Apache Storm 官方文档中文版

参考：<http://ifeve.com/apache-storm/>

# About

本项目是 Apache Storm 官方文档的中文翻译版，致力于为有实时流计算项目需求和对 Apache Storm 感兴趣的同学提供有价值的中文资料，希望能够对大家的工作和学习有所帮助。

虽然 Storm 的正式推出已经有好几个年头了，发行版也已经到了 1.0.x，但是目前网络上靠谱的学习资料仍然不多，很多比较有价值的资料都过时了（甚至官方网站自己的资料都没有及时更新，这大概也是发展太快的社区的通病），而较新的资料大多比较零碎，在关键内容的描述上也有些模棱两可，给初学者带来了很大的困扰。本人自己在初学 Storm 的阶段就非常痛苦，一直想有一份较系统、实用的资源来方便学习。最近借着整理工作的机会，就下定决心通过官方文档的翻译梳理出 Storm 的技术路线，于是就有了这个翻译项目。非常感谢并发编程网对本项目的支持，由于本人水平有限，翻译中仍然存在不少问题，还请大家不吝斧正。如果对本项目有任何问题，欢迎在评论中或者本项目的 Github 页面中（<https://github.com/weyo/Storm-Documents>）提出，另外也可以直接给本人发邮件（ivicoco at gmail.com），谢谢。

说明：如果没有特殊声明，本项目文档中所述 Storm 版本均为 0.9.x 版本。

# Storm 官方文档索引

原文资料来源（官方网站）：<http://storm.apache.org/documentation/Documentation.html>

## Storm 基础篇

* [Javadoc](http://storm.apache.org/javadoc/apidocs/index.html)1
* [基础概念](http://ifeve.com/storm-concepts/)
* [配置](http://ifeve.com/storm-configuration/)
* [消息的可靠性保障](http://ifeve.com/storm-guaranteeing-message-processing/)
* [容错性](http://ifeve.com/storm-fault-tolerance/)
* [命令行操作](http://ifeve.com/storm-command-line-client/)
* [理解 Storm 拓扑的并行度(parallelism)概念](http://ifeve.com/storm-understanding-the-parallelism-of-a-storm-topology/)
* [FAQ](http://ifeve.com/storm-faq/)

## Trident

*Trident*是 Storm 的一种高级操作接口，它能够提供可靠的数据流一次性处理模式、“事务型”数据持久化存储功能以及一系列数据流分析操作通用组件。

* [Trident 教程 —— 基本概念与参考手册](http://ifeve.com/storm-trident-tutorial/)
* [Trident API 概述 —— 数据的转换与整合操作](http://ifeve.com/storm-trident-api-overview/)
* [Trident State —— 恰好一次的数据处理与快速、持久化的聚合操作](http://ifeve.com/storm-trident-state/)
* [Trident Spouts —— 事务型与非事务型数据入口](http://ifeve.com/storm-trident-spouts/)

## 配置与部署

* [配置 Storm 集群](http://ifeve.com/storm-setting-up-a-storm-cluster/)
* [配置开发环境](http://ifeve.com/storm-setting-up-a-development-environment/)
* [本地模式](http://ifeve.com/storm-local-mode/)
* [问题与解决](http://ifeve.com/storm-troubleshooting/)
* [在生产环境中运行 topology](http://ifeve.com/storm-running-topologies-on-a-production-cluster/)
* [使用 Maven 构建 Storm 应用](http://ifeve.com/storm-building-storm-with-maven/)

## Storm 中级篇

* [序列化](http://ifeve.com/storm-serialization/)
* [常用模式](http://ifeve.com/storm-common-patterns/)
* Clojure DSL2
* [使用非 JVM 语言开发](http://ifeve.com/storm-using-non-jvm-languages-with-storm/)
* [分布式 RPC](http://ifeve.com/storm-distributed-rpc/)3
* 事务型拓扑4
* [Storm 与 Kestrel](http://ifeve.com/storm-kestrel-and-storm/)
* 直接数据流组5
* [Hooks](http://ifeve.com/storm-hooks/)
* [Metrics](http://ifeve.com/storm-metrics/)
* Trident tuple 的生命周期5

## Storm 高级篇

* [定义 Storm 的非 JVM 语言 DSL](http://ifeve.com/storm-defining-a-non-jvm-dsl-for-storm/)
* [多语言接口协议（如何定义其他语言的接口）](http://ifeve.com/storm-multilang-protocol/)
* [技术实现相关文档](http://ifeve.com/storm-internal-implementation/)

## 说明

1 JavaDoc 不在翻译计划之中。  
2 由于译者对 Clojure 不是很熟悉，相关内容暂时没有翻译，非常欢迎熟悉 Clojure 的同学来参与到翻译中。  
3 由于官方文档关于分布式 RPC 的部分内容已过时，这里改写了相关内容。  
4 事务型拓扑已经由 Trident 实现，之前的实现已经被标记为 @Deprecated，这里不再讨论。  
5 该文官方文档暂未提供。

# Apache Storm 官方文档 —— 基础概念

[原文链接](http://storm.apache.org/documentation/Concepts.html)    译者：[魏勇](http://weyo.me/)

Storm 系统中包含以下几个基本概念：

1. 拓扑（Topologies）
2. 流（Streams）
3. 数据源（Spouts）
4. 数据流处理组件（Bolts）
5. 数据流分组（Stream groupings）
6. 可靠性（Reliability）
7. 任务（Tasks）
8. 工作进程（Workers）

译者注：由于 Storm 的几个基础概念无论是直译还是意译均不够清晰，而且还会让习惯了 Storm 编程模型的读者感到困惑，因此后文在提及这些概念时大多还会以英文原文出现，希望大家能够谅解。

## 拓扑（Topologies）

Storm 的拓扑是对实时计算应用逻辑的封装，它的作用与 MapReduce 的任务（Job）很相似，区别在于 MapReduce 的一个 Job 在得到结果之后总会结束，而拓扑会一直在集群中运行，直到你手动去终止它。拓扑还可以理解成由一系列通过数据流（Stream Grouping）相互关联的 Spout 和 Bolt 组成的的拓扑结构。Spout 和 Bolt 称为拓扑的组件（Component）。我们会在后文中给出这些概念的解释。

**相关资料**

* [TopologyBuilder](http://storm.apache.org/javadoc/apidocs/backtype/storm/topology/TopologyBuilder.html)：在 Java 中使用此类构造拓扑
* [在生产环境中运行拓扑](http://ifeve.com/storm-running-topologies-on-a-production-cluster)
* [本地模式](http://ifeve.com/storm-local-mode)：通过本文学习如何在本地模式中开发、测试拓扑

## 数据流（Streams）

数据流（Streams）是 Storm 中最核心的抽象概念。一个数据流指的是在分布式环境中并行创建、处理的一组元组（tuple）的无界序列。数据流可以由一种能够表述数据流中元组的域（fields）的模式来定义。在默认情况下，元组（tuple）包含有整型（Integer）数字、长整型（Long）数字、短整型（Short）数字、字节（Byte）、双精度浮点数（Double）、单精度浮点数（Float）、布尔值以及字节数组等基本类型对象。当然，你也可以通过定义可序列化的对象来实现自定义的元组类型。

在声明数据流的时候需要给数据流定义一个有效的 id。不过，由于在实际应用中使用最多的还是单一数据流的 Spout 与 Bolt，这种场景下不需要使用 id 来区分数据流，因此可以直接使用 [OutputFieldsDeclarer](http://storm.apache.org/javadoc/apidocs/backtype/storm/topology/OutputFieldsDeclarer.html)来定义“无 id”的数据流。实际上，系统默认会给这种数据流定义一个名为“default”的 id。

**相关资料**

* 元组（Tuple）：数据流由多个元组构成
* [OutputFieldsDeclarer](http://storm.apache.org/javadoc/apidocs/backtype/storm/topology/OutputFieldsDeclarer.html)：用于声明数据流和数据流对应的模式
* [序列化（Serialization）](http://ifeve.com/storm-serialization)：关于 Storm 元组的动态类型以及声明自定义序列化模型的相关内容
* [ISerialization](http://storm.apache.org/javadoc/apidocs/backtype/storm/serialization/ISerialization.html)：自定义的序列化模型必须实现该接口
* [CONFIG.TOPOLOGY\_SERIALIZATIONS](http://storm.apache.org/javadoc/apidocs/backtype/storm/Config.html#TOPOLOGY_SERIALIZATIONS)：自定义的序列化模型可以通过这个配置项实现注册

## 数据源（Spouts）

数据源（Spout）是拓扑中数据流的来源。一般 Spout 会从一个外部的数据源读取元组然后将他们发送到拓扑中。根据需求的不同，Spout 既可以定义为**可靠的**数据源，也可以定义为**不可靠的**数据源。一个可靠的 Spout 能够在它发送的元组处理失败时重新发送该元组，以确保所有的元组都能得到正确的处理；相对应的，不可靠的 Spout 就不会在元组发送之后对元组进行任何其他的处理。

一个 Spout 可以发送多个数据流。为了实现这个功能，可以先通过 [OutputFieldsDeclarer](http://storm.apache.org/javadoc/apidocs/backtype/storm/topology/OutputFieldsDeclarer.html) 的 declareStream 方法来声明定义不同的数据流，然后在发送数据时在 [SpoutOutputCollector](http://storm.apache.org/javadoc/apidocs/backtype/storm/spout/SpoutOutputCollector.html) 的 emit 方法中将数据流 id 作为参数来实现数据发送的功能。

Spout 中的关键方法是 nextTuple。顾名思义，nextTuple 要么会向拓扑中发送一个新的元组，要么会在没有可发送的元组时直接返回。需要特别注意的是，由于 Storm 是在同一个线程中调用所有的 Spout 方法，nextTuple 不能被 Spout 的任何其他功能方法所阻塞，否则会直接导致数据流的中断（关于这一点，阿里的 JStorm 修改了 Spout 的模型，使用不同的线程来处理消息的发送，这种做法有利有弊，好处在于可以更加灵活地实现 Spout，坏处在于系统的调度模型更加复杂，如何取舍还是要看具体的需求场景吧——译者注）。

Spout 中另外两个关键方法是 ack 和 fail，他们分别用于在 Storm 检测到一个发送过的元组已经被成功处理或处理失败后的进一步处理。注意，ack 和 fail 方法仅仅对上述“可靠的” Spout 有效。

**相关资料**

* [IRichSpout](http://storm.apache.org/javadoc/apidocs/backtype/storm/topology/IRichSpout.html)：这是实现 Spout 的接口
* [消息的可靠性处理](http://ifeve.com/storm-guaranteeing-message-processing)

## 数据流处理组件（Bolts）

拓扑中所有的数据处理均是由 Bolt 完成的。通过数据过滤（filtering）、函数处理（functions）、聚合（aggregations）、联结（joins）、数据库交互等功能，Bolt 几乎能够完成任何一种数据处理需求。

一个 Bolt 可以实现简单的数据流转换，而更复杂的数据流变换通常需要使用多个 Bolt 并通过多个步骤完成。例如，将一个微博数据流转换成一个趋势图像的数据流至少包含两个步骤：其中一个 Bolt 用于对每个图片的微博转发进行滚动计数，另一个或多个 Bolt 将数据流输出为“转发最多的图片”结果（相对于使用2个Bolt，如果使用3个 Bolt 你可以让这种转换具有更好的可扩展性）。

与 Spout 相同，Bolt 也可以输出多个数据流。为了实现这个功能，可以先通过 [OutputFieldsDeclarer](http://storm.apache.org/javadoc/apidocs/backtype/storm/topology/OutputFieldsDeclarer.html) 的 declareStream 方法来声明定义不同的数据流，然后在发送数据时在 [OutputCollector](http://storm.apache.org/javadoc/apidocs/backtype/storm/task/OutputCollector.html) 的 emit 方法中将数据流 id 作为参数来实现数据发送的功能。

在定义 Bolt 的输入数据流时，你需要从其他的 Storm 组件中订阅指定的数据流。如果你需要从其他所有的组件中订阅数据流，你就必须要在定义 Bolt 时分别注册每一个组件。对于声明为默认 id（即上文中提到的“default”——译者注）的数据流，[InputDeclarer](http://storm.apache.org/javadoc/apidocs/backtype/storm/topology/InputDeclarer.html)支持订阅此类数据流的语法糖。也就是说，如果需要订阅来自组件“1”的数据流，declarer.shuffleGrouping("1") 与 declarer.shuffleGrouping("1", DEFAULT\_STREAM\_ID) 两种声明方式是等价的。

Bolt 的关键方法是 execute 方法。execute 方法负责接收一个元组作为输入，并且使用 [OutputCollector](http://storm.apache.org/javadoc/apidocs/backtype/storm/task/OutputCollector.html) 对象发送新的元组。如果有消息可靠性保障的需求，Bolt 必须为它所处理的每个元组调用 OutputCollector 的 ack 方法，以便 Storm 能够了解元组是否处理完成（并且最终决定是否可以响应最初的 Spout 输出元组树）。一般情况下，对于每个输入元组，在处理之后可以根据需要选择不发送还是发送多个新元组，然后再响应（ack）输入元组。[IBasicBolt](http://storm.apache.org/javadoc/apidocs/backtype/storm/topology/IBasicBolt.html) 接口能够实现元组的自动应答。

请注意 [OutputCollector](http://storm.apache.org/javadoc/apidocs/backtype/storm/task/OutputCollector.html) **不是**线程安全的对象，所有的 emit、ack 和 fail 操作都需要在同一个线程中进行处理。更多信息请参考[问题与解决](http://ifeve.com/storm-troubleshooting/)一文。

**相关资料**

* [IRichBolt](http://storm.apache.org/javadoc/apidocs/backtype/storm/topology/IRichBolt.html)：用于定义 Bolt 的基本接口
* [IBasicBolt](http://storm.apache.org/javadoc/apidocs/backtype/storm/topology/IBasicBolt.html): 用于定义带有过滤或者其他简单的函数操作功能的 Bolt 的简便接口
* [OutputCollector](http://storm.apache.org/javadoc/apidocs/backtype/storm/task/OutputCollector.html)：Bolt 使用此类来发送数据流
* [消息的可靠性处理](http://ifeve.com/storm-guaranteeing-message-processing)

## 数据流分组（Stream groupings）

为拓扑中的每个 Bolt 确定输入数据流是定义一个拓扑的重要环节。数据流分组定义了在 Bolt 的不同任务（tasks）中划分数据流的方式。

在 Storm 中有八种内置的数据流分组方式（原文有误，现在已经有八种分组模型——译者注），而且你还可以通过[CustomStreamGrouping](http://storm.apache.org/javadoc/apidocs/backtype/storm/grouping/CustomStreamGrouping.html) 接口实现自定义的数据流分组模型。这八种分组分时分别为：

1. 随机分组（Shuffle grouping）：这种方式下元组会被尽可能随机地分配到 Bolt 的不同任务（tasks）中，使得每个任务所处理元组数量能够能够保持基本一致，以确保集群的负载均衡。
2. 域分组（Fields grouping）：这种方式下数据流根据定义的“域”来进行分组。例如，如果某个数据流是基于一个名为“user-id”的域进行分组的，那么所有包含相同的“user-id”的元组都会被分配到同一个任务中，这样就可以确保消息处理的一致性。
3. 部分关键字分组（Partial Key grouping）：这种方式与域分组很相似，根据定义的域来对数据流进行分组，不同的是，这种方式会考虑下游 Bolt 数据处理的均衡性问题，在输入数据源关键字不平衡时会有更好的性能1。感兴趣的读者可以参考[这篇论文](https://melmeric.files.wordpress.com/2014/11/the-power-of-both-choices-practical-load-balancing-for-distributed-stream-processing-engines.pdf)，其中详细解释了这种分组方式的工作原理以及它的优点。
4. 完全分组（All grouping）：这种方式下数据流会被同时发送到 Bolt 的所有任务中（也就是说同一个元组会被复制多份然后被所有的任务处理），使用这种分组方式要特别小心。
5. 全局分组（Global grouping）：这种方式下所有的数据流都会被发送到 Bolt 的同一个任务中，也就是 id 最小的那个任务。
6. 非分组（None grouping）：使用这种方式说明你不关心数据流如何分组。目前这种方式的结果与随机分组完全等效，不过未来 Storm 社区可能会考虑通过非分组方式来让 Bolt 和它所订阅的 Spout 或 Bolt 在同一个线程中执行。
7. 直接分组（Direct grouping）：这是一种特殊的分组方式。使用这种方式意味着元组的发送者可以指定下游的哪个任务可以接收这个元组。只有在数据流被声明为直接数据流时才能够使用直接分组方式。使用直接数据流发送元组需要使用 [OutputCollector](http://storm.apache.org/javadoc/apidocs/backtype/storm/task/OutputCollector.html) 的其中一个 [emitDirect](http://storm.apache.org/javadoc/apidocs/backtype/storm/task/OutputCollector.html#emitDirect-int-java.lang.String-java.util.List-) 方法。Bolt 可以通过 [TopologyContext](http://storm.apache.org/javadoc/apidocs/backtype/storm/task/TopologyContext.html) 来获取它的下游消费者的任务 id，也可以通过跟踪 [OutputCollector](http://storm.apache.org/javadoc/apidocs/backtype/storm/task/OutputCollector.html) 的 emit 方法（该方法会返回它所发送元组的目标任务的 id）的数据来获取任务 id。
8. 本地或随机分组（Local or shuffle grouping）：如果在源组件的 worker 进程里目标 Bolt 有一个或更多的任务线程，元组会被随机分配到那些同进程的任务中。换句话说，这与随机分组的方式具有相似的效果。

**相关资料**

* [TopologyBuilder](http://storm.apache.org/javadoc/apidocs/backtype/storm/topology/TopologyBuilder.html)：使用此类构造拓扑
* [InputDeclarer](http://storm.apache.org/javadoc/apidocs/backtype/storm/topology/InputDeclarer.html)：在 TopologyBuilder 中调用 setBolt 方法时会返回这个对象的实例，通过该对象就可以定义 Bolt 的输入数据流以及数据流的分组方式
* [CoordinatedBolt](http://storm.apache.org/javadoc/apidocs/backtype/storm/task/CoordinatedBolt.html)：这个 Bolt 主要用于分布式 RPC 拓扑，其中大量使用了直接数据流与直接分组模型

## 可靠性（Reliability）

Storm 可以通过拓扑来确保每个发送的元组都能得到正确处理。通过跟踪由 Spout 发出的每个元组构成的元组树可以确定元组是否已经完成处理。每个拓扑都有一个“消息延时”参数，如果 Storm 在延时时间内没有检测到元组是否处理完成，就会将该元组标记为处理失败，并会在稍后重新发送该元组。

为了充分利用 Storm 的可靠性机制，你必须在元组树创建新结点的时候以及元组处理完成的时候通知 Storm。这个过程可以在 Bolt 发送元组时通过 [OutputCollector](http://storm.apache.org/javadoc/apidocs/backtype/storm/task/OutputCollector.html) 实现：在 emit 方法中实现元组的锚定（Anchoring），同时使用 ack方法表明你已经完成了元组的处理。

关于可靠性保障的更多内容可以参考这篇文章：[消息的可靠性处理](http://ifeve.com/storm-guaranteeing-message-processing)。

## 任务（Tasks）

在 Storm 集群中每个 Spout 和 Bolt 都由若干个任务（tasks）来执行。每个任务都与一个执行线程相对应。数据流分组可以决定如何由一组任务向另一组任务发送元组。你可以在 [TopologyBuilder](http://storm.apache.org/javadoc/apidocs/backtype/storm/topology/TopologyBuilder.html) 的 setSpout 方法和 setBolt 方法中设置 Spout/Bolt 的并行度。

## 工作进程（Workers）

拓扑是在一个或多个工作进程（worker processes）中运行的。每个工作进程都是一个实际的 JVM 进程，并且执行拓扑的一个子集。例如，如果拓扑的并行度定义为300，工作进程数定义为50，那么每个工作进程就会执行6个任务（进程内部的线程）。Storm 会在所有的 worker 中分散任务，以便实现集群的负载均衡。

**相关资料**

* [Config.TOPOLOGY\_WORKERS](http://storm.apache.org/javadoc/apidocs/backtype/storm/Config.html#TOPOLOGY_WORKERS)：这个配置项用于设置拓扑的工作进程数

1 Partial Key grouping 方式目前仅支持开发版，尚未加入 Storm 的正式发行版，不过可以通过 CustomStreamGrouping间接实现该分组功能，具体的实现可以参考 PartialKeyGrouping [源代码](https://github.com/apache/storm/blob/master/storm-core/src/jvm/backtype/storm/grouping/PartialKeyGrouping.java)。

# Apache Storm 官方文档 —— 配置

[原文链接](http://storm.apache.org/documentation/Configuration.html)    译者：[魏勇](http://weyo.me/)

Storm 有大量配置项用于调整 nimbus、supervisors 和拓扑的行为。有些配置项是系统级的配置项，在拓扑中不能修改，另外一些配置项则是可以在拓扑中修改的。

每一个配置项都在 Storm 代码库的 [defaults.yaml](https://github.com/apache/storm/blob/master/conf/defaults.yaml) 中有一个默认值。可以通过在 Nimbus 和 Supervisors 的环境变量中定义一个 storm.yaml 来覆盖默认值。最后，在使用 [StormSubmitter](http://storm.apache.org/javadoc/apidocs/backtype/storm/StormSubmitter.html) 提交拓扑时也可以定义基于具体拓扑的配置项。但是，基于拓扑的配置项仅仅能够覆盖那些以 “TOPOLOGY” 作为前缀的配置项。

Storm 0.7.0 以上版本支持覆写每个 Bolt/Spout 的配置信息。不过，使用这种方式只能修改以下几个配置项：

1. “topology.debug”
2. “topology.max.spout.pending”
3. “topology.max.task.parallelism”
4. “topology.kryo.register”：由于序列化对拓扑中的所有组件都是可见的，这一项与其他几项稍微有一些不同，详细信息可以参考 [Storm 的序列化](http://ifeve.com/storm-serialization)

Storm 的 Java API 支持两种自定义组件配置信息的方式：

1. 内置型：在需要配置的 Spout/Bolt 中覆写 getComponentConfiguration 方法，使其返回特定组件的配置表；
2. 外置型：TopologyBuilder 中的 setSpout 与 setBolt 方法会返回一个带有 addConfiguration 方法的ComponentConfigurationDeclarer 对象，通过 addConfiguration 方法就可以覆写对应组件的配置项（同时也可以添加自定义的配置信息——译者注）。

配置信息的优先级依次为：defaults.yaml < storm.yaml < 拓扑配置 < 内置型组件信息配置 < 外置型组件信息配置。

**相关资料**

* [Config](http://storm.apache.org/javadoc/apidocs/backtype/storm/Config.html)：此类包含所有可配置项的列表，对于创建拓扑配置信息很有帮助
* [defaults.yaml](https://github.com/apache/storm/blob/master/conf/defaults.yaml)：所有配置项的默认值
* [配置 Storm 集群](http://ifeve.com/storm-setting-up-a-storm-cluster)：说明了如何创建、配置一个 Storm 集群
* [在生产环境中运行拓扑](http://ifeve.com/storm-running-topologies-on-a-production-cluster)：列出了在集群中运行拓扑的一些有用的配置项
* [本地模式](http://ifeve.com/storm-local-mode)：列出了使用本地模式时比较有用的配置项

# Apache Storm 官方文档 —— 消息的可靠性保障

[原文链接](http://storm.apache.org/documentation/Guaranteeing-message-processing.html)    译者：[魏勇](http://weyo.me/)

Storm 能够保证每一个由 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。

## 消息的“完整性处理”是什么意思

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](http://storm.apache.org/javadoc/apidocs/backtype/storm/Config.html#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。

## 在消息得到完整性处理后或者处理失败后会发生什么

为了理解这个问题，让我们先了解一下 tuple 的生命周期。下面是定义 spout 的接口（可以在 [Javadoc](http://storm.apache.org/javadoc/apidocs/backtype/storm/spout/ISpout.html) 中查看更多细节信息）：

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，及时 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）。

## Storm 的可靠性 API

使用 Storm 的可靠性机制的时候你需要注意两件事：首先，在 tuple 树中创建新节点连接时务必通知 Storm；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(OutputFieldsDeclarer 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，这在流式连接或者聚合操作时很有用。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 树具象化的过程 —— 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）或者失效（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 进行应答或者失效，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(OutputFieldsDeclarer 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。聚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。

## 在 tuple 可以被重新处理的前提下，如何让我的应用可以得到正确的运行？

按照软件设计的一般思路，这个问题的答案是“取决于实际情况”。Storm 0.7.0 版本引入了“事务拓扑”的特性，它能够保证大多数计算过程都能够满足恰好一次（exactly-once）的消息语义的容错性要求。想要了解“事务拓扑”的更多内容可以参考[这篇文章](http://storm.apache.org/documentation/Transactional-topologies.html)。

## Storm 是以怎样一种高效的方式实现可靠性的？

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](http://storm.apache.org/javadoc/apidocs/backtype/storm/Config.html#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，接下来这个就是被我标记的新 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 匹配到 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，它也会给对应的 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 tuple 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。而且即使是这样，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](http://ifeve.com/storm-fault-tolerance)——译者注）。

综上所述，Storm 的可靠性机制完全具备分布的、可伸缩的、容错的特征。

## 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。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。

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。

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。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（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）。