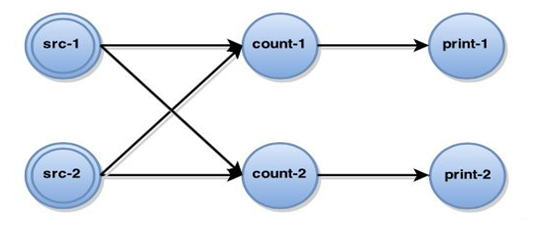
# 分布式数据流的异步快照(Flink的核心)

当我们有一个数据处理流中的各个处理节点，需要保持自己的状态的时候（比如接受一个消息之后，就根据消息更新自己的状态，比如消息记数），怎么保证node的自动failure recovery 的时候，不会造成各个节点的状态不统一从而影响后边的处理呢？

比如上图，当所有的src不再发出消息，那么最终count-1和count-2的计数必须是一样的，且print是只增的。然而当count-1节点挂掉，然后重启，怎么保证它和count-2一致呢？保证print-1和print-2最终会打出一样的数字呢？且保证print只增呢？

一种方法是所有节点记住所有发出的信息，失败重发，下游记住所有收到的消息和所有每个消息所导致的最新状态，failover之后要求所有上游从失败前的消息开始重发，且，只能重发给失败的节点(也就是src-1和src-2要记住不能给count-2重发)，且count-1要记住所有这些重发的消息不能导致print打出老的数字(否则违反只增性)，而重发结束就要立刻开始给print发出应有的消息。当系统内节点非常多和复杂的时候，记录整个图的消息流动会非常复杂和导致high cost。

Lightweight Asynchronous Snapshots这篇论文提供的算法，是Flink解决Exactly Once Message处理的核心算法。它的优点是不需要所有计算节点记住所有发出的信息。**只需要数据源可以replay就行了**，超级轻量级。【对比财大气粗的有完美infrastructure支持的Google DataFlow/MillWheel, 记录了所有Shuffle(因为一般shuffle才需要跨机器节点通信)的record在BigTable或者Spanner里，用一个保证consistent的垃圾回收算法来清理无限增长的record。回头讲MillWheel论文的时候会详细讲】

MillWheel [18] offers a complete end-to-end solution to processing guarantees, similarly to Flink. However, its heavy transactional nature, idempotency constraints and strong dependence on a high-throughput, always-available, replicated data store [28] makes this approach infeasible in commodity deployments.   
---- Flink的一篇论文对MillWheel的评价

# 单向图算法(无环)

1. **所有的通信channel需要是先进先出(FIFO)按顺序的。**
2. **中心coordinator：**需要有一个中心coordinator来不断广播持续增长的stage barrier到所有的src数据流里。(比如先给所有src发1，然后5秒后发2，10秒发3..... 如此增长)。
3. **数据源src**，当数据源收到第n个barrier的时候：

3.1 -保存状态，保证当需要replay从任意n开始的消息时，可以replay在自己收到barrier-n之后的所有消息。

3.2 -广播barrier给下游。

1. **中间处理节点或最终叶子节点**：假设一个中间处理节点或最终叶子节点需要m个input流，当在某个input流收到barrier-n的时候，

4.1 -block这个input流保证不再收取和处理。

* 1. -当收到所有m个input的barrier-n的时候，

4.2.1 --**Pi-LocalSnapshot-n**: 保存本地状态(take local snapshot n), 保证可以从这个状态恢复(比如存到云端, 在另外x台机器做replica)，假设我们每个logic processor都有一个id为Pi，那么每个logic processor的在收到所有inputs的barrier-n之后所保存的本地状态快照则设为**Pi-LocalSnapshot-n。**

* + 1. --向自己的下游广播barrier-n（如果是叶子节点没有下游，那么不需要广播）。

1. **CompleteGlobalSnapshot-n:**当所有的节点(源，中间处理，叶子节点)都处理完barrier-n且完成取快照(take snapshot)的任务之后，我们说我们有了一个完整的全局快照。这意味着我们的deterministic的进度，进步到了barrier-n。

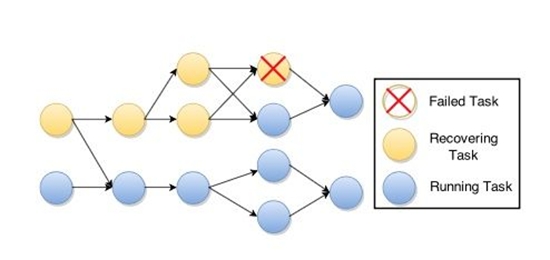
# 单向图Failover Recovery

当需要任意节点挂掉，我们从最近的**Complete Global Snapshot-n，**来重启整个系统；即，健康的节点rollback自己的状态到“接收到barrier-n时候所取的状态快照。fail掉的节点的逻辑Processor Pi被jobManager之类的东西，用自己在的**Pi-LocalSnapshot-n**重启设置本地状态之后，才开始接收上游的消息。

# 为什么这样就consistent了？

可以看到当failover的时候，全部节点的状态都回退到了barrier-n之前的数据源message所导致的全网状态，就好像数据源在barrier-n之后根本没有发过消息一样。不断发出的barrier就好像逻辑时钟一样，然而“时间”流动到不同地方的速度不同，只有当一个时间“点”全部流动到了全网，且全网把这个时间“点”的状态全部取了快照（注意当网络很大，最后一个节点取完快照，初始节点可能已经前进到n+5，n+10了，但是由于最后一个节点才刚取完快照，**CompleteGlobalSnapshot-n**只到n，n是全局consistent的记录点），

# 如何可以不用全网rollback？

如果没挂的节点的运算可以自动忽略老的已经处理过的消息(或者说replay导致的消息)，那么我们只需要重启所有从源到fail掉的节点的这条线即可(比如下图的黄色节点)。

# 带环图算法Failover

1. 当一个节点是环的msg流动的起点时(或者说这个节点正好同时是环的起点和终点)，它必定有一个input channel是来自自己的downstream节点的。
2. 这个节点不能像其他节点一样，等待所有的input channel的barrier到来，才take snapshot且广播barrier，因为它有一个或多个input channel的消息是被自己往“下游”发的消息所引发的。如果它自己不向下游广播barrier，那么这些回环input channel永远也不会有barrier发来，那么算法会永久等待。
3. 所以这个这个节点**只需要等待所有非回环input channel的barrier**到了，它就知道所有可能的barrier都到齐了，那么它就可以take本地snapshot且往“下游”广播barrier了（从而造成barrier会通过回路再次抵达这个节点）。
4. **重点:**此节点take完本地snapshot之后，需要记录所有回环input channel的msg到log里，直到从此回环input channel收到自己发出的barrier，当所有回环input channel都收到barrier-n. 此时**在Step3 take的本地snapshot，加上所有回环input channel的msg log一起，成为此节点在barrier-n的本地snapshot。**
5. Failover，failover的时候，除了从本地snapshot恢复状态之外，还需要replay所有input channel的msg。

# 简单的解释“为什么”

* 一个分布式系统的snapshot可以理解为时间静止时，系统的各个节点的状态和他们之间的channel的状态(channel里的msg list)，就是一个stream集合和table集合的剪影。stream指还在channel里没有融入到table里的msg list，tables可以理解为各个节点的本地状态。（注：Directed Cyclic Graph才要加msg list，即log）
* 从src开始，当src take一个snapshot之后，任何被src在snapshoting之前发出过的"历史msg"所引起的“蝴蝶效应”都必须被下游记录，才能构成完整的整个系统的consistent global snapshot,**难点在于, 我们无法时间静止，所以src在不知道下游什么状况的情况下，还是要继续往下游发msg，那么对所有下游节点，区分那些是“历史msg”和“历史msg引起的蝴蝶效应msg”，还是“new event”, "new msg引起的蝴蝶效应msg"是难点；**任何被src的eventA所因果导致的eventB,C,D,E，都必须记录，对任意系统的msg(或者说event),要么这个event已经被状态吸收，merge在某节点最终的table里(这时候这个节点有可能会因为接受到这个event而发出另外的别的event，也需要保证在下游记录)，要么这个event需要记录在这个节点的input stream log里。
* **对于环来说，所有的被"历史event“所因果导致的"发给本节点的需要记录的"未来状态"还未到来，但是已经有"new event"(比如从src来的新消息)来改本地状态了，所以不能等待回环的消息，而必须先把本地状态take snapshot了才行，作为”历史msg的因果导致的msg“，只能作为”未来event“记录在stream log里了。**
* 当然flink的算法也可以设计为，即使非回环input channel的barrier都到齐了，也不unblock input channel ，而是等待所有的回环input channel的barrier也都到齐了，才take本地snapshot，且一起unblock所有的input channel；这样就不需要维护stream log了。
* 但是这很蠢。降级为at-least-once。