Ui界面：Stormuihost:8080

任务列表：storm list

提交任务：storm jar ./storm-example-0.0.1-SNAPSHOT.jar com.blogchong.storm.helloworld.WordCountTopology task

杀死topology任务

storm kill topology-name [-w wait-time-secs] -w 等待多少秒后杀死

启动nimbus守护进程

storm nimbus

启动supervisor守护进程

storm supervisor

启动ui守护进程

storm ui

启动drpc守护进程

storm drpc

提供一个Web接口查看Storm日志文件,在storm ui上就可以观看

storm logviewer

激活指定的topology的spout：语法如下

storm activate topology-name

禁用指定的topology的spout:语法如下

storm deactivate topology-name

打印storm运行时的路径：语法如下

storm classpath

帮助命令：语法如下

storm help <command>

打印本地storm的配置文件，输出内容是storm.yaml和defaults.yaml的合并内容:语法如下

storm localconfvalue conf-name

打印本地storm的配置文件，输出内容是storm.yaml和defaults.yaml的合并内容  这个必须在集群模式下才会有效：

storm remoteconfvalue conf-name

storm版本输出：

storm version

#### 如何改变1个正在运行的拓扑的并发度

Storm有一个不错的特性，你可以在不需要重启集群或拓扑，来增加或减少worker进程和executor的数量。这样行为成为rebalancing。

你有两个选项可以rebalance1个拓扑：

1. 使用Storm的web UI来rebalance。
2. 像下面描述的那样，使用命令行工具来做：

# 重新配置拓扑 “mytopology” 使用5个worker进程。

# spout “blue-spout” 使用3个executor

# bolt “yellow-bolt” 使用10个executor

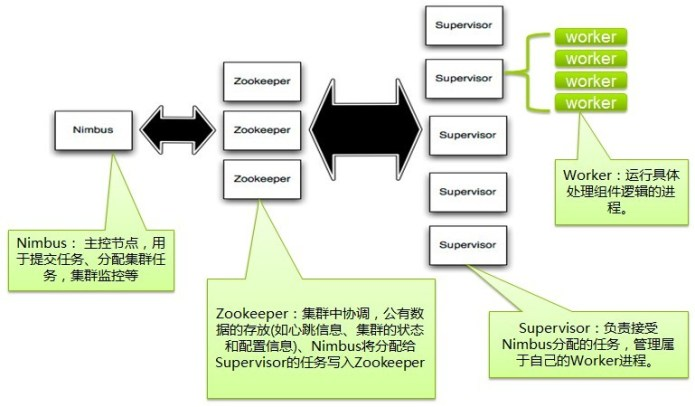
$ storm rebalance mytopology -n 5 -e blue-spout=3 -e yellow-bolt=10

Storm架构：

Storm与Hadoop的编程模型相似

| **结构** | **Hadoop** | **Storm** |
| --- | --- | --- |
| 主节点 | JobTracker | Nimbus |
| 从节点 | TaskTracker | Supervisor |
| 应用程序 | Job | Topology |
| 工作进程名称 | Child | Worker |
| 计算模型 | Map / Reduce | Spout / Bolt |

Storm集群采用主从架构方式，主节点是Nimbus，从节点是Supervisor，有关调度相关的信息存储到ZooKeeper集群中，架构如下图所示：



Nimbus  
Storm集群的Master节点，负责分发用户代码，指派给具体的Supervisor节点上的Worker节点，去运行Topology对应的组件（Spout/Bolt）的Task。  
Supervisor  
Storm集群的从节点，负责管理运行在Supervisor节点上的每一个Worker进程的启动和终止。通过Storm的配置文件中的supervisor.slots.ports配置项，可以指定在一个Supervisor上最大允许多少个Slot，每个Slot通过端口号来唯一标识，一个端口号对应一个Worker进程（如果该Worker进程被启动）。

Worker

运行具体处理组件逻辑的进程。Worker运行的任务类型只有两种，一种是Spout任务，一种是Bolt任务。

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，不同spout/bolt的task可能会共享一个物理线程，该线程称为executor。

ZooKeeper  
用来协调Nimbus和Supervisor，如果Supervisor因故障出现问题而无法运行Topology，Nimbus会第一时间感知到，并重新分配Topology到其它可用的Supervisor上运行

1. 我们利用前面所说的Storm可靠性机制，可以很容易的提供至少一次的处理(at least once processing)：也就是在一个tuple超时或者fail的时候，Storm会调用Spout的fail函数，在这里，我们可以实现一个重发tuple的机制，当然，这种重发一般都建立在消息队列中间件的重发功能上的。
2. 我们考虑一种计数Count场景，看简单的重发机制有什么问题。假如这个Topology一共有三个Bolt：第一个计数，第二个将技术结果持久化存入数据库（当前的统计加上数据库里面的统计），第三个显示报告。如果第三个Bolt失败了，那么会触发Spout重发，重发以后，第二个Bolt对于这重发的tuple计数如何处理？是更新数据还是不更新呢？当然是不更新，因为失败是发生在第三个Bolt上的，失败时，第二个Bolt已经做了入库处理了，所以重发后不需要再重新入库。那么如果错误发生在第一个Bolt呢？重发后，第二个Bolt是需要更新数据库的。所以，对于第二个Bolt，在重发tuple时，既有可能需要重新入库，也有可能不需要重新入库。所以，这里只有at least once processing就不够了，我们需要的是精确一次处理(exact once processing)机制，也即是事务机制。
3. 这里我们必须注意一个问题：我们假设第一个Bolt和第三个Bolt本身不含状态数据，即tuple和tuple之间在处理上没有状态关系，全部的状态数据都在第二个Bolt中，这点很重要，而且一般的书籍都不会强调这一点。笔者在开始学习的时候，就一直苦恼于此，因为计数Bolt一般都是在内存里面保存一个Map用来存储计数的，相当于也有状态，那么重发tuple以后，第一个Bolt又该如何处理Map里面缓存的数据呢？其实这种情况我们不考虑，我们假设tuple和tuple之间没有状态联系，状态都保存在数据库里面。那么如果我们想用Map来进行计数又该怎么办呢？基于Storm的事务机制也是可以的，只是不是tuple和tuple之间，而是batch和batch之间没有状态联系，这个我们在后面会看到。
4. 事务处理机制是在Storm 0.7版本中引入的，而随着Trident框架的引入，事务处理机制已经不用了，但是Trident的内部也是对事务机制的一种封装，所以，我们这里讲Storm的事务，对于理解Trident也是必要的。
5. 对于之前我们出现的问题，关键是在tuple进行重发时，将数据进行持久化的时候，我们无法确定该不该重新持久化。我们采用如下设计：
   1. 每一个tuple分配一个事务id txid，我们持久化存储数据时，不仅存储计数值，还要存储一个事务di。如果当前的txid和数据库里面的txid不同，我们就知道需要重新更新数据库，如果相同，说明是重发的tuple，而且数据库之前已经更新过了，所以就不需要重新更新数据库了。
   2. 第一点中其实还隐含了一种约束，即事务必须串行化，即1号事务必须在2号事务之前处理成功，否则2号事务则不能生效。这叫做强顺序性(strong ordering)。在这种约束下，只要判断txid的同与不同就行了，不需要判断大小关系。
6. 分配事务id好实现。那么如何实现这种强顺序性呢？
7. 我们先考虑一种简单的设计：等一个tuple入库完毕(即一个事务处理完毕)，才触发第二个tuple进入topology进行处理。这种设计的问题很明显：Storm的流水线处理性能就不复存在了，完全变成了串行处理，每次只能处理一个tuple。
8. 为了变串行为并行，我们考虑第二种设计：将一批(batch)tuple打包成一个事务，这一批tuple拥有一个txid，每次处理一批：处理完一批，再另一批进入topology进行处理。这种设计比第一种提高了很多性能：访问数据库的次数明显减少。但是这种设计也有它自己的问题：即batch与batch之间还是串行的，没有流起来。那么我们又有了第三种设计，也就是Storm采取的设计方案。
9. 为了要Batch和batch之间能并发，能流水线，我们必须把一个事务处理分成两种不同类型的处理：
   1. 处理阶段(processing stage)：只计算和当前batch相关的数据，batch和batch之间没有状态关联。这就是我在之前第三点提出的一个问题，采用Storm的事务机制，你也可以用Map保存计数结果，但是只是一个batch的计数结果，等另外一个batch进来以后，Bolt实例会重建，Map的数据就被清空了。
   2. 提交阶段(commit stage)：将计算结果保存到数据库。
10. 当将一个事务分成两个阶段后，processing阶段就可以在batch和batch之间并发了，而只需要commit阶段实现强顺序性。更加提高了性能。
11. 实现Transaction的Builder是：**TranasactionalTopologyBuilder**
12. 实现Transaction的Spout是：比较复杂，由一个Coordination Spout和许多emitter Bolt组成
13. 实现Transaction的Bolt根据是否具备commit功能，分为普通的Batch Bolt(不具备commit功能的Batch Bolt)，为**BaseBatchBolt**；以及具备commit功能的Batch Bolt。两者的唯一区别在于finishBatch函数调用的阶段不一样：具备commit功能的Batch Bolt的finishBatch函数在commit阶段调用，而不具备commit功能的Bolt，它的finishBatch函数在Batch处理完毕后调用，那么就有可能在processing阶段调用，也有可能在commit阶段调用。千万别简单地以为普通的Batch Bolt在processing阶段调用，而具备commit功能的Bolt在commit阶段调用。调用finishBatch函数时，Commiter Bolt只能在commit阶段，普通的Batch Bolt则不一定。那么调用execute函数呢？（笔者认为：不一定。如果Bolt1->Bolt2，如果Bolt1是committer Bolt，而它emit tuple是在finishBatch函数中完成的，那么Bolt2即使是个普通的Batch Bolt，那么它的execute函数也只能在commit阶段调用了，因为Bolt1的finishBatch只能在commit阶段调用。）
14. 有两种方式实现Committer Bolt：一种是实现**ICommitter**接口，它是个标签接口。另一种是builder在调用set Bolt时，调用**setCommitterBolt**函数，而不是调用setBolt函数。
15. 到这里我们一直是存在一个要求：你的Spout在接到Storm的失败或者超时通知时，能重传和之前txid相等的完全一样的batch tuple。当然我们在有些情况下，并不能满足这个要求，那么是否也能做到exact once processing呢？答案是肯定的，只是在存储状态时，不光要存储当前的状态，还要存储前一次的状态。为什么呢？因为如果重传时，batch内的tuple不一样了，那么重传的batch计算的结果，和之前失败batch计算的结果，两次结果可能是不一样的。那么，不管你持久化的txid和当前的txid是否一样，你都需要重新更新状态内容。怎么更新？由于更新也许是基于之前存储的状态的，那么我们必须先回滚到之前的状态，对，回滚，那么我们在数据库里面存储的前一次的状态就是相当于回滚后的状态。
16. 这种重传的内容可以和之前出错的内容不一样的Spout，我们叫做**Opaque(不透明) transactional Spout**，而之前讨论的重传内容和之前出错的内容完全一样的Spout，叫做**idempotent(等幂性) transactional spout。**
17. Opaque transactional Spout的这种重传特性，在失败后，有一些不同于idempotent transactional Spout的特殊性：如果txid=3的batch失败了，但是txid=4，txid=5的batch还没有最终完成(既可能在processing阶段，也可能在commit阶段)，那么txid=3的batch肯定要重传，而txid=4，txid=5的batch也要重传。原因就在于txid=3的重传内容可能没之前失败的batch不同，如果多了，那么就可能和txid=4，txid=5里的内容重复了；如果少了，tuple就会丢失掉，没有传。所以，Storm要求失败batch之后的batch也要重传。(作者注：如果Spout能做到发过的tuple做标记，不会重发，也不会漏发，那么是不是就不需要重发txid=4，txid=5的batch呢？)