http://weyo.me/pages/techs/storm-translations-guaranteeing-message-processing/

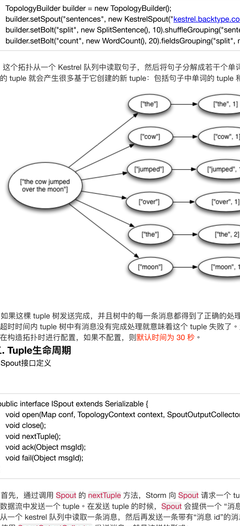
Storm 是够保证每一个由 Spout 发送的消息都能够得到完整地处理。本文详细解释了 Storm 如何实现这种保障机制，以及作为用户如何使用好 Storm 的可靠性机制。

一. 完整性处理

一个从 spout 中发送出的 tuple 会产生上千个基于它创建的 tuples。
例如，有这样一个 word-count 拓扑：

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("sentences", new KestrelSpout("kestrel.backtype.com", 22133, "sentence_queue", new StringScheme()));
builder.setBolt("split", new SplitSentence(), 10).shuffleGrouping("sentences");
builder.setBolt("count", new WordCount(), 20).fieldsGrouping("split", new Fields("word"));
```

这个拓扑从一个 Kestrel 队列中读取句子，然后将句子分解成若干个单词，然后将它每个单词和该单词的数量发送出去。这种情况下，从 spout 中发出的 tuple 就会产生很多基于它创建的新 tuple：包括句子中单词的 tuple 和 每个单词的个数的 tuple。这些消息构成了这样一棵树：



如果这棵 tuple 树发送完成，并且树中的每一条消息都得到了正确的处理，就表明发送 tuple 的 spout 已经得到了“完整性处理”。对应的，如果在指定的超时时间内 tuple 树中有消息没有完成处理就意味着这个 tuple 失败了。这个超时时间可以使用 Config.TOPOLOGY_MESSAGE_TIMEOUT_SECS 参数在构造拓扑时进行配置，如果不配置，则默认时间为 30 秒。

二. Tuple生命周期

Spout接口定义

```
public interface ISpout extends Serializable {
    void open(Map conf, TopologyContext context, SpoutOutputCollector collector);
    void close();
    void nextTuple();
    void ack(Object msgId);
    void fail(Object msgId);
}
```

首先，通过调用 Spout 的 nextTuple 方法，Storm 向 Spout 请求一个 tuple。Spout 会使用 open 方法中提供的 SpoutOutputCollector 向它的一个输出数据流中发送一个 tuple。在发送 tuple 的时候，Spout 会提供一个“消息 id”，这个 id 会在后续过程中用于识别 tuple。例如，上面的 KestrelSpout 就是从一个 kestrel 队列中读取一条消息，然后再发送一条带有“消息 id”的消息，这个 id 是由 Kestrel 提供的。
使用 SpoutOutputCollector 发送消息一般是这样的形式：

```
_collector.emit(new Values("field1", "field2", 3) , msgId);
```

随后，tuple 会被发送到对应的 bolt 中去，在这个过程中，Storm 会很小心地跟踪创建的消息树。如果 Storm 检测到某个 tuple 被完整处理。 Storm 会根据 Spout 提供的“消息 id”调用最初发送 tuple 的 Spout 任务的 ack 方法。对应的，Storm 在检测到 tuple 超时之后就会调用 fail 方法。注意，对于一个特定的 tuple，响应（ack）和失败处理（fail）都只会由最初创建这个 tuple 的任务执行。也就是说，即便 Spout 在集群中有很多个任务，某个特定的 tuple 也只会由创建它的那个任务 —— 而不是其他的任务 —— 来处理成功或失败的结果。

我们再以 KestrlSpout 为例来看看在消息的可靠性处理中 Spout 做了什么。在 KestrlSpout 从 Kestrel 队列中取出一条消息时，可以看作它“打开”了这条消息。也就是说，这条消息实际上并没有从队列中真正地取出来，而是保持着一个“挂起”状态，等待消息处理完成的信号。在挂起状态的消息不回被发送到其他的消费者中。另外，如果消费者（客户端）断开了连接，所有处于挂起状态的消息都会重新放回到队列中。在消息“打开”的时候 Kestrel 会给客户端同时提供消息体数据和一个唯一的 id。KestrelSpout 在使用 SpoutOutputCollector 发送 tuple 的时候就会把这个唯一的 id 当作“消息 id”。一段时间之后，在 KestrelSpout 的 ack 或者 fail 方法被调用的时候，KestrelSpout 就会通过这个消息 id 向 Kestrel 请求将消息从队列中移除（对应 ack 的情况）或者将消息重新放回队列（对应 fail 的情况）。

三. Storm可靠性API

使用 Storm 的可靠性机制的时候你需要注意两件事：

1. 首先，在 tuple 树中创建新节点连接时务必通知 Storm；
2. 其次，在每个 tuple 处理结束的时候也必须向 Storm 发出通知。通过这两个操作，Storm 就能够检测到 tuple 树会在何时完成处理，并适时地调用 ack 或者 fail 方法。Storm 的 API 提供了一种非常精确的方式来实现着两个操作。

Storm 中指定 tuple 树中的一个连接称为“锚定”（anchoring）。锚定是在发送新 tuple 的同时发生的。让我们以下面的 Bolt 为例说明这一点。这个 Bolt将一个包含句子的 tuple 分割成若干个单词 tuple：

```
public class SplitSentence extends BaseRichBolt {
    OutputCollector _collector;

    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
      _collector = collector;
    }

    public void execute(Tuple tuple) {
      String sentence = tuple.getString(0);
      for(String word: sentence.split(" ")) {
        _collector.emit(tuple, new Values(word));
      }
      _collector.ack(tuple);
    }

    public void declareOutputFields(OutputFields Declarer declarer) {
      declarer.declare(new Fields("word"));
    }
}
```

通过将输入 tuple 指定为 emit 方法的第一个参数，每个单词 tuple 都被“锚定”了。这样，如果单词 tuple 在后续处理过程中失败了，作为这棵 tuple 树的根节点的原始 Spout tuple 就会被重新处理。相对应的，如果这样发送 tuple：

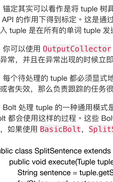
```
_collector.emit(new Values(word));
```

就称为“非锚定”。在这种情况下，下游的 tuple 处理失败不会触发原始 tuple 的任何处理操作。有时候发送这种“非锚定” tuple 也是必要的，这取决于你的拓扑的容错性要求。

一个输出 tuple 可以被锚定到多个输入 tuple 上，这在流式连接或者整合操作时很有用。显然，一个多锚定的 tuple 失败会导致 Spout 中多个 tuple 的重新处理。多锚定操作是通过指定一个 tuple 列表而不是单一的 tuple 来实现的。如下面的例子所示：

```
List<Tuple> anchors = new ArrayList<Tuple>();
anchors.add(tuple1);
anchors.add(tuple2);
_collector.emit(anchors, new Values(1, 2, 3));
```

多锚定操作会把输出 tuple 添加到多个 tuple 树中。注意，多锚定也可能会打破树的结构从而创建一个 tuple 的有向无环图（DAG），如下图所示：



Storm 的程序实现既支持对树的处理，同样也支持对 DAG 的处理（由于早期的 Storm 版本仅仅对树有效，所以“tuple 树”的这个糟糕的概念就一直沿袭下来了）。

锚定其实可以看作是将 tuple 树真象化的过程 —— 在结束对一棵 tuple 树中一个单独 tuple 的处理的时候，后续以及最终的 tuple 都会在 Storm 可靠性 API 的作用下得到标定。这是通过 OutputCollector 的 ack 和 fail 方法实现的。如果你再回过头看一下 SplitSentence 的例子，你就会发现输入 tuple 是在所有的单词 tuple 发送出去之后被 ack 的。

你可以使用 OutputCollector 的 fail 方法来使得位于 tuple 树根节点的 Spout tuple 立即失败。例如，你的应用可以在建立数据库连接的时候抓取异常，并且在异常出现的时候立即让输入 tuple 失败。通过这种立即失败的方式，原始 Spout tuple 就会比等待 tuple 超时的方式响应更快。

每个待处理的 tuple 都必须显式地应答（ack）或者失败（fail）。因为 Storm 是使用内存来跟踪每个 tuple 的，所以，如果你不对每个 tuple 进行应答或者失败，那么负责跟踪的任务很快就会发生内存溢出。

Bolt 处理 tuple 的一种通用模式是在 execute 方法中读取输入 tuple，发送出基于输入 tuple 的新 tuple，然后在方法末尾对 tuple 进行应答。大部分 Bolt 都会使用这样的过程。这些 Bolt 大多属于过滤器或者简单的处理函数一类。Storm 有一个可以简化这种操作的简便接口，称为 BasicBolt。例如，如果使用 BasicBolt，SplitSentence 的例子可以这样写：

```
public class SplitSentence extends BaseBasicBolt {
    public void execute(Tuple tuple, BasicOutputCollector collector) {
      String sentence = tuple.getString(0);
      for(String word: sentence.split(" ")) {
        collector.emit(new Values(word));
      }
    }

    public void declareOutputFields(OutputFields Declarer declarer) {
      declarer.declare(new Fields("word"));
    }
}
```

这个实现方式比之前的方式要简单许多，而且在语义上有着完全一致的效果。发送到 BasicOutputCollector 的 tuple 会被自动锚定到输入 tuple 上，而且输入 tuple 会在 execute 方法结束的时候自动应答。
相对应的，执行聚合或者联结操作的 Bolt 可能需要延迟应答 tuple，因为它需要等待一批 tuple 来完成某种结果计算。整合和联结操作一般也会需要对他们的输出 tuple 进行多锚定。这个过程已经超出了 IBasicBolt 的应用范围。

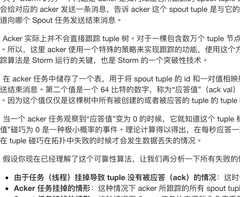
四. Storm如何实现高可靠性

Storm 的拓扑有一些特殊的称为“acker”的任务，这些任务负责跟踪每个 Spout 发出的 tuple 的 DAG。当一个 acker 发现一个 DAG 结束了，它就会给创建 spout tuple 的 Spout 任务发送一条消息，让这个任务来应答这个消息。你可以使用 Config.TOPOLOGY_ACKERS 来配置拓扑的 acker 数量。Storm 默认会将 acker 的数量设置为一，不过如果你有大量消息的处理需求，你可能需要增加这个数量。

理解 Storm 的可靠性实现的最好方式还是通过了解 tuple 和 tuple DAG 的生命周期。当一个 tuple 在拓扑中被创建出来的时候 —— 不管是在 Spout 中还是在 Bolt 中创建的 —— 这个 tuple 都会被配置一个随机的 64 位 id。acker 就是使用这些 id 来跟踪每个 spout tuple 的 tuple DAG 的。

Spout tuple 的 tuple 树中的每个 tuple 都知道 spout tuple 的 id。当你在 bolt 中发送一个新 tuple 的时候，输入 tuple 中的所有 spout tuple 的 id 都会被复制到新的 tuple 中。在 tuple 被 ack 的时候，它会通过回掉函数向合适的 acker 发送一条消息，这条消息显示了 tuple 树中发生的变化。也就是说，它会告诉 acker 这样一条消息：“在这个 tuple 树中，我的处理已经结束了，接下来这个 tuple 就要被我标记的新 tuple”。

以下图为例，如果 D tuple 和 E tuple 是由 C tuple 创建的，那么在 C 应答的时候 tuple 树就会发生变化：



由于在 D 和 E 添加到 tuple 树中的时候 C 已经从树中移除了，所以这个树并不会被过早地结束。

关于 Storm 如何跟踪 tuple 树还有更多的细节，正如上面所提到的，你可以随意设置拓扑中 acker 的数量。这就会引起下面的问题：当 tuple 在拓扑中被 ack 的时候，它是怎么知道向那个 acker 任务发送信息的？对于这个问题，Storm 实际上是使用哈希算法来将 spout tuple 已配到 acker 任务上的。由于每个 tuple 都会包含原始的 spout tuple id，所以他们会知道需要与哪个 acker 任务通信。

关于 Storm 的另一个问题是 acker 是如何知道它所跟踪的 spout tuple 是由哪个 Spout 任务处理的。实际上，在 Spout 任务发送新 tuple 的时候，它也会将对应的 acker 发送一条消息，告诉 acker 这个 spout tuple 是与它的任务 id 相关联的。随后，在 acker 观察到 tuple 树结束处理的时候，它就会知道向哪个 Spout 任务发送结束消息。

Acker 实际上并不会直接跟踪 tuple 树。对于一棵包含数万个 tuple 节点的树，如果直接跟踪其中的每个 tuple，显然会很快把这个 acker 的内存撑爆。所以，这里 acker 使用一个特殊的策略来实现跟踪的功能。使用这个方法对于每个 spout tuple 只需要占用固定的内存空间（大约 20 字节）。这个跟踪算法是 Storm 运行的关键，也是 Storm 的一个突破性技术。

在 acker 任务中储存了一个表，用于将 spout tuple 的 id 和一对值相映射。其中第一个值是创建这个 tuple 的任务 id，这个 id 主要用于在后续操作中发送结束消息。第二个值是一个 64 比特的数字，称为“应答值”（ack val）。这个应答值是整个 tuple 树的一个完整的状态表述，而且它与树的大小无关。因为这个值仅仅是这棵树中所有被创建的或者被应答的 tuple 的 id 进行异或运算的结果值。

当一个 acker 任务观察到“应答值”变为 0 的时候，它就知道这个 tuple 树已经完成处理了。因为 tuple id 实际上是随机生成的 64 比特数值，所以“应答值”碰巧为 0 是一种极小概率的事件。理论计算得以得出，在每秒应答一万次的情况下，需要 5000 万年才会发生一次错误。而且即便是这样，也仅仅会在 tuple 碰巧在拓扑中失败的时候才会发生数据丢失的情况。

假设你现在已经理解了这个可靠性算法，让我们再分析一下所有失败的情形，看看这些情形下 Storm 是如何避免数据缺失的：

- **由于任务（线程）挂掉导致 tuple 没有被应答（ack）的情况**：这时位于 tuple 树根节点的 spout tuple 会在任务超时后得到重新处理。
- **Acker 任务挂掉的情形**：这种情况下 acker 所跟踪的所有 spout tuple 都会由于超时被重新处理。
- **Spout 任务挂掉的情形**：这种情况下 Spout 任务的来源就会负责重新处理消息。例如，对于像 Kestrel 和 RabbitMQ 这样的消息队列就会在客户端断开连接时将所有的挂起状态的消息放回队列（关于挂起状态的概念可以参考Storm 的容错性 —— 译者注）。

五. 调整可靠性

由于 acker 任务是轻量级的，在拓扑中你并不需要很多 acker 任务。你可以通过 Storm UI 监控他们的性能（acker 任务的 id 为“__acker”）。如果发现观察结果存在问题，你可能就需要增加更多的 acker 任务。

如果你不关注消息的可靠性 —— 也就是说你不关心在失败情形下发生的 tuple 丢失 —— 那么你就可以通过不跟踪 tuple 树的处理来提升拓扑的性能。由于 tuple 树中的每个 tuple 都会带有一个应答消息，不追踪 tuple 树会使得传输的消息的数量减半。同时，下游数据流中的 id 也会变少，这样可以降低网络带宽的消耗。

有三种方法可以移除 Storm 的可靠性机制。

1. 第一种方法是将 Config.TOPOLOGY_ACKERS 设置为0，在这种情况下，Storm 会在 Spout 发送 tuple 之后立即调用 ack 方法，tuple 树叶就不会被跟踪了。
2. 第二种方法是基于消息本身移除可靠性。你可以通过在 SpoutOutputCollector.emit 方法中省略消息 id 来关闭 spout tuple 的跟踪功能。
3. 最后，如果你不关心拓扑中的下游 tuple 是否会失败，你可以在发送 tuple 的时候选择发送“非锚定”的（unanchored）tuple（（注意，在使用这种方法时，如果上游有 spout 或 bolt 仍然保持可靠性机制，那么需要在 execute 方法之初调用 OutputCollector.ack 来立即响应上游的消息，否则上游组件会误认为消息没有发送成功导致所有的消息会被反复发送 —— 译者注）。）由于这些 tuple 不会被标记到任何一个 spout tuple 中，显然在他们处理失败的时候不会引起任何 spout tuple 的重新处理。