# 课程介绍

课程名称：

Storm技术增强

注：学习本课程，请先学习Storm基础

课程目标：

通过本模块的学习，能够掌握Storm底层的通信机制、消息容错机制、storm目录树及任务提交流程。

课程大纲：

1. Storm程序的并发机制
2. Storm框架通信机制（worker内部通信与外部通信）
3. Storm组件本地目录树
4. Storm zookeeper目录树
5. Storm 任务提交的过程

# 课程内容

## 1、Storm程序的并发机制

### 1.1、概念

* Workers (JVMs): 在一个物理节点上可以运行一个或多个独立的JVM 进程。一个Topology可以包含一个或多个worker(并行的跑在不同的物理机上), 所以worker process就是执行一个topology的子集, **并且worker只能对应于一个topology**
* Executors (threads): 在一个worker JVM进程中运行着多个Java线程。一个executor线程可以执行一个或多个tasks。但一般默认每个executor只执行一个task。一个worker可以包含一个或多个executor, 每个component (spout或bolt)至少对应于一个executor, 所以可以说executor执行一个compenent的子集, 同时一个executor只能对应于一个component。
* Tasks(bolt/spout instances)：Task就是具体的处理逻辑对象，每一个Spout和Bolt会被当作很多task在整个集群里面执行。每一个task对应到一个线程，而stream grouping则是定义怎么从一堆task发射tuple到另外一堆task。你可以调用TopologyBuilder.setSpout和TopBuilder.setBolt来设置并行度 — 也就是有多少个task。

### 1.2、配置并行度

* 对于并发度的配置, 在storm里面可以在多个地方进行配置, 优先级为：

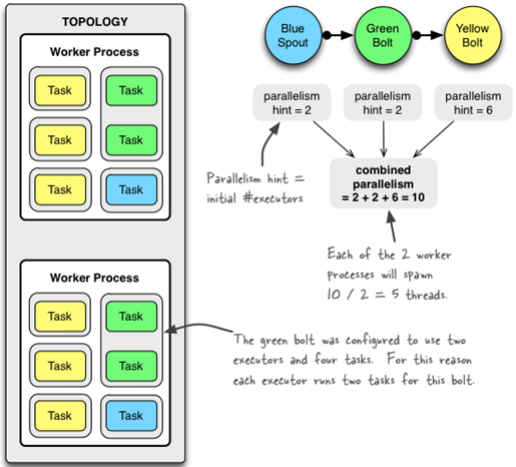
defaults.yaml < storm.yaml < topology-specific configuration

< internal component-specific configuration < external component-specific configuration

* worker processes的数目, 可以通过配置文件和代码中配置, worker就是执行进程, 所以考虑并发的效果, 数目至少应该大亍machines的数目
* executor的数目, component的并发线程数，只能在代码中配置(通过setBolt和setSpout的参数), 例如, setBolt("green-bolt", new GreenBolt(), 2)
* tasks的数目, 可以不配置, 默认和executor1:1, 也可以通过setNumTasks()配置

Topology的worker数通过config设置，即执行该topology的worker（java）进程数。它可以通过 storm rebalance 命令任意调整。

|  |
| --- |
| Config conf = newConfig();  conf.setNumWorkers(2); **//用2个worker**  topologyBuilder.setSpout("blue-spout", newBlueSpout(), 2); **//设置2个并发度**  topologyBuilder.setBolt("green-bolt", newGreenBolt(), 2).setNumTasks(4).shuffleGrouping("blue-spout"); **//设置2个并发度，4个任务**  topologyBuilder.setBolt("yellow-bolt", newYellowBolt(), 6).shuffleGrouping("green-bolt"); **//设置6个并发度**  StormSubmitter.submitTopology("mytopology", conf, topologyBuilder.createTopology()); |



3个组件的并发度加起来是10，就是说拓扑一共有10个executor，一共有2个worker，每个worker产生10 / 2 = 5条线程。

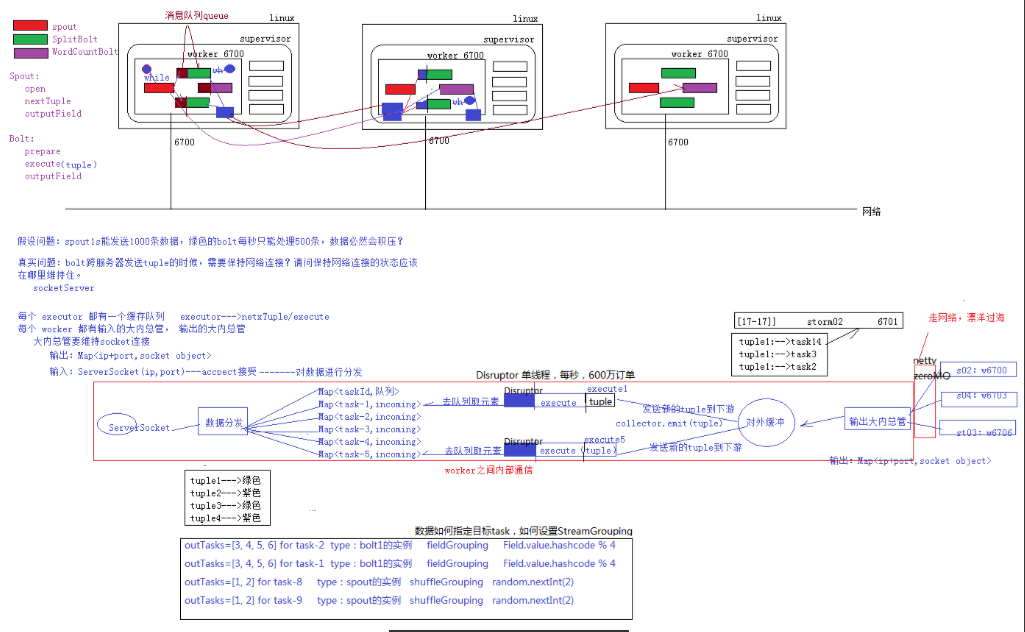
绿色的bolt配置成2个executor和4个task。为此每个executor为这个bolt运行2个task。

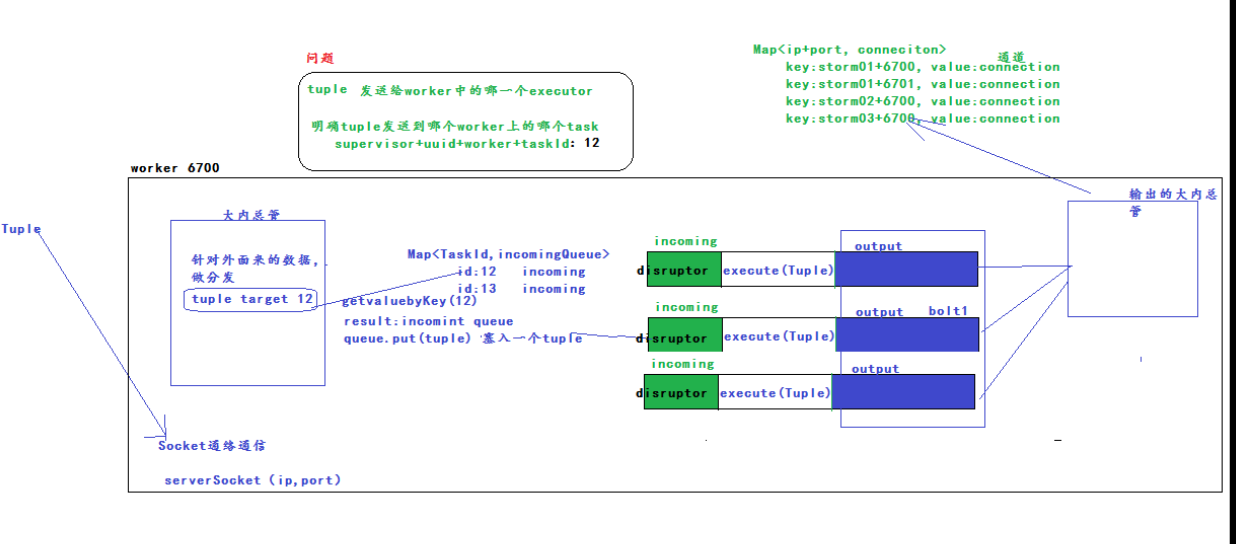
* 动态的改变并行度

Storm支持在不 restart topology 的情况下, 动态的改变(增减) worker processes 的数目和 executors 的数目, 称为rebalancing. 通过Storm web UI，或者通过storm rebalance命令实现：

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

## 2、Storm通信机制





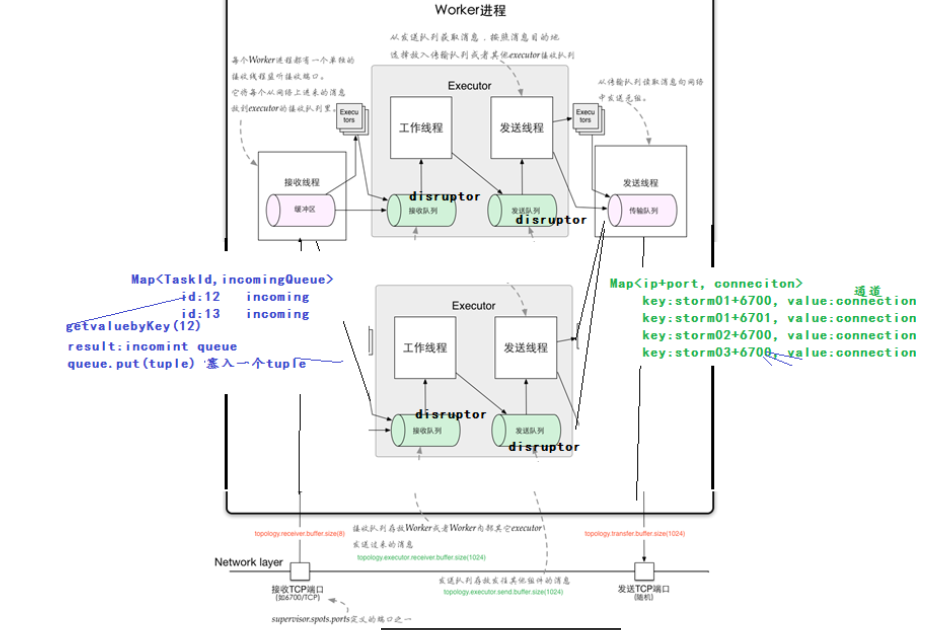
Worker间的通信经常需要通过网络跨节点进行，Storm使用ZeroMQ或Netty(0.9以后默认使用)作为进程间通信的消息框架。

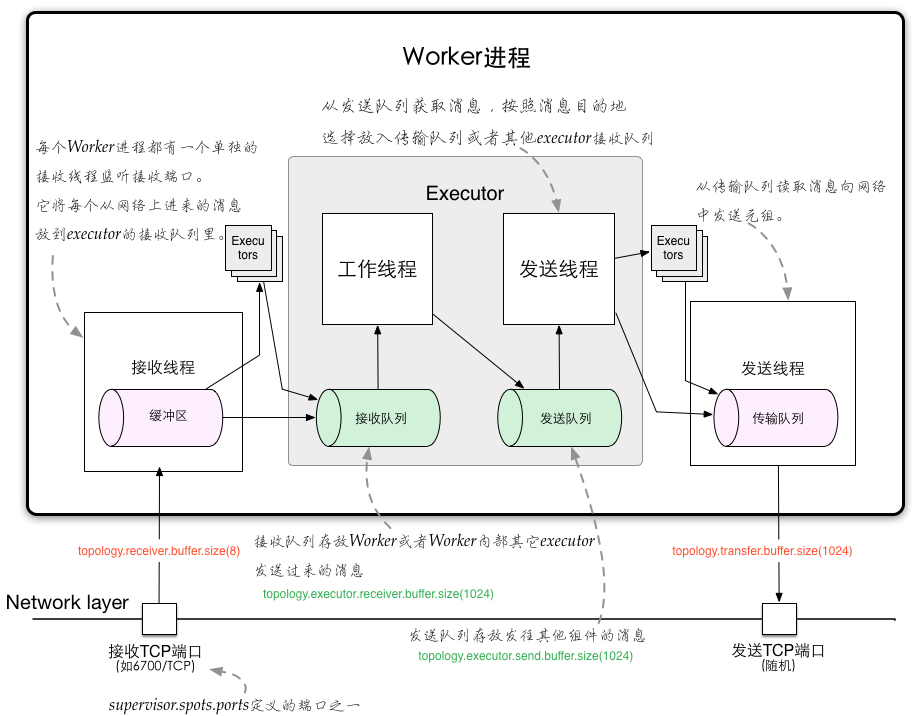
Worker进程内部通信：**不同worker的thread通信使用LMAX Disruptor来完成。**

  不同topologey之间的通信，Storm不负责，需要自己想办法实现，例如使用kafka等；

### 2.1、Worker进程间通信

worker进程间消息传递机制，消息的接收和处理的大概流程见下图





* + 对于worker进程来说，为了管理流入和传出的消息，每个worker进程有一个独立的接收线程(对配置的TCP端口supervisor.slots.ports进行监听);

对应Worker接收线程，每个worker存在一个独立的发送线程，它负责从worker的transfer-queue中读取消息，并通过网络发送给其他worker

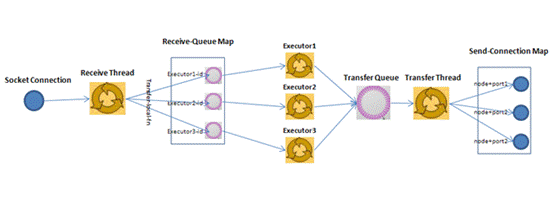
* + 每个executor有自己的incoming-queue和outgoing-queue。

Worker接收线程将收到的消息通过task编号传递给对应的executor(一个或多个)的incoming-queues;

每个executor有单独的线程分别来处理spout/bolt的业务逻辑，业务逻辑输出的中间数据会存放在outgoing-queue中，当executor的outgoing-queue中的tuple达到一定的阀值，executor的发送线程将批量获取outgoing-queue中的tuple,并发送到transfer-queue中。

* + 每个worker进程控制一个或多个executor线程，用户可在代码中进行配置。其实就是我们在代码中设置的并发度个数。

### 2.2、Worker进程间通信分析



1. Worker接受线程通过网络接受数据，并根据Tuple中包含的taskId，匹配到对应的executor；然后根据executor找到对应的incoming-queue，将数据存发送到incoming-queue队列中。
2. 业务逻辑执行现成消费incoming-queue的数据，通过调用Bolt的execute(xxxx)方法，将Tuple作为参数传输给用户自定义的方法
3. 业务逻辑执行完毕之后，将计算的中间数据发送给outgoing-queue队列，当outgoing-queue中的tuple达到一定的阀值，executor的发送线程将批量获取outgoing-queue中的tuple,并发送到Worker的transfer-queue中
4. Worker发送线程消费transfer-queue中数据，计算Tuple的目的地，连接不同的node+port将数据通过网络传输的方式传送给另一个的Worker。
5. 另一个worker执行以上步骤1的操作。

### 2.3、Worker进程间技术(Netty、ZeroMQ)

#### 2.3.1、Netty

Netty是一个NIO client-server(客户端服务器)框架，使用Netty可以快速开发网络应用，例如服务器和客户端协议。Netty提供了一种新的方式来使开发网络应用程序，这种新的方式使得它很容易使用和有很强的扩展性。Netty的内部实现时很复杂的，但是Netty提供了简单易用的api从网络处理代码中解耦业务逻辑。**Netty是完全基于NIO实现的，所以整个Netty都是异步的。**

**书籍：****Netty权威指南**

#### 2.3.2、ZeroMQ

ZeroMQ是一种基于消息队列的多线程网络库，其对套接字类型、连接处理、帧、甚至路由的底层细节进行抽象，提供跨越多种传输协议的套接字。ZeroMQ是网络通信中新的一层，介于应用层和传输层之间（按照TCP/IP划分），其是一个可伸缩层，可并行运行，分散在分布式系统间。

ZeroMQ定位为：一个简单好用的传输层，像框架一样的一个socket library，他使得Socket编程更加简单、简洁和性能更高。是一个消息处理队列库，可在多个线程、内核和主机盒之间弹性伸缩。ZMQ的明确目标是“成为标准网络协议栈的一部分，之后进入Linux内核”。

### 2.4、Worker 内部通信技术(Disruptor)

#### Disruptor的来历

* 一个公司的业务与技术的关系，一般可以分为三个阶段。第一个阶段就是跟着业务跑。第二个阶段是经历了几年的时间，才达到的驱动业务阶段。第三个阶段，技术引领业务的发展乃至企业的发展。所以我们在学习Disruptor这个技术时，不得不提LMAX这个机构，因为Disruptor这门技术就是由LMAX公司开发并开源的。
  + LMAX是在英国注册并受到FSA监管（监管号码为509778）的外汇黄金交易所。LMAX也是欧洲第一家也是唯一一家采用多边交易设施Multilateral Trading Facility（MTF）拥有交易所牌照和经纪商牌照的欧洲顶级金融公司
  + LAMX拥有最迅捷的交易平台，顶级技术支持。LMAX交易所使用“（MTF）分裂器Disruptor”技术，可以在极短时间内（一般在3百万秒之一内）处理订单，**在一个线程里每秒处理6百万订单**。所有订单均为撮合成交形式，无一例外。多边交易设施（MTF）曾经用来设计伦敦证券交易 所（london Stock Exchange）、德国证券及衍生工具交易所（Deutsche Borse）和欧洲证券交易所（Euronext）。
  + 2011年LMAX凭借该技术获得了金融行业技术评选大赛的最佳交易系统奖和甲骨文“公爵杯”创新编程框架奖。

#### 2.4.2、Disruptor是什么

* 1. 简单理解：Disruptor是一个Queue。**Disruptor是实现了“队列”的功能，而且是一个有界队列。**而队列的应用场景自然就是“生产者-消费者”模型。
  2. 在JDK中Queue有很多实现类，包括不限于ArrayBlockingQueue、LinkBlockingQueue，这两个底层的数据结构分别是数组和链表。数组查询快，链表增删快，能够适应大多数应用场景。
  3. 但是ArrayBlockingQueue、LinkBlockingQueue都是线程安全的。涉及到线程安全，就会有synchronized、lock等关键字，这就意味着CPU会打架。
  4. **Disruptor一种线程之间信息无锁的交换方式（**使用CAS（Compare And Swap/Set）操作）。

#### 2.4.2、Disruptor主要特点

1. 没有竞争=没有锁=非常快。
2. 所有访问者都记录自己的序号的实现方式，允许多个生产者与多个消费者共享相同的数据结构。
3. 在每个对象中都能跟踪序列号（ring buffer，claim Strategy，生产者和消费者），加上神奇的cache line padding，就意味着没有为伪共享和非预期的竞争。

#### Disruptor 核心技术点

Disruptor可以看成一个事件监听或消息机制，在队列中一边生产者放入消息，另外一边消费者并行取出处理.

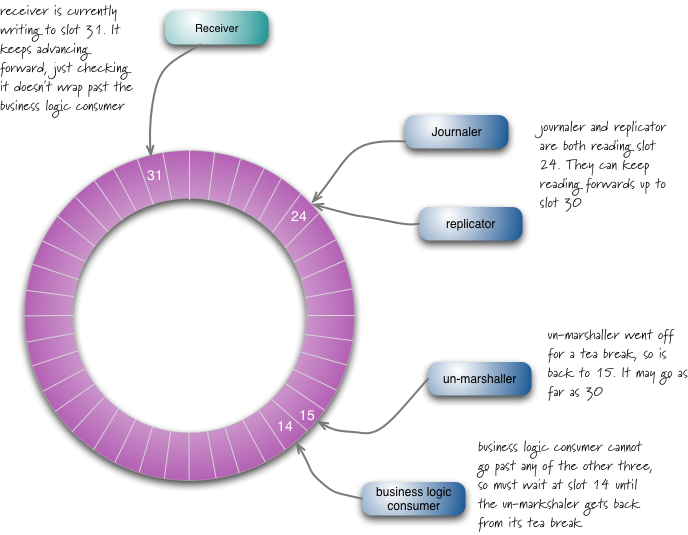
**底层是单个数据结构：一个ring buffer。**

每个生产者和消费者都有一个次序计算器，以显示当前缓冲工作方式。

每个生产者消费者能够操作自己的次序计数器的能够读取对方的计数器，生产者能够读取消费者的计算器确保其在没有锁的情况下是可写的。

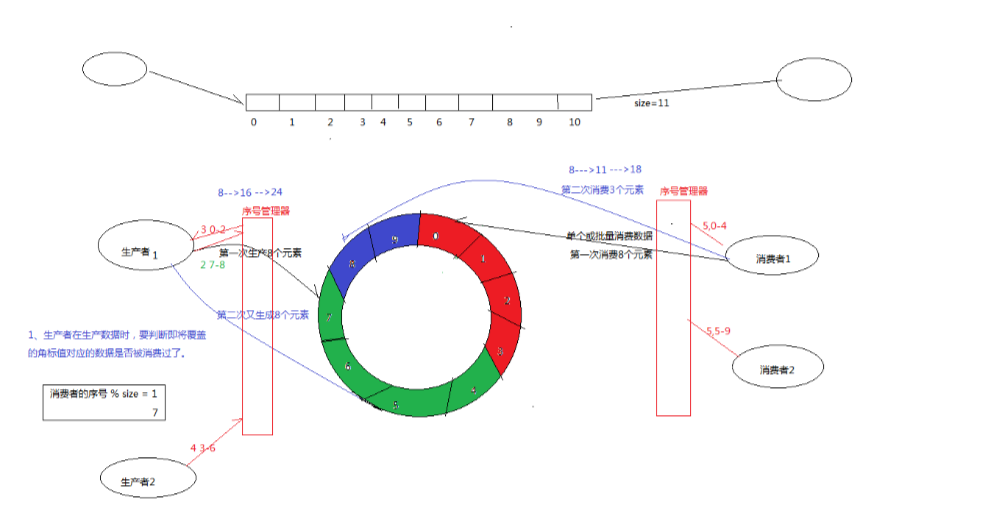
**核心组件**

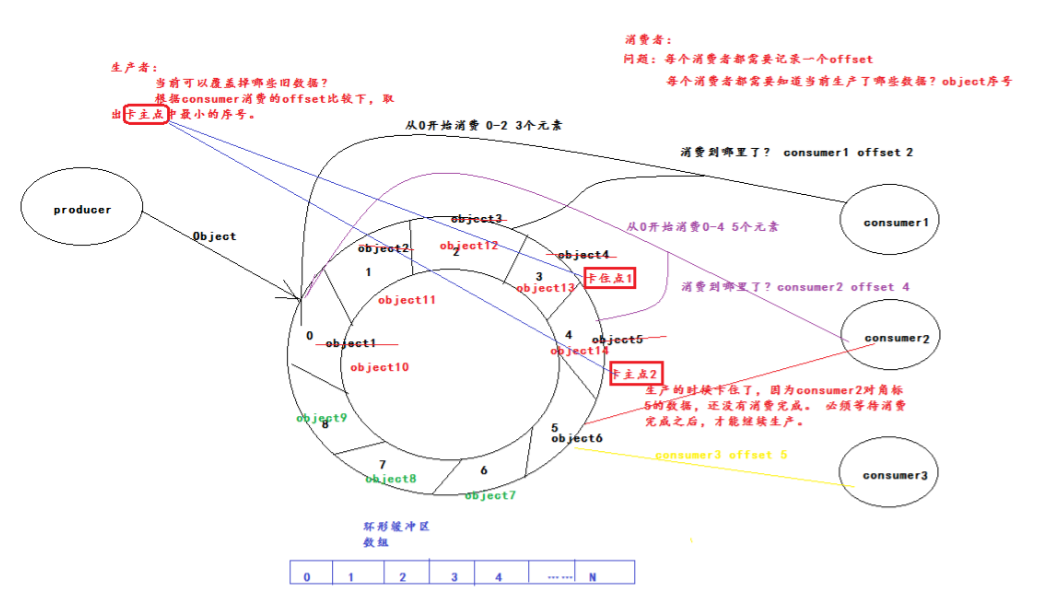
* Ring Buffer 环形的缓冲区，负责对通过 Disruptor 进行交换的数据（事件）进行存储和更新。
* Sequence 通过顺序递增的序号来编号管理通过其进行交换的数据（事件），对数据(事件)的处理过程总是沿着序号逐个递增处理。
* RingBuffer底层是个数组，次序计算器是一个64bit long 整数型，平滑增长。

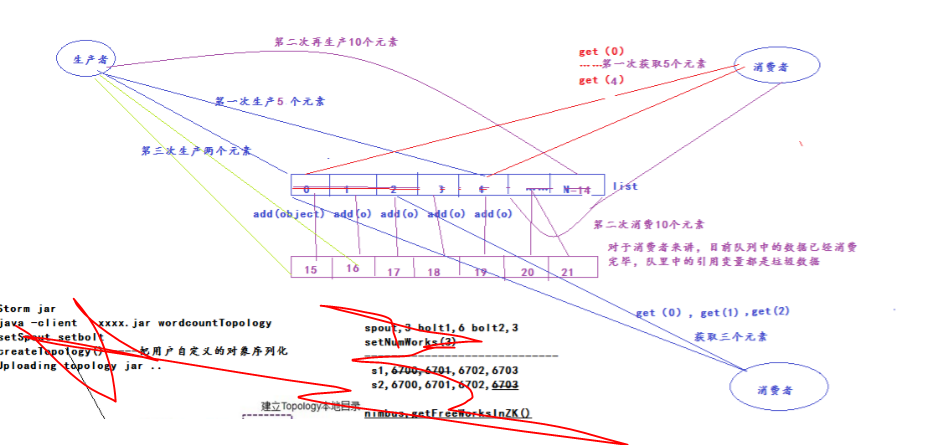


1. 接受数据并写入到脚标31的位置，之后会沿着序号一直写入，但是不会绕过消费者所在的脚标。
2. Joumaler和replicator同时读到24的位置，他们可以批量读取数据到30

3、消费逻辑线程读到了14的位置，但是没法继续读下去，因为他的sequence暂停在15的位置上，需要等到他的sequence给他序号。如果sequence能正常工作，就能读取到30的数据。



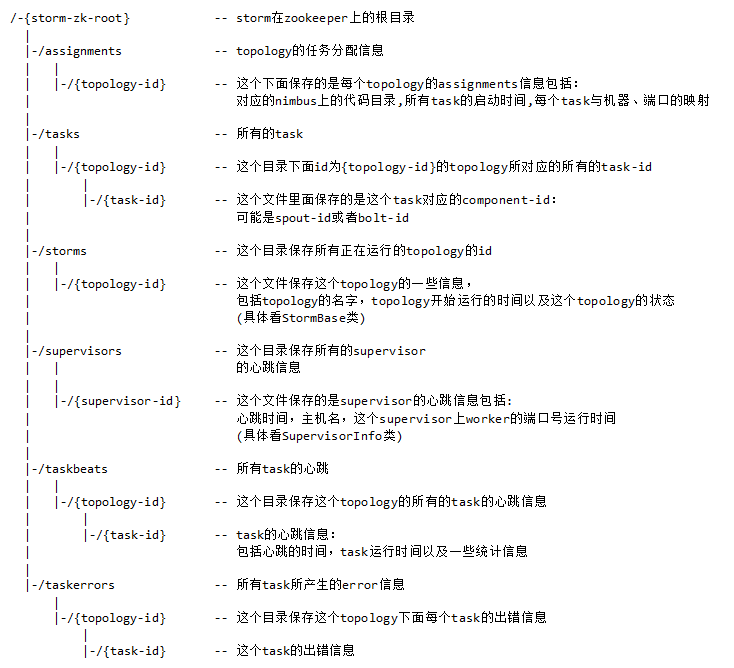


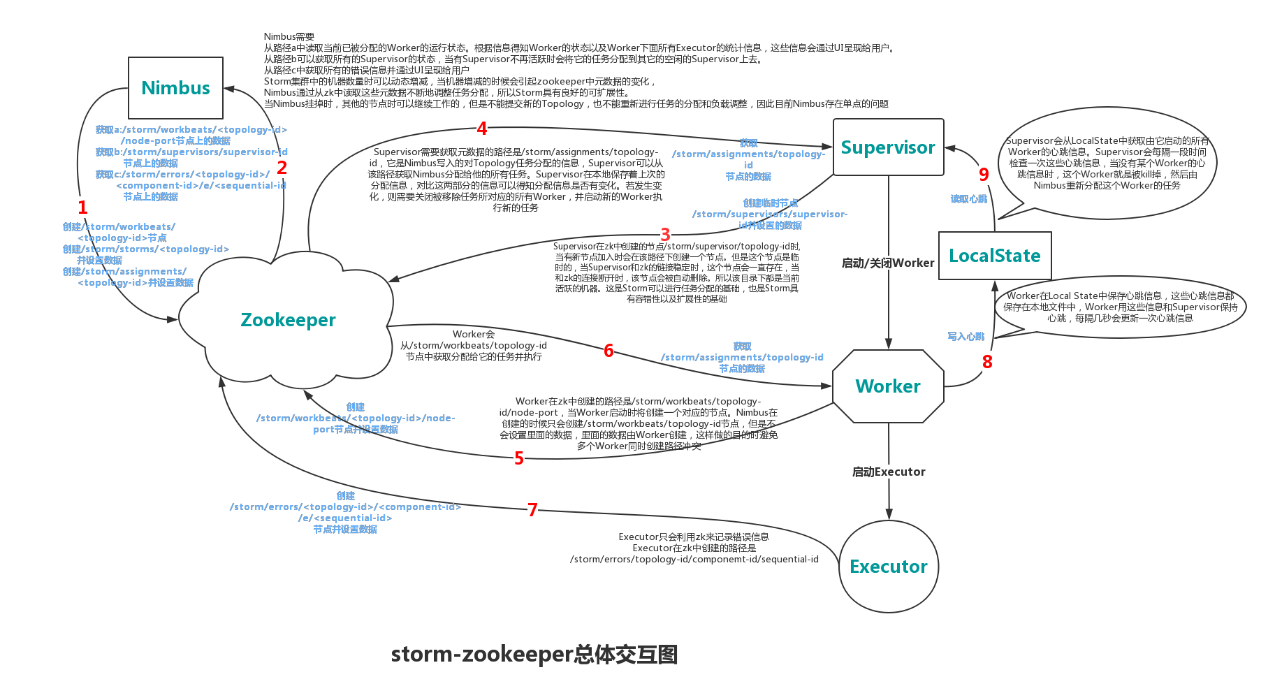


## 3、Storm组件本地目录树

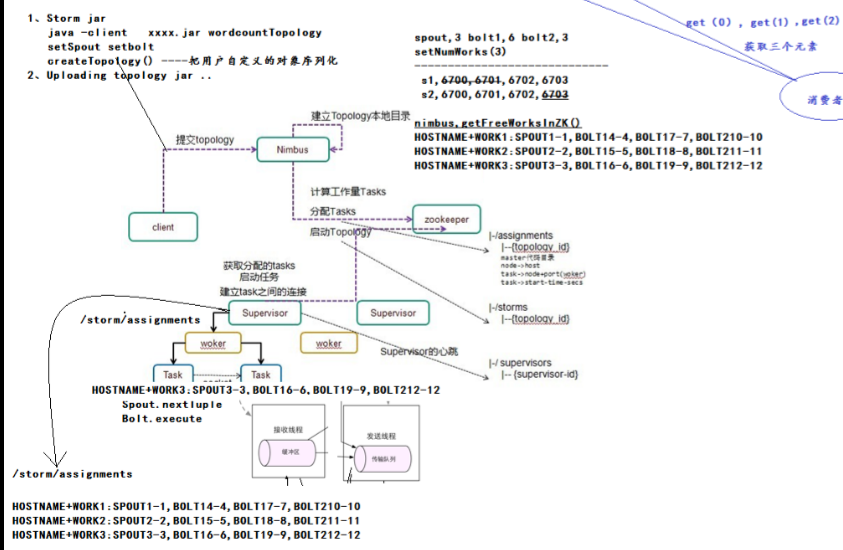


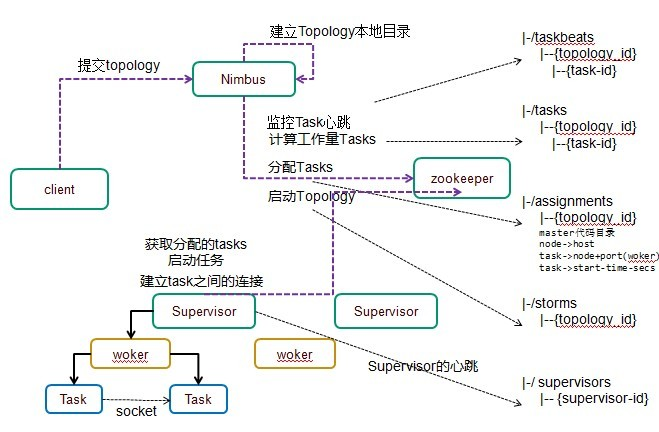
## 4、Storm zookeeper目录树



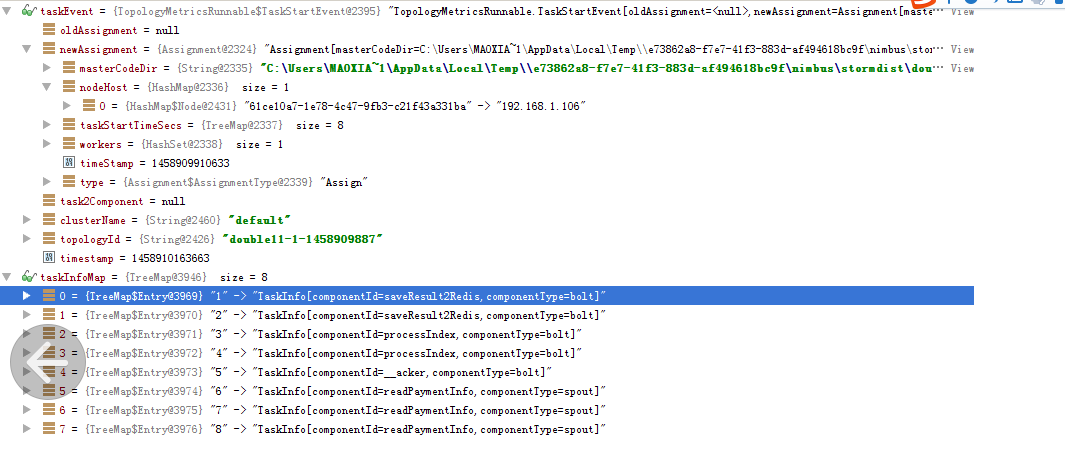


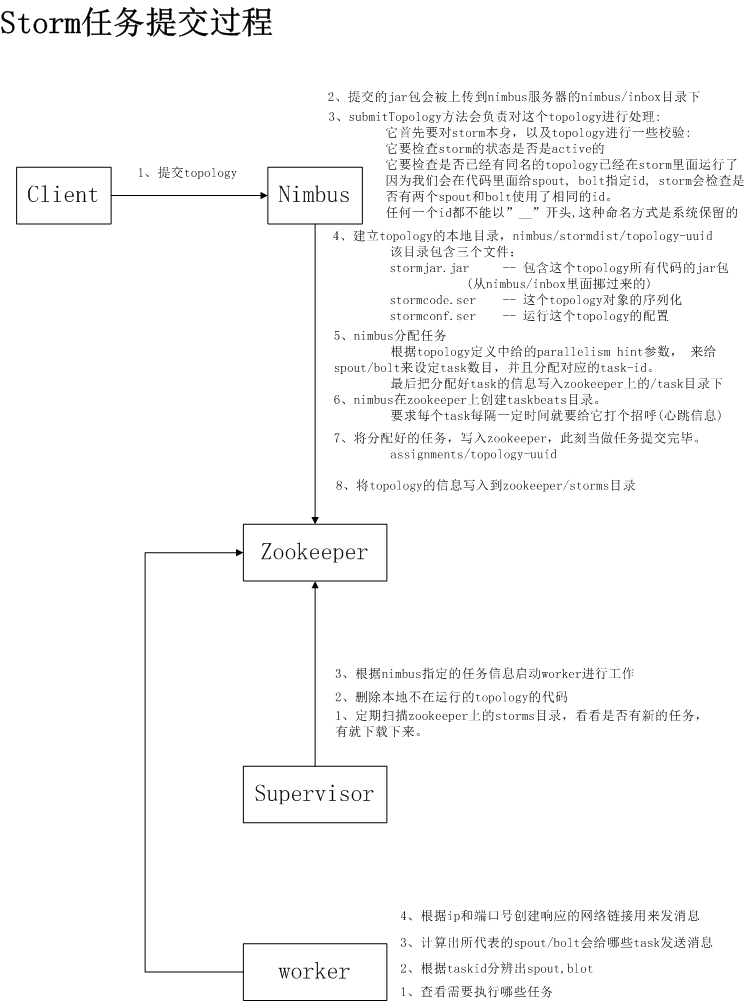
## 5、Storm 任务提交的过程





|  |
| --- |
| TopologyMetricsRunnable.TaskStartEvent[oldAssignment=<null>,newAssignment=Assignment[masterCodeDir=C:\Users\MAOXIA~1\AppData\Local\Temp\\e73862a8-f7e7-41f3-883d-af494618bc9f\nimbus\stormdist\double11-1-1458909887,nodeHost={61ce10a7-1e78-4c47-9fb3-c21f43a331ba=192.168.1.106},taskStartTimeSecs={1=1458909910, 2=1458909910, 3=1458909910, 4=1458909910, 5=1458909910, 6=1458909910, 7=1458909910, 8=1458909910},workers=[ResourceWorkerSlot[hostname=192.168.1.106,memSize=0,cpu=0,tasks=[1, 2, 3, 4, 5, 6, 7, 8],jvm=<null>,nodeId=61ce10a7-1e78-4c47-9fb3-c21f43a331ba,port=6900]],timeStamp=1458909910633,type=Assign],task2Component=<null>,clusterName=<null>,topologyId=double11-1-1458909887,timestamp=0] |





## 6、Storm 消息容错机制

### 6.1、总体介绍

* 在storm中，可靠的信息处理机制是从spout开始的。
* 一个提供了可靠的处理机制的spout需要记录他发射出去的tuple，当下游bolt处理tuple或者子tuple失败时spout能够重新发射。
* Storm通过调用Spout的nextTuple()发送一个tuple。为实现可靠的消息处理，**首先要给每个发出的tuple带上唯一的ID，并且将ID作为参数传递给SoputOutputCollector的emit()方法**：collector.emit(new Values("value1","value2"), msgId); messageid就是用来标示唯一的tupke的，而rootid是随机生成的

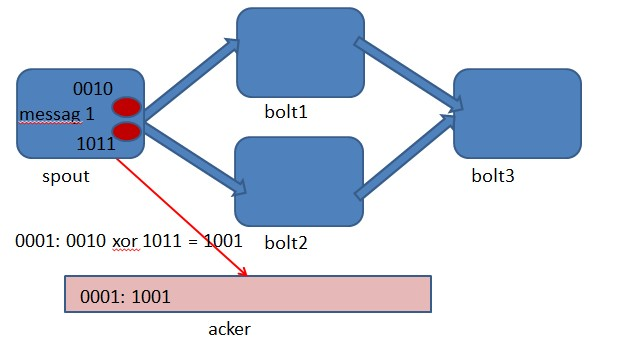
给每个tuple指定ID告诉Storm系统，无论处理成功还是失败，spout都要接收tuple树上所有节点返回的通知。如果处理成功，spout的ack()方法将会对编号是msgId的消息应答确认；如果处理失败或者超时，会调用fail()方法。

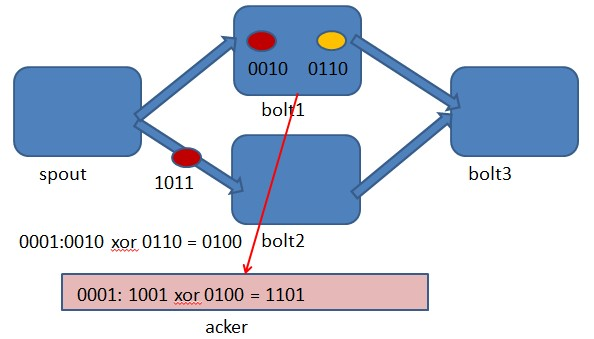
### 6.2、基本实现

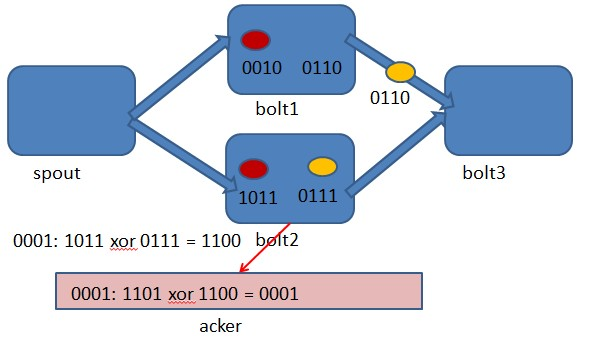
Storm 系统中有一组叫做"acker"的特殊的任务，它们负责跟踪DAG（有向无环图）中的每个消息。

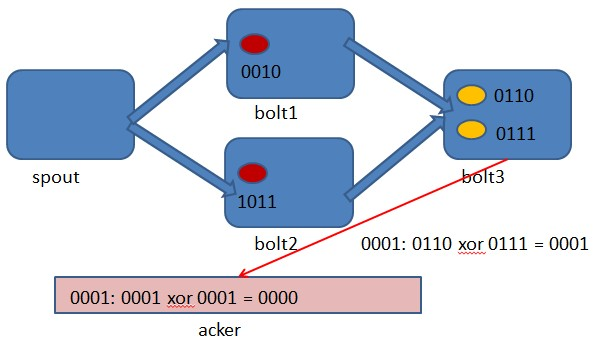
acker任务保存了spout id到一对值的映射。第一个值就是spout的任务id，通过这个id，acker就知道消息处理完成时该通知哪个spout任务。第二个值是一个64bit的数字，我们称之为"ack val"， 它是树中所有消息的随机id的异或计算结果。

ack val表示了整棵树的的状态，无论这棵树多大，只需要这个固定大小的数字就可以跟踪整棵树。当消息被创建和被应答的时候都会有相同的消息id发送过来做异或。 每当acker发现一棵树的ack val值为0的时候，它就知道这棵树已经被完全处理了









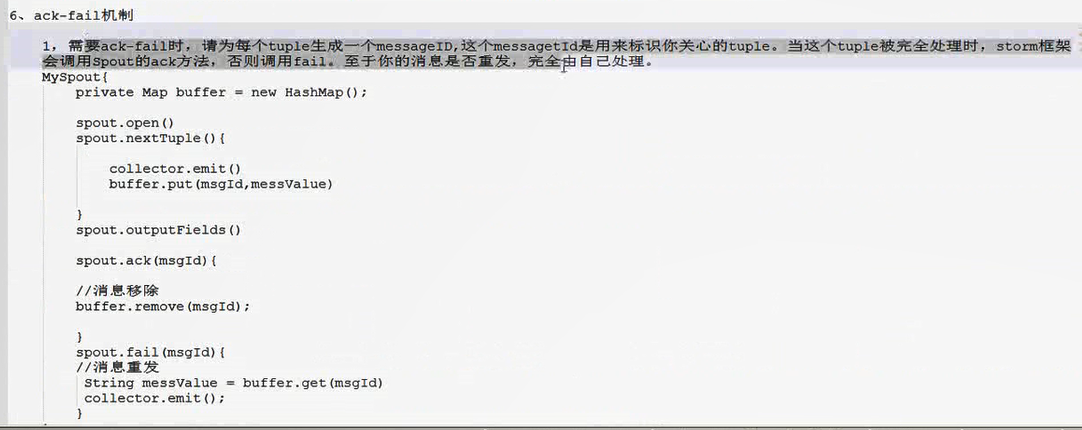
### 6.3、可靠性配置

有三种方法可以去掉消息的可靠性：

将参数Config.TOPOLOGY\_ACKERS设置为0，通过此方法，当Spout发送一个消息的时候，它的ack方法将立刻被调用；

Spout发送一个消息时，不指定此消息的messageID。当需要关闭特定消息可靠性的时候，可以使用此方法；

最后，如果你不在意某个消息派生出来的子孙消息的可靠性，则此消息派生出来的子消息在发送时不要做锚定，即在emit方法中不指定输入消息。因为这些子孙消息没有被锚定在任何tuple tree中，因此他们的失败不会引起任何spout重新发送消息。







<https://blog.csdn.net/qq_34092336/article/details/78408239>

概念：storm的ack-fail机制也就是storm的可靠消息处理机制，通俗来讲就是给spout发出的每个tuple带上一个messageid，然后这个spout下面的每一个bolt

都会给他返回一个完成情况，只有当每一个bolt都返回了正确的结果，整个发送过程才算成功，任何一个bolt处理不成功，则不成功。

**我对ack-fail机制的讲解分为三个层面：分别是api应用也就是写代码方面、ack-fail机制的处理过程。**

**首先是代码编写方面：**

假设我们在这个系统中有一种spout和一种bolt，如果你不使用ack-fail机制那么一个spout中有三个方法，分别是open(),nextTuple()和outputFields(),

open的作用是初始化那个outputCollector，nextTuple方法就是不断地取值然后发给下一个bolt或者就结束了，declareOutputFields方法就是声明一下我发

射出去的数据id，如果你使用了ack-fail机制那就多了俩方法，ack()和fail()，发送成功了就调ack方法，不成功就调fail，你在fail里可以进行重发或者什

么的，当然这些都是你自己决定，你要是不想做处理函数里就啥都不写，我们这里进行重发：

Myspout{

    Map<String,Values> buffer = new HashMap<>();  //缓存正在发送的tuple

    open();

    nextTuple(){

        String messageId = UUID.randomUUID().toString().replace("-", "");  //随机生成一个msgid

        buffer.put(messageId,tuple);  //放到缓存中

        collector.emit(value,messageId); //发射出去

    }

    fail(Object msgid){

        String value = buffer.get(msgid);  //取出value

        collector.emit(value,msgid);  //重发

    }

    ack(Object msgid){

        buffer.remove(msgid);  //从buffer中拿出来

    }

    declareOutputFields();

}

MyBolt方法本来的三个方法是prepare(),execute(),declareOutputFields(),prepare方法主要也就是初始化那个outputCollector，execute方法就是执行处理

过程，declareOutputFields也一样就是声明一下我发出去的是啥，而在应用了ack-fail机制的bolt中，这里要显示的声明我处理完了：

MyBolt{

    void execute(Tuple input){

        collector.emit(input,value);

        collector.ack(input);

    }

}

然后是ack-fail的处理过程方面：

spout---->tuple1---->bolt1---->ack(tuple1)

                     bolt1---->tuple1-1---->bolt2-1---->ack(tuple1-1);

                     bolt1---->tuple1-2---->bolt2-1---->ack(tuple1-2);

                                            ..........................

只有当每一个bolt都正确ack了，整个发送过程才算成功，任何一个bolt处理不成功，则不成功，重新处理。

那么ack这个东西他如何判断前者发射的tuple和ack返回的tuple是不是同一个呢，这里主要的概念是异或处理，对于每一个spout发射任务，ack维护了这样

一组数据，<spoutTaskId,<RootID,ackValue>,spoutTaskId标志着唯一的一个spouttask，RootId标志着整个过程的结果，ackValue记录着整个过程中不同

的tuple相异或的时候结果的变化，当ackValue最终等于0的时候，就标志着整个过程成功了，那么这个RootID如何计算呢，我们知道每一个tuple的发射过

程bolt都给了相应的返回tupleid，当这两个tupleid相同时就表明这一小阶段的任务完成了，而tuplid转化成二进制是0101形式的，如果返回的tupleid和

这个发送的tupleid相异或等于0，也就是ackValue等于0，就证明这两个是同一个id，也就表明这一小部分的任务成功了，但是整个过程中可能会有多层bolt，

每一个bolt的执行速度可能不同，所以注意，如果这些所有结果相异或后，ackValue等于0，就表明这个传输任务完成了。

**最后我们来从底层实现来讲一下ack-fail机制：**

我们运行storm程序时会发现有这样一个任务-ackTask，看一下源码他是继承了Bolt，他就是一个和其他数据处理的bolt一起存在一起处理的进程，而实际上

整个过程中是存在两种tuple的，分别是DataTuple和AckTuple，DataTuple主要负责数据的处理，AckTuple负责整个过程的排错，我们先来看这个AckTuple,

他其实封装了AckTuple<RootID,tupleId>,RootID标识了这个tuple属于哪个过程，而tupleId标识了每一个特定的tuple，这个AckTuple最终封装成一个

messageId这样一个对象，而DataTuple中就含有这个messageId。接下来我们来看整个过程，两种线程是一起进行的，ack的线程比较简单，当spout发射

一个DataTuple时同时就会发射一个AckTuple，然后他就在这等待响应，spout将DataTuple(messageId(AckTuple))发送给bolt，bolt.execute(dataTuple)

之后会应答也就是bolt.ack(dataTuple),而dataTuple中封装了ackTuple，就可以还原出这个ackTuple，这样acktask就等到了ack应答，也就是说这一阶段

处理成功，以此类推。

上述过程示意：

spout.emit(dataTuple(messageId(ackTuple)))--->bolt.excute--->bolt.ack(dataTuple(ackTuple))

spout.emit(ackTuple)

<https://blog.csdn.net/qq_21125183/article/details/80810112>

<https://blog.csdn.net/sheen1991/article/details/52788547>

tuple标记和数据结构

对于Spout而言，在发送一个tuple时，需要指定一个msgId，这时，系统会根据这个msgId来给这个tuple生成一个64位的spout tuple id，也可称之为root id。这个root id会一直跟着这个将要发送的tuple，不管是在当前spout中还是在下游的bolt中，作为该tuple的唯一标识。同时，不管在Spout还是在Bolt中，系统会给每个发送的tuple随机生成一个64位的tuple id，这个id只和当前tuple相关。例如某个spout中发送了tuple A，在下游bolt中接受了A并且发送了tuple B和C，且B和C被锚定在A上，那么A、B、C都会携带一个相同的root id，并且各自拥有一个随机的 tuple id。此外，在Spout发送一个源tuple时，负责“跟踪”的acker中会记录下当前Spout task的 task id和这个tuple的root id的对应关系，以便acker知道，当某个tuple树ack完全时，去通知哪个spout task。

这样一来，对于一个tuple，至少需要包含以下几项信息：

[root id (根据msgId生成的spout tuple id)、 tuple id]

对于acker中，需要存储一个映射表：

[msgId 、root id 、task id]

注：这里的数据结构只是帮助理解，storm的具体实现可能会与此处不同，但是思路是一致的，这里不去纠结。

ack确认

基于上述表述，我们知道在一个spout或者bolt发送一个tuple的时候都做了哪些工作。当我们在Bolt中进行ack确认的时候，Storm会将发送的tuple和接受到的tuple的tuple id进行异或运算，直到结果为0，则表示一棵tuple已经完成确认。