Storm入事务性拓扑

使用Storm编程，可以通过调用ack和fail方法来确保一条消息的处理成功或失败。不过当元组被重发时，会发生什么呢？你又该如何砍不会重复计算？

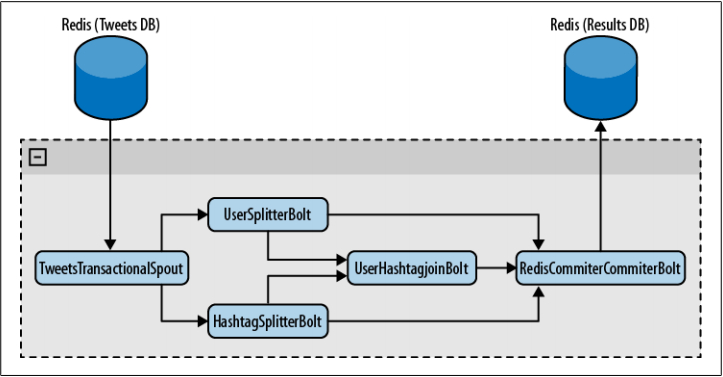
*Storm0.7.0*实现了一个新特性——事务性拓扑，这一特性使消息在语义上确保你可以安全的方式重发消息，并保证它们只会被处理一次。在不支持事务性拓扑的情况下，你无法在准确性，可扩展性，以空错性上得到保证的前提下完成计算。

**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。

**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8-1。[](http://ifeve.com/getting-started-of-storm8-2/figure8-1/)  图8-1 拓扑概览

正如你看到的，**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，在本章后面做更多讲解。

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

TransactionalTopologyBuilder builder=

new TransactionalTopologyBuilder("test", "spout", new TweetsTransactionalSpout());

builder.setBolt("users-splitter", new UserSplitterBolt(), 4).shuffleGrouping("spout");

buildeer.setBolt("hashtag-splitter", new HashtagSplitterBolt(), 4).shuffleGrouping("spout");

builder.setBolt("users-hashtag-manager", new UserHashtagJoinBolt(), r)

.fieldsGrouping("users-splitter", "users", new Fields("tweet\_id"))

.fieldsGrouping("hashtag-splitter", "hashtags", new Fields("tweet\_id"));

builder.setBolt("redis-commiter", new RedisCommiterBolt())

.globalGrouping("users-splitter", "users")

.globalGrouping("hashtag-splitter", "hashtags")

.globalGrouping("user-hashtag-merger");

接下来就看看如何在一个事务性拓扑中实现*spout*。

***Spout***

一个事务性拓扑的*spout*与标准*spout*完全不同。

public class TweetsTransactionalSpout extends BaseTransactionalSpout<TransactionMetadata>{

正如你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：

public class TransactionMetadata implements Serializable {

private static final long serialVersionUID = 1L;

long from;

int quantity;

public TransactionMetadata(long from, int quantity) {

this.from = from;

this.quantity = quantity;

}

}

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：

@Override

public ITransactionalSpout.Coordinator<TransactionMetadata> getCoordinator(

Map conf, TopologyContext context) {

return new TweetsTransactionalSpoutCoordinator();

}

@Override

public backtype.storm.transactional.ITransactionalSpout.Emitter<TransactionMetadata> getEmitter(Map conf, TopologyContext contest) {

return new TweetsTransactionalSpoutEmitter();

}

@Override

public void declareOutputFields(OuputFieldsDeclarer declarer) {

declarer.declare(new Fields("txid", "tweet\_id", "tweet"));

}

**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。

**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。

public class RQ {

public static final String NEXT\_READ = "NEXT\_READ";

public static final String NEXT\_WRITE = "NEXT\_WRITE";

Jedis jedis;

public RQ() {

jedis = new Jedis("localhost");

}

public long getavailableToRead(long current) {

return getNextWrite() - current;

}

public long getNextRead() {

String sNextRead = jedis.get(NEXT\_READ);

if(sNextRead == null) {

return 1;

}

return Long.valueOf(sNextRead);

}

public long getNextWrite() {

return Long.valueOf(jedis.get(NEXT\_WRITE));

}

public void close() {

jedis.disconnect();

}

public void setNextRead(long nextRead) {

jedis.set(NEXT\_READ, ""+nextRead);

}

public List<String> getMessages(long from, int quantity) {

String[] keys = new String[quantity];

for (int i = 0; i < quantity; i++) {

keys[i] = ""+(i+from);

}

return jedis.mget(keys);

}

}

仔细阅读每个方法，确保自己理解了它们的用处。

**协调者Coordinator**下面是本例的协调者实现。

public static class TweetsTransactionalSpoutCoordinator implements ITransactionalSpout.Coordinator<TransactionMetadata> {

TransactionMetadata lastTransactionMetadata;

RQ rq = new RQ();

long nextRead = 0;

public TweetsTransactionalSpoutCoordinator() {

nextRead = rq.getNextRead();

}

@Override

public TransactionMetadata initializeTransaction(BigInteger txid, TransactionMetadata prevMetadata) {

long quantity = rq.getAvailableToRead(nextRead);

quantity = quantity > MAX\_TRANSACTION\_SIZE ? MAX\_TRANSACTION\_SIZE : quantity;

TransactionMetadata ret = new TransactionMetadata(nextRead, (int)quantity);

nextRead += quantity;

return ret;

}

@Override

public boolean isReady() {

return rq.getAvailableToRead(nextRead) > 0;

}

@Override

public void close() {

rq.close();

}

}

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。

**Emitter**

创建事务性*spout*的最后一步是实现分发器（Emitter）。实现如下：

public static class TweetsTransactionalSpoutEmitter implements ITransactionalSpout.Emitter<TransactionMetadata> {

</pre>

<pre>    RQ rq = new RQ();</pre>

<pre>    public TweetsTransactionalSpoutEmitter() {}</pre>

<pre>    @Override

    public void emitBatch(TransactionAttempt tx, TransactionMetadata coordinatorMeta, BatchOutputCollector collector) {

        rq.setNextRead(coordinatorMeta.from+coordinatorMeta.quantity);

        List<String> messages = rq.getMessages(coordinatorMeta.from, <span style="font-family: Georgia, 'Times New Roman', 'Bitstream Charter', Times, serif; font-size: 13px; line-height: 19px;">coordinatorMeta.quantity);

</span> long tweetId = coordinatorMeta.from;

for (String message : messages) {

collector.emit(new Values(tx, ""+tweetId, message));

   tweetId++;

}

}

@Override

public void cleanupBefore(BigInteger txid) {}

@Override

public void close() {

rq.close();

}</pre>

<pre>

}

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。

***Bolts***

首先看一下这个拓扑中的标准*bolt*：

public class UserSplitterBolt implements IBasicBolt{

private static final long serialVersionUID = 1L;

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

declarer.declareStream("users", new Fields("txid","tweet\_id","user"));

}

@Override

public Map<String, Object> getComponentConfiguration() {

return null;

}

@Override

public void prepare(Map stormConf, TopologyContext context) {}

@Override

public void execute(Tuple input, BasicOutputCollector collector) {

String tweet = input.getStringByField("tweet");

String tweetId = input.getStringByField("tweet\_id");

StringTokenizer strTok = new StringTokenizer(tweet, " ");

HashSet<String> users = new HashSet<String>();

while(strTok.hasMoreTokens()) {

String user = strTok.nextToken();

//确保这是个真实的用户，并且在这个tweet中没有重复

if(user.startsWith("@") && !users.contains(user)) {

collector.emit("users", new Values(tx, tweetId, user));

users.add(user);

}

}

}

@Override

public void cleanup(){}

}

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。

public class HashtagSplitterBolt implements IBasicBolt{

private static final long serialVersionUID = 1L;

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

declarer.declareStream("hashtags", new Fields("txid","tweet\_id","hashtag"));

}

@Override

public Map<String, Object> getComponentConfiguration() {

return null;

}

@Override

public void prepare(Map stormConf, TopologyContext context) {}

@Oerride

public void execute(Tuple input, BasicOutputCollector collector) {

String tweet = input.getStringByField("tweet");

String tweetId = input.getStringByField("tweet\_id");

StringTokenizer strTok = new StringTokenizer(tweet, " ");

TransactionAttempt tx = (TransactionAttempt)input.getValueByField("txid");

HashSet<String> words = new HashSet<String>();

while(strTok.hasMoreTokens()) {

String word = strTok.nextToken();

if(word.startsWith("#") && !words.contains(word)){

collector.emit("hashtags", new Values(tx, tweetId, word));

words.add(word);

}

}

}

@Override

public void cleanup(){}

}

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。

public void execute(Tuple tuple) {

String source = tuple.getSourceStreamId();

String tweetId = tuple.getStringByField("tweet\_id");

if("hashtags".equals(source)) {

String hashtag = tuple.getStringByField("hashtag");

add(tweetHashtags, tweetId, hashtag);

} else if("users".equals(source)) {

String user = tuple.getStringByField("user");

add(userTweets, user, tweetId);

}

}

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。

@Override

public void finishBatch() {

for(String user:userTweets.keySet()){

Set<String> tweets = getUserTweets(user);

HashMap<String, Integer> hashtagsCounter = new HashMap<String, Integer>();

for(String tweet:tweets){

Set<String> hashtags=getTweetHashtags(tweet);

if(hashtags!=null){

for(String hashtag:hashtags){

Integer count=hashtagsCounter.get(hashtag);

if(count==null){count=0;}

count++;

hashtagsCounter.put(hashtag,count);

}

}

}

for(String hashtag:hashtagsCounter.keySet()){

int count=hashtagsCounter.get(hashtag);

collector.emit(new Values(id,user,hashtag,count));

}

}

}

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。（译者注：https://github.com/storm-book/examples-ch08-transactional-topologies这个仓库里没有代码，谁知道哪里有代码麻烦说一声。）

**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

public class RedisCommiterCommiterBolt extends BaseTransactionalBolt implements ICommitter {

public static final String LAST\_COMMITED\_TRANSACTION\_FIELD = "LAST\_COMMIT";

TransactionAttempt id;

BatchOutputCollector collector;

Jedis jedis;

@Override

public void prepare(Map conf, TopologyContext context,

BatchOutputCollector collector, TransactionAttempt id) {

this.id = id;

this.collector = collector;

this.jedis = new Jedis("localhost");

}

HashMap<String, Long> hashtags = new HashMap<String,Long>();

HashMap<String, Long> users = new HashMap<String, Long>();

HashMap<String, Long> usersHashtags = new HashMap<String, Long>();

private void count(HashMap<String, Long> map, String key, int count) {

Long value = map.get(key);

if(value == null){value = (long)0;}

value += count;

map.put(key,value);

}

@Override

public void execute(Tuple tuple) {

String origin = tuple. getSourceComponent();

if("sers-splitter".equals(origin)) {

String user = tuple.getStringByField("user");

count(users, user, 1);

} else if("hashtag-splitter".equals(origin)) {

String hashtag = tuple.getStringByField("hashtag");

count(hashtags, hashtag, 1);

} else if("user-hashtag-merger".quals(origin)) {

String hashtag = tuple.getStringByField("hashtag");

String user = tuple.getStringByField("user");

String key = user + ":" + hashtag;

Integer count = tuple.getIntegerByField("count");

count(usersHashtags, key, count);

}

}

@Override

public void finishBatch() {

String lastCommitedTransaction = jedis.get(LAST\_COMMITED\_TRANSACTION\_FIELD);

String currentTransaction = ""+id.getTransactionId();

if(currentTransaction.equals(lastCommitedTransaction)) {return;}

Transaction multi = jedis.multi();

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

Set<String> keys = hashtags.keySet();

for (String hashtag : keys) {

Long count = hashtags.get(hashtag);

multi.hincrBy("hashtags", hashtag, count);

}

keys = users.keySet();

for (String user : keys) {

Long count =users.get(user);

multi.hincrBy("users",user,count);

}

keys = usersHashtags.keySet();

for (String key : keys) {

Long count = usersHashtags.get(key);

multi.hincrBy("users\_hashtags", key, count);

}

multi.exec();

}

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {}

}

这个实现很简单，但是在**finishBatch**有一个细节。

...

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

...

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。

**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**。

public class TweetsPartitionedTransactionalSpout extends

BasePartitionedTransactionalSpout<TransactionMetadata> {

...

}

然后告诉Storm谁是你的协调器。

public static class TweetsPartitionedTransactionalCoordinator implements Coordinator {

@Override

public int numPartitions() {

return 4;

}

@Override

public boolean isReady() {

return true;

}

@Override

public void close() {}

}

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：

public static class TweetsPartitionedTransactionalEmitter

implements Emitter<TransactionMetadata> {

PartitionedRQ rq = new ParttionedRQ();

@Override

public TransactionMetadata emitPartitionBatchNew(TransactionAttempt tx,

BatchOutputCollector collector, int partition,

TransactionMetadata lastPartitioonMeta) {

long nextRead;

if(lastPartitionMeta == null) {

nextRead = rq.getNextRead(partition);

}else{

nextRead = lastPartitionMeta.from + lastPartitionMeta.quantity;

rq.setNextRead(partition, nextRead); //移动游标

}

long quantity = rq.getAvailableToRead(partition, nextRead);

quantity = quantity > MAX\_TRANSACTION\_SIZE ? MAX\_TRANSACTION\_SIZE : quantity;

TransactionMetadata metadata = new TransactionMetadata(nextRead, (int)quantity);

emitPartitionBatch(tx, collector, partition, metadata);

return metadata;

}

@Override

public void emitPartitionBatch(TransactionAttempt tx, BatchOutputCollector collector,

int partition, TransactionMetadata partitionMeta) {

if(partitionMeta.quantity <= 0){

return;

}

List<String> messages = rq.getMessages(partition, partitionMeta.from,

partitionMeta.quantity);

long tweetId = partitionMeta.from;

for (String msg : messages) {

collector.emit(new Values(tx, ""+tweetId, msg));

tweetId++;

}

}

@Override

public void close() {}

}

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。

**NOTE:** 完整的源码请见：<https://github.com/storm-book/examples-ch08-transactional-topologies>（译者注：原文如此，实际上这个仓库里什么也没有）

**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。

public static class TweetsOpaquePartitionedTransactionalSpoutCoordinator implements IOpaquePartitionedTransactionalSpout.Coordinator {

@Override

public boolean isReady() {

return true;

}

}

public static class TweetsOpaquePartitionedTransactionalSpoutEmitter

implements IOpaquePartitionedTransactionalSpout.Emitter<TransactionMetadata> {

PartitionedRQ rq = new PartitionedRQ();

@Override

public TransactionMetadata emitPartitionBatch(TransactionAttempt tx,

BatchOutputCollector collector, int partion,

TransactionMetadata lastPartitonMeta) {

long nextRead;

if(lastPartitionMeta == null) {

nextRead = rq.getNextRead(partition);

}else{

nextRead = lastPartitionMeta.from + lastPartitionMeta.quantity;

rq.setNextRead(partition, nextRead);//移动游标

}

long quantity = rq.getAvailabletoRead(partition, nextRead);

quantity = quantity > MAX\_TRANSACTION\_SIZE ? MAX\_TRANSACTION\_SIZE : quantity;

TransactionMetadata metadata = new TransactionMetadata(nextRead, (int)quantity);

emitMessages(tx, collector, partition, metadata);

return metadata;

}

private void emitMessage(TransactionAttempt tx, BatchOutputCollector collector,

int partition, TransactionMetadata partitionMeta) {

if(partitionMeta.quantity <= 0){return;}

List<String> messages = rq.getMessages(partition, partitionMeta.from, partitionMeta.quantity);

long tweetId = partitionMeta.from;

for(String msg : messages) {

collector.emit(new Values(tx, ""+tweetId, msg));

tweetId++;

}

}

@Override

public int numPartitions() {

return 4;

}

@Override

public void close() {}

}

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。