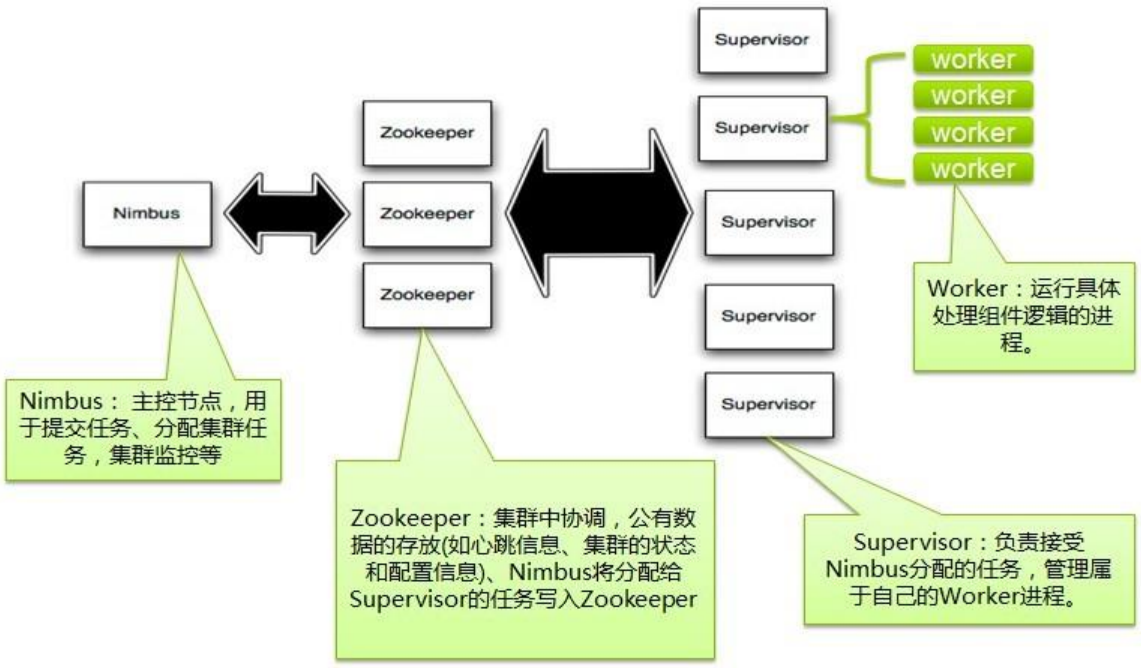
Storm 原理

Storm 的集群基本概念

	Hadoop	Storm
系统角色	JobTracker	Nimbus
	TaskTracker	Supervisor
	Child	Worker
应用名称	Job	Topology
组件接口	Mapper/Reducer	Spout/Bolt

Nimbus：负责资源分配和任务调度。
Supervisor：负责接受 nimbus 分配的任务，启动和停止属于自己管理的 worker 进程。
Worker：运行具体处理组件逻辑的进程。
Task：worker 中每一个 spout/bolt 的线程称为一个 task. 在 storm0.8 之后，task 不再与物理线程对应，同一个 spout/bolt 的 task 可能会共享一个物理线程，该线程称为 executor。



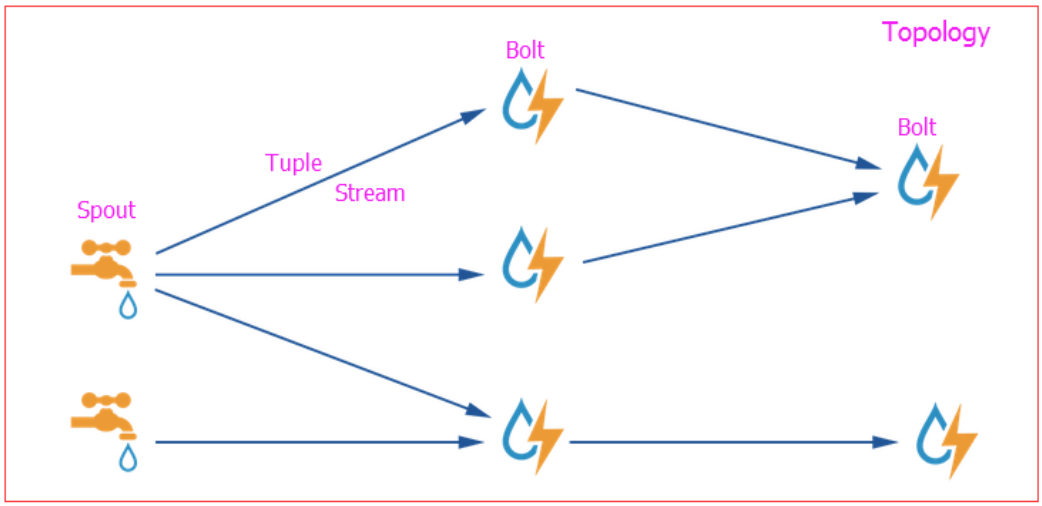
工作原理

Nimbus 负责在集群分发的代码，topology 只能在 nimbus 机器上提交，将任务分配给其他机器，和故障监测。

Supervisor，监听分配给它的节点，根据 Nimbus 的委派在必要时启动和关闭工作进程。每个工作进程执行 topology 的一个子集。一个运行中的 topology 由很多运行在很多机器上的工作进程组成。

Nimbus 和 Supervisor 之间的所有协调工作都是通过一个 Zookeeper 集群来完成。并且，nimbus 进程和 supervisor 都是快速失败（fail-fast)和无状态的。所有的状态要么在 Zookeeper 里面， 要么在本地磁盘上。这也就意味着你可以用 kill -9 来杀死 nimbus 和 supervisor 进程， 然后再重启它们，它们可以继续工作， 就好像什么都没有发生过似的。这个设计使得 storm 不可思议的稳定。

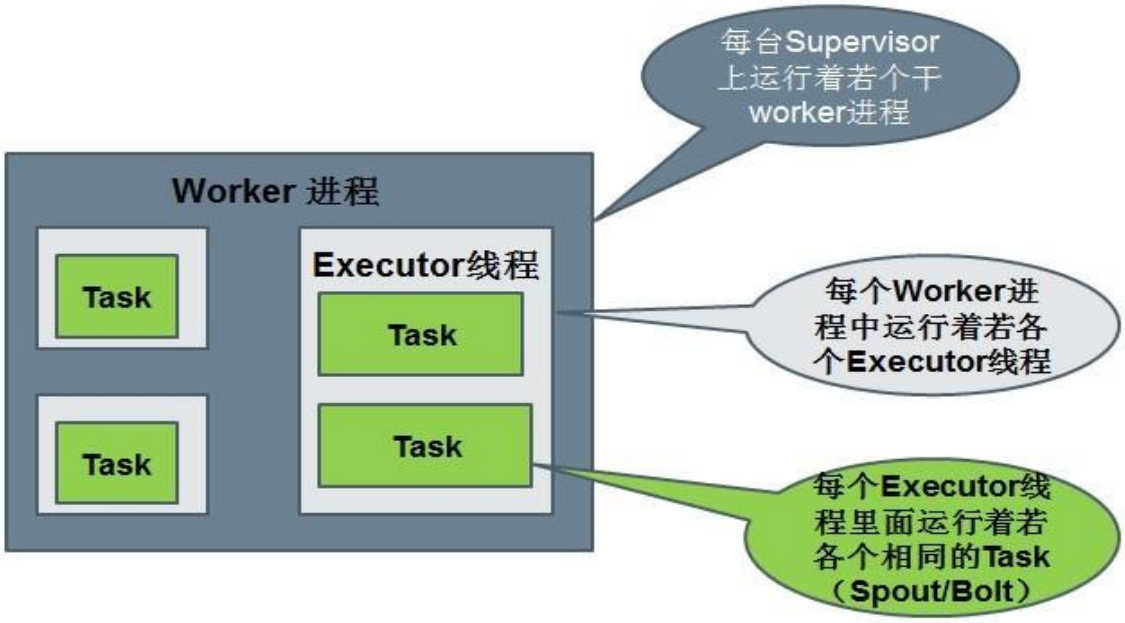
数据处理模型



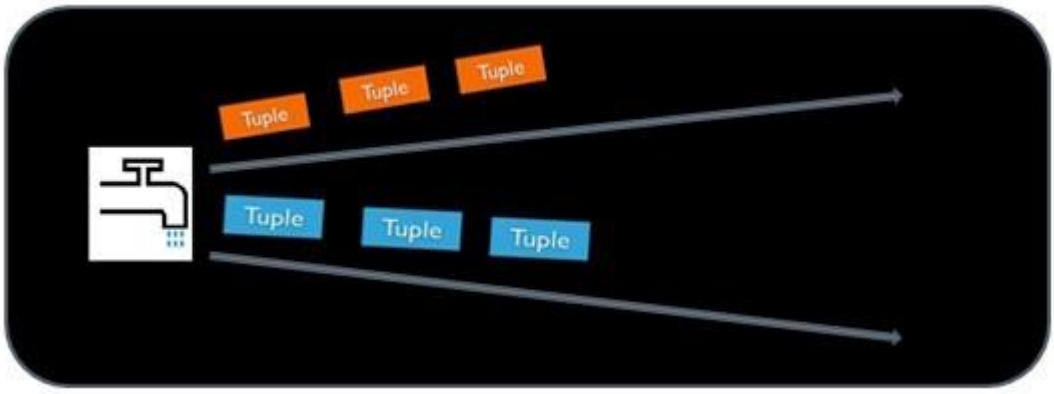
Topology：storm 中运行的一个实时应用程序，因为各个组件间的消息流动形成逻辑上的一个拓扑结构。

运行中的 Topology 主要由以下三个组件组成的：

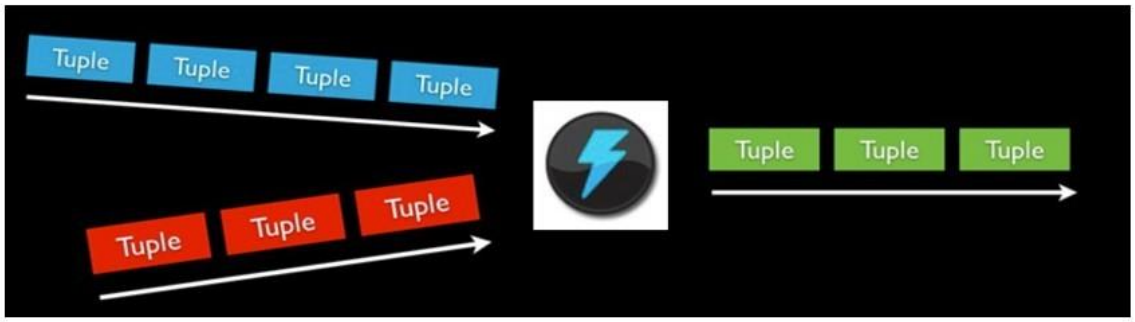
Worker processes（进程）
Executors (threads)（线程）
Tasks



Spout：在一个 topology 中产生源数据流的组件。通常情况下 spout 会从外部数据源中读取数据，然后转换为 topology 内部的源数据。Spout 是一个主动的角色，其接口中有个 nextTuple()函数，storm 框架会不停地调用此函数，用户只要在其中生成源数据即可。



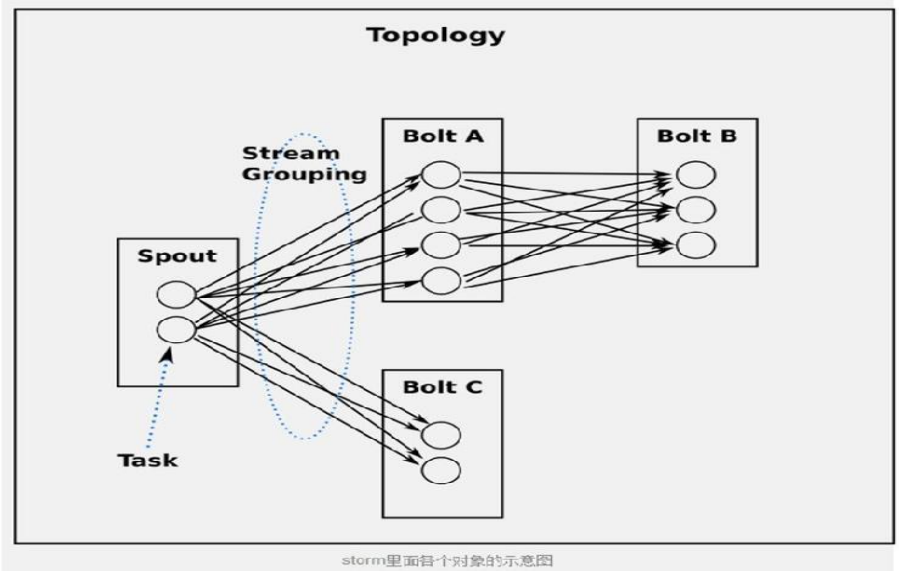
Bolt：在一个 topology 中接受数据然后执行处理的组件。Bolt 可以执行过滤、函数操作、合并、写数据库等任何操作。Bolt 是一个被动的角色，其接口中有个 execute(Tuple input)函数,在接受到消息后会调用此函数，用户可以在其中执行自己想要的操作。



Tuple：一次消息传递的基本单元。本来应该是一个 key-value 的 map，但是由于各个组件间传递的 tuple 的字段名称已经事先定义好，所以 tuple 中只要按序填入各个 value 就行了，所以就是一个 value list.

Stream：源源不断传递的 tuple 就组成了 stream。

Stream Grouping：即消息的 partition 方法。Storm 中提供若干种实用的 grouping 方式，包括 shuffle, fields hash, all, global, none, direct 和 localOrShuffle 等



storm里面各个对象的示意图

几种常用的 grouping 说明：

ShuffleGrouping: 随机选择一个 Task 来发送。

FiledGrouping: 根据 Tuple 中 Fields 来做一致性 hash，相同 hash 值的 Tuple 被发送到相同的 Task。

AllGrouping: 广播发送，将每一个 Tuple 发送到所有的 Task。

GlobalGrouping: 所有的 Tuple 会被发送到某个 Bolt 中的 id 最小的那个 Task。

NoneGrouping: 不关心 Tuple 发送给哪个 Task 来处理，等价于 ShuffleGrouping。

DirectGrouping: 直接将 Tuple 发送到指定的 Task 来处理。

Storm 记录级容错的基本原理

首先来看一下什么叫做记录级容错？storm 允许用户在 spout 中发射一个新的源 tuple 时为其指定一个 message id，这个 message id 可以是任意的 object 对象。多个源 tuple 可以共用一个 message id，表示这多个源 tuple 对用户来说是同一个消息单元。storm 中记录级容错的意思是说，storm 会告知用户每一个消息单元是否在指定时间内被完全处理了。那什么叫做完全处理呢，就是该 message id 绑定的源 tuple 及由该源 tuple 后续生成的 tuple 经过了 topology 中每一个应该到达的 bolt 的处理。举个例子。在图 4-1 中，在 spout 由 message 1 绑定的 tuple1 和 tuple2 经过了 bolt1 和 bolt2 的处理生成两个新的 tuple，并最终都流向了 bolt3。当这个过程完成处理完时，称 message 1 被完全处理了。

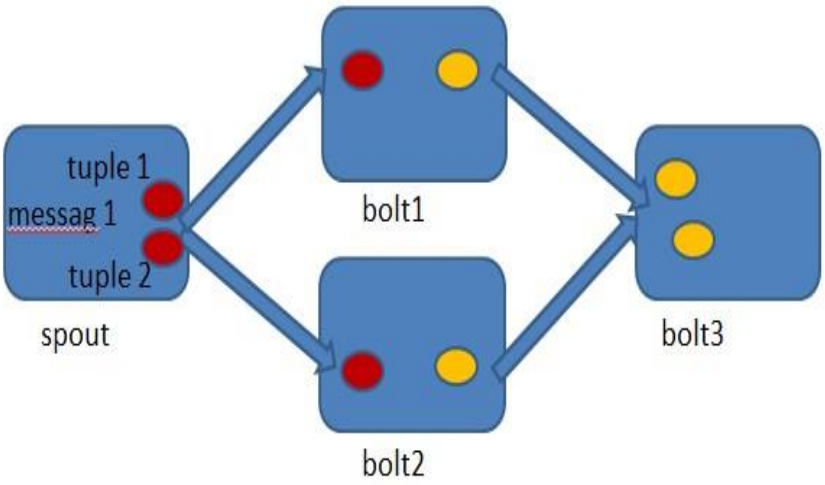


图 4-1

在 storm 的 topology 中有一个系统级组件，叫做 acker。这个 acker 的任务就是追踪从 spout 中流出来的每一个 message id 绑定的若干 tuple 的处理路径，如果在用户设置的最大超时时间内这些 tuple 没有被完全处理，那么 acker 就会告知 spout 该消息处理失败了，相反则会告知 spout 该消息处理成功了。在刚才的描述中，我们提到了”记录 tuple 的处理路径”，如果曾经尝试过这么做的同学可以仔细地思考一下这件事的复杂程度。但是 storm 中却是使用了一种非常巧妙的方法做到了。在说明这个方法之前，我们来复习一个数学定理。

A xor A = 0.

A xor B…xor B xor A = 0，其中每一个操作数出现且仅出现两次。

storm 中使用的巧妙方法就是基于这个定理。具体过程是这样的：在 spout 中系统会为用户指定的 message id 生成一个对应的 64 位整数，作为一个 root id。root id 会传递给 acker 及后续的 bolt 作为该消息单元的唯一标识。同时无论是 spout 还是 bolt 每次新生成

一个 tuple 的时候，都会赋予该 tuple 一个 64 位的整数的 id。Spout 发射完某个 message id 对应的源 tuple 之后，会告知 acker 自己发射的 root id 及生成的那些源 tuple 的 id。而 bolt 呢，每次接受到一个输入 tuple 处理完之后，也会告知 acker 自己处理的输入 tuple 的 id 及新生成的那些 tuple 的 id。Acker 只需要对这些 id 做一个简单的异或运算，就能判断出该 root id 对应的消息单元是否处理完成了。下面通过一个图示来说明这个过程。

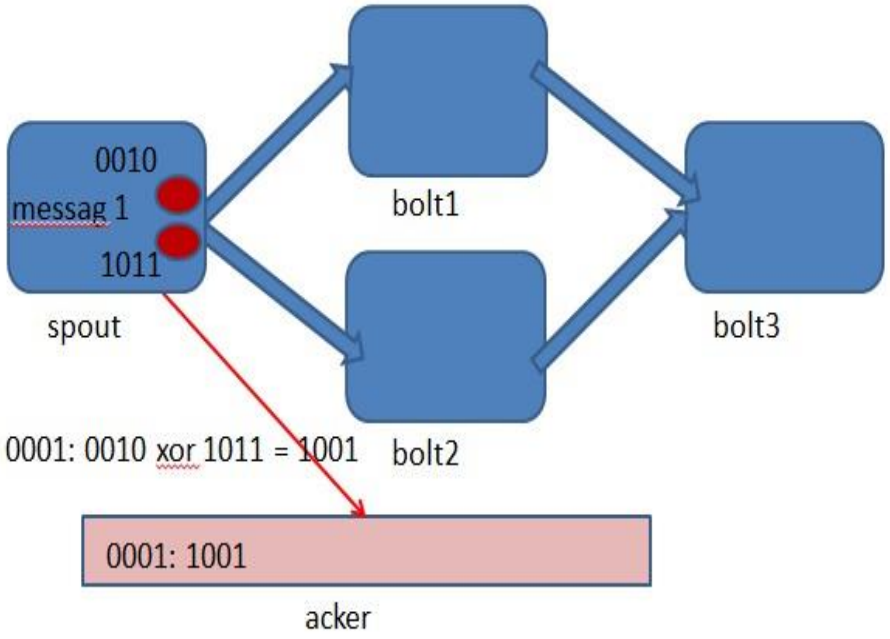


图 4-1 spout 中绑定 message 1 生成了两个源 tuple，id 分别是 0010 和 1011.

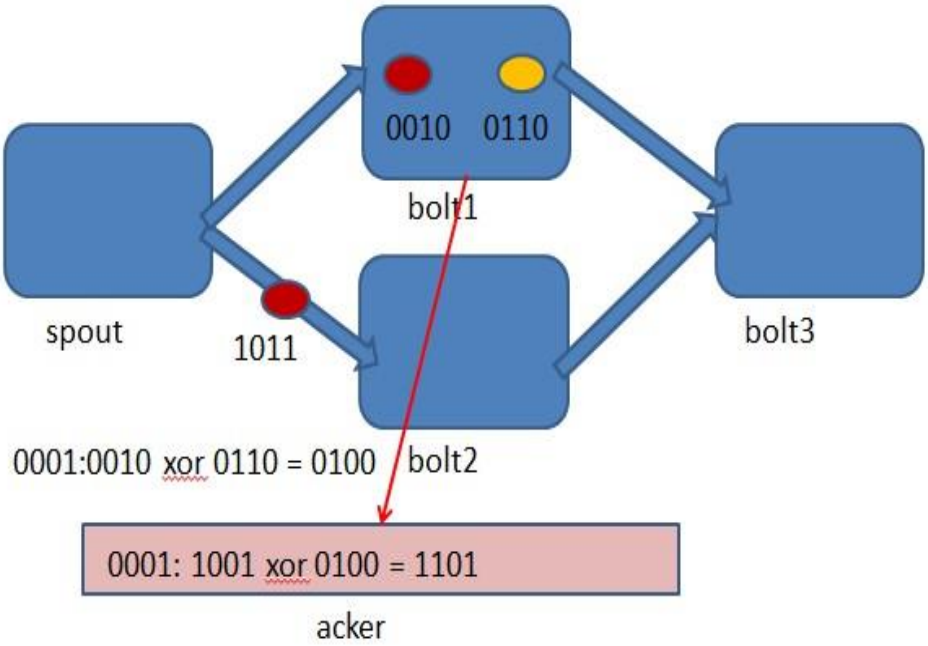


图 4-2 bolt1 处理 tuple 0010 时生成了一个新的 tuple，id 为 0110.

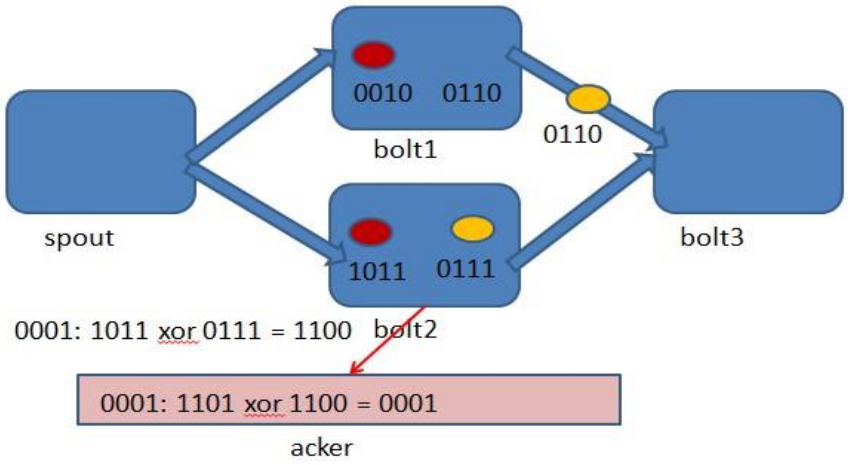


图 4-3 bolt2 处理 tuple 1011 时生成了一个新的 tuple，id 为 0111.

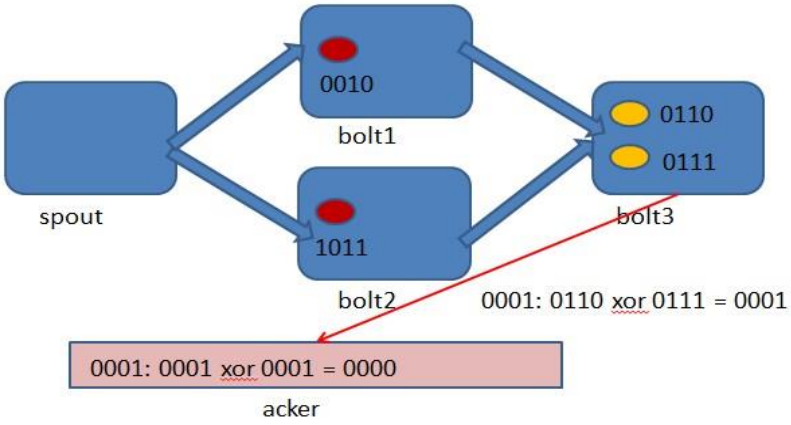


图 4-4 bolt3 中接收到 tuple 0110 和 tuple 0111，没有生成新的 tuple.

可能有些细心的同学会发现，容错过程存在一个可能出错的地方，那就是，如果生成的 tuple id 并不是完全各异的，acker 可能会在消息单元完全处理完成之前就错误的计算为 0。这个错误在理论上的确是存在的，但是在实际中其概率是极低极低的，完全可以忽略。

Storm 的事务拓扑

事务拓扑(transactional topology)是 storm0.7 引入的特性，在最近发布的 0.8 版本中已经被封装为 Trident，提供了更加便利和直观的接口。在此对事务拓扑做一个简单的介绍。

事务拓扑的目的是为了满足对消息处理有着极其严格要求的场景，例如实时计算某个用户的成交笔数，要求结果完全精确，不能多也不能少。Storm 的事务拓扑是完全基于它底层的 spout/bolt/acker 原语实现的，通过一层巧妙的封装得出一个优雅的实现。个人觉得这也是 storm 最大的魅力之一。

事务拓扑简单来说就是将消息分为一个个的批(batch)，同一批内的消息以及批与批之间的消息可以并行处理，另一方面，用户可以设置某些 bolt 为 committer，storm 可以保证 committer 的 finishBatch()操作是按严格不降序的顺序执行的。用户可以利用这个特性通过简单的编程技巧实现消息处理的精确。