# Storm概述

2018年9月13日 星期四　　上午 9:14

## 1. 实时分析概念

### a. 离线分析

通常是 需要一段时间的数据积累 积累到一定数量数据后 开始离线分析 无论数据量多大 离线分析 有开始 也有结束 最终得到一个处理的结果 这样的分析过程 得到的结果是有较大的延迟的

### b. 实时分析

通常 数据不停的到来 随着数据的到来 来进行增量的运算 立即得到新数据的处理结果 并没有一个数据积累的过程 有开始 但是没有明确的结束的时刻 数据实时的进行运算 基本没有延迟

## 2. Storm概述

Storm是一个开源的分布式实时计算系统，可以简单、可靠的处理大量的数据流。

Storm有很多使用场景：如实时分析，在线机器学习，持续计算，分布式RPC，ETL等等。

Storm支持水平扩展，具有高容错性，保证每个消息都会得到处理。

Storm性能优良，处理速度很快(在一个小集群中，每个结点每秒可以处理数以百万计的消息)。

Storm的部署和运维都很便捷，而且更为重要的是可以使用任意编程语言来开发应用。

## 3. Storm组件

Storm将实时运算的过程 拆分为若干简单的步骤 再组装在一起完成复杂计算任务 以便于实现分布式的流式处理

由这些简单步骤组装起来的运算过程 称之为一个Topology(拓扑)

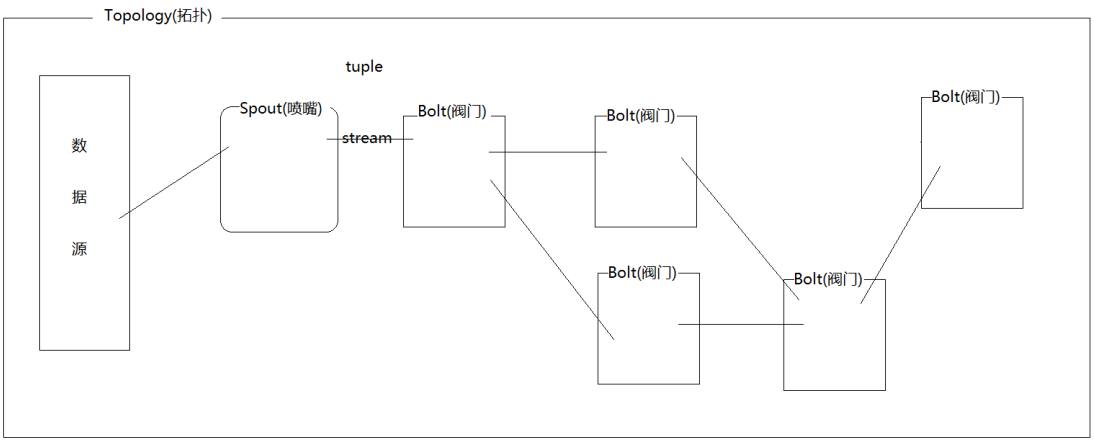
Topology由Spout(喷嘴) 和 Bolt(阀门)组成

Spout负责连接外部数据源 接收数据 并将数据 转换为Tuple格式 向后发送 可以发送给 一个或多个Bolt,Spout通常只负责接收 转换 发送数据,不会进行任何的业务处理

Bolt负责接收上游传入的Tuple数据,进行相应的运算,再将结果发送给后续的零个或多个Bolt

从而利用Spuot和Bolt组建起复杂的数据流处理网络 实现实时处理

在整个Topology中传递的Tuple就组成了Stream(数据流)



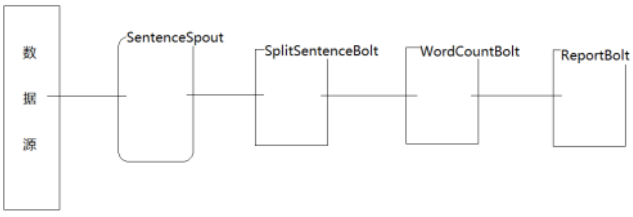
# Storm入门案例

2018年9月13日 星期四　上午 9:53

1. Storm入门案例 - WordCount案例
   a. 单机程序处理流程:

```
String [] sentences  = ...
Map<String,Integer> map = ...
//1.得到每条语句
for(String sentence : sentences){
    //2.切分语句
    String [] words = sentence.split(" ");
    //3.统计数量
    for(String word : words){
        map.put(word,map.contains(word) ? map.get(word)+1 : 1);
    }
}
//4.打印结果
syso(map)
```

   b. Topology设计:




2. SentenceSpout开发

**void cn.tedu.storm.wc.SentenceSpout.open(Map conf, TopologyContext context, SpoutOutputCollector collector)**

@Override

Called when a task for this component is initialized within a worker on the cluster. It provides the spout with the environment in which the spout executes.

This includes the:

**Specified by:** open(...) in ISpout

**Parameters:**

**conf** The Storm configuration for this spout. This is the configuration provided to the topology merged in with cluster configuration on this machine.

**context** This object can be used to get information about this task's place within the topology, including the task id and component id of this task, input and output information, etc.

**collector** The collector is used to emit tuples from this spout. Tuples can be emitted at any time, including the open and close methods. The collector is thread-safe and should be saved as an instance variable of this spout object.

**void cn.tedu.storm.wc.SentenceSpout.nextTuple()**

@Override

When this method is called, Storm is requesting that the Spout emit tuples to the output collector. This method should be non-blocking, so if the Spout has no tuples to emit, this method should return. nextTuple, ack, and fail are all called in a tight loop in a single thread in the spout task. When there are no tuples to emit, it is courteous to have nextTuple sleep for a short amount of time (like a single

millisecond) so as not to waste too much CPU.

**Specified by:** nextTuple() in ISpout

**void cn.tedu.storm.wc.SentenceSpout.declareOutputFields(OutputFieldsDeclarer declarer)**

@Override

Declare the output schema for all the streams of this topology.

**Specified by:** declareOutputFields(...) in IComponent

**Parameters:**

**declarer** this is used to declare output stream ids, output fields, and whether or not each output stream is a direct stream

```
package cn.tedu.storm.wc;


import java.util.Map;


import backtype.storm.spout.SpoutOutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichSpout;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Values;

public class SentenceSpout extends BaseRichSpout {

        String sentences [] = {
                "my name is park",
                "i am so shuai",
                "do you like me",
                "are you sure you do not like me",
                "ok i am sure"
        };

        private SpoutOutputCollector collector = null;

        /**
         * 初始化的方法
         * 在当前组件被初始化时被调用 执行初始化相关操作
         *
         * conf: 当前Topology的配置信息
         * context: 当前组件运行的上下文环境信息
         * collector: 可以用来发送tuple,可以在任意位置发送tuple,此对象线程安全,通常保存到类内部作为成员变量,方便在其他方法中访问
         */
        @Override
        public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
                this.collector = collector;
        }

        /**
         * storm会不停的调用这个方法 要求发送tuple
         * 此方法不应该被阻塞 所以如果没有任何tuple要发送 直接返回即可
         * 在底层 其实就是一个单一的线程内循环不停的再调用此方法 所以如果真的没有任何tuple要发送 最好再这个方法中 睡眠一小段时间 以便于不至于浪费过多的cpu
         */
        private int index = 0;
        @Override
        public void nextTuple() {
                if(index < sentences.length){
                        collector.emit(new Values(sentences[index]));
                        index++;
                }else{
                        try {
                                Thread.sleep(1);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                        return;
                }
        }

        /**
         * 用来声明当前组件输出的tuple的基本信息
         * declare:所有的组件如果想要发送tuple,都必须先声明再发送,此对象就是用来声明tuple的基本信息
         */
        @Override
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
                declarer.declare(new Fields("sentence"));
        }
}
```

3. SplitSentenceBolt开发

**void cn.tedu.storm.wc.SplitSentenceBolt.prepare(Map stormConf, TopologyContext context,**

**OutputCollector collector)**

@Override

Called when a task for this component is initialized within a worker on the cluster. It provides the bolt with the environment in which the bolt executes.

This includes the:

**Specified by:** prepare(...) in IBolt

**Parameters:**

**stormConf** The Storm configuration for this bolt. This is the configuration provided to the topology merged in with cluster configuration on this machine.

**context** This object can be used to get information about this task's place within the topology, including the task id and component id of this task, input and output information, etc.

**collector** The collector is used to emit tuples from this bolt. Tuples can be emitted at any time, including the prepare and cleanup methods. The collector is thread-safe and should be saved as an instance variable of this bolt object.

**void cn.tedu.storm.wc.SplitSentenceBolt.execute(Tuple input)**

@Override

Process a single tuple of input. The Tuple object contains metadata on it about which component/stream/task it came from. The values of the Tuple can be accessed using Tuple#getValue. The IBolt does not have to process the Tuple immediately. It is perfectly fine to hang onto a tuple and process it later (for instance, to do an aggregation or join).

Tuples should be emitted using the OutputCollector provided through the prepare method. It is required that all input tuples are acked or failed at some point using the OutputCollector. Otherwise, Storm will be unable to determine when tuples coming off the spouts have been completed.

For the common case of acking an input tuple at the end of the execute method, see IBasicBolt which automates this.

**Specified by:** execute(...) in IBolt

**Parameters:**

**input** The input tuple to be processed.

**void cn.tedu.storm.wc.SplitSentenceBolt.declareOutputFields(OutputFieldsDeclarer declarer)**

@Override

Declare the output schema for all the streams of this topology.

**Specified by:** declareOutputFields(...) in IComponent

**Parameters:**

**declarer** this is used to declare output stream ids, output fields, and whether or not each output stream is a direct stream

```
1  package cn.tedu.storm.wc;
2
3
4  import java.util.Map;
5
6
7  import backtype.storm.task.OutputCollector;
8  import backtype.storm.task.TopologyContext;
9  import backtype.storm.topology.OutputFieldsDeclarer;
10 import backtype.storm.topology.base.BaseRichBolt;
11 import backtype.storm.tuple.Fields;
12 import backtype.storm.tuple.Tuple;
13 import backtype.storm.tuple.Values;
14
```

```
public class SplitSentenceBolt extends BaseRichBolt{

	private OutputCollector collector = null;

	/**
	 * 初始化的方法
	 * 在当前组件被初始化时被调用 执行初始化相关操作
	 *
	 * conf: 当前Topology的配置信息
	 * context: 当前组件运行的上下文环境信息
	 * collector: 可以用来发送tuple,可以在任意位置发送tuple,此对象线程安全,通常保存到类内部作为成员变量,方便在其他方法中访问
	 */
	@Override
	public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
		this.collector = collector;
	}

	/**
	 * 处理上游发送的tuple,每个tuple都会进入一次此方法 进行处理
	 * 处理过后可以发送零个或多个tuple给后续的组件
	 * 如果需要,也可以不立即处理tuple,而是先持有tuple再在后续需要时处理
	 *
	 * input:当前传入的要处理的tuple
	 */
	@Override
	public void execute(Tuple input) {
		String sentence =  input.getStringByField("sentence");
		String []  words = sentence.split(" ");
		for(String word : words){
			collector.emit(new Values(word));
		}
	}

	/**
	 * 用来声明当前组件输出的tuple的基本信息
	 * declare:所有的组件如果想要发送tuple,都必须先声明再发送,此对象就是用来声明tuple的基本信息
	 */
	@Override
	public void declareOutputFields(OutputFieldsDeclarer declarer) {
		declarer.declare(new Fields("word"));
	}
}
```

4. WordCountBolt

```
package cn.tedu.storm.wc;

import java.util.HashMap;
import java.util.Map;

import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;

public class WordCountBolt extends BaseRichBolt {
	private OutputCollector collector = null;
	@Override
	public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
		this.collector = collector;
	}


	private Map<String,Integer> map = new HashMap<>();
	@Override
	public void execute(Tuple input) {
		String word = input.getStringByField("word");
		map.put(word, map.containsKey(word) ? map.get(word) + 1 : 1);
		collector.emit(new Values(word,map.get(word)));
	}


	@Override
	public void declareOutputFields(OutputFieldsDeclarer declarer) {
		declarer.declare(new Fields("word","count"));
	}


}
```

5. ReportBolt

```
package cn.tedu.storm.wc;


import java.util.Map;


import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichBolt;
import backtype.storm.tuple.Tuple;

public class ReportBolt extends BaseRichBolt {


	private OutputCollector collector = null;
```

```
17
18          @Override
19          public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
20                  this.collector = collector;
21          }
22
23
24          @Override
25          public void execute(Tuple input) {
26                  String word =input.getStringByField("word");
27                  int count = input.getIntegerByField("count");
28                  System.err.println("===单词数量发生变化：word["+word+"],count["+count+"]====");
29          }
30
31          @Override
32          public void declareOutputFields(OutputFieldsDeclarer declarer) {

            }
    }
```

## 6. WcTopology

```
1   package cn.tedu.storm.wc;
2
3
4   import backtype.storm.Config;
5   import backtype.storm.LocalCluster;
6   import backtype.storm.generated.StormTopology;
7   import backtype.storm.topology.TopologyBuilder;
8   import backtype.storm.tuple.Fields;
9
10  public class WordCountTopology {
11        public static void main(String[] args) throws Exception {
12              //1.创建出所有组件
13              SentenceSpout spout = new SentenceSpout();
14              SplitSentenceBolt splitSentenceBolt = new SplitSentenceBolt();
15              WordCountBolt wordCountBolt = new WordCountBolt();
16              ReportBolt reportBolt = new ReportBolt();
17
18              //2.创建拓扑构建者
19              TopologyBuilder builder = new TopologyBuilder();
20
21              //3.向拓扑构建者告知 拓扑结构
22              builder.setSpout("Sentence_Spout", spout);
23              builder.setBolt("Split_Sentence_Bolt", splitSentenceBolt).shuffleGrouping("Sentence_Spout");
24              builder.setBolt("Word_Count_Bolt", wordCountBolt).fieldsGrouping("Split_Sentence_Bolt", new Fields("word"));
25              builder.setBolt("Report_Bolt", reportBolt).globalGrouping("Word_Count_Bolt");
26
27              //4.利用拓扑构建者创建拓扑
28              StormTopology topology = builder.createTopology();
29
30              //5.将拓扑提交到集群中运行
31  //          Config conf = new Config();
32  //          StormSubmitter.submitTopology("Wc_Topology", conf, topology);
33
34
35              //5.在本地集群中模拟运行拓扑
36              LocalCluster cluster = new LocalCluster();
37              Config conf = new Config();
38              cluster.submitTopology("Wc_Topology", conf, topology);
39
40              Thread.sleep(10 * 1000);
41
42              cluster.killTopology("Wc_Topology");
43              cluster.shutdown();
        }
    }
```

# Storm的并发机制

2018年9月13日 星期四　　下午 2:07

## 1. Storm的并发级别

Storm可以在如下的四个级别上存在并发：

1) Node服务器

配置在Storm集群中的一个服务器，会执行Topology的一部分运算,一个Storm集群中包含一个或者多个Node

2) Worker进程

JVM虚拟机、进程。指一个Node上相互独立运作的JVM进程，每个Node可以配置运行一个或多个worker。一个Topology会分配到一个或者多个worker上运行。

3) Executor线程

指一个worker的jvm中运行的java线程。多个task可以指派给同一个executer来执行。除非是明确指定，Storm默认会给每个executor分配一个task。

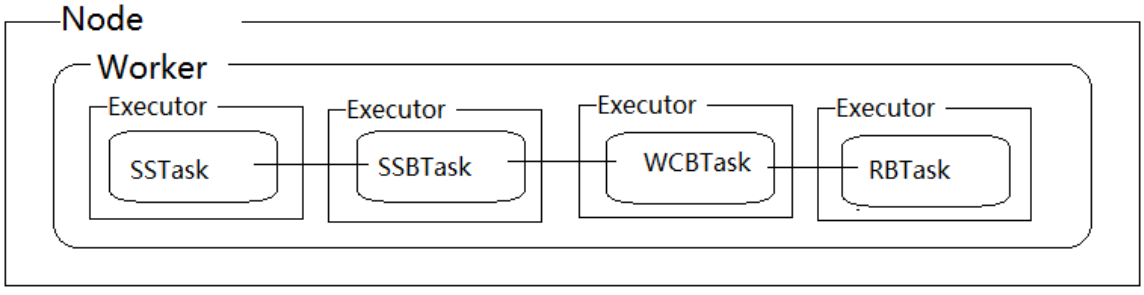
4) Task任务

bolt/spout实例。task是spout或bolt的实例，他们的nextTuple()和execute()方法会被executors线程调用执行。一个线程可以同时处理多个Task。

## 2. storm的并发控制

a. 默认并发控制

如果不进行任何配置，默认并发度为1，也即，默认情况下一个Node服务器上运行一个Worker进程，一个Worker进程里分配多个Executor线程，而每个Executor线程分别负责一个Task任务，此时唯一的并发处在线程级别。




b. 人为指定并发度

i. Node级别

Node级别的并发度无法通过代码控制，只能增加服务器硬件资源。

ii. Worker级别

可以通过代码为指定的Topology配置多个Worker

```
Config conf = new Config();
conf.setNumWorkers(2);
```

**单机测试模式下，设定Worker的数量没有意义，不会起效果。

iii. Executor级别

可以通过代码指定Spout或Bolt需要几个线程

builder.setSpout(spout_id,spout,2)

builder.setBolt(bolt_id,bolt,2)

** 如果只是指定线程数量，而未指定Task数量，则这个组件就会有指定数量的线程来处理，而默认每个线程内都执行一个该组件的Task

iv. Task级别

可以通过代码指定Spout或Bolt需要并发几个Task

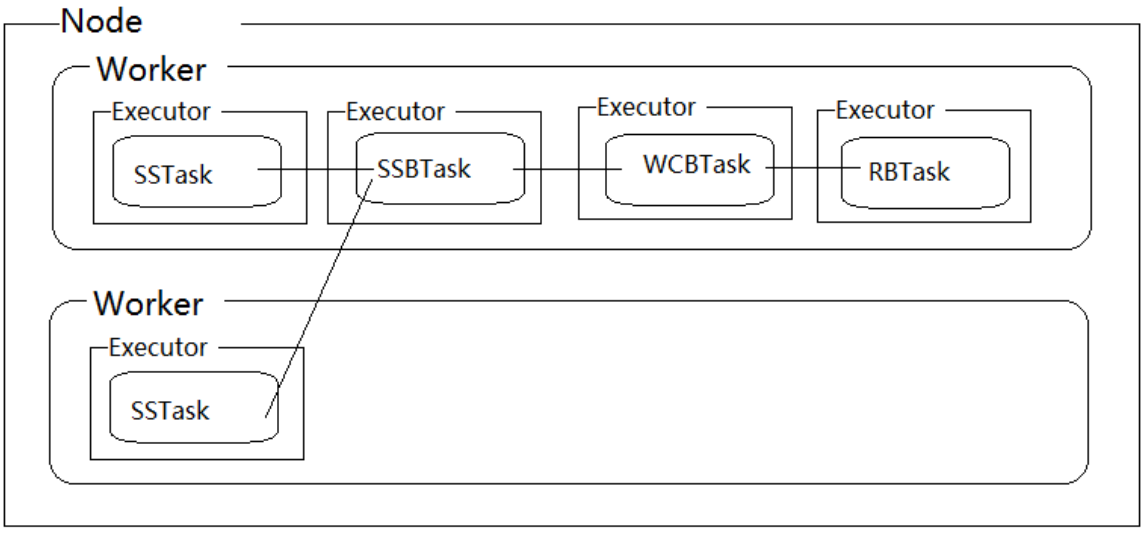
builder.setSpout(...).setNumTasks(2);

builder.setBolt(...).setNumTasks(2);

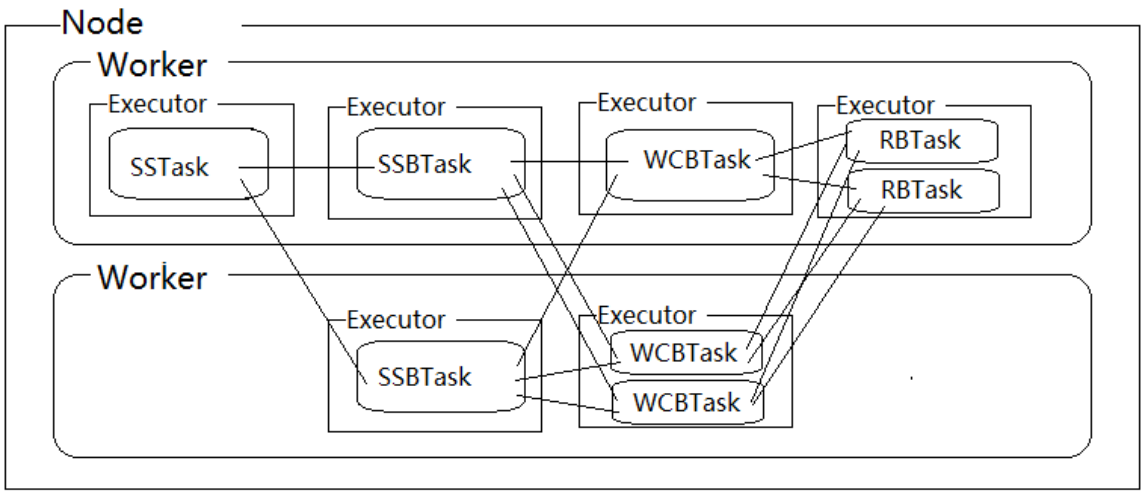
**指定的Task会在线程内部执行，如果指定的Task的数量多于线程的数量，则这些Task自动分配到这些线程内执行。

案例:

```
builder.setSpout("Sentence_Spout", spout,2);
builder.setBolt("Split_Sentence_Bolt", splitSentenceBolt);
builder.setBolt("Word_Count_Bolt", wordCountBolt);
builder.setBolt("Report_Bolt", reportBolt);
conf.setNumWorkers(2);
```



```
conf.setNumWorkers(2);
builder.setSpout("Sentence_Spout", spout);
builder.setBolt("Split_Sentence_Bolt", splitSentenceBolt,2)
builder.setBolt("Word_Count_Bolt", wordCountBolt,2).setNumTasks(3)
builder.setBolt("Report_Bolt", reportBolt).setNumTasks(2)
```




### 3. 数据流分组方式

在设定完并发度后，可以发现，组件向后发送tuple时，涉及到如何为后续组件的多个task分配这些tuple的问题，这是通过在连接组件时使用不同的方法，从而指定数据流分组方式来解决的。

storm内置了七种数据流分组方式：

1) Shuffle Grouping(随机分组)

   随机分发数据流中的tuple给bolt中的各个task，每个task接收到的tuple数量相同。

2) Fields Grouping(按字段分组)

   根据指定字段的值进行分组。

   指定字段具有相同值的tuple会路由到同一个bolt中的task中。

3) All Grouping(全复制分组)

   所有的tuple复制后分发给后续bolt的所有的task。

4) Globle Grouping(全局分组)

   这种分组方式将所有的tuple路由到唯一一个task上

   Storm按照最小task id来选取接受数据的task

   这种分组方式下配置bolt和task的并发度没有意义。

   这种方式会导致所有tuple都发送到一个JVM实例上，可能会引起Strom集群中某个JVM或者服务器出现性能瓶颈或崩溃。

5) None Grouping(不分组)

   在功能上和随机分组相同，为将来预留。

6) Direct Grouping(指向型分组)

   数据源会通过emitDirect()方法来判断一个tuple应该由哪个Strom组件来接受。

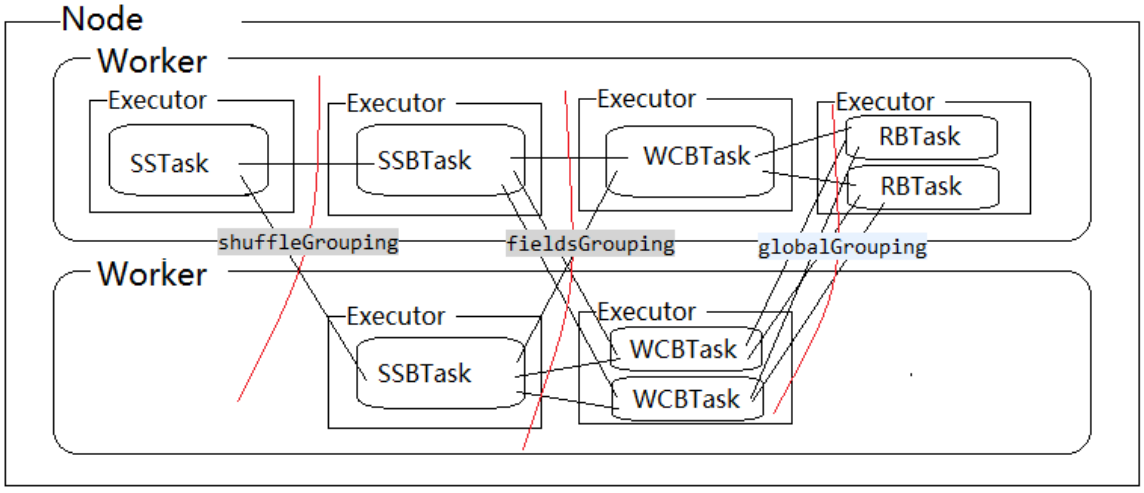
   只能在声明为指向型的数据流上使用。

7) Local or shuffle Grouping(本地或随机分组)

和随机分组类似，但是，会优先将tuple分发给同一个worker内的bolt的task，只有在同一个Worker内没有目标bolt的task的情况下采用随机分组方式。

这种方式可以减少网络传输，从而提高topology的性能。

案例：



# Storm可靠性保证

2018年9月13日 星期四　　下午 3:07

## 1. Storm可靠性概述

Storm是分布式程序，进程之间是通过网络来通信的，此时可能因为网络的不稳定、节点宕机等原因造成数据可能存在丢失的情况，这就给Storm的实时处理带来了数据丢失的可能，如何保障Storm处理数据时能够不丢数据 不多数据，使处理的结果就有可信度，这就是Storm可靠性相关的问题。

Storm中的可靠性分为三个级别：

至多一次 - 不多数据，但是可能会丢数据

至少一次 - 不丢数据，但是可能会多数据

恰好一次 - 不丢也不多，恰好完整的处理

这三种级别，越往下可靠性越高，但效率越低，越往上可靠性越低，但是效率越高，在真正的开发中应该根据业务需求，选择一个在满足的要求的可靠性的前提下，性能尽量好的可靠性级别。

## 2. 至多一次

之前WC案例中，没有任何并发控制，此时就是 至多一次的并发状态，不会多数据，但是可能会因为网络延迟 程序出错 造成数据的丢失。

## 3. 至少一次

在分布式的环境下，因为网络的不稳定性，数据丢失是无法彻底避免的，所以为了保证数据不丢失，应该检测数据丢失的情况，并重发丢失的数据。

这种方式下可以保证不丢数据，但是可能因为重发造成多数据。

a. 对于Spout：

i. 持有已发数据一段时间

持有已经发送的数据一段时间，以便于在数据丢失的情况下能够重发，直到该数据完成了整个Storm处理流程，才可以删除。

ii. 实现ack()和fail()

覆盖ack方法 和fail方法 ，storm会在某条数据完成整个处理环节过后，来调用ack方法，此时可以在ack方法中删除之前缓存的数据。而在某条数据处理失败时，storm会自动调用fail方法，此时可以在fail方法中实现重发的过程。

**void cn.tedu.storm.wc.SentenceSpout.ack(Object msgId)**

@Override

Storm has determined that the tuple emitted by this spout with the msgId identifier has been fully processed. Typically, an implementation of this method will take that message off the queue and prevent it from being replayed.

**Overrides:** ack(...) in BaseRichSpout

**Parameters:**

**msgId**

**void cn.tedu.storm.wc.SentenceSpout.fail(Object msgId)**

@Override

The tuple emitted by this spout with the msgId identifier has failed to be fully processed. Typically, an implementation of this method will put that message back on the queue to be replayed at a later time.

**Overrides:** fail(...) in BaseRichSpout

**Parameters:**

**msgId**

b. 对于Bolt：

i. 需要在发送tuple的过程中实现锚定。

所谓的锚定就是将父tuple和子tuple的派生关系记录在collector中，以便于在后续tuple出错时，可以通过collector中维系的锚定关系，来回溯找到最初的tuple来实现重复

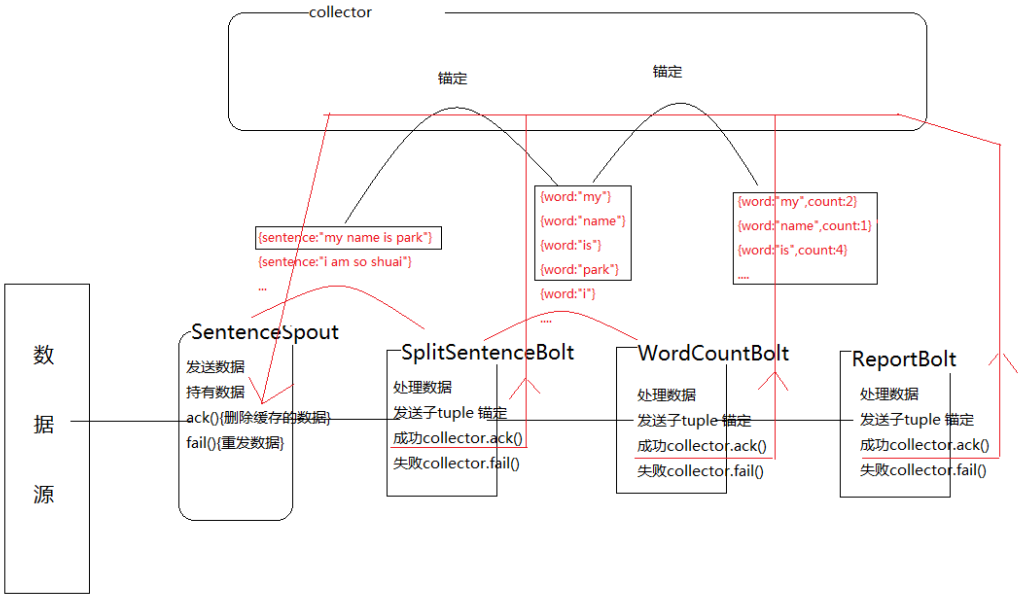
ii. 调用collector.ack() 或 collector.fail()

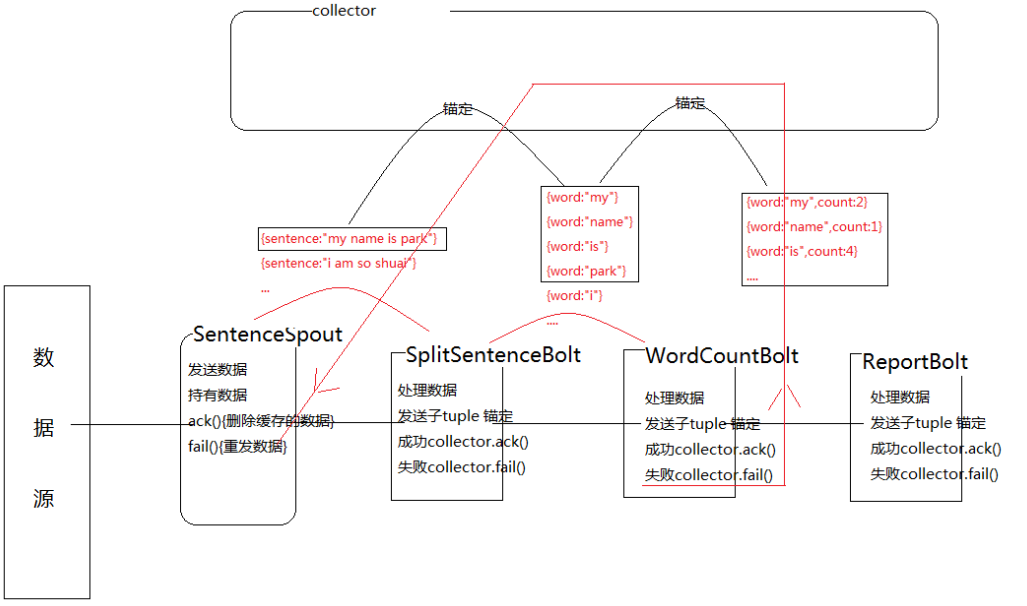
在整个bolt处理完一个tuple后调用collector提供的ack方法，或，在某个tuple处理过程中出错要调用collector的fail方法。

以此向collector告知某个tuple在某个bolt中的处理状态，实现对tuple的处理的监控。

当对于某一个spout发送的tuple，整个拓扑中的所有bolt都处理成功，调用的是ack，则collector认为该tuple完成了整个处理流程会自动调用spout的ack方法。

而任何一个bolt处理失败，调用的是fail方法，则collector认为该tuple处理失败，会自动调用spout的fail方法。




案例：参考今日代码

## 4. 恰好一次

在实现了至少一次语义之后，已经可以保证数据不丢，但是可能会多

想要实现恰好一次的语义，需要在以上的基础上，去除重复的数据

想要去除重复的数据，需要能够识别每条数据，并在处理过程中对已经处理过的数据的重发的数据进行抛弃

方案1：

想要唯一识别每条数据，可以为每条数据指定唯一的编号，这个编号是一个递增的数字，在重发时保持不变，通过这个编号唯一的识别tuple

后续处理过程中，记录所有已经处理过的tuple的编号，后续的tuple要处理之前，需要和这些已经处理过的tuple的编号做比较，如果发现已经处理过，则抛弃，如果没有处理过就正常处理

此方案，需要记录所有已经处理过的tuple的编号，且由于在任何时间点都无法确定之后会不会有指定tuple的重发，这些编号都不能删除，这样一来，随着storm的运行，编号越记越多，浪费大量存储空间，且，比较的速度越来越慢。在空间和时间上都无法接收。此方法不可行。

方案2：

想要解决方案1的问题，可以让所有的tuple的编号严格按照顺序递增，且要求整个topology严格按照顺序处理这些tuple，则只需要记录最后一个处理过的tuple的编号即可，后续再有tuple过来，如果编号小于等于记录中的tuple编号，则说明是之前已经处理过的tuple重发的数据，抛弃该数据。如果编号大于记录中的编号，则是正常数据，进行处理并更新记录中的最后一次处理的tuple编号即可。

此方案，需要严格保证tuple处理的顺序，可以要求所有的tuple依次处理，一个tuple处理时，其他tuple排队等待，可以保证顺序，但是效率非常低下。此方案不可行。

方案3：

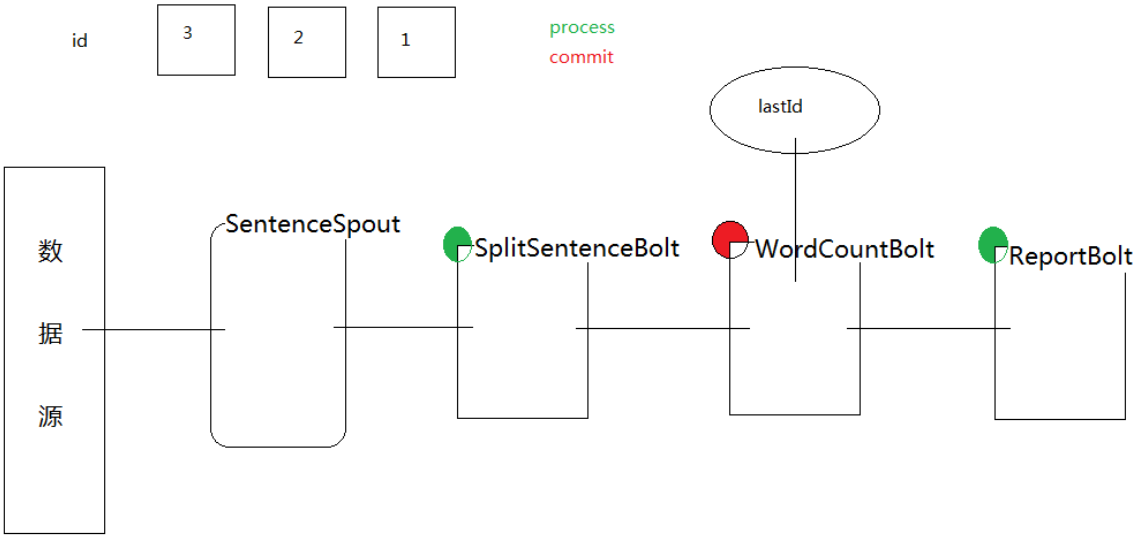
想要解决方案2的问题，则要求每次发一个批的数据，一个批中有若干tuple，每个批都被赋予一个递增的编号，批中任何一个tuple出错，整批重发，重发过程中编号不变。所有的批依次处理，一个批处理的过程中，其他批排队等待，保证顺序，批之间串行，但批内部有很多tuple，是可以并发的，这种方案，比起方案2，效率有了一定提升。

此方案，虽然比方案2效率提升了，但是批和批之间仍然串行处理，性能依然受限。

方案4：

通过分析发现，在整个Topology中，并不是所有的Bolt都需要严格保证一次且一次的语义，只要抓住最关键的一个或几个步骤，保证一次且一次的语义，就可以保障数据计算不出问题，基于这样的分析，可以将整个Topology中的Bolt分为Process阶段 和 Commit阶段两类。批在Porcess阶段的Bolt中随意并发保证性能，但一旦遇到Commit阶段的Bolt，则要严格按照顺序进入，保证顺序，再在Commit阶段的Bolt中记录最后一个处理过的批的编号，和后续批的编号做比较，进行去重操作，就可以保证一次且一次的语义。

此方案，即保证了一次且一次的语义，也在最大程度上，保证了程序的并发效率，Storm采用的正是这个方案。


id
3
2
1
process
commit
lastId
数
据
源
SentenceSpout
SplitSentenceBolt
WordCountBolt
ReportBolt

# Storm事务型拓扑

2018年9月14日 星期五　　上午 11:53

## 1. 事务性拓扑

根据之前分析的恰好一次语义的实现方式，storm设计了事务型拓扑API来实现恰好一次的语义，即TransactionTopology API。

在TransactionTopology API中 存在TransactionSpout 和 TransactionBolt 来实现事务型拓扑。

## 2. TransactionSpout的开发

### a. BaseTransactionalSpout

写一个类继承BaseTransactionalSpout<metadata>

其中包含三个方法：

**Coordinator<Object> cn.tedu.storm.wc3.TransSentenceSpout.getCoordinator(Map conf, TopologyContext context)**

@Override

The coordinator for a TransactionalSpout runs in a single thread and indicates when batches of tuples should be emitted and w hen transactions should commit. The Coordinator that you provide in a TransactionalSpout provides metadata for each transaction s o that the transactions can be replayed.

**Specified by:** getCoordinator(...) in ITransactionalSpout

**Parameters:**

**conf**

**context**

**Emitter<Object> cn.tedu.storm.wc3.TransSentenceSpout.getEmitter(Map conf, TopologyContext context)**

@Override

The emitter for a TransactionalSpout runs as many tasks across the cluster. Emitters are responsible for emitting batches of tuples for a transaction and must ensure that the same batch of tuples is always emitted for the same transaction id.

**Specified by:** getEmitter(...) in ITransactionalSpout

**Parameters:**

**conf**

**context**

**void cn.tedu.storm.wc3.TransSentenceSpout.declareOutputFields(OutputFieldsDeclarer declarer)**

@Override

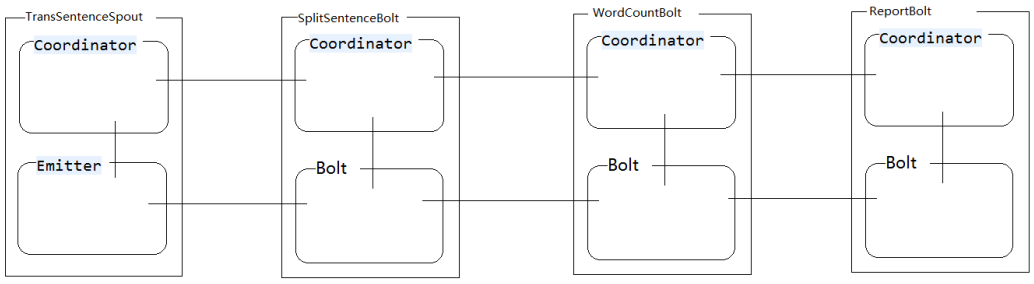
Declare the output schema for all the streams of this topology.

**Specified by:** declareOutputFields(...) in IComponent

**Parameters:**

**declarer** this is used to declare output stream ids, output fields, and whether or not each output stream is a direct stream

其中：




Coordinator：

组织批的元数据信息，将批的元数据信息交给Emitter真正的发送批

联系下游的Coordinator，沟通收发数据的信息，如果一致则发送批

结束，如果不一致，则说明丢失数据，进行重发。

Emitter：

根据Coordinator指定的元数据，真正的组织一个批进行发送

## b. Coordinator

**boolean cn.tedu.storm.wc3.SentenceCoordinator.isReady()**

@Override

Returns true if its ok to emit start a new transaction, false otherwise (will skip this transaction). You should sleep here if you want a delay between asking for the next transaction (this will be called repeatedly in a loop).

**Specified by:** isReady() in Coordinator

**Object cn.tedu.storm.wc3.SentenceCoordinator.initializeTransaction(BigInteger txid, Object prevMetadata)**

@Override

Create metadata for this particular transaction id which has never been emitted before. The metadata should contain whatever is necessary to be able to replay the exact batch for the transaction at a later point. The metadata is stored in Zookeeper. Storm uses the Kryo serializations configured in the component configuration for this spout to serialize and deserialize the metadata.

**Specified by:** initializeTransaction(...) in Coordinator

**Parameters:**

**txid** The id of the transaction.

**prevMetadata** The metadata of the previous transaction

**Returns:**

the metadata for this new transaction

**void cn.tedu.storm.wc3.SentenceCoordinator.close()**

@Override

Release any resources from this coordinator.

**Specified by:** close() in Coordinator

## c. Emitter：

**void cn.tedu.storm.wc3.SentenceEmitter.emitBatch(TransactionAttempt tx, SentenceMetaData coordinatorMeta, BatchOutputCollector collector)**

@Override

Emit a batch for the specified transaction attempt and metadata for the transaction. The metadata was created by the Coordinator in the initializeTranaction method. This method must always emit the same batch of tuples across all tasks for the same transaction id. The first field of all emitted tuples must contain the provided TransactionAttempt.

**Specified by:** emitBatch(...) in Emitter

**Parameters:**

**tx**

**coordinatorMeta**

**collector**

**void cn.tedu.storm.wc3.SentenceEmitter.cleanupBefore(BigInteger txid)**

@Override

Any state for transactions prior to the provided transaction id can be safely cleaned up, so this method should clean up that state.

**Specified by:** cleanupBefore(...) in Emitter

**Parameters:**

**txid**

**void cn.tedu.storm.wc3.SentenceEmitter.close()**

@Override

Release any resources held by this emitter.

**Specified by:** close() in Emitter

## d. 代码：

```
package cn.tedu.storm.wc3;


public class SentenceDB {
        private static String [] sentence = {
                "my name is park",
                "i am so shuai",
                "do you like me",
                "are you sure you do not like me",
                "ok i am sure",
                "how are you",
                "fine thank you and you",
                "i am ok"
        };
        private SentenceDB() {
        }

        public static String [] getData(){
                return sentence;
        }
}
```

```
package cn.tedu.storm.wc3;


import java.io.Serializable;


public class SentenceMetaData implements Serializable {
        private int begin;
        private int end;

        public SentenceMetaData() {
        }


        public SentenceMetaData(int begin, int end) {
                this.begin = begin;
                this.end = end;
        }


        public int getBegin() {
                return begin;
        }
        public void setBegin(int begin) {
                this.begin = begin;
        }
        public int getEnd() {
                return end;
        }
        public void setEnd(int end) {
                this.end = end;
        }


}
```

```
package cn.tedu.storm.wc3;


import java.math.BigInteger;


import backtype.storm.transactional.ITransactionalSpout.Coordinator;


public class SentenceCoordinator implements Coordinator<SentenceMetaData> {
        //--批大小
        private int batchSize = 3;
        //--是否有新的批
        private boolean hasMoreBatch = true;
        /**
         * 当isReady方法返回true时，storm回来调用当前方法，要求真的组织一个批的元数据返回
         * 这个元数据对象中应该包含任何可能需要的信息，必须保证能够基于这些信息组织一个批来发送，
         * 且在批出问题时，应该可以利用这个元数据对象重新组织出一模一样的批出来
         */
        @Override
```

```
        public SentenceMetaData initializeTransaction(BigInteger txid, SentenceMetaData prevMetadata) {
                //--begin = 是第一个批吗 ? 0 : 上一个批的结尾
                int begin = prevMetadata == null
                                ? 0 : prevMetadata.getEnd();
                //--end = begin + 批大小 < 数组长度 ? begin+批大小 : 数组的结尾
                int end = begin+batchSize < SentenceDB.getData().length
                                ? begin+batchSize : SentenceDB.getData().length;
                //--是否还有新的批 = end 是否等于 数组的结尾
                hasMoreBatch = end != SentenceDB.getData().length;
                //--将begin和end 组织为SentenceMetaData返回
                return new SentenceMetaData(begin, end);
        }

        /**
         * storm不停的在一个单一循环中调用此方法，询问是否准备好发送一个新的批
         * 如果准备好了发送一个新的批，就返回true,否则返回false
         * 也可以在此方法中睡眠一段事件，以便于在一段时间后发送当前批
         */
        @Override
        public boolean isReady() {
                return hasMoreBatch;
        }

        /**
         * 当前Coordinator被销毁之前调用
         * 可以用来释放一些资源
         */
        @Override
        public void close() {
        }

}
```

```
package cn.tedu.storm.wc3;

import java.math.BigInteger;

import backtype.storm.coordination.BatchOutputCollector;
import backtype.storm.transactional.ITransactionalSpout.Emitter;
import backtype.storm.transactional.TransactionAttempt;
import backtype.storm.tuple.Values;

public class SentenceEmitter implements Emitter<SentenceMetaData> {

        /**
         * 根据传入的事务编号 和 元数据对象 组织一个批发送
         * 要保证对于同一个批的编号,要保证发送的数据必须一模一样
         * ,为了保证这一点,通常需要在Emitter内部保留已经发送的批的数据一段时间
         * ,直到批完成了整个处理才可以删除
         * 要注意,所有发送的tuple第一个字段要留给批编号
         * tx:storm出入的批的编号
         * coordinatorMeta:批的元数据信息,来自于Coordinator
         * collector:用来发送tuple的对象
         */
        @Override
        public void emitBatch(TransactionAttempt tx, SentenceMetaData coordinatorMeta,
BatchOutputCollector collector) {
                int begin = coordinatorMeta.getBegin();
                int end = coordinatorMeta.getEnd();
                for(int i = begin;i<end;i++){
                        String sentence = SentenceDB.getData()[i];
                        collector.emit(new Values(tx,sentence));
                }
        }

        /**
         * 当一个批完成了整个处理流程 此方法会被调用
         * 主要在这个方法中 清理之前缓存的批相关的数据
         */
        @Override
        public void cleanupBefore(BigInteger txid) {

        }

        /**
         * 在当前组件销毁之前会调用的方法
```

```
         * 主要用来释放资源
         */
        @Override
        public void close() {

        }


}
```

```
package cn.tedu.storm.wc3;


import java.util.Map;


import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseTransactionalSpout;
import backtype.storm.tuple.Fields;


public class TransSentenceSpout extends BaseTransactionalSpout<SentenceMetaData> {


      @Override
      public Coordinator<SentenceMetaData> getCoordinator(Map conf,TopologyContext context) {
            return new SentenceCoordinator();
      }


      @Override
      public Emitter<SentenceMetaData> getEmitter(Map conf,TopologyContext context) {
            return new SentenceEmitter();
      }


      @Override
      public void declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields("txid","sentence"));
      }


}
```