# [Storm如何保证可靠的消息处理](http://www.cnblogs.com/Jack47/p/guaranteeing-message-processing-in-storm.html)

Storm可以保证从Spout发出的每个消息都能被完全处理。Storm的可靠性机制是完全分布式的(distributed)，可伸缩的(scalable)，容错的(fault-tolerant)。Storm中处理的消息是用元组这种数据结构来表示的。

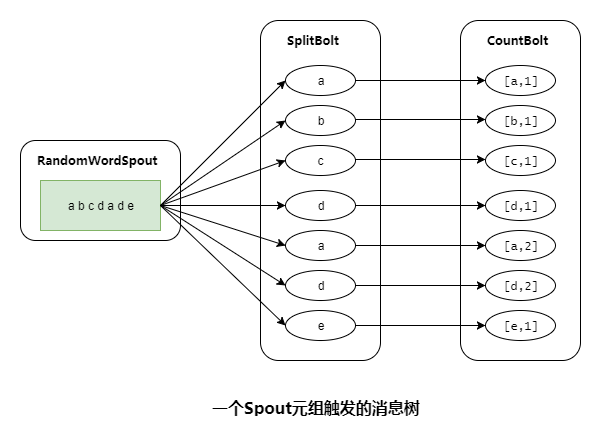
## wordcount的拓扑

|  |
| --- |
| // 创建一个拓扑  TopologyBuilder builder = **new** TopologyBuilder();  // 设置Spout，这个Spout的名字叫做"Spout"，设置并行度为2  builder.setSpout("Spout", **new** RandomWordSpout(), 2);  // 设置Blot——“Split”，并行度为2，它的数据来源是Spout的  builder.setBolt("Split", **new** SplitWordBolt(), 2).shuffleGrouping("Spout");  // 设置slot——“Count”,你并行度为2，它的数据来源是Split的word字段相同的数据  builder.setBolt("Count", **new** WordCountBolt(), 2).fieldsGrouping("Split", **new** Fields("word")); |

|  |
| --- |
| 这个拓扑由3个处理单元组成：一个叫"Spout"的Spout，负责随机生成一组单词作为新的Spout元组发送出去。名称为"Split"的Bolt是Spout元组的下游消费方，它把接收到句子切分成单词并发送出去。名称为"Count"的Bolt是"Split" Bolt的下游消费方，它使用HashMap<String, Long>存储了每个任务中每个单词出现的次数，每次读取到新的单词元组就让该单词的计数加一。"Count" Bolt接收"Split" Bolt发出的消息时，是使用元组中的"word"(单词)字段来作为路由策略，所以相同的单词元组会被路由到相同的任务(task)里，这样就能够计数了。 |

## 消息(元组)树(message tree or tuple tree)

在下游的Bolt中会基于某个Spout元组发射出很多新的元组：一组单词中的每个单词会生成一个新元组(在Split Bolt完成)，每个单词的计数更新后(在Count Bolt完成)也会触发一个新的元组。某个Spout元组触发的消息树如下图：



可以看到这棵消息树的根节点是Spout产生的句子内容为"a b c d a d e"的元组。这个Spout元组在"Split"这个Bolt里被切分为7个单词，触发了7个单词元组，"Count" Bolt接收到这7个单词元组后，更新了每个单词的计数并为之产生了一个新的元组。

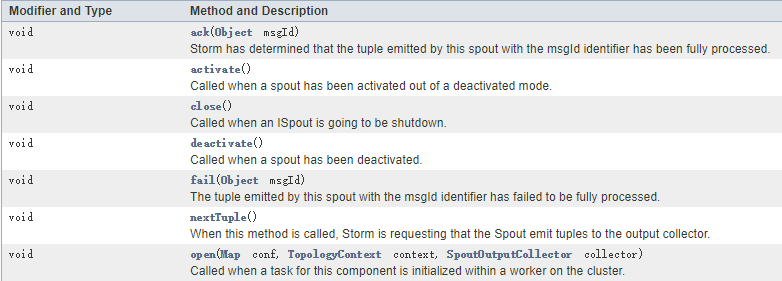
## 一条消息被“完整处理”

|  |
| --- |
| 指一个从Spout发出的元组所触发的消息树中所有的消息都被Storm处理了。如果在指定的超时时间里，这个Spout元组触发的消息树中有任何一个消息没有处理完，就认为这个Spout元组处理失败了。  这个超时时间是通过每个拓扑的Config.TOPOLOGY\_MESSAGE\_TIMEOUT\_SECS配置项来进行配置的，默认是30秒。 |

在前面消息树的例子里，只有消息树中所有的消息(包含一条Spout消息，七条split Bolt消息，七条count Bolt消息)都被Storm处理完了，才算是这条Spout消息被完整处理了。

## 消息被完整处理或者处理失败

当消息没有被完整处理或者处理失败了会怎么样？为了理解这个问题，应该首先看一下Spout发出的一个元组的生命周期。Spout需要实现的接口([接口文档](http://storm.apache.org/releases/1.2.2/javadocs/index.html))如下：



|  |
| --- |
| 首先，Storm通过调用Spout的nextTuple函数来从Spout请求一个元组，Spout任务使用open函数入参中提供的SpoutOutputCollector来给Spout任务的某个输出流发射一个新元组。当发射一个元组时，Spout提供了一个"消息标识"(message-id)，用来后续识别这个元组。例如：我们用UUID来作为MessageID发送出去。  \_collector.emit(**new** Values(word), UUID.*randomUUID*().toString());  接下来，元组就被发送到下游的Bolt进行消费，Storm会负责跟踪这个Spout元组创建的消息树。如果Storm检测到一个元组被完整地处理了，Storm会调用产生这个元组的Spout任务(Spout Bolt有多个任务来运行) 的ack函数，参数是Spout之前发送这个消息时提供给Storm的message-id。类似的，当元组处理超时或处理失败时，Storm会在元组对应的Spout任务上调用fail函数，参数是之前Spout发送这个消息时提供给Storm的message-id。这样应用程序通过实现Spout Bolt中的ack接口和fail接口来处理消息处理成功和失败的情况。例如当消息处理成功时记录当前处理的进度，当处理失败时，重新发送消息来对这个消息进行重新处理。注：1. 在消息队列中，消费消息不会被立即移除，而是处于挂起状态，下次从队列取消息，仍然会取到这些消息，只有处理成功后，才会从队列中移除这些消息。2. 如果Spout漏掉了消息ID，或者没有设置，那么Storm将不会跟踪消息。并且Spout也不会收到任何ack和fail消息的回调。 |

### 1. open()

**void** open(Map conf, TopologyContext context, SpoutOutputCollector collector);

该组件的一个任务此在集群的工作进程内被初始化时调​​用。提供了Spout执行所需要的环境。

conf 🡪 是这个Spout的Storm配置，提供给拓扑与这台主机上的集群配置一起进行合并。

context 🡪 可以用来获取关于这个任务在拓扑中的位置信息，包括该任务的id、该任务组件的id、输入和输出信息等。

collector 🡪 收集器，用于从这个 Spout 发射元组。元组可以随时被发射，包括 open

和 doseO方法。收集器是线程安全的，应该作为这个Spout 对象的实例变量进行保存。

## Storm的可靠性API

作为Storm开发者，如果想利用Storm的可靠性，需要做三件事：

|  |
| --- |
| 1. Spout发射元组时需提供消息ID  \_collector.emit(**new** Values(word), UUID.*randomUUID*().toString());  2. 创建一个元组时(消息树上创建一个新节点)需要通知Storm  /\*\*  \* 处理上游的每一个数据  \*/  @Override  **public** **void** execute(Tuple input) {  **if** (***LOGGER***.isDebugEnabled()) {  ***LOGGER***.debug("2.execute(Tuple input)...");  ***LOGGER***.debug("input mid == " + input.getMessageId());  }  String word = input.getStringByField("word");  String[] wordArray = word.split("\\s", -1);  **for** (String string : wordArray) {  \_collector.emit(input, **new** Values(string));  }  \_collector.ack(input);  }  3. 处理完一个元组，需要通知Storm  \_collector.ack(input);  // \_collector.fail(input); |

通过这两个操作，当消息树被完全处理完，Storm就可以立即检测到，从而可以正确地确认这个Spout元组处理成功或者失败。Storm的API提供了一套简洁地处理这些操作的方法。

### Spout发射元组时提供消息ID

\_collector.emit(**new** Values(word), UUID.*randomUUID*().toString());

注：如果Spout漏掉了消息ID，或者没有设置，那么Storm将不会跟踪消息。并且Spout也不会收到任何ack和fail消息的回调。

### 元组创建时通知Storm

在Storm消息树(元组树)中添加一个子结点的操作叫做锚定(anchoring)。在应用程序发送一个新元组时候，Storm会在幕后做锚定。

|  |
| --- |
| \_collector.emit(input, **new** Values(string)); |

每个单词元组是通过把输入的元组作为emit函数中的第一个参数来做锚定的。通过锚定，Storm就能够得到元组之间的关联关系(输入元组触发了新的元组)，继而构建出Spout元组触发的整个消息树。所以当下游处理失败时，就可以通知Spout当前消息树根节点的Spout元组处理失败，让Spout重新处理。相反，如果在emit的时候没有指定输入的元组，叫做不锚定：

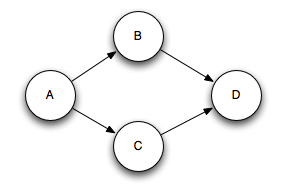
|  |
| --- |
| \_collector.emit(**new** Values(string)); |

这样发射单词元组，会导致这个元组不被锚定(unanchored)，这样Storm就不能得到这个元组的消息树，继而不能跟踪消息树是否被完整处理。这样下游处理失败，不能通知到上游的Spout任务。不同的应用的有不同的容错处理方式，有时候需要这样不锚定的场景。

一个输出的元组可以被锚定到多个输入元组上，叫做多锚定(multi-anchoring)。这在做流的合并或者聚合的时候非常有用。一个多锚定的元组处理失败，会导致Spout上重新处理对应的多个输入元组。多锚定是通过指定一个多个输入元组的列表而不是单个元组来完成的。例如：

|  |
| --- |
| List<Tuple> anchors = **new** ArrayList<Tuple>();  anchors.add(tuple1);  anchors.add(tuple2 );  \_collector.emit(anchors, **new** Values(word)); |

多锚定会把这个新输出的元组添加到多棵消息树上。注意多锚定可能会打破消息的树形结构，变成有向无环图(DAG)，Storm的实现既支持树形结构，也支持有向无环图(DAG)。在本文中，提到的消息树跟有向无环图是等价的。消息之间的关系是有向无环图的例子见下图：

消息形成的有向无环图

Spout元组A触发了B和C两个元组，而这两个元组作为输入，共同作用后触发D元组。

### 元组处理完后通知Storm

锚定的作用就是指定元组树的结构--下一步是当元组树中某个元组已经处理完成时，通知Storm。通知是通过OutputCollector中的ack和fail函数来完成的。例如上面流式计算单词个数例子中的Split Bolt的实现SplitBolt类，可以看到句子被切分成单词后，当所有的单词元组都被发射后，会确认(ack)输入的元组处理完成。

可以利用OutputCollector的fail函数来立即通知Storm，当前消息树的根元组处理失败了。例如，应用程序可能捕捉到了数据库客户端的一个异常，就显示地通知Storm输入元组处理失败。通过显示地通知Storm元组处理失败，这个Spout元组就不用等待超时而能更快地被重新处理。

Storm需要占用内存来跟踪每个元组，所以每个被处理的元组都必须被确认。因为如果不对每个元组进行确认，任务最终会耗光可用的内存。

做聚合或者合并操作的Bolt可能会延迟确认一个元组，直到根据一堆元组计算出了一个结果后，才会确认。聚合或者合并操作的Bolt，通常也会对他们的输出元组进行多锚定。

## Storm怎样高效的实现可靠性？

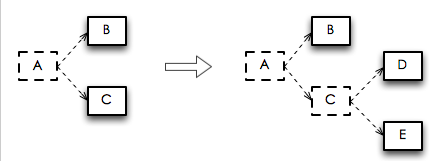
### acker任务

一个Storm拓扑有一组特殊的"acker"任务，它们负责跟踪由每个Spout元组触发的消息的处理状态。当一个"acker"看到一个Spout元组产生的有向无环图中的消息被完全处理，就通知当初创建这个Spout元组的Spout任务，这个元组被成功处理。可以通过拓扑配置项Config.TOPOLOGY\_ACKER\_EXECUTORS来设置一个拓扑中acker任务executor的数量。Storm默认TOPOLOGY\_ACKER\_EXECUTORS和拓扑中配置的Worker的数量相同，--对于需要处理大量消息的拓扑来说，需要增大acker executor的数量。

### 元组的生命周期

理解Storm的可靠性实现方式的最好方法是查看元组的生命周期和元组构成的有向无环图。当拓扑的Spout或者Bolt中创建一个元组时，Storm系统会为用户指定的MessageId生成一个对应的64位的整数，作为整个Tuple Tree的RootId。RootId会被传递给Acker以及后续的Bolt来作为该消息单元的唯一标识。同时，无论Spout还是Bolt每次新生成一个Tuple时，都会赋予该Tuple一个唯一的64位整数的Id。acker任务使用这些id来跟踪每个Spout元组产生的有向无环图的处理状态。在Bolt中产生一个新的元组时，会从锚定的一个或多个输入元组中拷贝所有Spout元组的message-id，所以每个元组都携带了自己所在元组树的根节点Spout元组的message-id。当确认一个元组处理成功了，Storm就会给对应的acker任务发送特定的消息--通知acker当前这个Spout元组产生的消息树中某个消息处理完了，而且这个特定消息在消息树中又产生了一个新消息(新消息锚定的输入是这个特定的消息)。

举个例子，假设"D"元组和"E"元组是基于“C”元组产生的，那么下图描述了确认“C”元组成功处理后，元组树的变化。图中虚线框表示的元组代表已经在消息树上被删除了：



确认元组成功处理后消息树的变化

由于在“C”从消息树中删除(通过acker函数确认成功处理)的同时，“D”和“E”也被添加到(通过emit函数来锚定的)元组树中，所以这棵树从来不会被提早处理完。

正如上面已经提到的，在一个拓扑中，可以有任意数量的acker任务。这导致了如下的两个问题：

1. 当拓扑中的一个元组确认被处理完，或者产生一个新的元组时，Storm应该通知哪个acker任务？

2. 通知了acker任务后，acker任务如何通知到对应的Spout任务？

Storm采用对元组中携带的Spout元组message-id哈希取模的方法来把一个元组映射到一个acker任务上(所以同一个消息树里的所有消息都会映射到同一个acker任务)。因为每个元组携带了自己所处的元组树中根节点Spout元组(可能有多个)的标识，所以Storm就能决定通知哪个acker任务。

当一个Spout任务产出一个新的元组，仅需要简单的发送一个消息给对应的acker(Spout元组message-id哈希取模)来告知Spout的任务标示(task id)，以此来通知acker当前这个Spout任务负责这个消息。当acker看到一个消息树被完全处理完，它就能根据处理的元组中携带的Spout元组message-id来确定产生这个Spout元组的task id，然后通知这个Spout任务消息树处理完成(调用 Spout任务的ack函数)。

## 调整可靠性

acker任务是轻量级的，所以在一个拓扑中不需要太多的acker任务。可以通过Storm UI(id为"\_\_acker"的组件）来观察acker任务的性能。如果吞吐量看起来不正常，就需要添加更多的acker任务。

## 去掉可靠性

如果可靠性无关紧要－－例如你不关心元组失败场景下的消息丢失－－那么你可以通过不跟踪元组的处理过程来提高性能。不跟踪一个元组树会让传递的消息数量减半，因为正常情况下，元组树中的每个元组都会有一个确认消息。另外，这也能减少每个元组需要存储的id的数量(指每个元组存储的Spout message-id)，减少了带宽的使用。

有三种方法来去掉可靠性：

1. 设置Config.TOPOLOGY\_ACKERS为0。这种情况下，Storm会在Spout吐出一个元组后立马调用Spout的ack函数。这个元组树不会被跟踪。

2. 当产生一个新元组调用emit函数的时候通过忽略消息message-id参数来关闭这个元组的跟踪机制。

3. 如果你不关心某一类特定的元组处理失败的情况，可以在调用emit的时候不要使用锚定。由于它们没有被锚定到某个Spout元组上，所以当它们没有被成功处理，不会导致Spout元组处理失败。

# Storm定时机制（TickTuple）

Storm中内置了一种定时机制——tick，它能够让任何bolt的所有task每隔一段时间（精确到秒级，用户可以自定义）收到一个来自systemd的tick stream的tick tuple，bolt收到这样的tuple后可以根据业务需求完成相应的处理。

## 为bolt设置tick

若希望某个bolt每隔一段时间做一些操作，那么可以将bolt继承BaseBasicBolt/BaseRichBolt，并重写getComponentConfiguration()方法。在方法中设置Config.TOPOLOGY\_TICK\_TUPLE\_FREQ\_SECS的值，单位是秒。

|  |
| --- |
| @Override  **public** Map<String, Object> getComponentConfiguration() {  // return super.getComponentConfiguration();  Config config = **new** Config();  config.put(Config.***TOPOLOGY\_TICK\_TUPLE\_FREQ\_SECS***, 60);  **return** config;  } |

这样设置之后，此bolt的所有task都会每隔一段时间收到一个来自systemd的tick stream的tick tuple。

判断收到的Tuple是否为Tick\_Tuple

|  |
| --- |
| /\*\*  \* 根据传送过来的Tuple，判断本Tuple是否是tickTuple 如果是tickTuple，则触发相应的动作  \* **@param** tuple  \* **@return**  \*/  **public** **static** **boolean** isTickTuple(Tuple tuple){  **return** tuple.getSourceComponent().equals(Constants.***SYSTEM\_COMPONENT\_ID***)  && tuple.getSourceStreamId().equals(Constants.***SYSTEM\_TICK\_STREAM\_ID***);  } |

## 全局tick

若希望Topology中的每个bolt都每隔一段时间做一些操作，那么可以定义一个Topology全局的tick，同样是设置Config.TOPOLOGY\_TICK\_TUPLE\_FREQ\_SECS的值：

|  |
| --- |
| Config config = **new** Config();  config.put(Config.***TOPOLOGY\_TICK\_TUPLE\_FREQ\_SECS***, 60);  StormSubmitter.*submitTopology*(toponame, conf, builder.createTopology()); |

## 优先级

当我们在整个Topology上设置tick和我们单个运算bolt上冲突时，其优先级如何呢？事实是在更小范围的bolt设置的tick优先级更高

## 定时精度问题

Config.TOPOLOGY\_TICK\_TUPLE\_FREQ\_SECS是精确到秒级的。例如某bolt设置Config.TOPOLOGY\_TICK\_TUPLE\_FREQ\_SECS为10s，理论上说bolt的每个task应该每个10s收到一个tick tuple。实际测试发现，这个时间间隔的精确性是很高的，一般延迟（而不是提前）时间在1-2ms左右。

# Storm集成Kafka

在调试过程中遇到的几个问题：（1）topic的路径错误（提示找不到topic），我kafka集群配置中zookeeper.connect=risbig14:2181,risbig15:2181,risbig16:2181/kafka，所以brokerZkPath就变成了/kafka/brokers，默认情况下，kafka的topic在zk上的目录为/brokers；

ZooKeeper连接字符串的格式为：hostname:port，此处hostname和port分别是ZooKeeper集群中某个节点的host和port；为了当某个host宕掉之后你能通过其他ZooKeeper节点进行连接，你可以按照一下方式制定多个hosts：

hostname1:port1, hostname2:port2, hostname3:port3.

ZooKeeper 允许你增加一个“chroot”路径，将集群中所有kafka数据存放在特定的路径下。当多个Kafka集群或者其他应用使用相同ZooKeeper集群时，可以使用这个方式设置数据存放路径。这种方式的实现可以通过这样设置连接字符串格式，如下所示：

hostname1：port1，hostname2：port2，hostname3：port3/chroot/path

这样设置就将所有kafka集群数据存放在/chroot/path路径下。注意，在你启动broker之前，你必须创建这个路径，并且consumers必须使用相同的连接格式。

需要手动在ZooKeeper中创建路径/kafka，使用如下命令连接到任意一台ZooKeeper服务器：

cd /usr/local/zookeeper

bin/zkCli.sh

在ZooKeeper执行如下命令创建chroot路径：

create /kafka ''

这样，每次连接Kafka集群的时候（使用--zookeeper选项），也必须使用带chroot路径的连接字符串