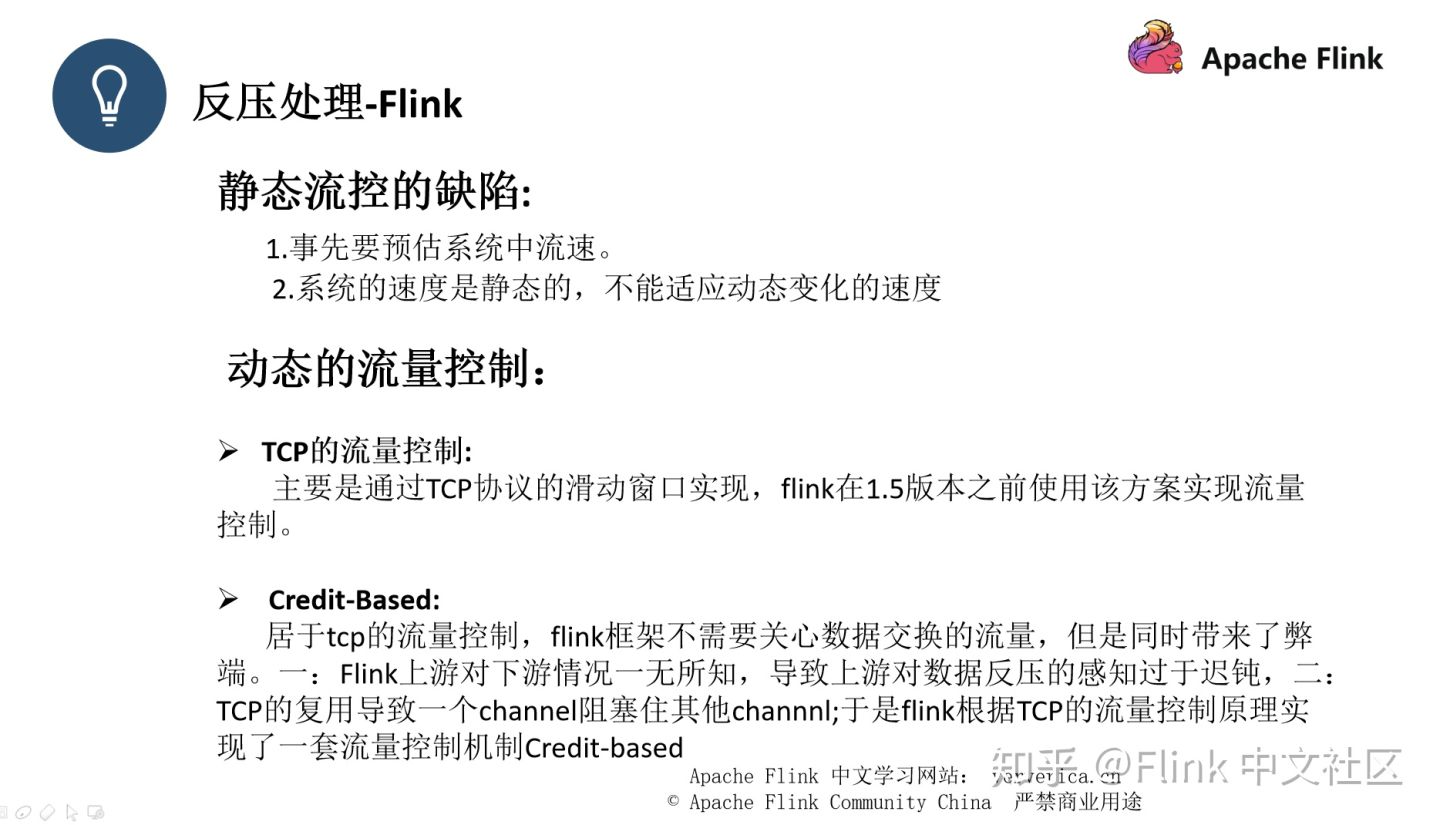
假设我现在的 Producer 是使用了无界 buffer 来进行存储，当我们的生产者生产速度远大于消费者消费的速度时，生产端的数据会因为消费端的消费能力低下而导致数据积压，最终导致 OOM 的产生。



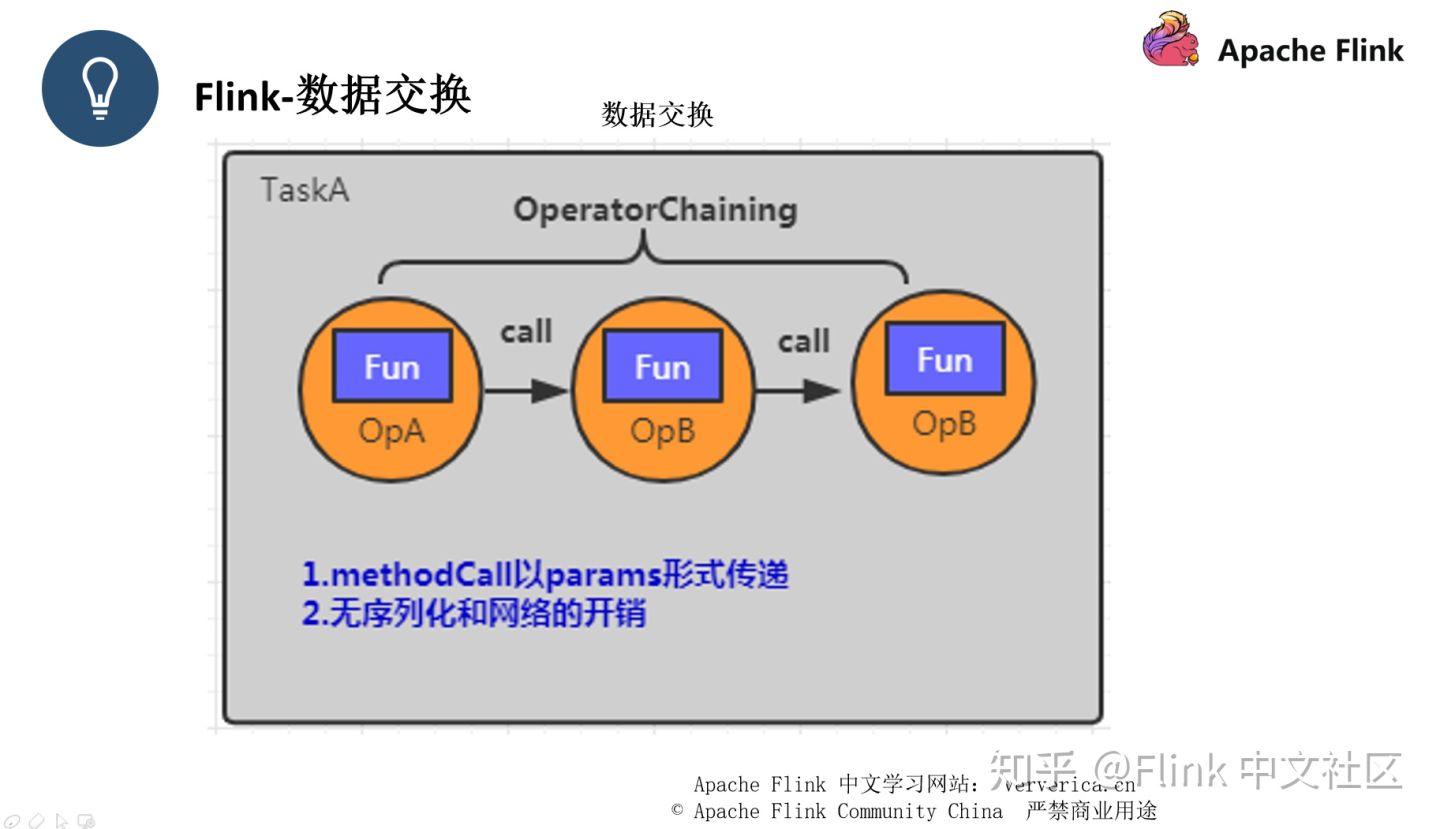
而就算使用了有界 buffer，同样消费者端的消费能力低下，当 buffer 被积满时生产者就会停止生产，这样还不能完全地解决我们的问题，所以就需要根据不同的情况进行调整。  
Flink 也是通过有界 buffer 来进行不同 TaskManager 的数据交换。而且做法分为了静态控流和动态控流两种方式。



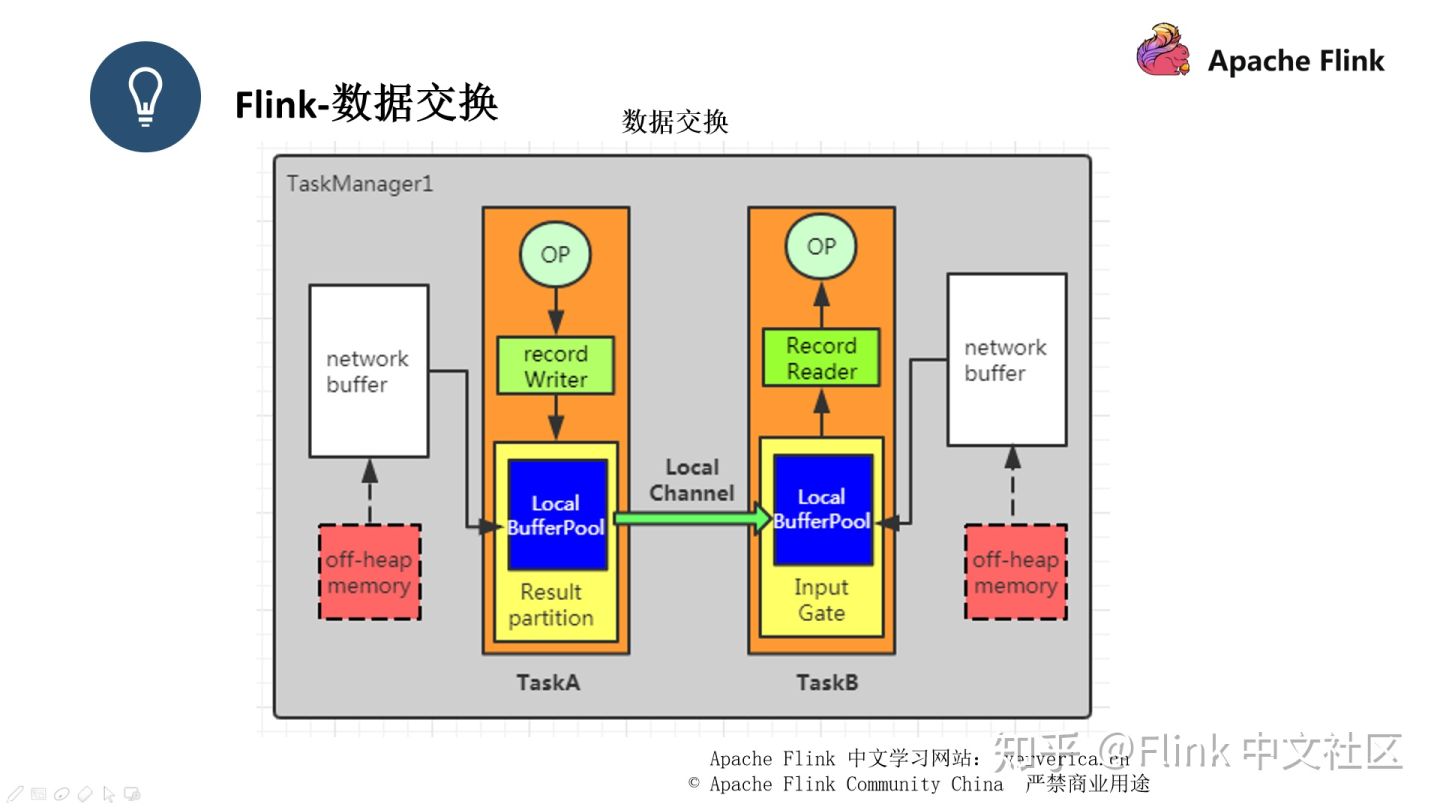
简单来说就是当生产者比消费者的 TPS 多时，我们采用溢写的方式，使用 batch 来封装好我们的数据，然后分批发送出去，每次发送完成后再 sleep 一段时间，这个时间的计算方式是 left（剩余的数据）/ tps，但是这个做法是很难去预估系统的情况的。



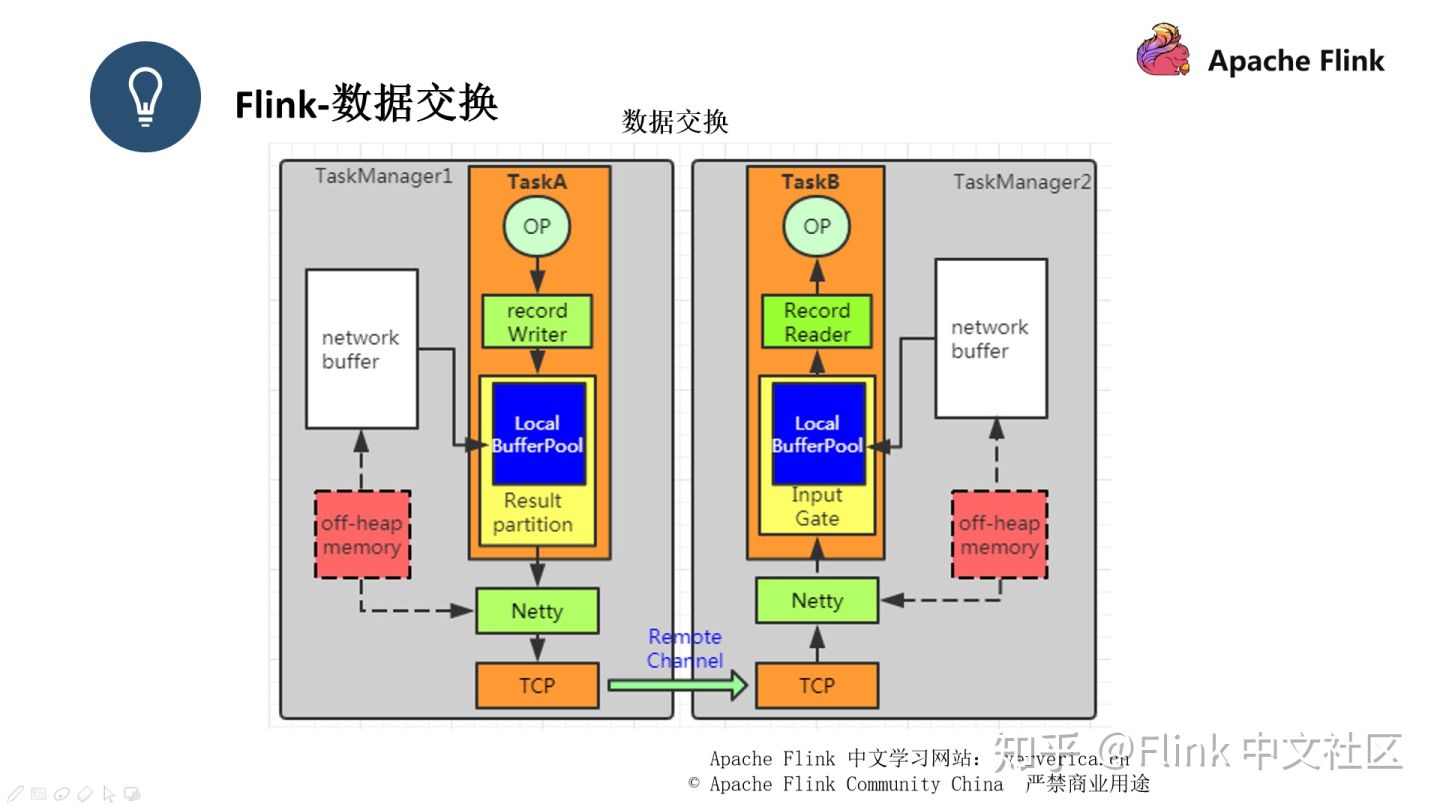
Flink 1.5 之前的流控是基于 TCP 的滑动窗口实现的，在之前的课程中已经有提到过了。而 Flink 在1.5之后已经弃用了该机制，所以这里不展开说明。在此网络模型中，数据生成节点只能通过检查当前的 channel 是否可写来决定自己是否要向消费端发送数据，它对下游数据消费端的真实容量情况一概不知。这就导致，当生成节点发现 channel 已经不可写的时候，有可能下游消费节点已经积压了很多数据。  
Credit-Based 我们用下面的数据交换的例子说明：  
Flink 的数据交换大致分为三种，一种是同一个 Task 的数据交换，另一种是 不同 Task 同 JVM 下的数据交换。第三种就是不同 Task 且不同 JVM 之间的交换。



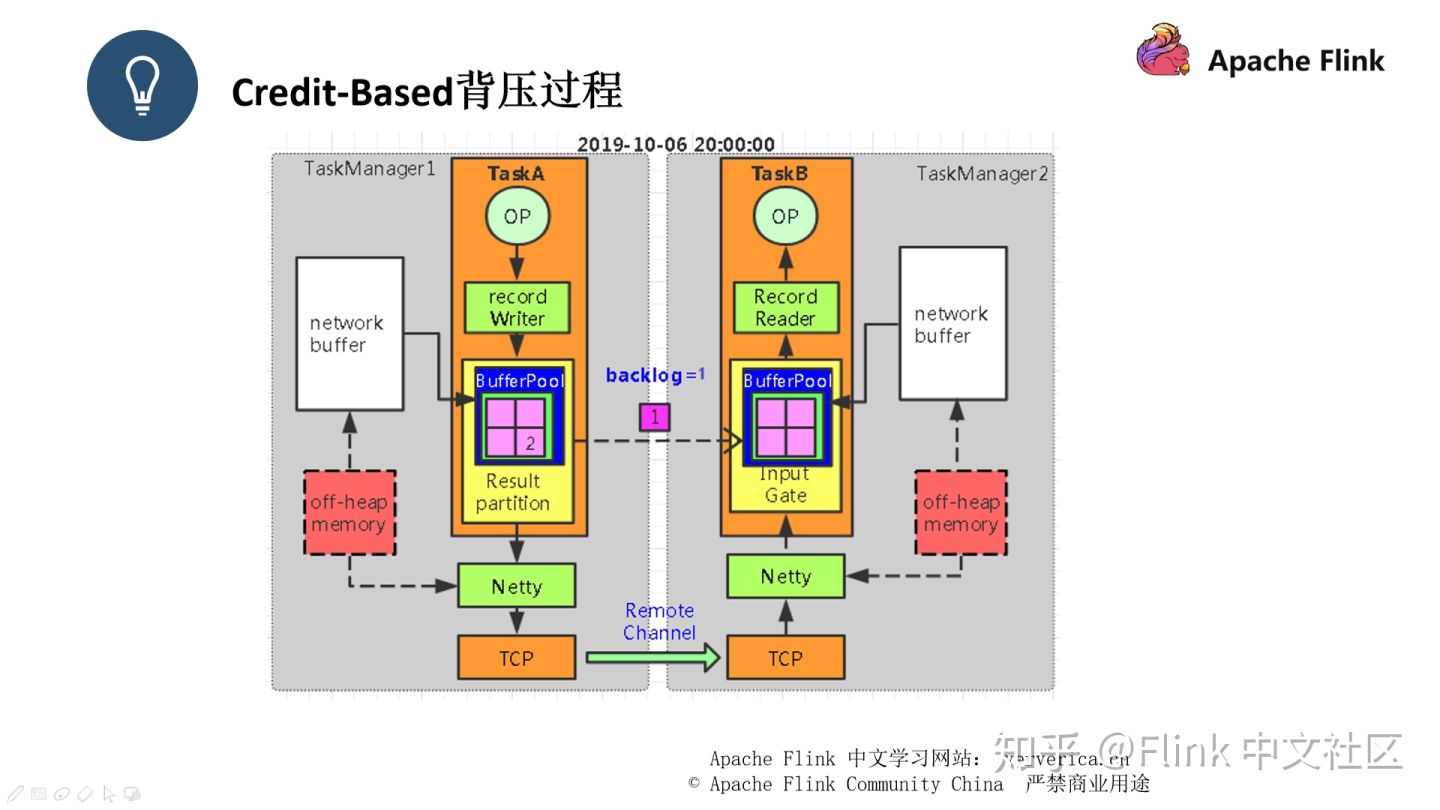
同一个 Task 的数据交换就是我们刚刚提到的 forward strategy 方式，主要就是避免了序列化和网络的开销。



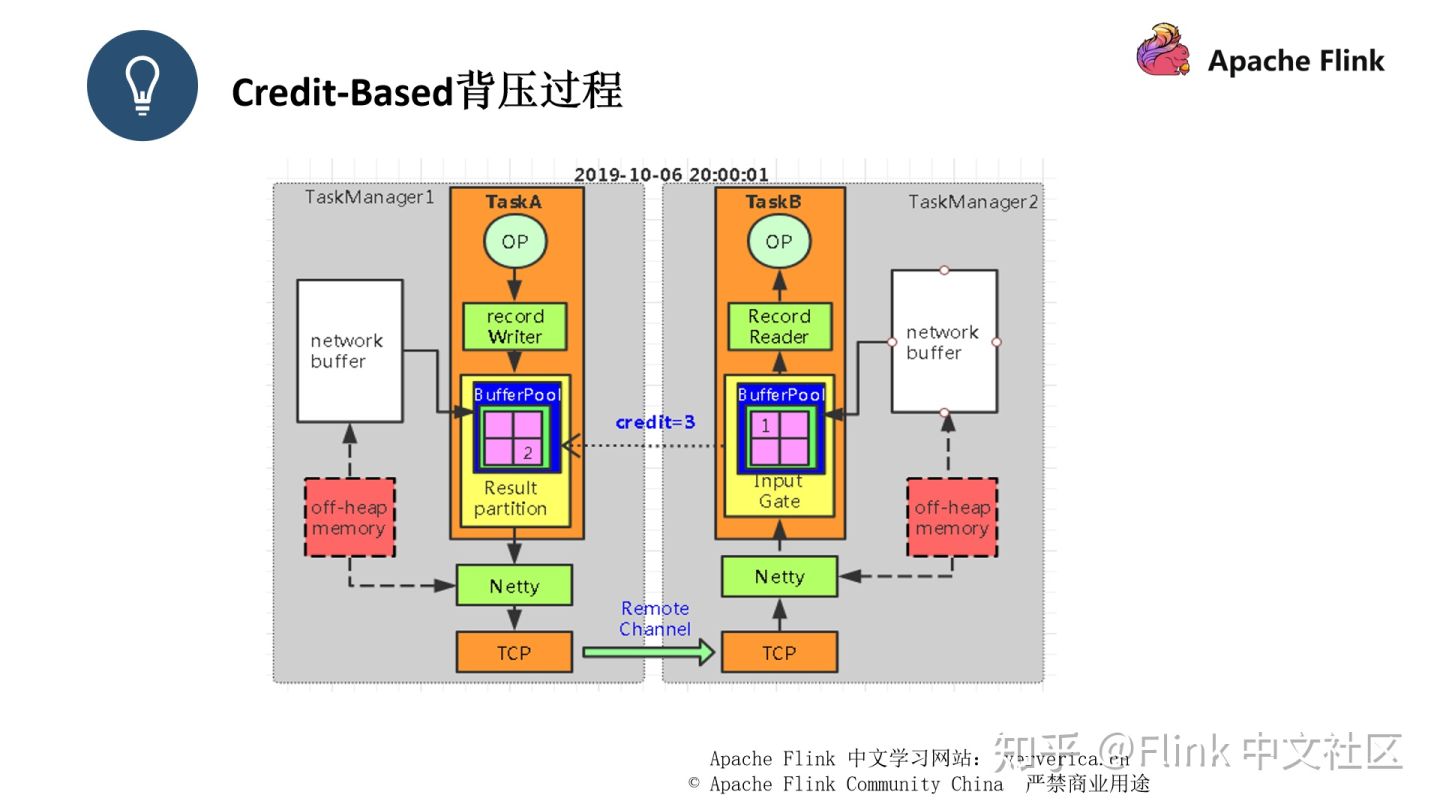
第二种数据交换的方式就是数据会先通过一个 record Writer ，数据在里面进行序列化之后再传递给 Result Partition ，之后数据会通过 local channel 传递给另外一个 Task 的 Input Gate 里面，再进行反序列化，推送给 Record Reader 之后进行操作。



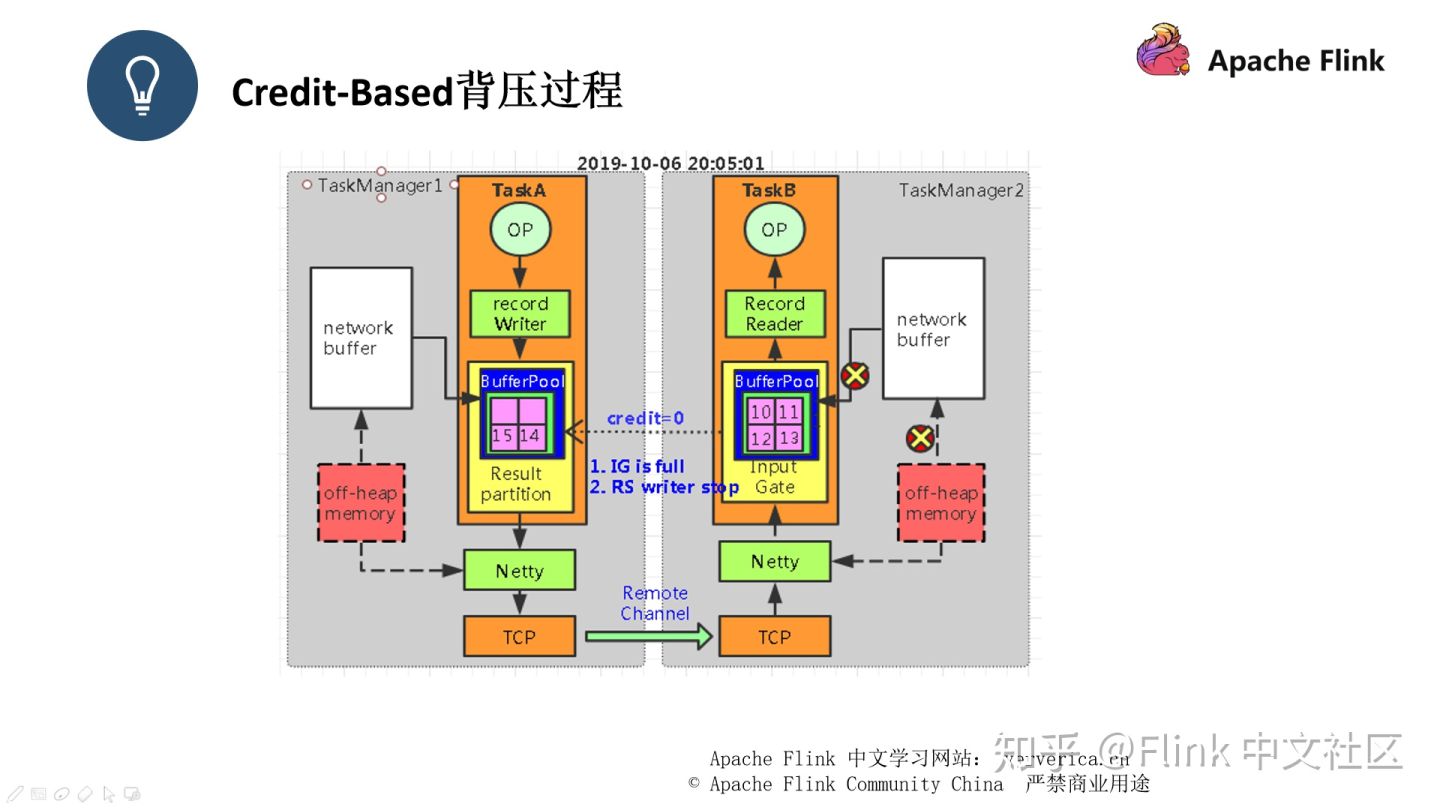
因为第三种数据交换涉及到了不同的 JVM，所以会有一定的网络开销，和第二种的区别就在于它先推给了 Netty ，通过netty把数据推送到远程端的 Task 上。  
■ Credit-Based



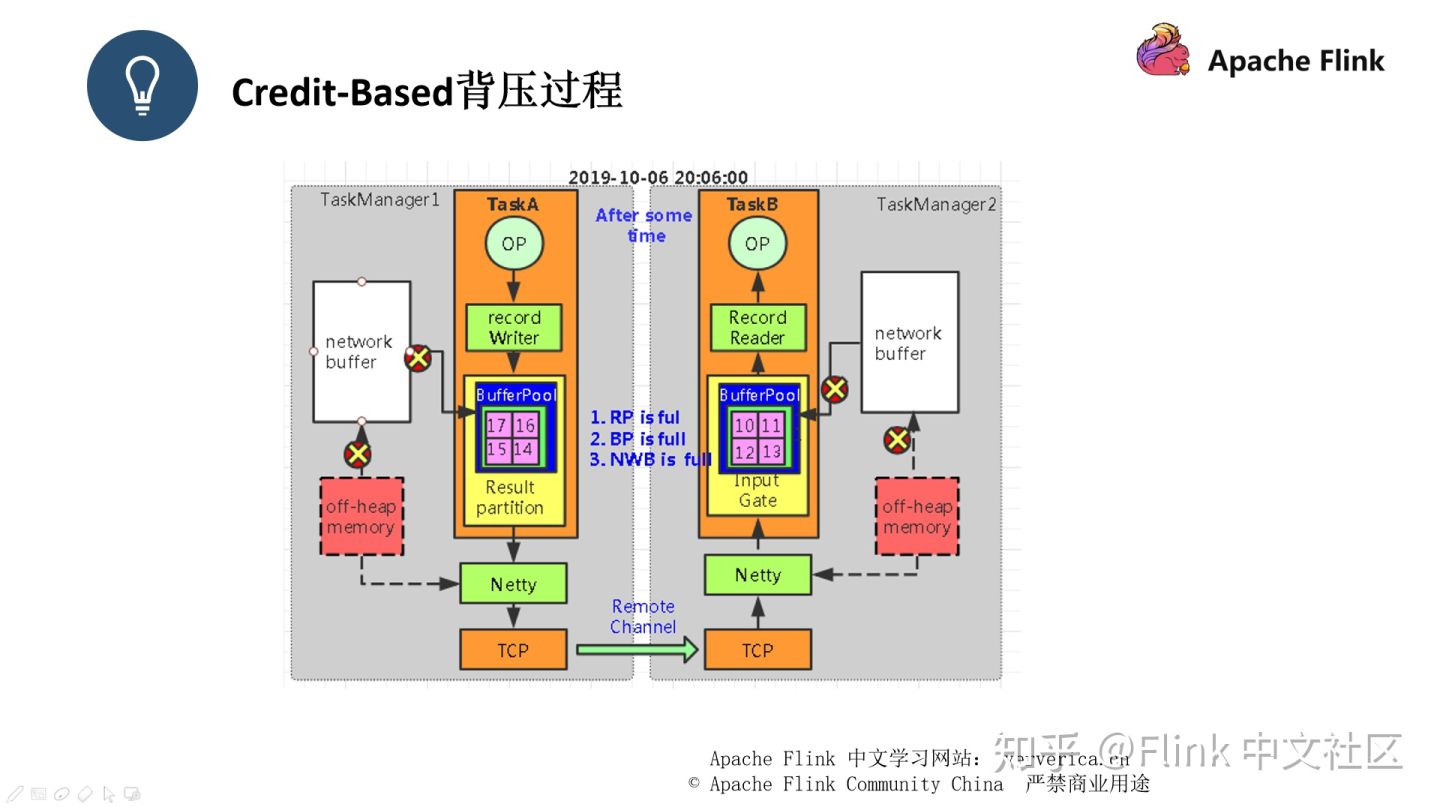
此时我们可以看到 event1 已经连带一个 backlog = 1 推送给了 TaskB，backlog 的作用其实只是为了让消费端感知到我们生产端的情况

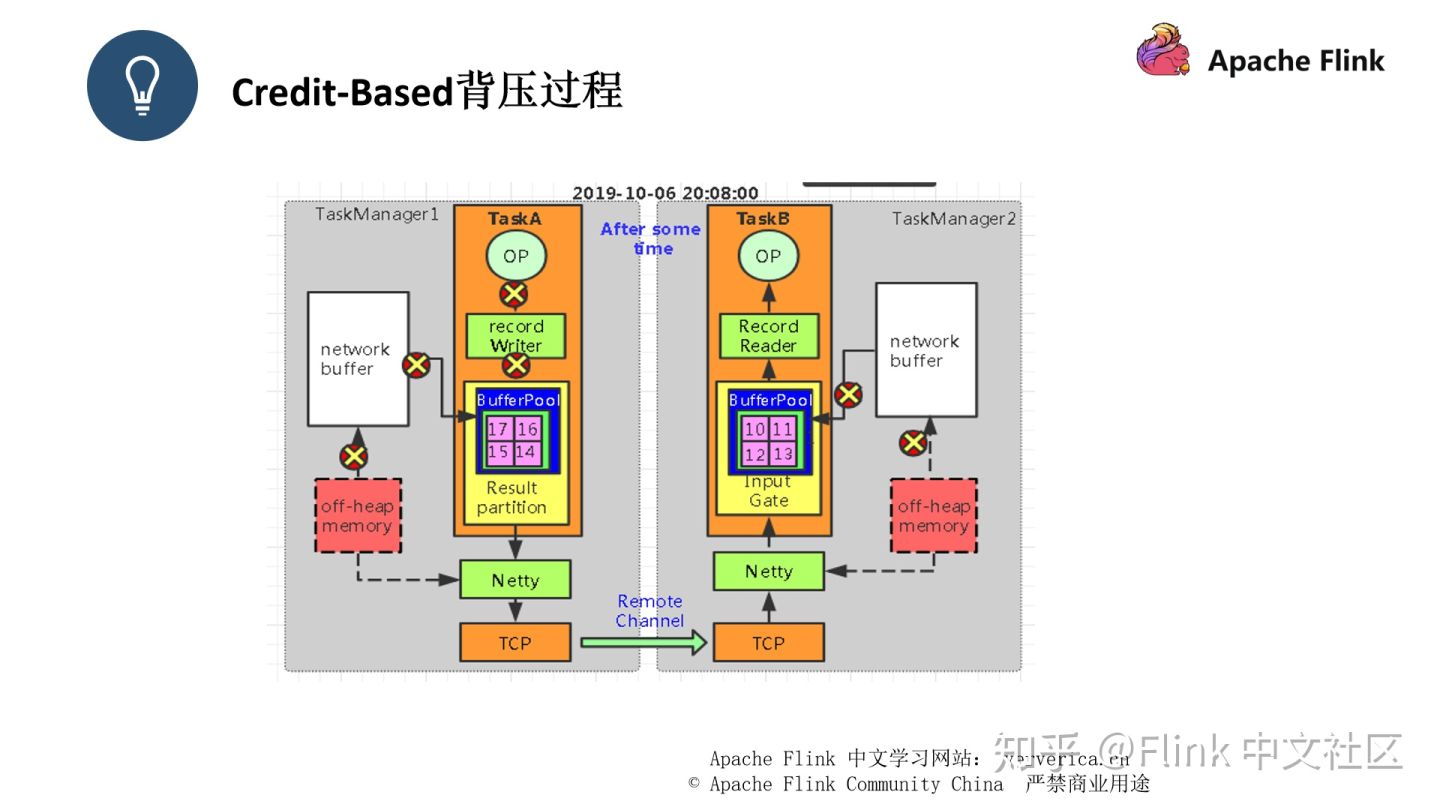


此时 event1 被 TaskB 接收后，TaskB会返回一个 ack 给 TaskA，同时返回一个credit = 3，这个是告知 TaskA 它还能接收多少条数据，Flink 就是通过这种互相告知的方式，来让生产者和消费者都能感知到对方的状态。

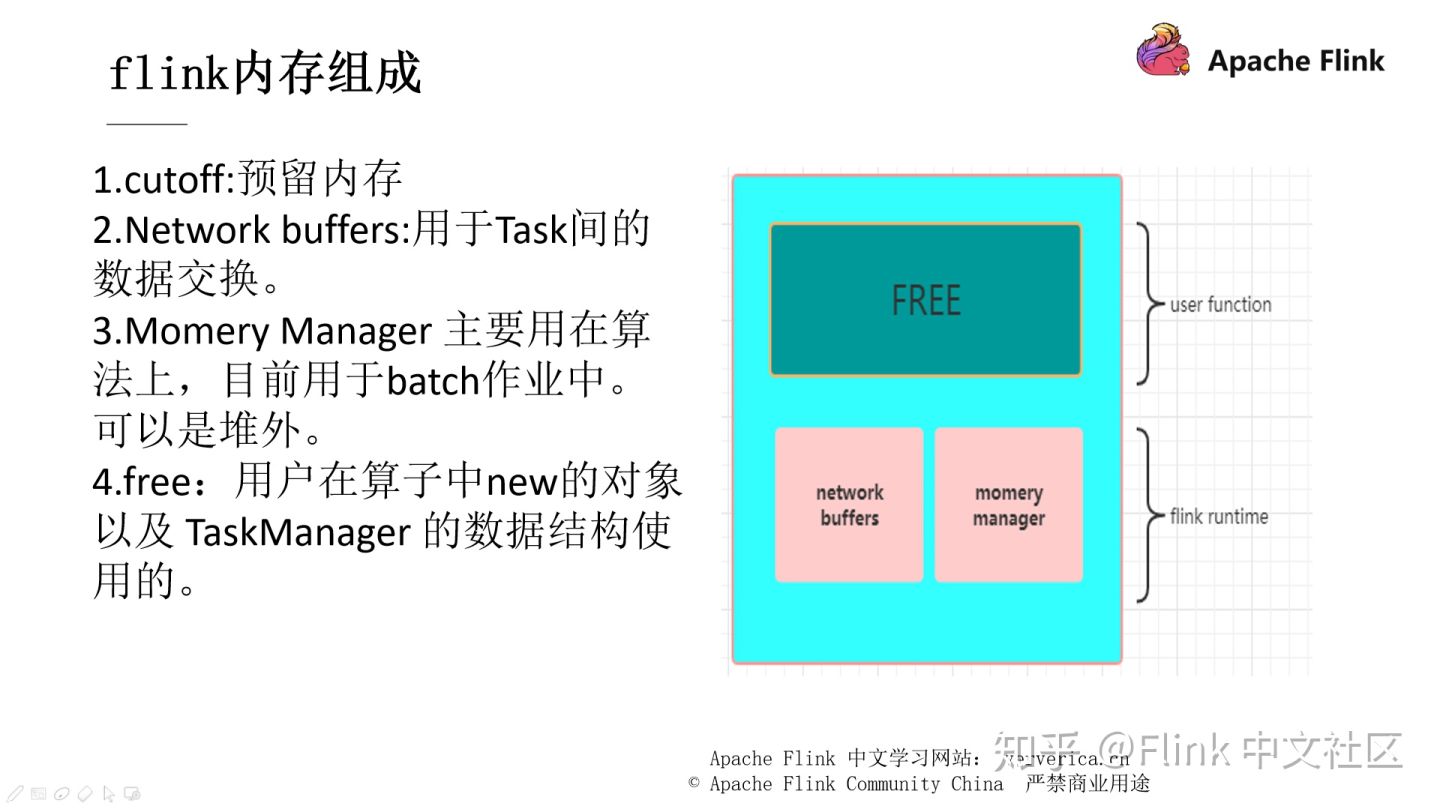


此时经过一段时间之后，TaskB中的有界 buffer 已经满了，此时 TaskB回复 credit = 0 给 TaskA，此时 channel 通道将会停止工作，TaskA 不再将数据发往 TaskB。

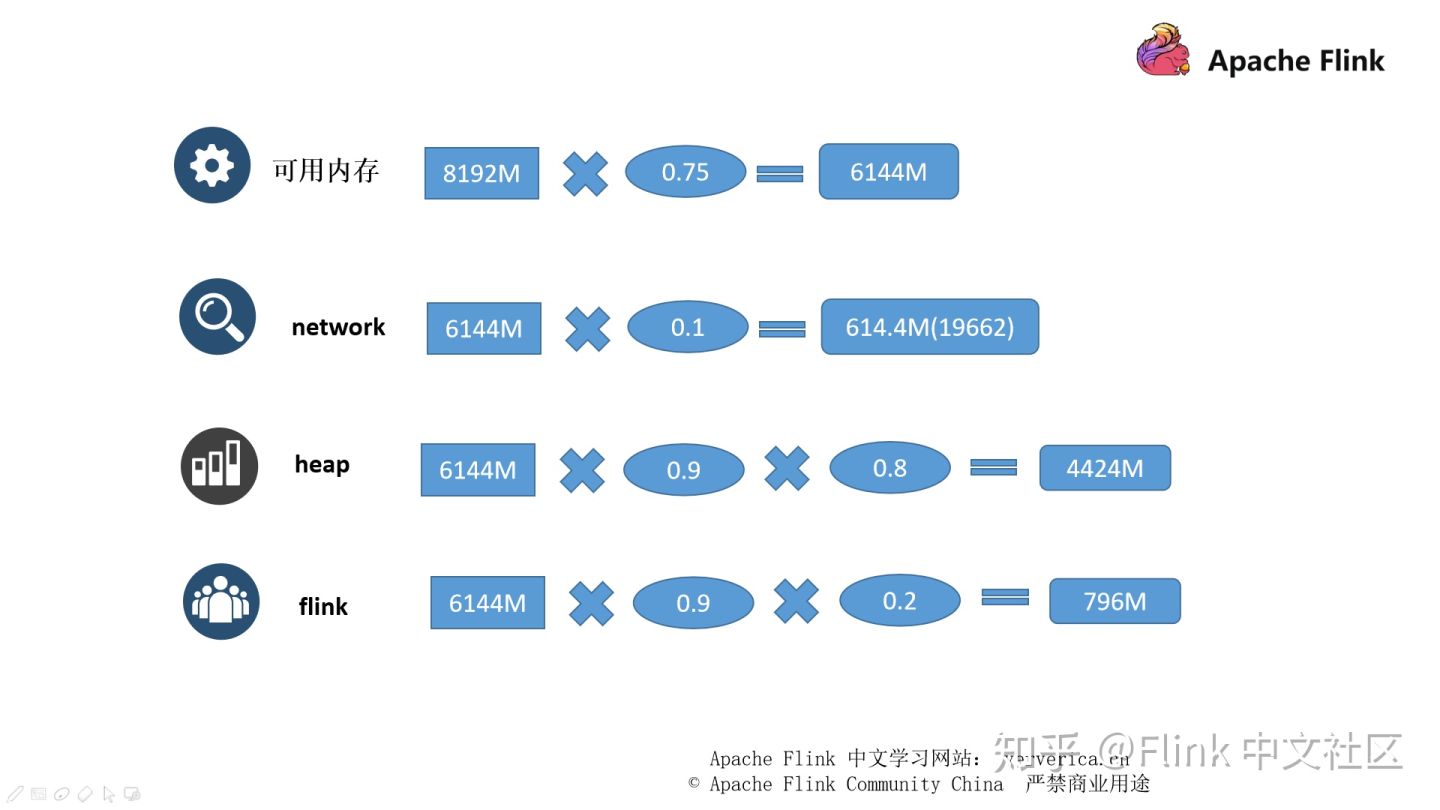




此时再经过一段时间，TaskA 中的有界 Buffer 也已经出现了数据积压，所以我们平时遇到的吞吐下降，处理延迟的问题，就是因为此时整个系统相当于一个停滞的状态，如图二示，所有的过程都被打上 “X”，表示这些过程都已经停止工作。



JVM 是一个非常复杂的系统，当其内存不足时会造成 OOM ，导致系统的崩溃。Flink 在拿到我们分配的内存之后会先分配一个 cutoff 预留内存，保证系统的安全性。Netword buffers 其实就是对应我们刚刚一直提到的有界 buffer，momery manager 是一个内存池，这部分的内存可以设置为堆内或者堆外的内存，当然在流式作业中我们一般设置其为堆外内存，而 Free 部分就是提供给用户使用的内存块。  
现在我们假设分配给此 TaskManager 的内存是 8g。



1. 首先是要砍掉 cutoff 的部分，默认是0.25，所以我们的可用内存就是 8gx0.75
2. network buffers 占用可用内存的 0.1 ，所以是 6144x0.1
3. 堆内/堆外内存为可用内存减去 network buffers 的部分，再乘以 0.8
4. 给到用户使用的内存就是堆内存剩下的 0.2 那部分

其实真实情况是 Flink 是先知道了 heap 内存的大小然后逆推出其它内存的大小。  
Flink 作业的问题定位  
1.问题定位口诀  
“**一压二查三指标，延迟吞吐是核心**。  
**时刻关注资源量 , 排查首先看GC**。”  
一压是指背压，遇到问题先看背压的情况，二查就是指 checkpoint ，对齐数据的时间是否很长，state 是否很大，这些都是和系统吞吐密切相关的，三指标就是指 Flink UI 那块的一些展示，我们的主要关注点其实就是延迟和吞吐，系统资源，还有就是 GC logs。

* 看反压：通常最后一个被压高的 subTask 的下游就是 job 的瓶颈之一。
* 看 Checkpoint 时长：Checkpoint 时长能在一定程度影响 job 的整体吞吐。
* 看核心指标：指标是对一个任务性能精准判断的依据，延迟指标和吞吐则是其中最为关键的指标。
* 资源的使用率：提高资源的利用率是最终的目的。

■ 常见的性能问题

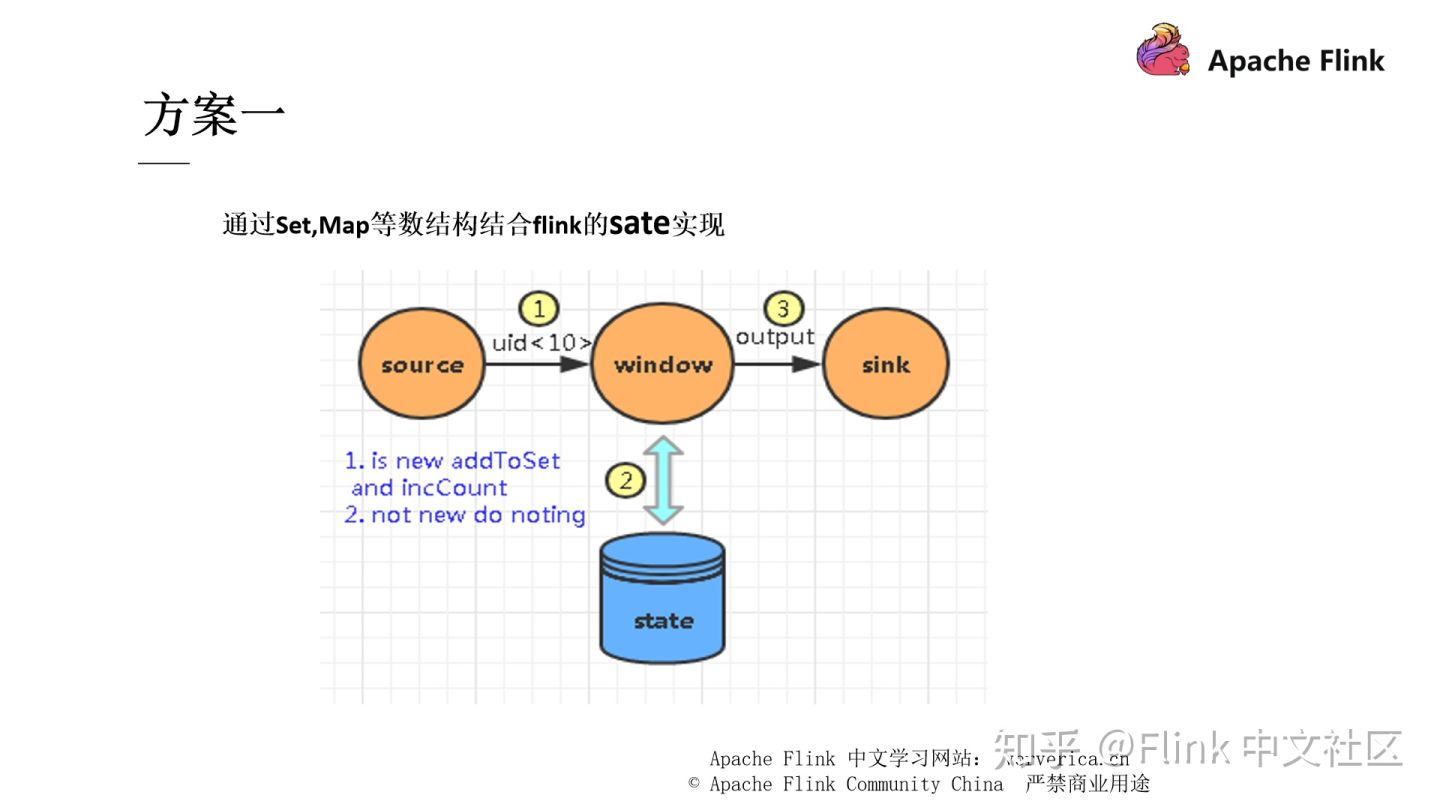


简单解释一下：

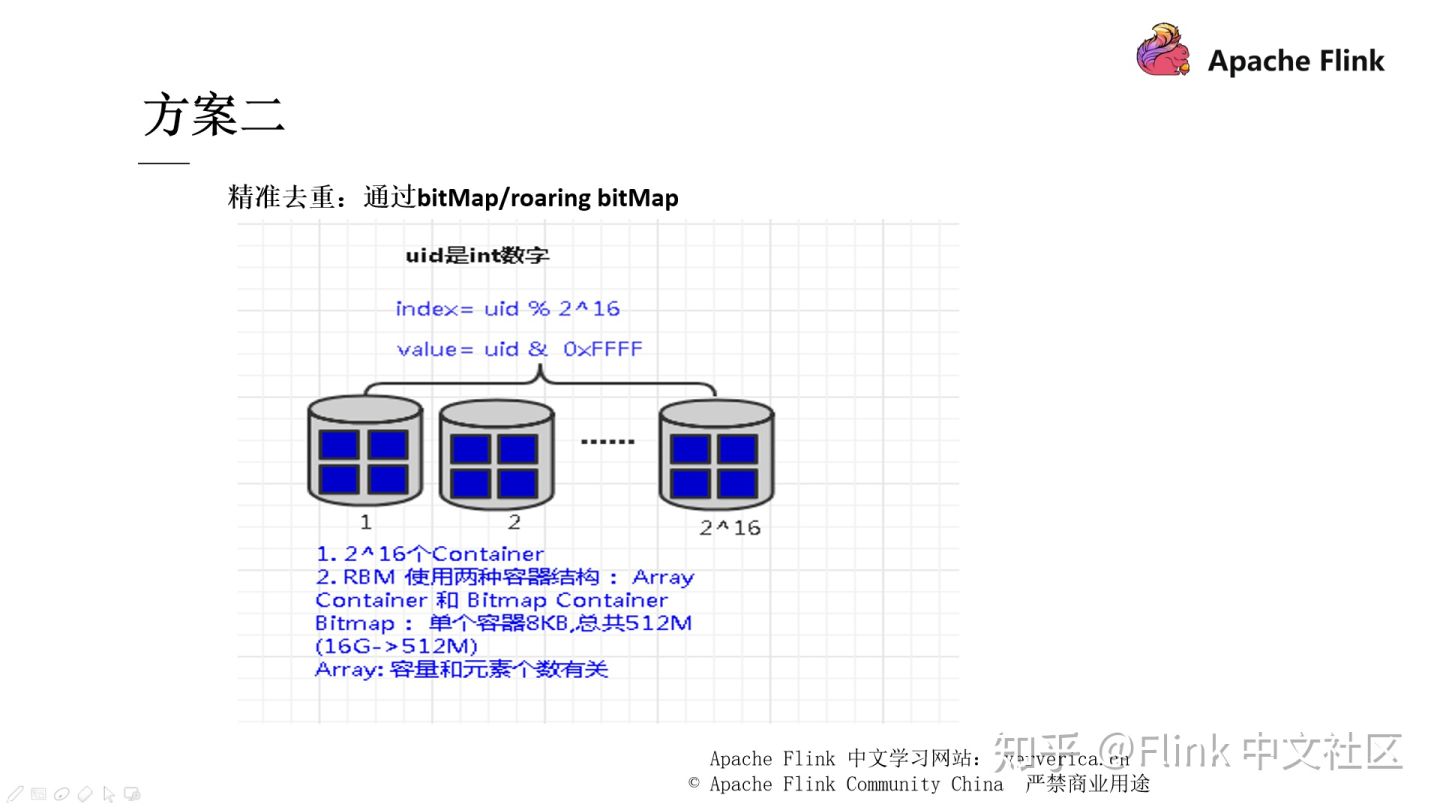
1. 在关注背压的时候大家往往忽略了数据的序列化和反序列化过程所造成的性能问题。
2. 一些数据结构，比如 HashMap 和 HashSet 这种 key 需要经过 hash 计算的数据结构，在数据量大的时候使用 keyby 进行操作， 造成的性能影响是非常大的。
3. 数据倾斜是我们的经典问题，后面再进行展开。
4. 如果我们的下游是 MySQL，HBase 这种，我们都会进行一个批处理的操作，就是让数据存储到一个 buffer 里面，在达到某些条件的时候再进行发送，这样做的目的就是减少和外部系统的交互，降低网络开销的成本。
5. 频繁 GC ，无论是 CMS 也好，G1 也好，在进行 GC 的时候，都会停止整个作业的运行，GC 时间较长还会导致 JobManager 和 TaskManager 没有办法准时发送心跳，此时 JobManager 就会认为此 TaskManager 失联，它就会另外开启一个新的 TaskManager
6. 窗口是一种可以把无限数据切割为有限数据块的手段。比如我们知道，使用滑动窗口的时候数据的重叠问题，size = 5min 虽然不属于大窗口的范畴，可是 step = 1s 代表1秒就要进行一次数据的处理，这样就会造成数据的重叠很高，数据量很大的问题。

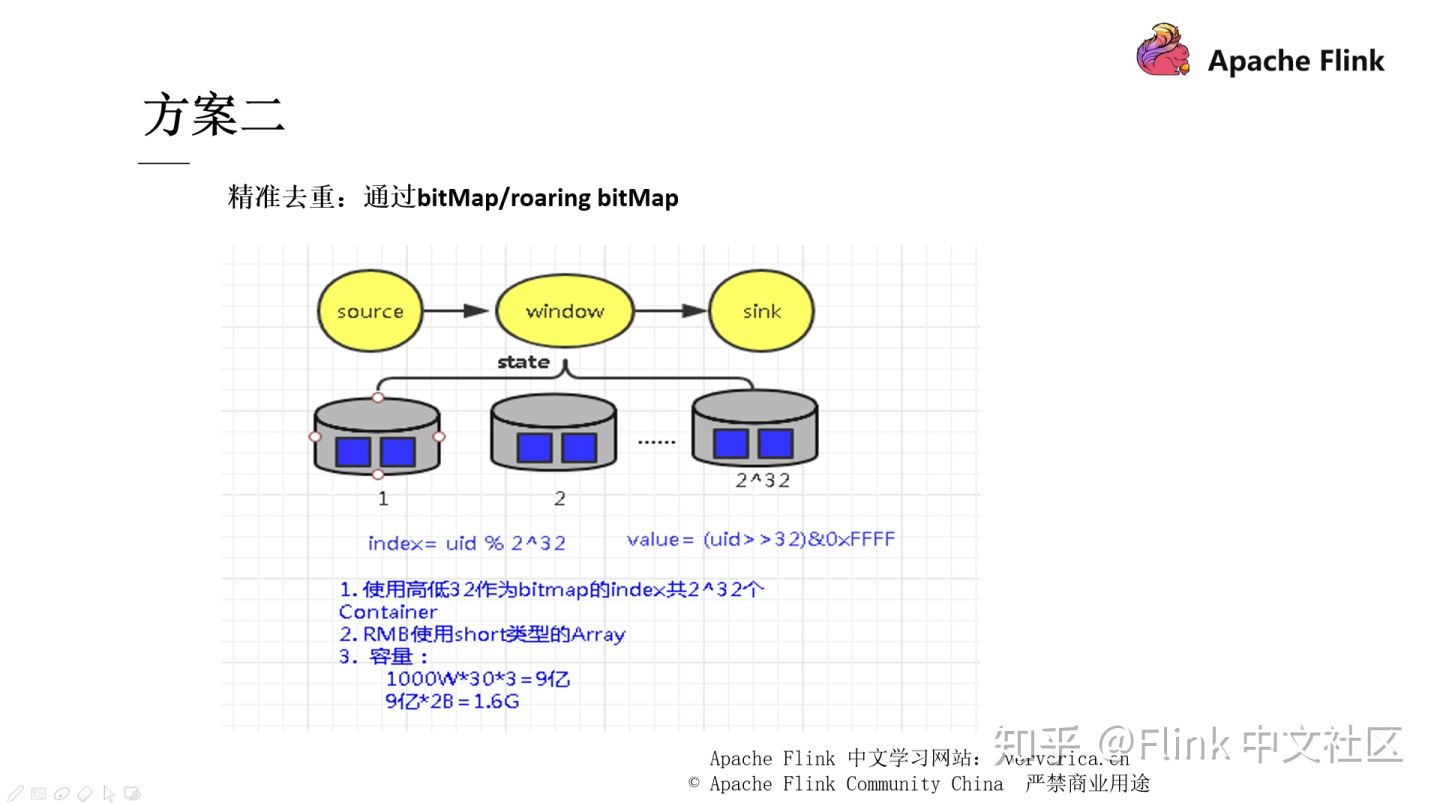
2.Flink 作业调优

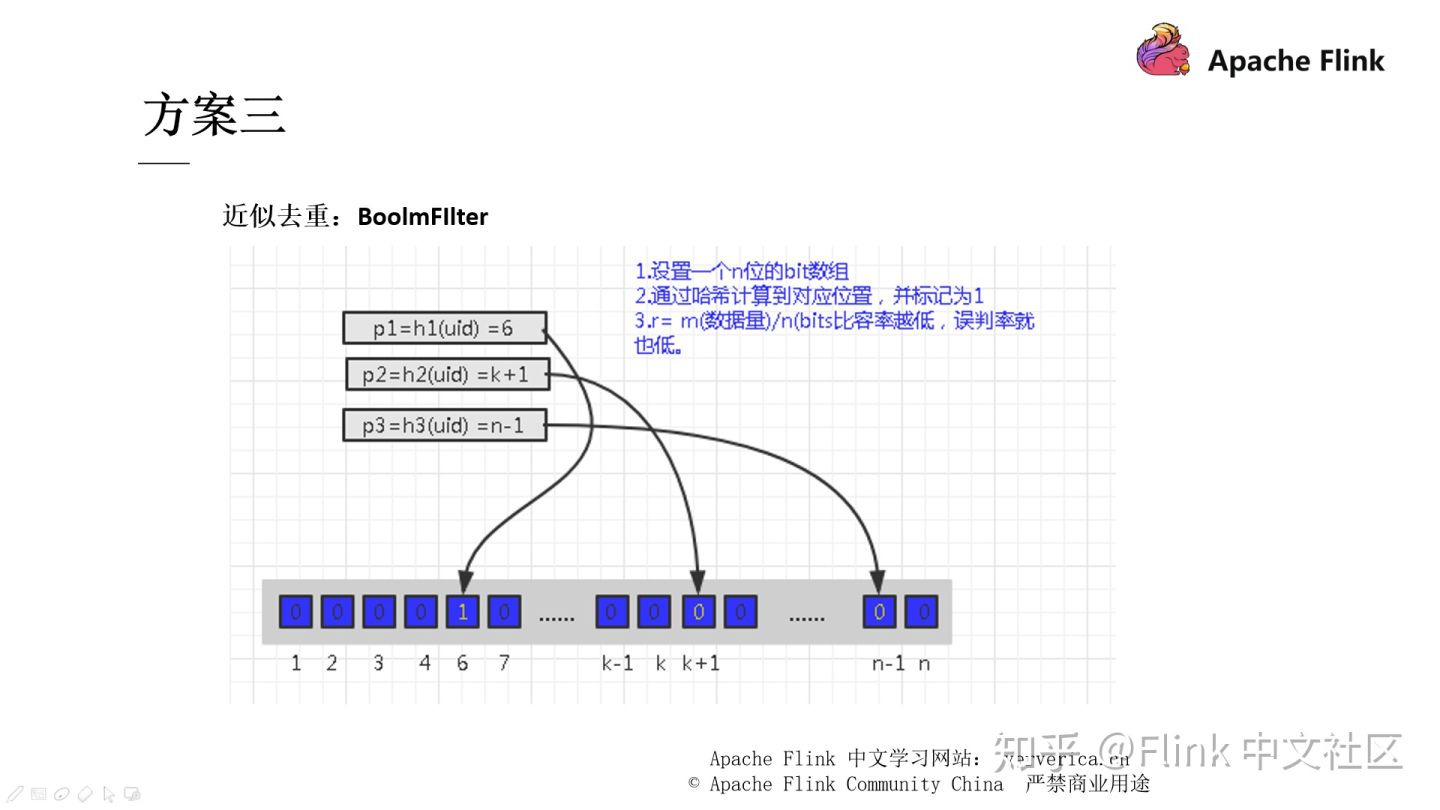


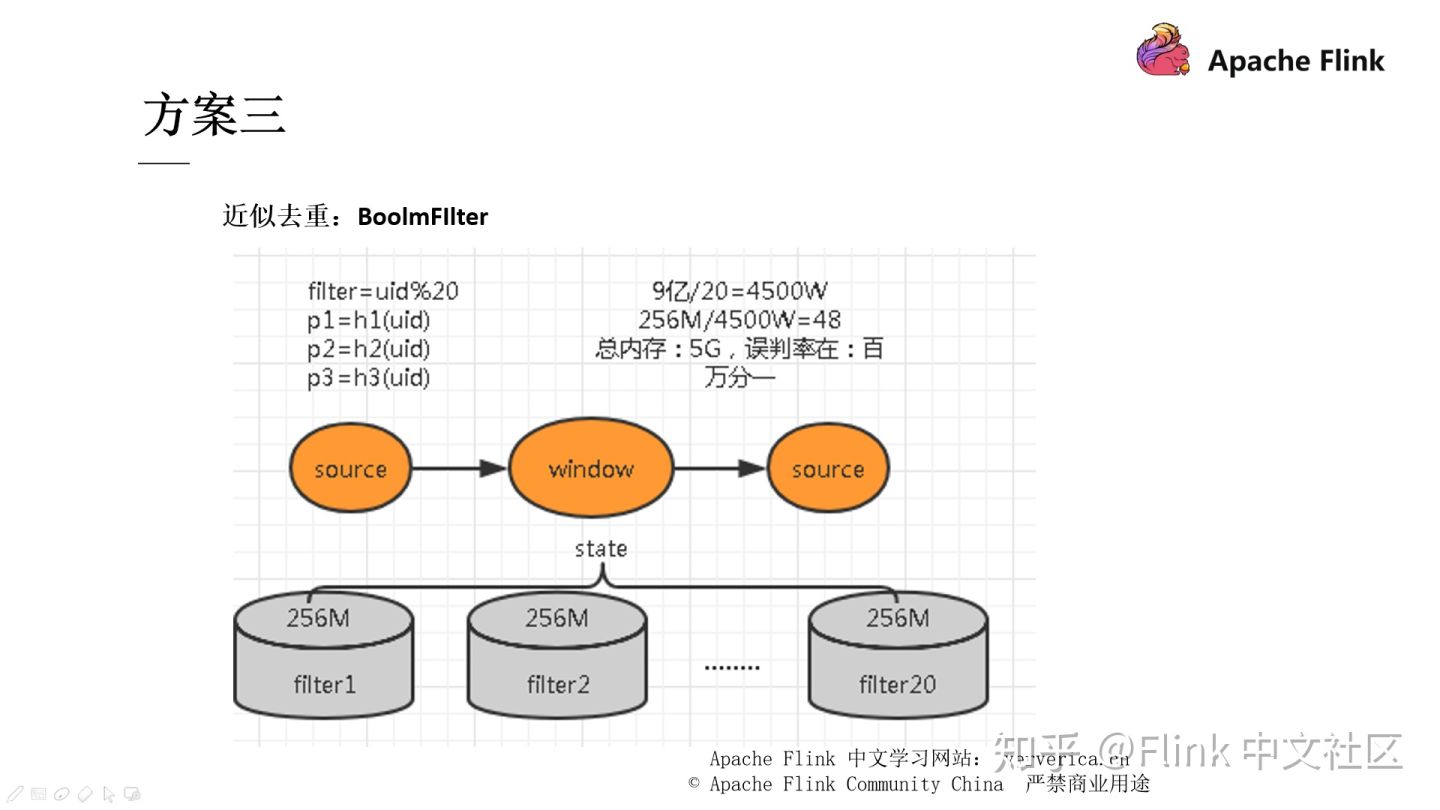


我们可以通过一些数据结构，比如 Set 或者 Map 来结合 Flink state 进行去重。但是这些去重方案会随着数据量不断增大，从而导致性能的急剧下降，比如刚刚我们分析过的 hash 冲突带来的写入性能问题，内存过大导致的 GC 问题，TaskManger 的失联问题。





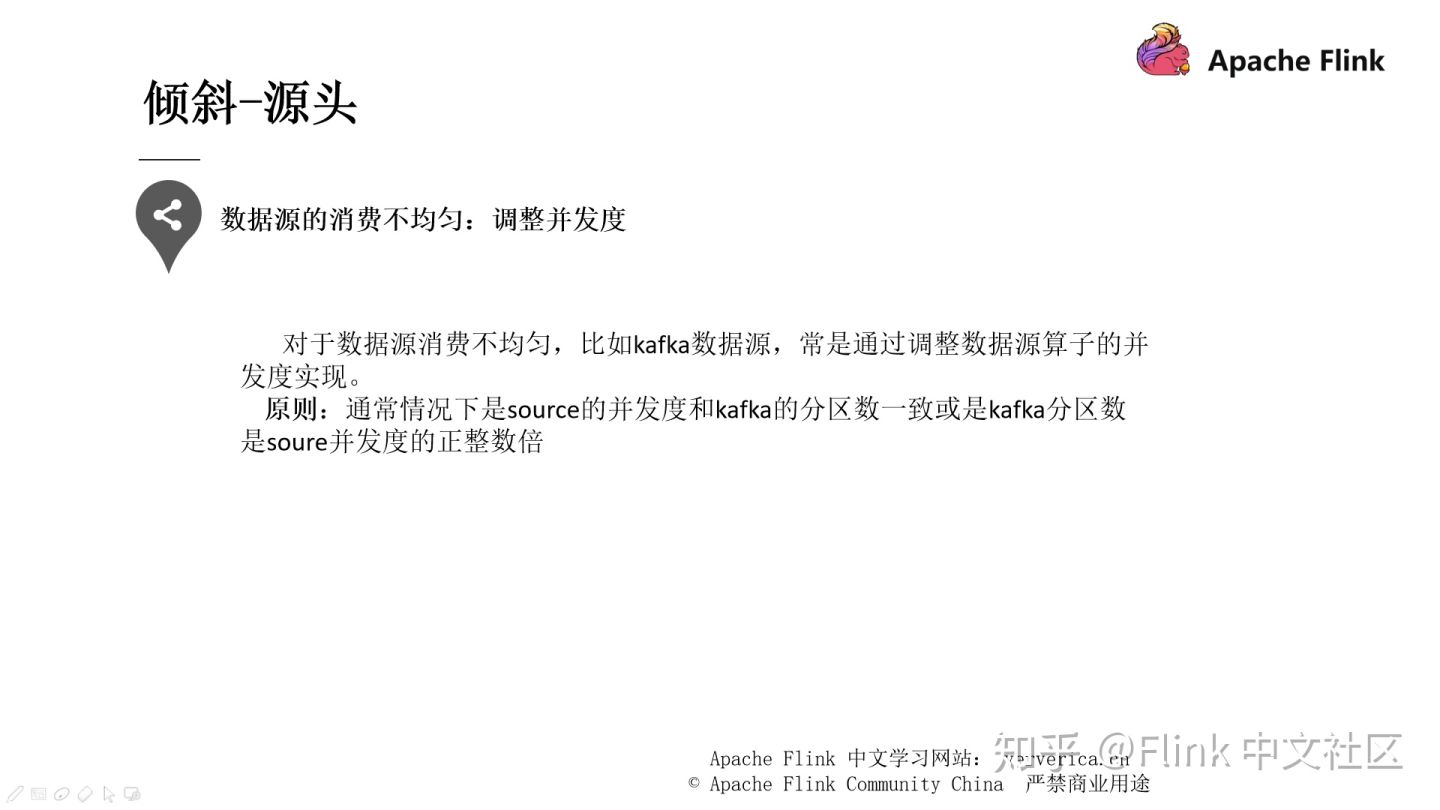




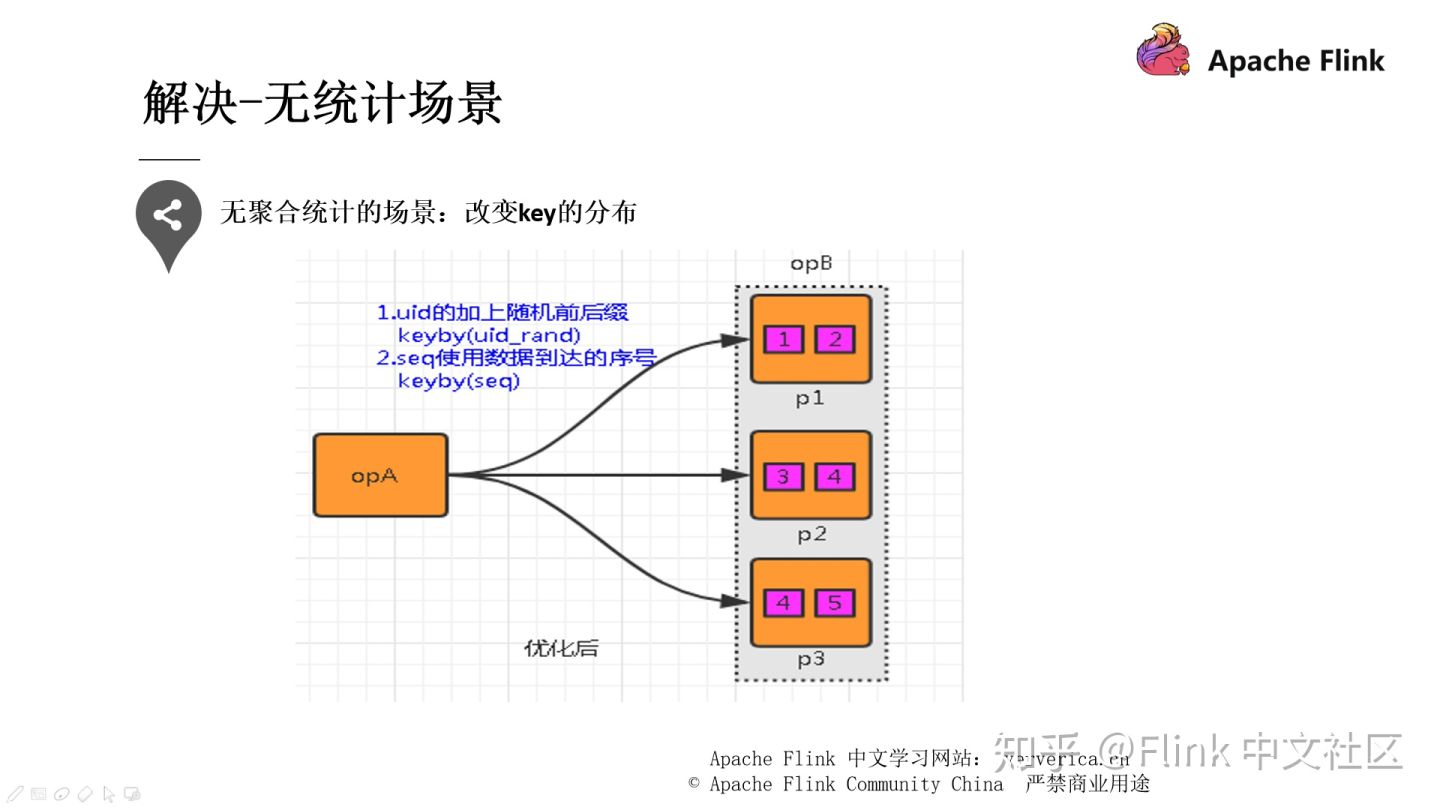
方案二和方案三也都是通过一些数据结构的手段去进行去重，有兴趣的同学可以自行下去了解，在这里不再展开。  
■ 数据倾斜



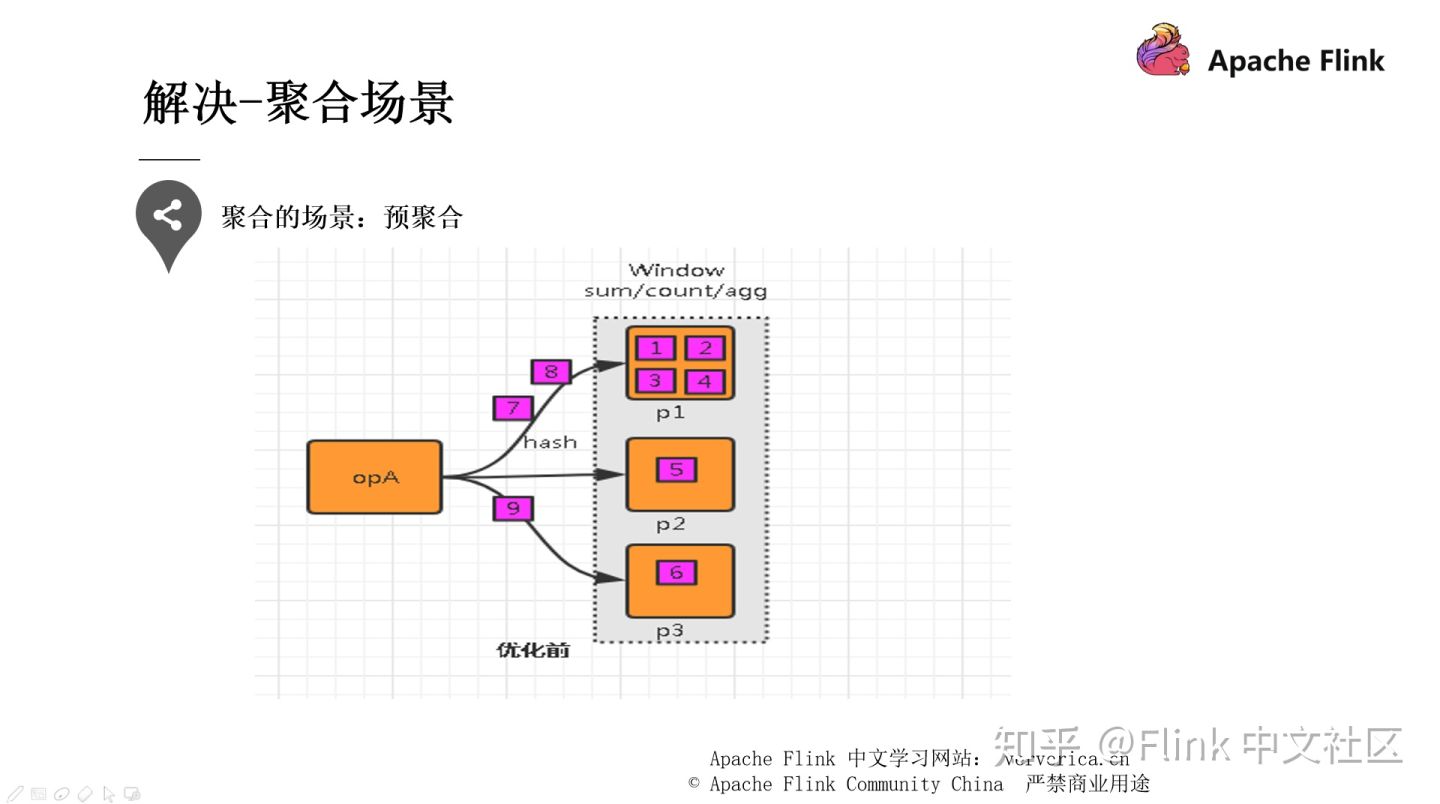
数据倾斜是大家都会遇到的高频问题，解决的方案也不少。



第一种场景是当我们的并发度设置的比分区数要低时，就会造成上面所说的消费不均匀的情况。



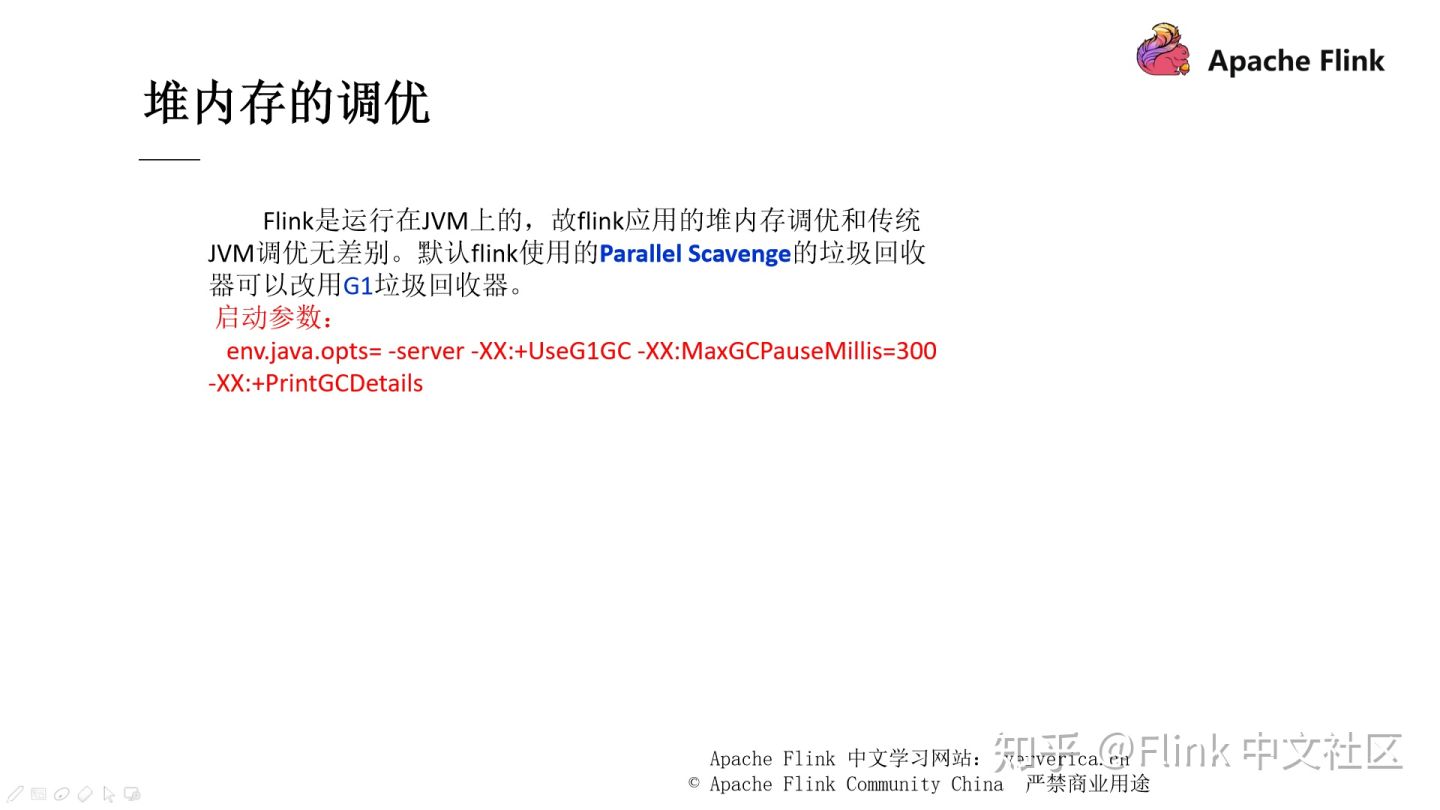
第二种提到的就是 key 分布不均匀的情况，可以通过添加随机前缀打散它们的分布，使得数据不会集中在几个 Task 中。



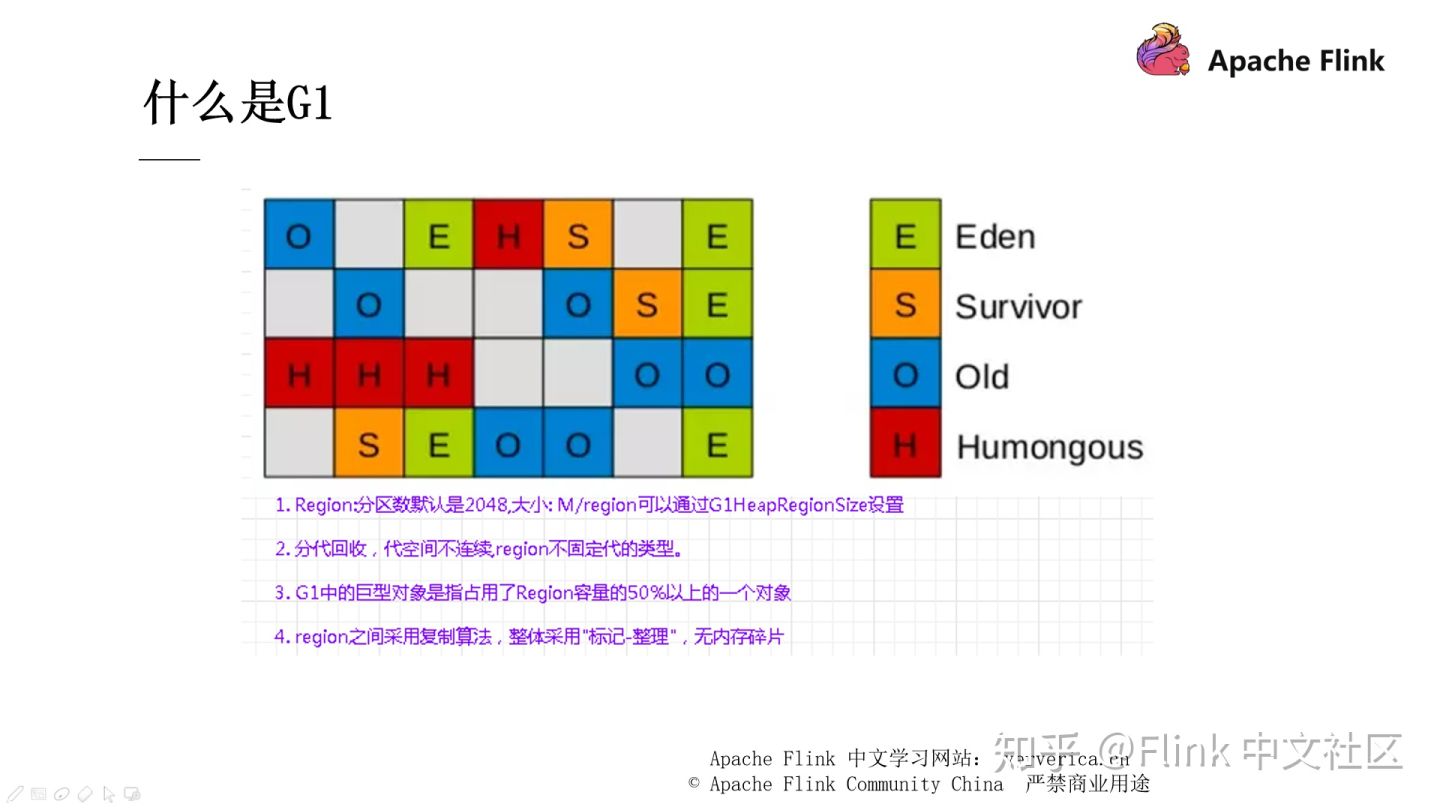
在每个节点本地对相同的 key 进行一次聚合操作，类似于 MapReduce 中的本地 combiner。map-side 预聚合之后，每个节点本地就只会有一条相同的 key，因为多条相同的 key 都被聚合起来了。其他节点在拉取所有节点上的相同 key 时，就会大大减少需要拉取的数据数量，从而也就减少了磁盘 IO 以及网络传输开销。  
■ 内存调优  
Flink 的内存结构刚刚我们已经提及到了，所以我们清楚，调优的方面主要是针对 非堆内存 Network buffer ，manager pool 和堆内存的调优，这些基本都是通过参数来进行控制的。

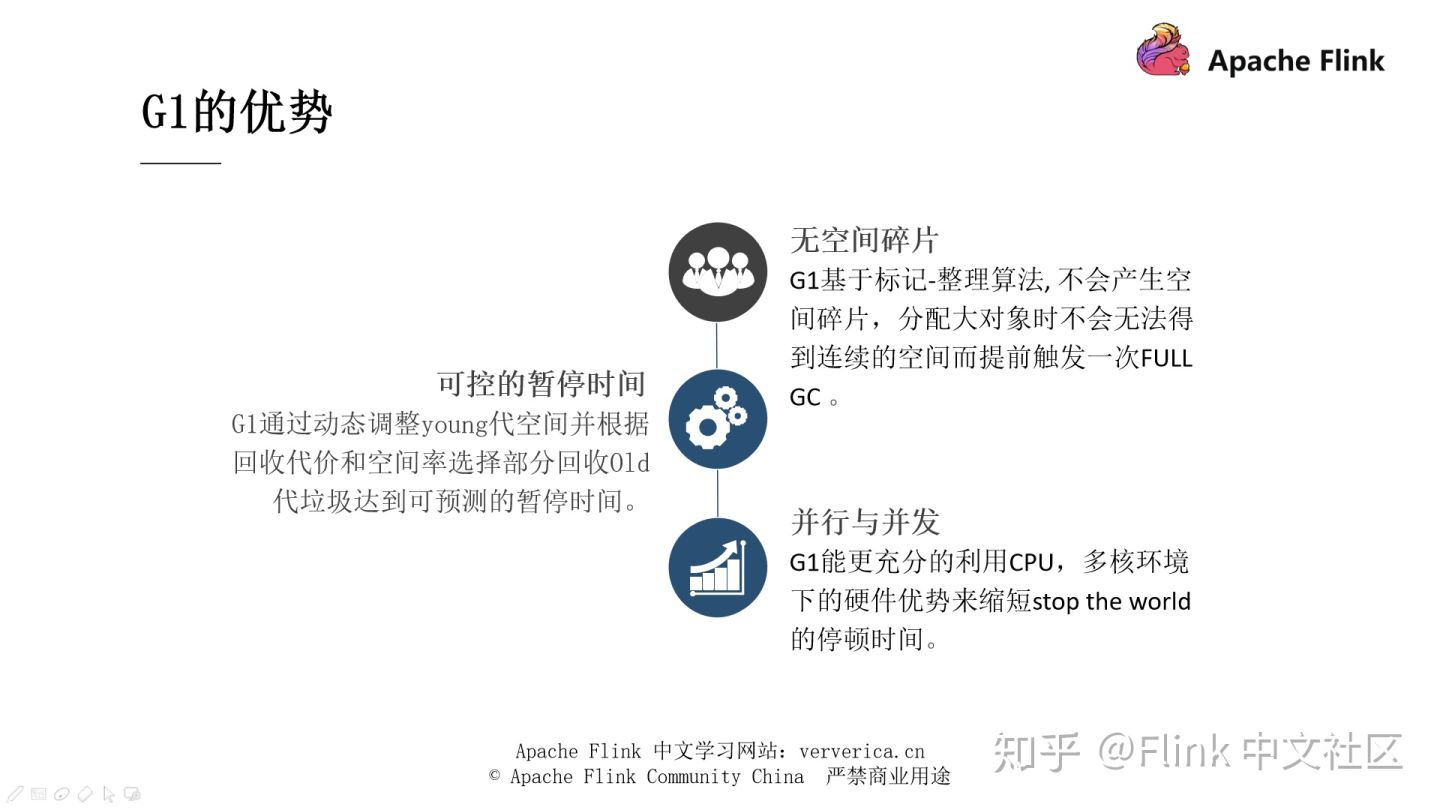


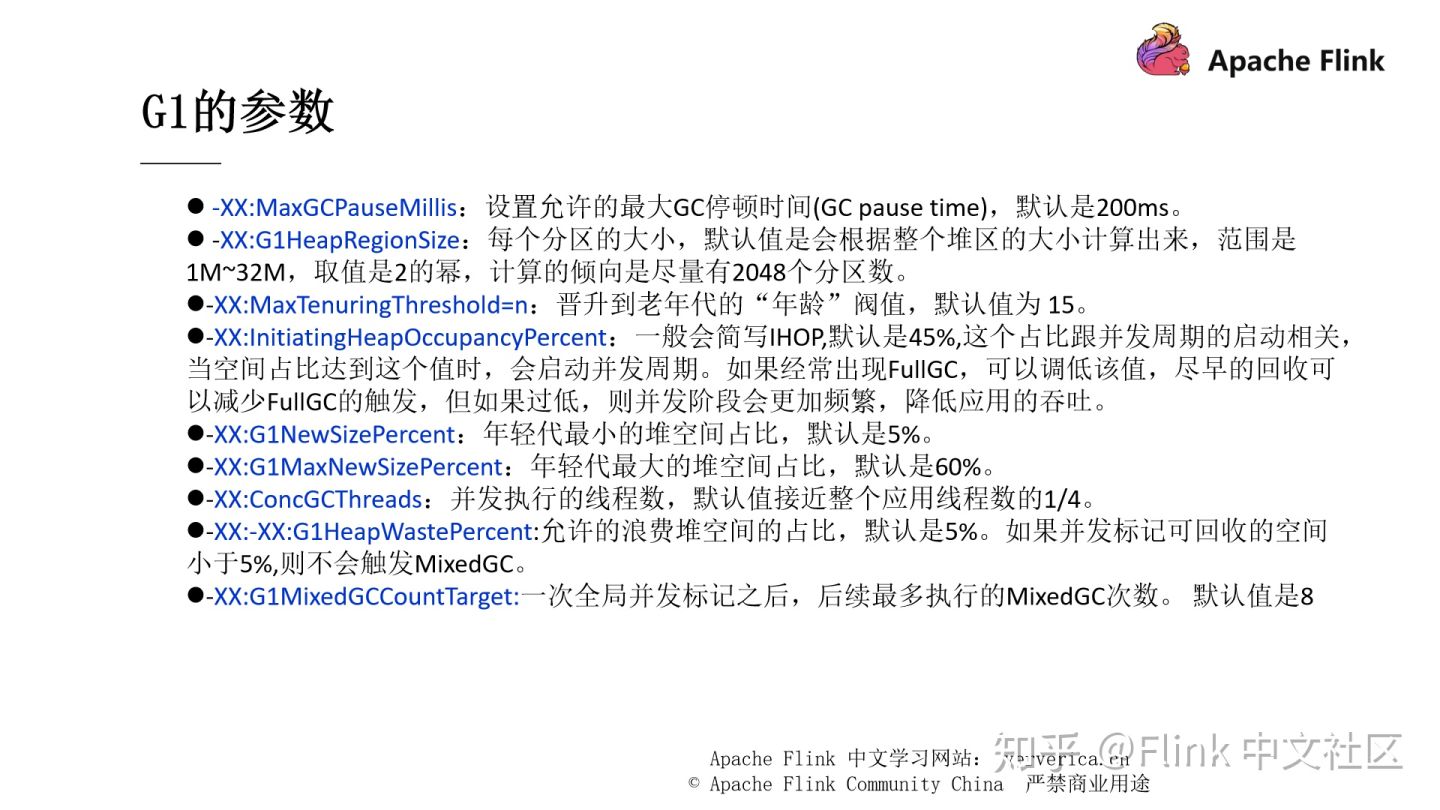
这些参数我们都需要结合自身的情况去进行调整，这里只给出一些建议。而且对于 ManagerBuffer 来说，Flink 的流式作业现在并没有过多使用到这部分的内存，所以我们都会设置得比较小，不超过0.3。



堆内存的调优是关于 JVM 方面的，主要就是将默认使用的垃圾回收器改为 G1 ，因为默认使用的 Parallel Scavenge 对于老年代的 GC 存在一个串行化的问题，它的 Full GC 耗时较长，下面是关于 G1 的一些介绍，网上资料也非常多，这里就不展开说明了。







总 结  
本文带大家了解了 Flink 的 CheckPoint 机制，反压机制及 Flink 的内存模型和基于内存模型分析了一些调优的策略。希望能对大家有所帮助，原文分享的视频回顾可移步下方链接：  
[https://ververica.cn/developers/flink-training-course-operation/](https://link.zhihu.com/?target=https%3A//ververica.cn/developers/flink-training-course-operation/)

消息中间件 存储 分布式计算 网络协议 Java 关系型数据库 Kafka 数据处理 流计算 容器