目录

[1.storm介绍 2](#_Toc494117114)

[1.1Storm集群组件(Components of a Storm cluster) 2](#_Toc494117115)

1.2拓扑(Topologies) ................................................................................................................2

1.3 流.......................................................................................................................................4

1.4 数据结构...........................................................................................................................5

[1.5 在本地模式下运行ExclamationTopology 7](#_Toc494117116)

2.消息处理.......................................................................................................................................7

2.1[(Running ExclamationTopology in local mode) 7](#_Toc494117117)

[2.2 (What happens if a message is fully processed or fails to be fully processed?) 11](#_Toc494117118)

[2.3 (What is Storm's reliability API) 11](#_Toc494117119)

[2.4 (How do I make my applications work correctly given that tuples can be replayed) 13](#_Toc494117120)

[2.5 (How does Storm implement reliability in an efficient way) 14](#_Toc494117121)

[2.6 (Tuning reliability) 15](#_Toc494117122)

[3.事务性拓扑 15](#_Toc494117123)

[3.1事务实践 16](#_Toc494117124)

[3.2 Spout 17](#_Toc494117125)

[3.3 RQ类 17](#_Toc494117126)

[3.4协调者Coordinator 18](#_Toc494117127)

[3.5 Emitter 19](#_Toc494117128)

[3.6 Bolts 19](#_Toc494117129)

[3.7提交者bolts 21](#_Toc494117130)

[3.8 分区的事务Spouts 22](#_Toc494117131)

[3.9 模糊的事务性拓扑 24](#_Toc494117132)

[4.strom安装 25](#_Toc494117133)

[4.1准备开始 25](#_Toc494117134)

[4.2操作模式 25](#_Toc494117135)

[4.3本地模式 25](#_Toc494117136)

[4.4远程模式 25](#_Toc494117137)

[4.5Hello World 25](#_Toc494117138)

[4.6 Java安装检查 26](#_Toc494117139)

[4.7创建工程 26](#_Toc494117140)

[4.8创建我们的第一个Topology 28](#_Toc494117141)

[4.8.1Spout 28](#_Toc494117142)

[4.8.2 Bolts 31](#_Toc494117143)

[4.8.3主类 34](#_Toc494117144)

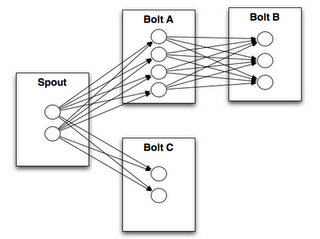
[4.8.4观察运行情况 36](#_Toc494117145)

# 1.storm介绍

Storm是一个分布式实时计算系统。就像Hadoop提供一组通用原语来进行批量处理(batch processing)一样，Storm也提供了一组通用原语来进行实时计算(realtime computation)。Storm非常简单，能用于任意编程语言，被很多大的公司采用。   
    本教程中，你会学习如何创建Storm拓扑(topologies)，以及如何部署它们到Storm集群中。Java是我们使用的主要语言，但是一些例子会使用到Python来展示Storm的多语言能力。

## **1.1** Storm集群组件(Components of a Storm cluster)

    Storm集群看起来类似于Hadoop集群。然而，在Hadoop上你运行的是”MapReduce jobs”，在Storm上你运行的是” topologies”。”Jobs”和”topologies”本身就非常的不同，其中一个主要的一个区别是，MapReduce的任务最终会结束，而拓扑则永远在处理消息(直到你干掉它)。   
    Storm集群上有两种类型的节点：主节点(master node)和工作节点(worker node)。主节点运行一个称为Nimbus的守护进程，类似于Hadoop的JobTracker。Nimbus负责分发代码到集群中，分配任务到机器，并且监控失败。   
    每一个工作节点运行一个称为Supervisor的守护进程。supervisor监听分配给这台机器的工作，并且基于Nimbus分配情况在必要时启动和停止工作进程。每一个工作进程执行一个拓扑的子集合；一个运行中的拓扑由横跨多个机器的多个工作进程组成。



   Numbus和Supervisor的所有协调同步工作是由Zookeeper集群完成的。另外，Nimbus和Supervisor守护进程是快速失败和无状态的，所有的状态保存在Zookeeper或者本地磁盘中。这意味着，你可以kill -9Nimbus或者Supervisor，它们会像什么都没有发生一样再启动起来。这个设计使得Storm集群变得难以置信的稳定。 

## **1.2**拓扑(Topologies)

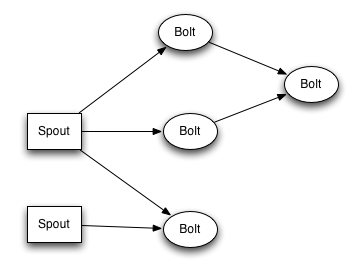
    为了在Storm上进行实时计算，你需要创建一个我们称为拓扑的东西。一个拓扑就是一个计算图谱。拓扑中的每一个节点包含着处理逻辑，以及表明数据如何在节点间传递的很多链接。   
    运行一个拓扑非常的简单。首先，你将你所有的代码和依赖打包进一个jar中。然后，运行类似下面这样的命令行：

storm jar all-my-code.jar backtype.storm.MyTopology arg1 arg2

上面使用参数arg1和arg2来运行backtype.storm.MyTopology类。这个类的主要功能是定义了拓扑，并且将其提交给Nimbus。storm jar部分负责连接到Nimbus并且上传这个jar文件。   
    因为拓扑的定义就是Thrift结构体，Nimbus是一个Thrift服务，所以你可以创建和提交然一编程语言的拓扑。上面的例子是基于JVM语言实现的最简单方式。要想了解如何在生产环境集群中运行拓扑，持续关注本系列后续文章(译者注)。 

## **1.3**流(Streams)

    Storm中的核心抽象概念就是流。流是无边界的元组(tuples)的序列。Storm提供以分布式的、可靠的方式转换一个流到一个新流的原语。例如，你可以转换一个微博的流到一个流行话题的流。   
    Storm提供处理流转换的基本原语是spouts和bolts。，实现spout和bolt提供的接口来运行你应用程序相关的逻辑。   
    spout是流的源头。例如，一个spout从Kestrel队列读取元组，然后作为流发射(emit)它们。一个spout也可以连接到Twitter API，然后发射一个微博的流。   
    一个bolt消费任意数量的输入流，进行一些处理，然后可能发射新的流。复杂的流转换，例如从微博流中计算流行话题的流，需要更多的步骤以及更多的bolt。bolt可以干任意事情，从运行函数，过滤元组，聚合(aggregations)流，连接(joins)流，到与数据库交互，等等。   
    spout和bolt组成的网络被打包进一个拓扑中，这就是你提交给Storm集群执行的最高层次抽象。拓扑就是一个流的转换图谱，每一个节点就是一个spout或者bolt。图中的边表示哪个流被哪个流订阅。当一个spout或者bolt发射一个元组到一个流，也会发送这个元组到每一个订阅这个流的bolt。



拓扑中节点间的链接表示元组该如何被传递。例如，Spout A和Spout B之间有一个链接，Spout A到Bolt C有一个链接，Bolt B到Bolt C有一个链接，那么每次Spout A发射一个元组，也会发送这个元组到Bolt B和Bolt C。Bolt B的所有输出元组也会到达Bolt C。   
    Storm拓扑中的每个节点是并行执行的。在你的拓扑中，你可以为每个节点指定你想要的并行度(parallelism)，然后Storm会在集群中分配这么多数量的线程来执行。   
    拓扑永远在运行中，除非你干掉它。Storm会自动重新分配失败的任务。另外，Storm保证不会有数据丢失，及时机器挂掉了以及消息丢失了。

## **1.4**数据模型(Data model)

    Storm使用元组作为数据模型。元组是命名的值列表，元组中的字段可以是任何类型的对象。Storm支持所有基本类型，字符串和梓杰数组作为元组的字段值。要使用一个其他类型的对象，你只需要为该类型实现一个序列化器即可。   
    拓扑中的每一个节点必须声明它发射的元组的输出字段。例如，下面的bolt声明其发射double和triple字段的一个二元组。

1. public class DoubleAndTripleBolt extends BaseRichBolt {
2. private OutputCollectorBase \_collector;
4. @Override
5. public void prepare(Map conf, TopologyContext context, OutputCollectorBase collector) {
6. \_collector = collector;
7. }
9. @Override
10. public void execute(Tuple input) {
11. int val = input.getInteger(0);
12. \_collector.emit(input, new Values(val\*2, val\*3));
13. \_collector.ack(input);
14. }
16. @Override
17. public void declareOutputFields(OutputFieldsDeclarer declarer) {
18. declarer.declare(new Fields("double", "triple"));
19. }
20. }

    declareOutputFields函数为组件声明了输出字段["double", "triple"]。这个bolt的剩下部分会在下面的小节详细解释。   
  
   1.4.1一个简单的拓扑(A simple topology)     让我们来看一个简单的拓扑来探索更多的概念，然后看看这些代码是怎么来的。我们看看storm-starter中的ExclamationTopology定义：

1. TopologyBuilder builder = new TopologyBuilder();
2. builder.setSpout("words", new TestWordSpout(), 10);
3. builder.setBolt("exclaim1", new ExclamationBolt(), 3)
4. .shuffleGrouping("words");
5. builder.setBolt("exclaim2", new ExclamationBolt(), 2)
6. .shuffleGrouping("exclaim1");

  这个拓扑包含1个spout和2个bolt。spout发射单词，每个bolt追加字符串”!!!”到输入。节点被线性安排：spout发射到第一个bolt，然后第一个bolt发射给第二个bolt。如果spout发射元组[“bob”]和[“john”]，之后第二个bolt会发射单词[“bob!!!!!!”]和[“john!!!!!!”]。   
    这段代码使用setSpout和setBolt方法来定义节点。这个方法一个用户相关的id，一个包含处理逻辑的对象，以及这个节点你想要的并行度作为输入。在这个例子中，spout被赋予的id为”words”，bolt被赋予的id为”exclaim1”和”exclaim2”。   
    包含处理逻辑的对象实现了IrichSpout接口作为spout，实现了IrichBolt接口作为bolt。   
最后一个参数，你期待的节点并行度，是可选的。它表明集群中该使用多少个线程执行这个组件。如果你忽略它，Storm只会为这个节点分配一个线性。   
    setBolt返回一个InputDeclarer对象，用于定义到bolt的输入。这里，组件exclaim1通过shuffle grouping声明它想要读取组件words发射的所有元组，组件exclaim2通过shuffle grouping声明它想读取组件exclaim1发射的所有元组。shuffle grouping意思是元组应该从输入任务被随机的分布到bolt任务。有多种在组件间分组数据的方式。这些将会在下面的小节中详细解释。   
    如果你想让组件exclaim2读取所有由组件words和组件exclaim1发射的元组，你需要像下面这样定义组件exclaim2：

1. builder.setBolt("exclaim2", new ExclamationBolt(), 5)
2. .shuffleGrouping("words")
3. .shuffleGrouping("exclaim1");

    如你所见，输入声明可以是链式的以为bolt指定多个来源。   
    让我们再深入研究一下这个拓扑中的spout和bolt实现。spout负责发射新消息到拓扑中。这个拓扑中，TestWordSpout每隔100ms从 ["nathan", "mike", "jackson", "golda", "bertels"]中发射一个随机单词一元组。TestWordSpout 中的nextTuple()实现看起来像这样：

1. public void nextTuple() {
2. Utils.sleep(100);
3. final String[] words = new String[] {"nathan", "mike", "jackson", "golda", "bertels"};
4. final Random rand = new Random();
5. final String word = words[rand.nextInt(words.length)];
6. \_collector.emit(new Values(word));
7. }

你可以看到，实现非常的简单。   
    ExclamationBolt追加字符串”!!!”到其输入。让我们看一下ExclamationBolt的完整实现：

1. public static class ExclamationBolt implements IRichBolt {
2. OutputCollector \_collector;
4. public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
5. \_collector = collector;
6. }
8. public void execute(Tuple tuple) {
9. \_collector.emit(tuple, new Values(tuple.getString(0) + "!!!"));
10. \_collector.ack(tuple);
11. }
13. public void cleanup() {
14. }
16. public void declareOutputFields(OutputFieldsDeclarer declarer) {
17. declarer.declare(new Fields("word"));
18. }
20. public Map getComponentConfiguration() {
21. return null;
22. }
23. }

  prepare方法提供一个OutputCollector给bolt，用于bolt发射元组。元组可以在任意时刻从bolt被发射——在prepare，execute或者cleanup方法中，或者甚至在另一个线程中异步发射。这个prepare实现只是保存OutputCollector作为一个实例变量，以在后面的execute方法中使用。   
    execute方法接受来自某个bolt输入中的元组。ExclamationBolt从一个元组中抓取第一个字段，然后发射一个被追加”!!!”的新的元组。如果你实现一个订阅多个输入源的bolt，你可以通过使用Tuple#getSourceComponent方法来分辨元组来自于哪个组件。   
    execute方法中还有一些其他的信息，也就是输入元组作为第一个参数传递给emit，并且输入元组在最后一行被确认(acked)，这些是属于Storm reliability API的部分，以保证数据不会被丢失，后文中会详细解释。   
    cleanup方法当Bolt被关闭时被调用，用于清理我们打开的任意资源。集群中这个方法是否被调用没有保证：例如，如果运行任务的机器着火了，就没有办法调用这个方法。cleanup方法用来当你在本地模式下运行拓扑时(在进程中模拟Storm集群)，你想能够运行和干掉许多拓扑而免遭任何资源泄露。   
    declareOutputFields方法声明ExclamationBolt发射只有一个word字段的一元组。   
getComponentConfiguration方法允许你配置这个组件如何运行的方方面面。这是一个更高级的主题，我们会在本系列后续文章中深入解释。   
    cleanup和getComponentConfiguration方法通常不需要实现。你可以使用一个基类更方便的定义bolt，基类提供了默认的合适的实现。ExclamationBolt可以通过继承BaseRichBolt更方便的实现，像这样：

1. public static class ExclamationBolt extends BaseRichBolt {
2. OutputCollector \_collector;
4. public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
5. \_collector = collector;
6. }
8. public void execute(Tuple tuple) {
9. \_collector.emit(tuple, new Values(tuple.getString(0) + "!!!"));
10. \_collector.ack(tuple);
11. }
13. public void declareOutputFields(OutputFieldsDeclarer declarer) {
14. declarer.declare(new Fields("word"));
15. }
16. }

## **1.5 在本地模式下运行ExclamationTopology**

## **(Running ExclamationTopology in local mode)**

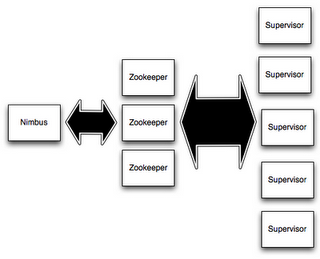
    让我们看看如何在本地模式下运行ExclamationTopology，以及它是怎么运转的。   
    Storm有两种运行模式：本地模式和分布式模式。在本地模式下，Storm完全在用线程模拟工作节点的进程中运行。本地模式在测试和开发拓扑时非常有用。当你运行storm-starter中的拓扑时，它们就运行在本地模式，你可以看到每个组件发射了什么消息。你可以在另一篇中看到在本地模式运行拓扑的更详细信息(译者注)。   
    在分布式模式下，Storm像一个机器集群一样运行。当你提交一个拓扑到主节点，你也提交了所有的必要代码来运行这个拓扑。主节点负责发布你的代码并且分配工作节点来运行你的拓扑。如果工作节点挂掉了，主节点会重新分配它们到其他地方。   
    下面是在本地模式运行ExclamationTopology的代码：

1. Config conf = new Config();
2. conf.setDebug(true);
3. conf.setNumWorkers(2);
5. LocalCluster cluster = new LocalCluster();
6. cluster.submitTopology("test", conf, builder.createTopology());
7. Utils.sleep(10000);
8. cluster.killTopology("test");
9. cluster.shutdown();

  最开始，代码通过创建LocalCluster对象来定义一个进程内集群。提交一个拓扑到这个虚拟集群等同于提交拓扑到分布式集群。通过调用submitTopology来提交一个拓扑到LocalCluster，以一个拓扑名字，一个拓扑配置，以及拓扑本身作为参数。   
    名字用于识别这个拓扑，这样你可以在稍后干掉它。一个拓扑会无限的运行下去直到你干掉它。   
    配置用于跳着拓扑运行的方方面面。下面的两个配置非常常用：   
    1 TOPOLOGY\_WORKERS (使用setNumWorkers设置)指定你想在集群中分配多少个进程来运行这个拓扑。拓扑中的每一个组件都会作为许多线程运行。某个组件被分配的线程数通过setBolt和setSpout方法来配置。这些线程存在于工作进程中。每一个工作进程包含若干组件的若干线程在其中。例如，你在你所有的组件中指定了300个线程，在配置中丁丁了50个工作进程。每个工作进程将会执行6个线程，每一个属于一个不同的组件。你通过调整每个组件的并行性以及运行这些线程的工作者进程的数量来调优Storm拓扑的性能。   
    2 TOPOLOGY\_DEBUG (使用setDebug设置)当设置为true时，告诉Storm记录每个组件发射的每个消息。这在本地模式下测试拓扑时非常有用，但是你很可能想要在集群中运行拓扑时保持这个关闭。   
还有很多其他的配置你可以用于设置拓扑。更多细节请参考Config的Javadoc。 

## **1.6**流分组(Stream groupings)

    流分组告诉拓扑如何在组件间发送元组。记住，spout和bolt以很多任务的形式在集群中并行执行。如果你从任务级别来观察拓扑如何执行，看起来就像是这样：



    当Bolt A的一个任务发射一个元组给Bolt B，这个元组会被发送给哪个任务？   
    流分组通过告诉Storm如何在任务集合间发送元组来回答这个问题。在我们深入到不同的流分组类型之前，让我们一起看看storm-starter中的另一个拓扑。WordCountTopology中的spout读取句子，然后WordCountBolt输出每个单词之前出现的次数：

1. TopologyBuilder builder = new TopologyBuilder();
3. builder.setSpout("sentences", new RandomSentenceSpout(), 5);
4. builder.setBolt("split", new SplitSentence(), 8)
5. .shuffleGrouping("sentences");
6. builder.setBolt("count", new WordCount(), 12)
7. .fieldsGrouping("split", new Fields("word"));

  SplitSentence发射每个它收到的句子的每个单词的元组，WordCount在内存中保存一个从单词到技术的映射。每次WordCount收到一个单词，就更新它的状态，然后发射一个新的单词计数。   
  
    有一些不同的流分组类型：   
  最简单的分组类型被称为” shuffle grouping”，发送元组给随机的任务。  WordCountTopology使用一个shuffle grouping来从RandomSentenceSpout向SplitSentence bolt发送任务。结果就是，处理元组的工作被平均分配在SpliteSentence bolt的所有任务中。   
  
    更有意思的一种分组类型是”fields grouping”. fields grouping被用于SplitSentence bolt和WordCount bolt之间。WordCount bolt中一个很关键的功能是想通的单子总是去往同一个任务。否则，多余一个的任务会看到相同的单词，它们会分别发射一个错误的值来计数，因为每一个都只有不完整的信息。fields grouping允许你通过字段的子集来分组一个流。这导致字段子集中相等的值去往同一个任务。因为WordCount使用fields grouping在word字段上订阅了SplitSentence的输出流，相同的单词总是去往相同的任务，这样bolt就产生了正确的输出。   
  
    fields grouping是实现流连接和流聚合的基础，也适用于很多其他情形。进一步说，fields grouping是通过使用取模哈希实现的。

# 2.消息处理保证

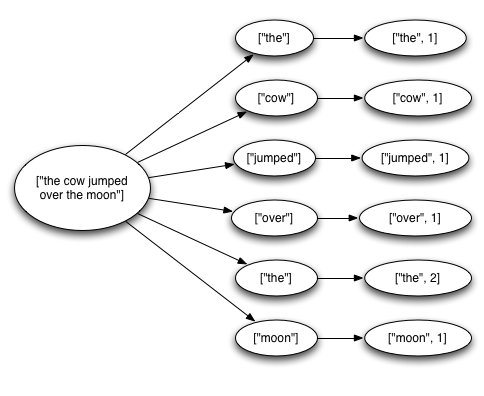
   Strom保证来自spout的每一个消息都会被完全处理。本文描述Storm是如何做到这个保证的，以及作为用户需要干些什么从而受益于Storm的可靠性能力。

## **2.1** (What does it mean for a message to be "fully processed")

    来自于spout的元组可以触发基于该元组的成千上万的元组被创建。例如，考虑一下，单词计数拓扑：

1. TopologyBuilder builder = **new** TopologyBuilder();
2. builder.setSpout("sentences", **new** KestrelSpout("kestrel.backtype.com",
3. 22133,
4. "sentence\_queue",
5. **new** StringScheme()));
6. builder.setBolt("split", **new** SplitSentence(), 10)
7. .shuffleGrouping("sentences");
8. builder.setBolt("count", **new** WordCount(), 20)
9. .fieldsGrouping("split", **new** Fields("word"));

  这个拓扑从Kestrel队列读取句子，将其拆分成逐个单词，然后发射每个单词之前遇到的次数。来自spout的元组触发了许多基于该元组的元组被创建：句子中的每一个单词有一个元组，每个单词的计数也有一个元组。消息树看起来大概是这样的：



    当元组树(tuple tree)耗尽或者树中的每一个消息都被处理了时，来自于spout的元组则被Storm认为完全处理了。当树中的消息在指定的超时时间内没有被完全处理时，则这个元组被认为处理失败。这个超时时间可以为特定的拓扑进行设置，使用Config.TOPOLOGY\_MESSAGE\_TIMEOUT\_SECS配置选项即可，默认为30秒。

## 2.2 ****(What happens if a message is fully processed or fails to be fully processed?)****

 为了理解这个问题，让我们来瞧瞧来自于spout的元组的生命周期。例如，下面是spout需要实现的接口(详细信息请看Javadoc)：

1. **public** **interface** ISpout **extends** Serializable {
2. **void** open(Map conf, TopologyContext context, SpoutOutputCollector collector);
3. **void** close();
4. **void** nextTuple();
5. **void** ack(Object msgId);
6. **void** fail(Object msgId);
7. }

  首先，Storm通过调用Spout的nextTuple方法来从spout中请求一个元组。spout使用open方法中提供的SpoutOutputCollector来发射一个元组到其中某个输出流。当发射一个元组时，spout提供了一个消息id，用于在之后标识这个元组。例如，KestrelSpout从kestrel队列中读取消息，并且将Kestrel提供的id作为消息id来发射消息。发射一个消息到SpoutOutputCollector看起来像这样：

1. \_collector.emit(**new** Values("field1", "field2", 3) , msgId);

接下来，元组被发往消费bolt，Storm负责追踪创建的消息树。如果Storm检测到元组被完全处理了，Storm就会使用spout提供给Storm的消息id，在源头spout上调用ack方法。同样地，如果元组超时了，Storm就会在spout上调用fail方法。注意，元组只会被创建它的同一个spout任务确认(acked)或失败(failed)。所以，如果spout作为很多任务在集群上执行时，元组不会被不同的任务确认或失败(这里的失败是动词，下同，译者注)，而只能是创建它的那个。   
    让我们继续使用KestrelSpout这个例子，来看看spout需要做什么来保证消息处理。当KestrelSpout从Kestrel队列中取得一个消息，就”打开”了这个消息。这意思就是说，消息还没有真正从队列中取出，而是处于”等待(pending)”状态，等待消息被处理的确认信息。当处于等待状态时，消息不会被发送给其他的队列消费者。另外，如果客户端断开了连接，为这个客户端等待的所有消息将会被重新放入队列。当消息被打开了，Kestrel为客户端提供了消息的数据以及消息的唯一id。KestrelSpout使用这个唯一id作为发送元组到   SpoutOutputCollector的消息id。之后的某个时间，当KestrelSpout上的ack或者fail被调用时，KestrelSpout发送一个确认或失败消息给Kestrel，使用这个消息id来从队列中取出这个消息或者将其放回队列。

## 2.3 ****(What is Storm's reliability API)****

   作为用户，你可以干两件事情来从Storm的可靠性能力中获益。第一，每当你在元组树中创建一个新的链接时，你需告知Storm。第二，当你完成处理某个元组时，也需要告知Storm。通过干这两件事情，Storm就能够检测到元组树什么时候被完全处理，能够准确地对spout的元组进行确认或者失败。Storm API提供了一种简单的方式来做这两件事情。   
    在元组树中指定一个链接被称为锚定(anchoring)。锚定在你发射一个新的元组时同时完成了。让我们使用下面的bolt作为例子。这个bolt拆解一个包含句子的元组为逐个单词的元组：

1. **public** **class** SplitSentence **extends** BaseRichBolt {
2. OutputCollector \_collector;
4. **public** **void** prepare(Map conf, TopologyContext context, OutputCollector collector) {
5. \_collector = collector;
6. }
8. **public** **void** execute(Tuple tuple) {
9. String sentence = tuple.getString(0);
10. **for**(String word: sentence.split(" ")) {
11. \_collector.emit(tuple, **new** Values(word));
12. }
13. \_collector.ack(tuple);
14. }
16. **public** **void** declareOutputFields(OutputFieldsDeclarer declarer) {
17. declarer.declare(**new** Fields("word"));
18. }
19. }

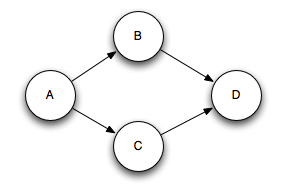
 每一个单词元组通过指定输入元组为emit方法的第一参数而被锚定。因为单词元组被锚定了，如果单词元组在往下游执行的过程中失败，处于树根的spout元组会被稍后重放(replayed)。为了对比，我们一起看看，如果单词元组像这样发射会发生什么：

**\_collector.emit(new Values(word));**

  按照这样的方式发射单词元组将导致其没有被锚定。如果元组在往下游的处理过程中失败，根元组不会被重放。取决于在你的拓扑中你想要的容错保证，但通常发射未被锚定的元组是合适的。   
    一个输出元组可以被锚定到多个输入元组。这在进行流连接和流聚合时非常有用。一个处理失败的多锚定(multi-anchored)元组会导致spout的多个元组被重放。多锚定通过指定一个元组列表而不是单个元组来实现。例如：

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

  多锚定将输出元组添加到多个元组树。注意，也有可能多锚定打破树结构而创建了元组的有向无环图(DAG)，像这样：



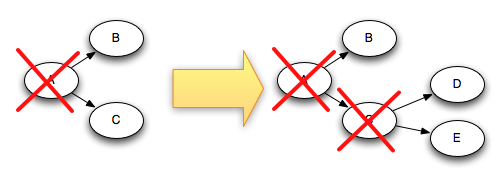
 Storm的实现既可以处理有向无环图，也可以处理树(之前的版本只能处理树，但元组树这个名字就这样固定下来了)。   
    锚定就是如何指定元组树。Storm可靠性API的下一部分，也就是最后一部分(功能)，就是指定你什么时候完成了处理元组树中的一个元组。这是通过使用OutputCollector的ack和fail方法实现的。如果你回过头来看看SplitSentence这个例子，你会看到在所有的单词元组被发射之后输入元组被确认了。   
    你可以使用OutputCollector的fail方法来立即失败处于元组树根部的spout元组。例如，你的应用程序可以选择捕获一个数据库客户端的异常，然后显式地失败这个输入元组。通过显式失败元组，相比于你等待这个元组超时，spout元组会被更快的重放。   
    你处理的每一个元组必须被确认或者失败。Storm使用内存来追踪每一个元组，所以如果你不确认或者失败每一个元组，任务最终会耗尽内存。   
    许多bolt都遵循着相同的模式来读取输入元组，基于它发射元组，然后在execute方法的结尾确认这个元组。这些bolts按照过滤器和简单功能进行分类。Storm有一个BasicBolt的接口为你封装了这个模式。SplitSentence的例子可以作为一个BasicBolt写成下面这样：

1. **public** **class** SplitSentence **extends** BaseBasicBolt {
2. **public** **void** execute(Tuple tuple, BasicOutputCollector collector) {
3. String sentence = tuple.getString(0);
4. **for**(String word: sentence.split(" ")) {
5. collector.emit(**new** Values(word));
6. }
7. }
9. **public** **void** declareOutputFields(OutputFieldsDeclarer declarer) {
10. declarer.declare(**new** Fields("word"));
11. }
12. }

实现比之前的实现更简单，并且在语义上是等价的。元组被发射到BasicOutputCollector自动被锚定到输入元组了，当execute方法完成时，输入元组也为你自动确认了。   
    作为对比，执行聚合和连接的bolt可能会延迟确认元组，直到其计算出一批元组的结果出来以后。聚合和连接也通常多锚定其输出流。这些事情超出了IbasicBolt的简单模式范畴。

2.4 (How do I make my applications work correctly given that tuples can be replayed)   
    在软件设计领域，答案永远是“具体问题具体分析(it depends)”。Storm 0.7.0引入了”事务性拓扑”特性，使得你可以在大多数计算中获取精确一次消息送达的语义的完全容错能力。

2.5 (How does Storm implement reliability in an efficient way)   
    一个Storm拓扑有一个特殊的”acker”任务集合，为每一个spout元组追踪元组的有向无环图。当一个acker发现一个DAG完成了，就会发送一个消息到创建这个spout元组的spout任务来确认这个消息。你可以在拓扑的配置中使用Config.TOPOLOGY\_ACKERS选项来设置拓扑的acker任务数量。Storm默认Config.TOPOLOGY\_ACKERS为一个任务。你需要为处理大量消息的拓扑增加这个值。   
    理解Storm可靠性实现的最佳途径是关注元组和元组有向无环图的生命周期。当拓扑中一个元组被创建，无论是在spout中还是在bolt中，都会被赋予一个随机的64位id。这个id被acker用来为每一个spout元组追踪其有向无环图。   
    每一个元组知道其所在的元组树中的全部spout元组的id。当你在bolt中发射一个新的元组时，spout元组的id就会从元组的锚点中被拷贝到这个新的元组中。当一个元组被确认，就会发送一个消息给合适的acker任务，这个任务知道元组树是如何变化的相关信息。具体来说，就是告诉acker”我已经在树中完成了这个spout元组，这个是树中被锚定给我的新元组”。   
    例如，如果元组D和E是基于元组C被创建，下面就是当C被确认时元组树时如何变化的：



  因为在C被从树中删除的同时，D和E被添加进去，所以树不会很快地结束。   
    还有一个关于Storm如何追踪元组树的更多细节。正如已经提到的，你可以拥有任意数量的acker任务在拓扑中。那么问题来了：当一个元组在拓扑中被确认，它是如何知道应该发送信息给哪一个acker任务呢？   
    Storm使用模哈希(mod hashing)来映射spout元组id到acker任务。因为每个元组都携带着其所在的所有树中的spout元组id(一个元组可能存在于多个树中，译者注)，它们知道应该与哪一个acker任务进行通信。   
    Storm的另一个细节是，acker任务是如何知道每一个spout元组该由哪一个acker进行追踪。当一个spout任务发射一个新元组，它只是发送消息给一个合适的acker，来告诉它这个spout元组的任务id。然后当一个acker发现一个树完成了，就知道发送完成消息给哪一个任务id。   
     acker任务并不显式完全的追踪元组树。在一个拥有成千上万(或者更多)个节点的大型元组树中，追踪所有的元组树可能会超过acker内存上限。取而代之的是，acker采取了一种不同的策略，每个spout元组只需要固定数量的空间(大约20字节)。这个追踪算法是Storm运转的关键，也是其主要的突破之一。   
    一个acker任务存储了从一个spout元组id到一对值的映射。第一个值是创建这个spout元组的任务id，后面用于发送完成消息。第二个值是一个64位的数字，被称为”ack val”。ack val是整个元组树的状态表示，无论这个树有多大或者多小。它只是简单地对树中所有创建的或者/以及确认的元组id进行异或。   
    当一个acker任务发现一个ack val变成了0，它就知道元组树完成了。因为元组id是64位的随机数字，所以ack val碰巧变成0的可能性是极低的。如果你用数学算一算，以每秒10K次确认的频率，也需要50,000,000年才能遇到一个错误。即便如此，也只是在拓扑中的元组碰巧失败的情况下才会导致数据丢失。   
    现在，你理解了可靠性算法，让我们一起来过一下所有可能失败的情形，看看每种情况下Storm是如何避免数据丢失的： 

**由于任务死掉了导致元组没有被确认**：在这种情况下，处于失败元祖所在树的根部的spout元组会超时并被重放。

**acker任务死掉**：在这种情况下，这个acker追踪的所有spout元组会超时并被重放。

**spout任务死掉**：在这种情况下，与spout对话的源头来负责重放消息。例如，像Kestrel和RabbitMQ这样的队列会在客户端断开时将所有顶戴的消息重新防火到队列中。

    正如你所见，Storm的可靠性机制完全是分布式的，可伸缩的和容错的。

2.6 (Tuning reliability)   
    acker任务是轻量级的，所以你在拓扑中不需要太多。你可以通过Storm UI(组件id是\_\_acker)来追踪其性能。如果吞吐量看起来不正常，不就需要添加更多的acker任务。   
    如果可靠性对你来说并不重要——也就是说你不在意在失败情形下的元组丢失——那么你可以通过不追踪spout元组的元组树来改善性能。不追踪元组树将会减少一半的消息传输量，因为正常情况下，元组树中的每一个元组都对应有一个确认消息。另外，下游的元组也只需要保存更少的id从而节省了带宽使用。   
    有三种移除可靠性的方式。第一种是设置Config.TOPOLOGY\_ACKERS为0。这种方式下，Storm会在spout发射一个元组后立即在spout上调用ack方法。   
     第二种移除可靠性的方式是通过消息本身。你可以通过在SpoutOutputCollector.emit方法中忽略消息id来关闭对某个spout元组的追踪。   
    最后，如果你对下游的部分元组处理失败不是很在意，你可以作为非锚定的元组来发射它们。因为它们没有被锚定到任何spout元组，所以如果它们没有被确认也不会导致任何spout元组失败。

# 3.事务性拓扑

*Storm0.7.0*实现了一个新特性——事务性拓扑，这一特性使消息在语义上确保你可以安全的方式重发消息，并保证它们只会被处理一次。在不支持事务性拓扑的情况下，你无法在准确性，可扩展性，以空错性上得到保证的前提下完成计算。

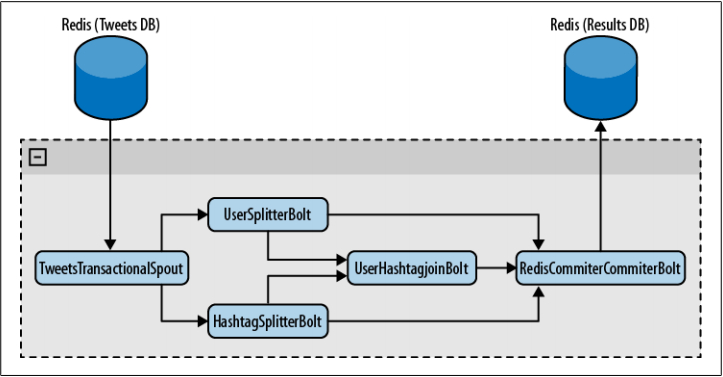
**NOTE:**事务性拓扑是一个构建于标准Storm spout和bolt之上的抽象概念。

**设计**

在事务性拓扑中，Storm以并行和顺序处理混合的方式处理元组。spout并行分批创建供bolt处理的元组（译者注：下文将这种分批创建、分批处理的元组称做批次）。其中一些bolt作为提交者以严格有序的方式提交处理过的批次。这意味着如果你有每批五个元组的两个批次，将有两个元组被bolt并行处理，但是直到提交者成功提交了第一个元组之后，才会提交第二个元组。 **NOTE：** 使用事务性拓扑时，数据源要能够重发批次，有时候甚至要重复多次。因此确认你的数据源——你连接到的那个spout——具备这个能力。 这个过程可以被描述为两个阶段： 处理阶段 纯并行阶段，许多批次同时处理。 提交阶段 严格有序阶段，直到批次一成功提交之后，才会提交批次二。 这两个阶段合起来称为一个Storm事务。 **NOTE：** Storm使用zookeeper储存事务元数据，默认情况下就是拓扑使用的那个zookeeper。你可以修改以下两个配置参数键指定其它的zookeeper——transactional.zookeeper.servers和transactional.zookeeper.port。

## 3.1事务实践

下面我们要创建一个Twitter分析工具来了解事务的工作方式。我们从一个Redis数据库读取tweets，通过几个bolt处理它们，最后把结果保存在另一个Redis数据库的列表中。处理结果就是所有话题和它们的在tweets中出现的次数列表，所有用户和他们在tweets中出现的次数列表，还有一个包含发起话题和频率的用户列表。 这个工具的拓扑见图。



到的，**TweetsTransactionalSpout**会连接你的tweet数据库并向拓扑分发批次。**UserSplitterBolt**和**HashTagSplitterBolt**两个bolt，从spout接收元组。**UserSplitterBolt**解析tweets并查找用户——以@开头的单词——然后把这些单词分发到名为users的自定义数据流组。**HashtagSplitterBolt**从tweet查找**#**开头的单词，并把它们分发到名为hashtags的自定义数据流组。第三个bolt，**UserHashtagJoinBolt**，接收前面提到的两个数据流组，并计算具名用户的一条tweet内的话题数量。为了计数并分发计算结果，这是个**BaseBatchBolt**（稍后有更多介绍）。

最后一个bolt——**RedisCommitterBolt**——接收以上三个bolt的数据流组。它为每样东西计数，并在对一个批次完成处理时，把所有结果保存到redis。这是一种特殊的bolt，叫做提交者

用**TransactionalTopologyBuilder**构建拓扑，代码如下：



接下来就看看如何在一个事务性拓扑中实现spout。

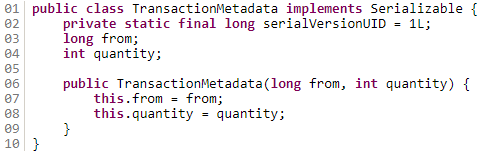
## ****3.2 Spout****

务性拓扑的spout与标准spout完全不同。

C:\Users\liulang\Desktop\screenshot-ifeve.com-2017-09-25-10-02-03-540.png

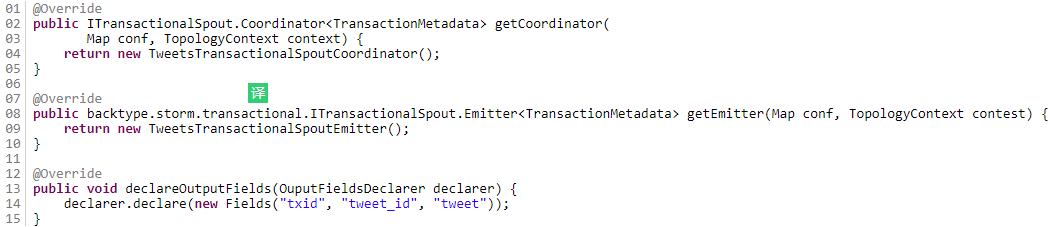
在这个类定义中看到的，TweetsTransactionalSpout继承了带范型的**BaseTransactionalSpout**。指定的范型类型的对象是事务元数据集合。它将在后面的代码中用于从数据源分发批次。

在这个例子中，**TransactionMetadata**定义如下：



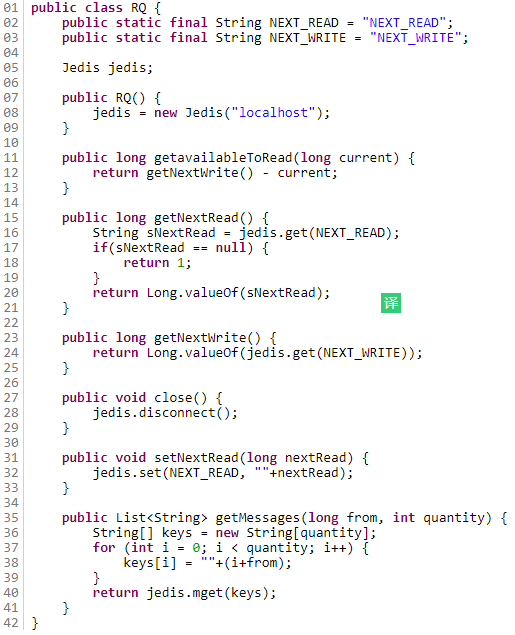
该类的对象维护着两个属性**from**和**quantity**，它们用来生成批次。

spout的最后需要实现下面的三个方法：



**getCoordinator**方法，告诉Storm用来协调生成批次的类。**getEmitter**，负责读取批次并把它们分发到拓扑中的数据流组。最后，就像之前做过的，需要声明要分发的域。

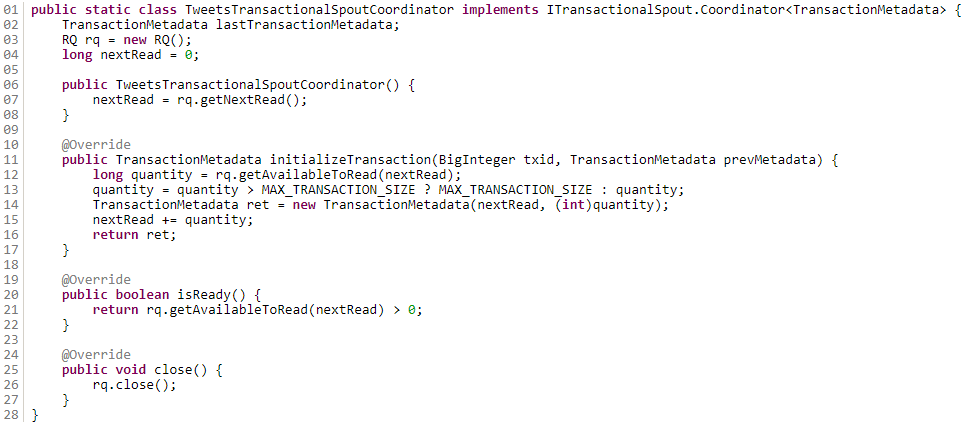
3.3 RQ类 为了让例子简单点，我们决定用一个类封装所有对Redis的操作。



仔细阅读每个方法，确保自己理解了它们的用处。

## 3.4协调者Coordinator

下面是本例的协调者实现。



值得一提的是，在整个拓扑中只会有一个提交者实例。创建提交者实例时，它会从redis读取一个从1开始的序列号，这个序列号标识要读取的tweet下一条。

第一个方法是**isReady**。在**initializeTransaction**之前调用它确认数据源已就绪并可读取。此方法应当相应的返回**true**或**false**。在此例中，读取tweets数量并与已读数量比较。它们之间的不同就在于可读tweets数。如果它大于0，就意味着还有tweets未读。

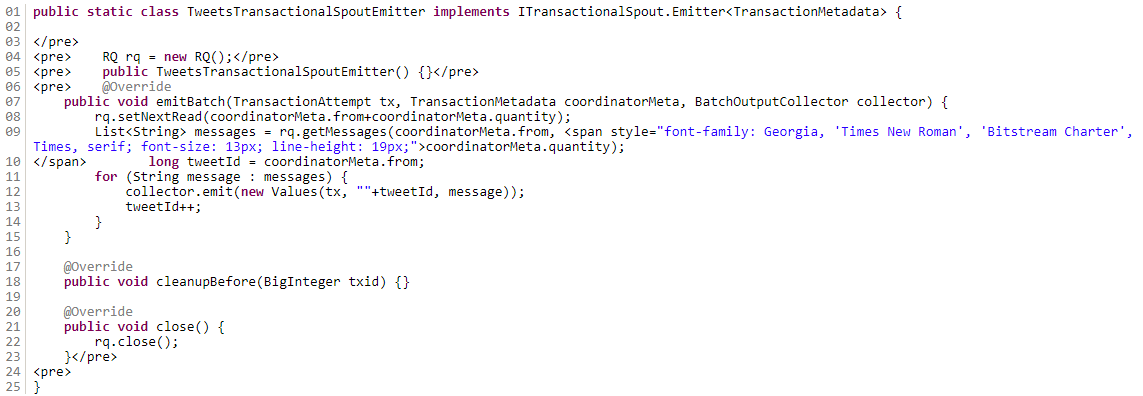
最后，执行**initializeTransaction**。正如你看到的，它接收**txid**和**prevMetadata**作为参数。第一个参数是Storm生成的事务ID，作为批次的惟一性标识。**prevMetadata**是协调器生成的前一个事务元数据对象。

在这个例子中，首先确认有多少tweets可读。只要确认了这一点，就创建一个TransactionMetadata对象，标识读取的第一个tweet（译者注：对象属性**from**），以及读取的tweets数量（译者注：对象属性**quantity**）。

元数据对象一经返回，Storm把它跟**txid**一起保存在zookeeper。这样就确保了一旦发生故障，Storm可以利用分发器(译者注：**Emitter**，见下文)重新发送批次。

## ****3.5 Emitter****

创建事务性spout的最后一步是实现分发器（Emitter）。实现如下：

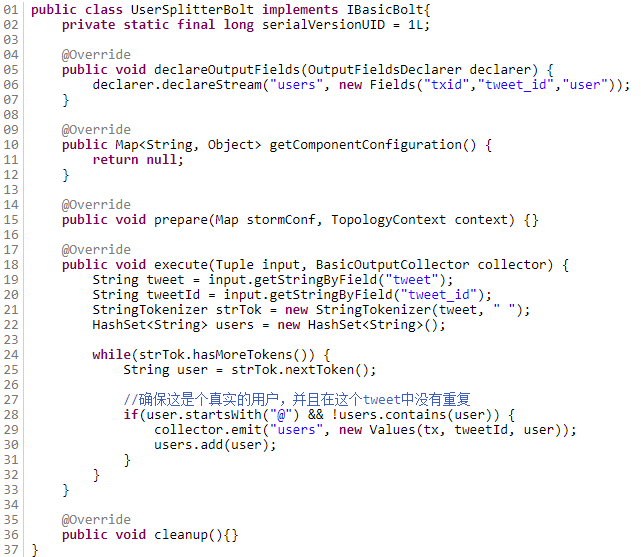


分发器从数据源读取数据并从数据流组发送数据。分发器应当问题能够为相同的事务id和事务元数据发送相同的批次。这样，如果在处理批次的过程中发生了故障，Storm就能够利用分发器重复相同的事务id和事务元数据，并确保批次已经重复过了。Storm会在**TransactionAttempt**对象里为尝试次数增加计数（译者注：**attempt id**）。这样就能知道批次已经重复过了。

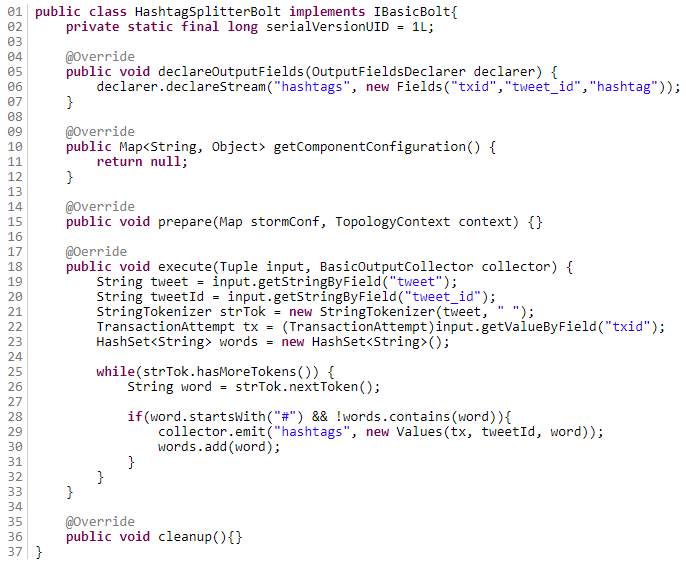
在这里**emitBatch**是个重要方法。在这个方法中，使用传入的元数据对象从redis得到tweets，同时增加redis维持的已读tweets数。当然它还会把读到的tweets分发到拓扑。

## ****3.6 Bolts****

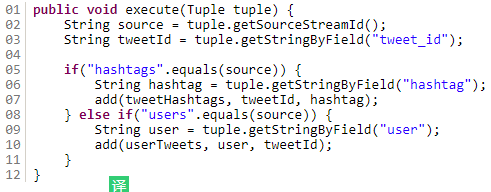
首先看一下这个拓扑中的标准bolt：



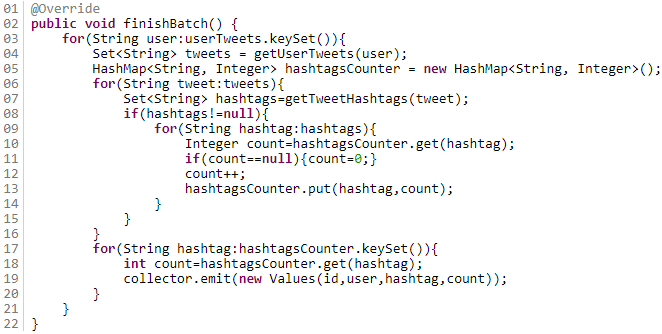
正如本章前面提到的，**UserSplitterBolt**接收元组，解析tweet文本，分发@开头的单词————tweeter用户。**HashtagSplitterBolt**的实现也非常相似。



现在看看**UserHashTagJoinBolt**的实现。首先要注意的是它是一个**BaseBatchBolt**。这意味着，**execute**方法会操作接收到的元组，但是不会分发新的元组。批次完成时，Storm会调用**finishBatch**方法。



既然要结合tweet中提到的用户为出现的所有话题计数，就需要加入前面的bolts创建的两个数据流组。这件事要以批次为单位进程，在批次处理完成时，调用**finishBatch**方法。



这个方法计算每对用户-话题出现的次数，并为之生成和分发元组。

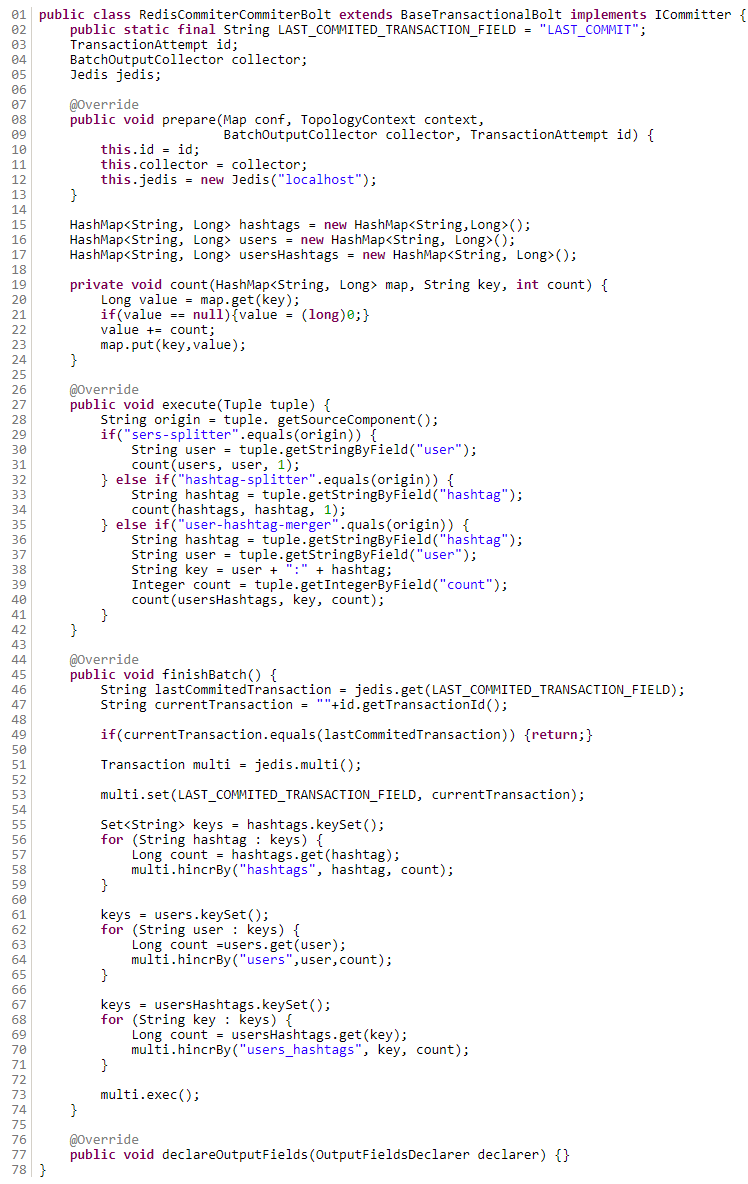
你可以在GitHub上找到并下载完整代码。

## ****3.7提交者****bolts

我们已经学习了，批次通过协调器和分发器怎样在拓扑中传递。在拓扑中，这些批次中的元组以并行的，没有特定次序的方式处理。

协调者bolts是一类特殊的批处理bolts，它们实现了**IComh mitter**或者通过**TransactionalTopologyBuilder**调用**setCommiterBolt**设置了提交者bolt。它们与其它的批处理bolts最大的不同在于，提交者bolts的**finishBatch**方法在提交就绪时执行。这一点发生在之前所有事务都已成功提交之后。另外，**finishBatch**方法是顺序执行的。因此如果同时有事务ID1和事务ID2两个事务同时执行，只有在ID1没有任何差错的执行了**finishBatch**方法之后，ID2才会执行该方法。

下面是这个类的实现



这个实现很简单，但是在**finishBatch**有一个细节。

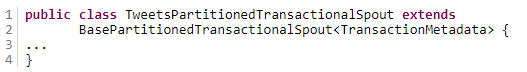
|  |  |
| --- | --- |
| 1 | ... |

|  |  |
| --- | --- |
| 2 | multi.set(LAST\_COMMITED\_TRANSACTION\_FIELD, currentTransaction); |

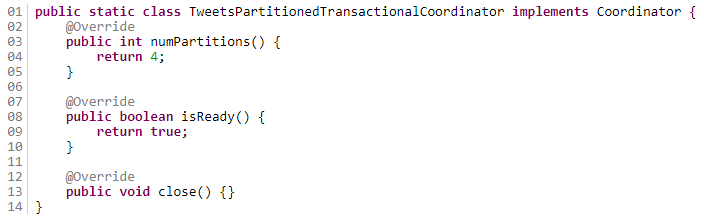
|  |  |
| --- | --- |
| 3 | ... |
|  |  |

在这里向数据库保存提交的最后一个事务ID。为什么要这样做？记住，如果事务失败了，Storm将会尽可能多的重复必要的次数。如果你不确定已经处理了这个事务，你就会多算，事务拓扑也就没有用了。所以请记住：保存最后提交的事务ID，并在提交前检查。

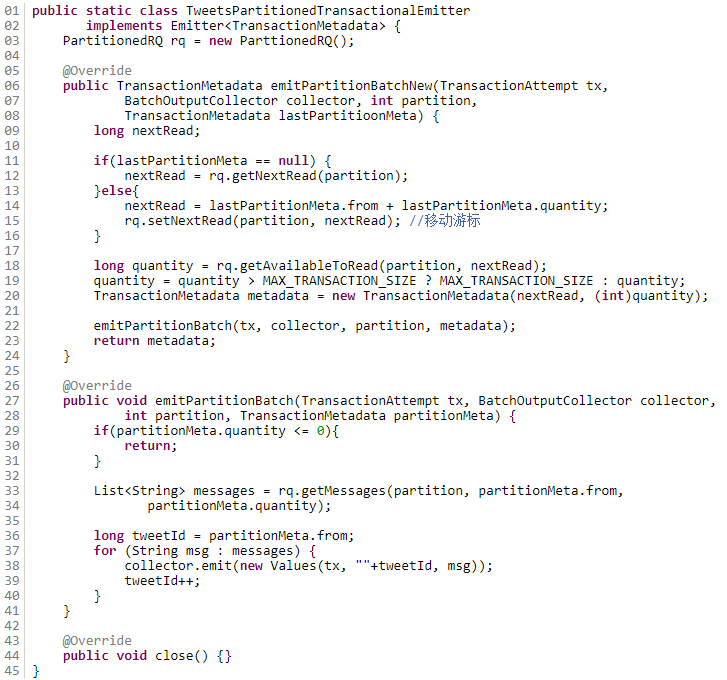
3.8 分区的事务Spouts  
 对一个spout来说，从一个分区集合中读取批次是很普通的。接着这个例子，你可能有很多redis数据库，而tweets可能会分别保存在这些redis数据库里。通过实现IPartitionedTransactionalSpout，Storm提供了一些工具用来管理每个分区的状态并保证重播的能力。  
 下面我们修改TweetsTransactionalSpout，使它可以处理数据分区。  
首先，继承BasePartitionedTransactionalSpout，它实现了IPartitionedTransactionalSpout。



然后告诉Storm谁是你的协调器。



在这个例子里，协调器很简单。numPartitions方法，告诉Storm一共有多少分区。而且你要注意，不要返回任何元数据。对于IPartitionedTransactionalSpout，元数据由分发器直接管理。  
下面是分发器的实现：



这里有两个重要的方法，**emitPartitionBatchNew**，和**emitPartitionBatch**。对于**emitPartitionBatchNew**，从Storm接收分区参数，该参数决定应该从哪个分区读取批次。在这个方法中，决定获取哪些tweets，生成相应的元数据对象，调用**emitPartitionBatch**，返回元数据对象，并且元数据对象会在方法返回时立即保存到zookeeper。  
 Storm会为每一个分区发送相同的事务ID，表示一个事务贯穿了所有数据分区。通过**emitPartitionBatch**读取分区中的tweets，并向拓扑分发批次。如果批次处理失败了，Storm将会调用**emitPartitionBatch**利用保存下来的元数据重复这个批次。

## ****3.9 模糊的事务性拓扑****

到目前为止，你可能已经学会了如何让拥有相同事务ID的批次在出错时重播。但是在有些场景下这样做可能就不太合适了。然后会发生什么呢？

事实证明，你仍然可以实现在语义上精确的事务，不过这需要更多的开发工作，你要记录由Storm重复的事务之前的状态。既然能在不同时刻为相同的事务ID得到不同的元组，你就需要把事务重置到之前的状态，并从那里继续。

比如说，如果你为收到的所有tweets计数，你已数到5，而最后的事务ID是321，这时你多数了8个。你要维护以下三个值——previousCount=5,currentCount=13，以及lastTransactionId=321。假设事物ID321又发分了一次，而你又得到了4个元组，而不是之前的8个，提交器会探测到这是相同的事务ID，它将会把结果重置到**previousCount**的值5，并在此基础上加4，然后更新**currentCount**为9。

另外，在之前的一个事务被取消时，每个并行处理的事务都要被取消。这是为了确保你没有丢失任何数据。

你的spout可以实现**IOpaquePartitionedTransactionalSpout**，而且正如你看到的，协调器和分发器也很简单。



最有趣的方法是**emitPartitionBatch**，它获取之前提交的元数据。你要用它生成批次。这个批次不需要与之前的那个一致，你可能根本无法创建完全一样的批次。剩余的工作由提交器bolts借助之前的状态完成。

# 4.storm安装

## 4.1准备开始

在本章，我们要创建一个Storm工程和我们的第一个Strom拓扑结构。

**NOTE**: 下面假设你的JRE版本在1.6以上。

## 4.2操作模式

开始之前，有必要了解一下Storm的操作模式。下面有两种方式。

## 4.3本地模式

在本地模式下，Storm拓扑结构运行在本地计算机的单一JVM进程上。这个模式用于开发、测试以及调试，因为这是观察所有组件如何协同工作的最简单方法。在这种模式下，我们可以调整参数，观察我们的拓扑结构如何在不同的Storm配置环境下运行。要在本地模式下运行，我们要下载Storm开发依赖，以便用来开发并测试我们的拓扑结构。我们创建了第一个Storm工程以后，很快就会明白如何使用本地模式了。

**NOTE**: 在本地模式下，跟在集群环境运行很像。不过很有必要确认一下所有组件都是线程安全的，因为当把它们部署到远程模式时它们可能会运行在不同的JVM进程甚至不同的物理机上，这个时候它们之间没有直接的通讯或共享内存。

我们要在本地模式运行所有例子。

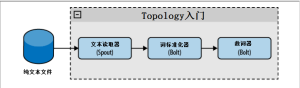
## 4.4远程模式

在远程模式下，我们向Storm集群提交拓扑，它通常由许多运行在不同机器上的流程组成。远程模式不会出现调试信息， 因此它也称作生产模式。不过在单一开发机上建立一个Storm集群是一个好主意，可以在部署到生产环境之前，用来确认拓扑在集群环境下没有任何问题。

## 4.5Hello World

我们在这个工程里创建一个简单的拓扑，数单词数量。我们可以把这个看作Storm的“Hello World”。不过，这是一个非常强大的拓扑，因为它能够扩展到几乎无限大的规模，而且只需要做一些小修改，就能用它构建一个统计系统。举个例子，我们可以修改一下工程用来找出Twitter上的热点话题。

要创建这个拓扑，我们要用一个spout读取文本，第一个bolt用来标准化单词，第二个bolt为单词计数，如图1所示。



你可以从这个网址下载源码压缩包， <https://github.com/storm-book/examples-ch02-getting_started/zipball/master>。

## 4.6 Java安装检查

构建Storm运行环境的第一步是检查你安装的Java版本。打开一个控制台窗口并执行命令：java -version。控制台应该会显示出类似如下的内容：

java -version

java version "1.6.0\_26"

Java(TM) SE Runtime Enviroment (build 1.6.0\_26-b03)

Java HotSpot(TM) Server VM (build 20.1-b02, mixed mode)

## 4.7创建工程

开始之前，先为这个应用建一个目录（就像你平常为Java应用做的那样）。这个目录用来存放工程源码。

接下来我们要下载Storm依赖包，这是一些jar包，我们要把它们添加到应用类路径中。你可以采用如下两种方式之一完成这一步：

下载所有的依赖，解压缩他们，把它们添加到类路径

使用Apache Maven

**NOTE**: Maven是一个软件项目管理的综合工具。它可以用来管理项目的开发周期的许多方面，从包依赖到版本发布过程。在这本书中，我们将广泛使用它。如果要检查是否已经安装了。

要运行我们的拓扑，我们可以编写一个包含基本组件的pom.xml文件。

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>storm.book</groupId>

<artifactId>Getting-Started</artifactId>

<version>0.0.1-SNAPSHOT</version>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>2.3.2</version>

<configuration>

<source>1.6</source>

<target>1.6</target>

<compilerVersion>1.6</compilerVersion>

</configuration>

</plugin>

</plugins>

</build>

<repositories>

<!-- Repository where we can found the storm dependencies -->

<repository>

<id>clojars.org</id>

<url>http://clojars.org/repo</url>

</repository>

</repositories>

<dependencies>

<!-- Storm Dependency -->

<dependency>

<groupId>storm</groupId>

<artifactId>storm</artifactId>

<version>0.6.0</version>

</dependency>

</dependencies>

</project>

开头几行指定了工程名称和版本号。然后我们添加了一个编译器插件，告知Maven我们的代码要用Java1.6编译。接下来我们定义了Maven仓库（Maven支持为同一个工程指定多个仓库）。clojars是存放Storm依赖的仓库。Maven会为运行本地模式自动下载必要的所有子包依赖。

一个典型的Maven Java工程会拥有如下结构：

我们的应用目录/

├── pom.xml

└── src

└── main

└── java

| ├── spouts

| └── bolts

└── resources

java目录下的子目录包含我们的代码，我们把要统计单词数的文件保存在resource目录下。

**NOTE**：命令mkdir -p 会创建所有需要的父目录。

## 4.8创建我们的第一个Topology

我们将为运行单词计数创建所有必要的类。可能这个例子中的某些部分，现在无法讲的很清楚。

### 4.8.1Spout

spout WordReader类实现了IRichSpout接口。我们将在第四章看到更多细节。WordReader负责从文件按行读取文本，并把文本行提供给第一个bolt。

**NOTE**: 一个spout发布一个定义域列表。这个架构允许你使用不同的bolts从同一个spout流读取数据，它们的输出也可作为其它bolts的定义域，以此类推。

例1包含WordRead类的完整代码（我们将会分析下述代码的每一部分）。

/\*\*

\* 例1.src/main/java/spouts/WordReader.java

\*/

package spouts;

import java.io.BufferedReader;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.util.Map;

import backtype.storm.spout.SpoutOutputCollector;

import backtype.storm.task.TopologyContext;

import backtype.storm.topology.IRichSpout;

import backtype.storm.topology.OutputFieldsDeclarer;

import backtype.storm.tuple.Fields;

import backtype.storm.tuple.Values;

public class WordReader implements IRichSpout {

private SpoutOutputCollector collector;

private FileReader fileReader;

private boolean completed = false;

private TopologyContext context;

public boolean isDistributed() {return false;}

public void ack(Object msgId) {

System.out.println("OK:"+msgId);

}

public void close() {}

public void fail(Object msgId) {

System.out.println("FAIL:"+msgId);

}

/\*\*

\* 这个方法做的惟一一件事情就是分发文件中的文本行

\*/

public void nextTuple() {

/\*\*

\* 这个方法会不断的被调用，直到整个文件都读完了，我们将等待并返回。

\*/

if(completed){

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

//什么也不做

}

return;

}

String str;

//创建reader

BufferedReader reader = new BufferedReader(fileReader);

try{

//读所有文本行

while((str = reader.readLine()) != null){

/\*\*

\* 按行发布一个新值

\*/

this.collector.emit(new Values(str),str);

}

}catch(Exception e){

throw new RuntimeException("Error reading tuple",e);

}finally{

completed = true;

}

}

/\*\*

\* 我们将创建一个文件并维持一个collector对象

\*/

public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {

try {

this.context = context;

this.fileReader = new FileReader(conf.get("wordsFile").toString());

} catch (FileNotFoundException e) {

throw new RuntimeException("Error reading file ["+conf.get("wordFile")+"]");

}

this.collector = collector;

}

/\*\*

\* 声明输入域"word"

\*/

public void declareOutputFields(OutputFieldsDeclarer declarer) {

declarer.declare(new Fields("line"));

}

}

第一个被调用的spout方法都是public void open(Map conf, TopologyContext context, SpoutOutputCollector collector)。它接收如下参数：配置对象，在定义topology对象是创建；TopologyContext对象，包含所有拓扑数据；还有SpoutOutputCollector对象，它能让我们发布交给bolts处理的数据。下面的代码主是这个方法的实现。

public void open(Map conf, TopologyContext context,

SpoutOutputCollector collector) {

try {

this.context = context;

this.fileReader = new FileReader(conf.get("wordsFile").toString());

} catch (FileNotFoundException e) {

throw new RuntimeException("Error reading file ["+conf.get("wordFile")+"]");

}

this.collector = collector;

}

我们在这个方法里创建了一个FileReader对象，用来读取文件。接下来我们要实现public void nextTuple()，我们要通过它向bolts发布待处理的数据。在这个例子里，这个方法要读取文件并逐行发布数据。

public void nextTuple() {

if(completed){

try {

Thread.sleep(1);

} catch (InterruptedException e) {

//什么也不做

}

return;

}

String str;

BufferedReader reader = new BufferedReader(fileReader);

try{

while((str = reader.readLine()) != null){

this.collector.emit(new Values(str));

}

}catch(Exception e){

throw new RuntimeException("Error reading tuple",e);

}finally{

completed = true;

}

}

**NOTE**: Values是一个ArrarList实现，它的元素就是传入构造器的参数。

nextTuple()会在同一个循环内被ack()和fail()周期性的调用。没有任务时它必须释放对线程的控制，其它方法才有机会得以执行。因此nextTuple的第一行就要检查是否已处理完成。如果完成了，为了降低处理器负载，会在返回前休眠一毫秒。如果任务完成了，文件中的每一行都已被读出并分发了。

**NOTE**:元组(tuple)是一个具名值列表，它可以是任意java对象（只要它是可序列化的）。默认情况，Storm会序列化字符串、字节数组、ArrayList、HashMap和HashSet等类型。

### 4.8.2 Bolts

现在我们有了一个spout，用来按行读取文件并每行发布一个元组，还要创建两个bolts，用来处理它们（看图2-1）。bolts实现了接口backtype.storm.topology.IRichBolt。

bolt最重要的方法是void execute(Tuple input)，每次接收到元组时都会被调用一次，还会再发布若干个元组。

**NOTE**: 只要必要，bolt或spout会发布若干元组。当调用nextTuple或execute方法时，它们可能会发布0个、1个或许多个元组。

第一个bolt，WordNormalizer，负责得到并标准化每行文本。它把文本行切分成单词，大写转化成小写，去掉头尾空白符。

首先我们要声明bolt的出参：

public void declareOutputFields(OutputFieldsDeclarer declarer){

declarer.declare(new Fields("word"));

}

这里我们声明bolt将发布一个名为“word”的域。

下一步我们实现public void execute(Tuple input)，处理传入的元组：

public void execute(Tuple input){

String sentence=input.getString(0);

String[] words=sentence.split(" ");

for(String word : words){

word=word.trim();

if(!word.isEmpty()){

word=word.toLowerCase();

//发布这个单词

collector.emit(new Values(word));

}

}

//对元组做出应答

collector.ack(input);

}

第一行从元组读取值。值可以按位置或名称读取。接下来值被处理并用collector对象发布。最后，每次都调用collector对象的ack()方法确认已成功处理了一个元组。

例 2 是这个类的完整代码。

//例 2 src/main/java/bolts/WordNormalizer.java

package bolts;

import java.util.ArrayList;

import java.util.List;

import java.util.Map;

import backtype.storm.task.OutputCollector;

import backtype.storm.task.TopologyContext;

import backtype.storm.topology.IRichBolt;

import backtype.storm.topology.OutputFieldsDeclarer;

import backtype.storm.tuple.Fields;

import backtype.storm.tuple.Tuple;

import backtype.storm.tuple.Values;

public class WordNormalizer implements IRichBolt{

private OutputCollector collector;

public void cleanup(){}

/\*\*

\* \*bolt\*从单词文件接收到文本行，并标准化它。

\* 文本行会全部转化成小写，并切分它，从中得到所有单词。

\*/

public void execute(Tuple input){

String sentence = input.getString(0);

String[] words = sentence.split(" ");

for(String word : words){

word = word.trim();

if(!word.isEmpty()){

word=word.toLowerCase();

//发布这个单词

List a = new ArrayList();

a.add(input);

collector.emit(a,new Values(word));

}

}

//对元组做出应答

collector.ack(input);

}

public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {

this.collector=collector;

}

/\*\*

\* 这个\*bolt\*只会发布“word”域

\*/

public void declareOutputFields(OutputFieldsDeclarer declarer) {

declarer.declare(new Fields("word"));

}

}

**NOTE**:通过这个例子，我们了解了在一次execute调用中发布多个元组。如果这个方法在一次调用中接收到句子“This is the Storm book”，它将会发布五个元组。

下一个bolt，WordCounter，负责为单词计数。这个拓扑结束时（cleanup()方法被调用时），我们将显示每个单词的数量。

**NOTE**: 这个例子的bolt什么也没发布，它把数据保存在map里，但是在真实的场景中可以把数据保存到数据库。

package bolts;

import java.util.HashMap;

import java.util.Map;

import backtype.storm.task.OutputCollector;

import backtype.storm.task.TopologyContext;

import backtype.storm.topology.IRichBolt;

import backtype.storm.topology.OutputFieldsDeclarer;

import backtype.storm.tuple.Tuple;

public class WordCounter implements IRichBolt{ Integer id; String name; Map<String,Integer> counters; private OutputCollector collector;

/\*\*

\* 这个spout结束时（集群关闭的时候），我们会显示单词数量

\*/

@Override

public void cleanup(){

System.out.println("-- 单词数 【"+name+"-"+id+"】 --");

for(Map.Entry<String,Integer> entry : counters.entrySet()){

System.out.println(entry.getKey()+": "+entry.getValue());

}

}

/\*\*

\* 为每个单词计数

\*/

@Override

public void execute(Tuple input) {

String str=input.getString(0);

/\*\*

\* 如果单词尚不存在于map，我们就创建一个，如果已在，我们就为它加1

\*/

if(!counters.containsKey(str)){

conters.put(str,1);

}else{

Integer c = counters.get(str) + 1;

counters.put(str,c);

}

//对元组作为应答

collector.ack(input);

}

/\*\*

\* 初始化

\*/

@Override

public void prepare(Map stormConf, TopologyContext context, OutputCollector collector){

this.counters = new HashMap<String, Integer>();

this.collector = collector;

this.name = context.getThisComponentId();

this.id = context.getThisTaskId();

}

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {}

}

execute方法使用一个map收集单词并计数。拓扑结束时，将调用clearup()方法打印计数器map。（虽然这只是一个例子，但是通常情况下，当拓扑关闭时，你应当使用cleanup()方法关闭活动的连接和其它资源。）

### 4.8.3主类

你可以在主类中创建拓扑和一个本地集群对象，以便于在本地测试和调试。LocalCluster可以通过Config对象，让你尝试不同的集群配置。比如，当使用不同数量的工作进程测试你的拓扑时，如果不小心使用了某个全局变量或类变量，你就能够发现错误。

**NOTE**：所有拓扑节点的各个进程必须能够独立运行，而不依赖共享数据（也就是没有全局变量或类变量），因为当拓扑运行在真实的集群环境时，这些进程可能会运行在不同的机器上。

接下来，TopologyBuilder将用来创建拓扑，它决定Storm如何安排各节点，以及它们交换数据的方式。

pologyBuilder builder = new TopologyBuilder();

builder.setSpout("word-reader", new WordReader());

builder.setBolt("word-normalizer", new WordNormalizer()).shuffleGrouping("word-reader");

builder.setBolt("word-counter", new WordCounter()).shuffleGrouping("word-normalizer");

在spout和bolts之间通过shuffleGrouping方法连接。这种分组方式决定了Storm会以随机分配方式从源节点向目标节点发送消息。

下一步，创建一个包含拓扑配置的Config对象，它会在运行时与集群配置合并，并通过prepare方法发送给所有节点。

Config conf = new Config();

conf.put("wordsFile", args[0]);

conf.setDebug(true);

由spout读取的文件的文件名，赋值给wordFile属性。由于是在开发阶段，设置debug属性为true，Strom会打印节点间交换的所有消息，以及其它有助于理解拓扑运行方式的调试数据。

正如之前讲过的，你要用一个LocalCluster对象运行这个拓扑。在生产环境中，拓扑会持续运行，不过对于这个例子而言，你只要运行它几秒钟就能看到结果。

LocalCluster cluster = new LocalCluster();

cluster.submitTopology("Getting-Started-Topologie", conf, builder.createTopology());

Thread.sleep(2000);

cluster.shutdown();

调用createTopology和submitTopology，运行拓扑，休眠两秒钟（拓扑在另外的线程运行），然后关闭集群。

例 3 是完整的代码

//例 3 src/main/java/TopologyMain.java

import spouts.WordReader;

import backtype.storm.Config;

import backtype.storm.LocalCluster;

import backtype.storm.topology.TopologyBuilder;

import backtype.storm.tuple.Fields;

import bolts.WordCounter;

import bolts.WordNormalizer;

public class TopologyMain {

public static void main(String[] args) throws InterruptedException {

//定义拓扑

TopologyBuilder builder = new TopologyBuilder());

builder.setSpout("word-reader", new WordReader());

builder.setBolt("word-normalizer", new WordNormalizer()).shuffleGrouping("word-reader");

builder.setBolt("word-counter", new WordCounter(),2).fieldsGrouping("word-normalizer", new Fields("word"));

//配置

Config conf = new Config();

conf.put("wordsFile", args[0]);

conf.setDebug(false);

//运行拓扑

conf.put(Config.TOPOLOGY\_MAX\_SPOUT\_PENDING, 1);

LocalCluster cluster = new LocalCluster();

cluster.submitTopology("Getting-Started-Topologie", conf, builder.createTopology();

Thread.sleep(1000);

cluster.shutdown();

}

}

### 4.8.4观察运行情况

你已经为运行你的第一个拓扑准备好了。在这个目录下面创建一个文件，/src/main/resources/words.txt，一个单词一行，然后用下面的命令运行这个拓扑：mvn exec:java -Dexec.mainClass=”TopologyMain” -Dexec.args=”src/main/resources/words.txt。

举个例子，如果你的words.txt文件有如下内容： Storm test are great is an Storm simple application but very powerful really Storm is great 你应该会在日志中看到类似下面的内容： is: 2 application: 1 but: 1 great: 1 test: 1 simple: 1 Storm: 3 really: 1 are: 1 great: 1 an: 1 powerful: 1 very: 1 在这个例子中，每类节点只有一个实例。但是如果你有一个非常大的日志文件呢？你能够很轻松的改变系统中的节点数量实现并行工作。这个时候，你就要创建两个WordCounter实例。

builder.setBolt("word-counter", new WordCounter(),2).shuffleGrouping("word-normalizer");

时，你将看到： — 单词数 【word-counter-2】 — application: 1 is: 1 great: 1 are: 1 powerful: 1 Storm: 3 — 单词数 [word-counter-3] — really: 1 is: 1 but: 1 great: 1 test: 1 simple: 1 an: 1 very: 1 棒极了！修改并行度实在是太容易了（当然对于实际情况来说，每个实例都会运行在单独的机器上）。不过似乎有一个问题：单词is和great分别在每个WordCounter各计数一次。怎么会这样？当你调用shuffleGrouping时，就决定了Storm会以随机分配的方式向你的bolt实例发送消息。在这个例子中，理想的做法是相同的单词问题发送给同一个WordCounter实例。你把shuffleGrouping(“word-normalizer”)换成fieldsGrouping(“word-normalizer”, new Fields(“word”))就能达到目的。试一试，重新运行程序，确认结果。