Storm 笔记

试想一下你自己实现实时计算的方案？

数据源务必实时，所以采用Message Queue作为数据源，消息处理Comsumer实时从MQ获取数据进行处理，返回结果到Web或写DB。

这种方式有以下几个缺陷：

1、单机模式，能处理的数据量有限

2、不健壮，服务器挂掉即结束。而Storm集群节点挂掉后，任务会重新分配给其他节点，作业不受影响。

3、失败重试、事务等，你需要在代码上进行控制，过多精力放在业务开发之外。

4、伸缩性差： 当一个消息处理者的消息量达到阀值，你需要对这些数据进行分流， 你需要配置这些新的处理者以让他们处理分流的消息。

Storm的一些关键特性

1. **适用场景广泛**： storm可以实时处理消息和更新DB，对一个数据量进行持续的查询并返回客户端（持续计算），对一个耗资源的查询作实时并行化的处理(分布式方法调用，即DRPC），storm的这些基础API可以满足大量的场景。
2. **可伸缩性高**:  Storm的可伸缩性可以让storm每秒可以处理的消息量达到很高。扩展一个实时计算任务，你所需要做的就是加机器并且提高这个计算任务的并行度 。Storm使用ZooKeeper来协调集群内的各种配置使得Storm的集群可以很容易的扩展。

3. **保证无数据丢失**： 实时系统必须保证所有的数据被成功的处理。 那些会丢失数据的系统的适用场景非常窄， 而storm保证每一条消息都会被处理， 这一点和S4相比有巨大的反差。

4. **异常健壮**： storm集群非常容易管理，轮流重启节点不影响应用。

5. **容错性好**：在消息处理过程中出现异常， storm会进行重试

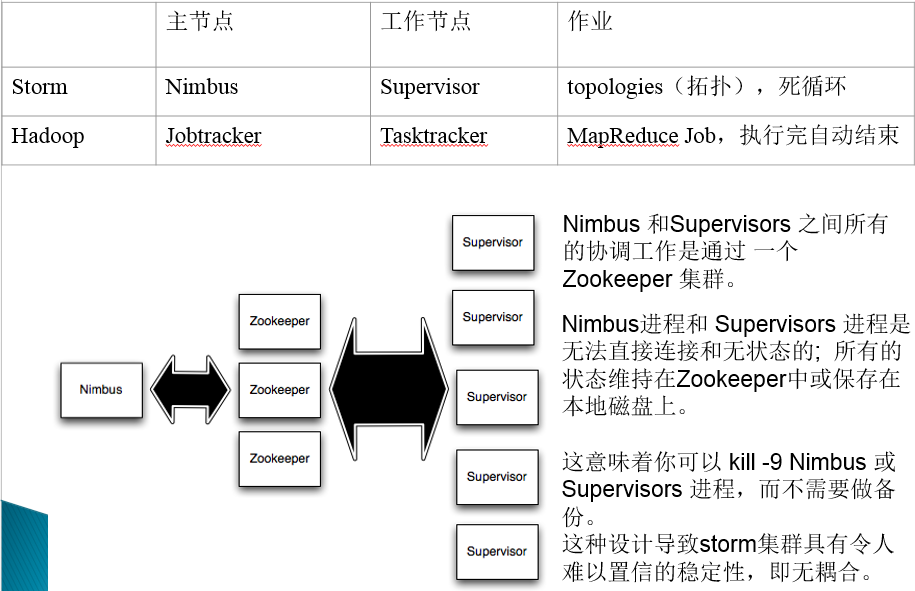
6. **语言无关性**： Storm的topology和消息处理组件(Bolt)可以用任何语言来定义， 这一点使得任何人都可以使用storm.

# Storm原理和概念详解

## Storm 集群架构

Nimbus（主节点，master）和Supervisor（工作节点，slave）之间的所有协调工作都是通过一个Zookeeper集群来完成。

**并且，nimbus进程和supervisor都是快速失败（fail-fast)和无状态的。所有的状态要么在Zookeeper里面， 要么在本地磁盘上。**



当storm 集群需要重启的时候，可以逐台机器进行重启，以保证数据不会丢失。

## 工作原理

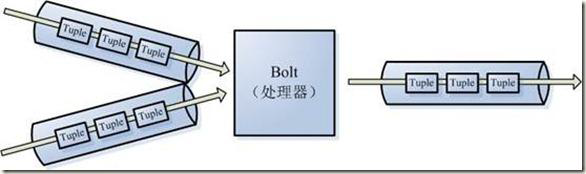
### Nimbus

负责在集群分发的代码，topo只能在nimbus机器上提交，将任务分配给其他机器，和故障监测。

### Supervisor

监听分配给它的节点，根据Nimbus 的委派在必要时启动和关闭工作进程。 每个工作进程执行topology 的一个子集。一个运行中的topology 由很多运行在很多机器上的工作进程组成。

在Storm中有对于流stream的抽象，流是一个不间断的无界的连续tuple，注意Storm在建模事件流时，把流中的事件抽象为tuple即元组

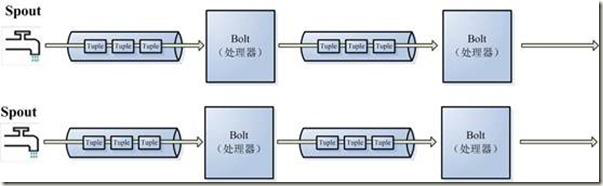


### Spout And Bolt

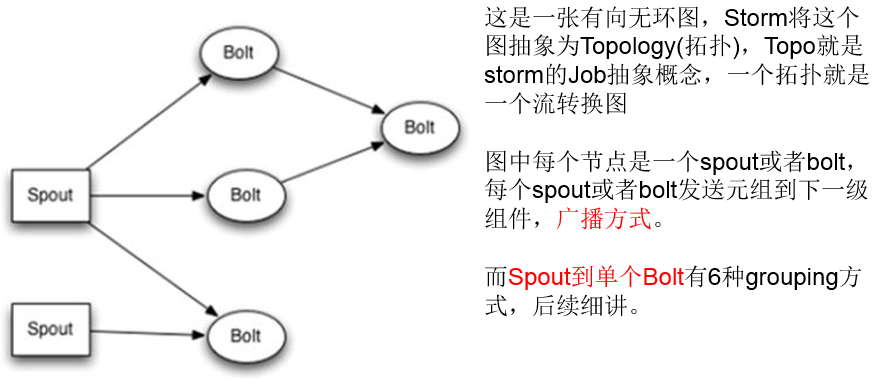
Storm认为每个stream都有一个源，也就是原始元组的源头，叫做Spout（管口）

处理stream内的tuple，抽象为Bolt，bolt可以消费任意数量的输入流，只要将流方向导向该bolt，同时它也可以发送新的流给其他bolt使用，这样一来，只要打开特定的spout再将spout中流出的tuple导向特定的bolt，又bolt对导入的流做处理后再导向其他bolt或者目的地。

可以认为spout就是水龙头，并且每个水龙头里流出的水是不同的，我们想拿到哪种水就拧开哪个水龙头，然后使用管道将水龙头的水导向到一个水处理器（bolt），水处理器处理后再使用管道导向另一个处理器或者存入容器中。

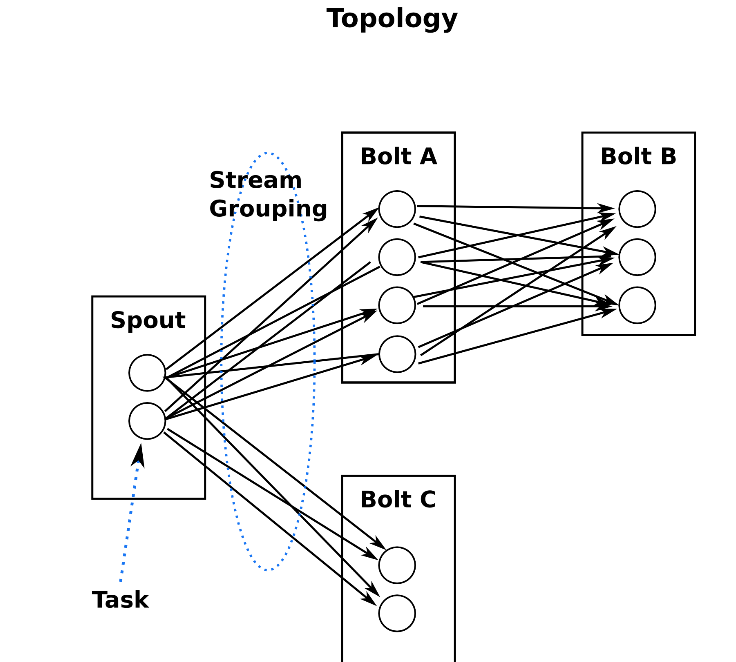


为了增大水处理效率，我们很自然就想到在同个水源处接上多个水龙头并使用多个水处理器，这样就可以提高效率。



### Topology

Storm将流中元素抽象为tuple，**一个tuple就是一个值列表value list**，list中的每个value都有一个name，并且该value可以是任意可序列化的类型。拓扑的每个节点都要说明它所发射出的元组的字段的name，其他节点只需要订阅该name就可以接收处理。



#### Streams：消息流

消息流是一个没有边界的tuple序列，而这些tuples会被以一种分布式的方式并行创建和处理。 每个tuple可以包含多列，字段类型可以是： integer, long, short, byte, string, double, float, boolean和byte array。 你还可以自定义类型 — 只要你实现对应的序列化器。

#### Spouts：消息源

Spouts是topology消息生产者。Spout从一个外部源(消息队列)读取数据向topology发出tuple。 **消息源Spouts可以是可靠的也可以是不可靠的**。一个可靠的消息源可以重新发射一个处理失败的tuple， 一个不可靠的消息源Spouts不会。

Spout类的方法nextTuple不断发射tuple到topology，storm在检测到一个tuple被整个topology成功处理的时候调用ack, 否则调用fail。

storm只对可靠的spout调用ack和fail。

#### Bolts：消息处理者

消息处理逻辑被封装在bolts里面，Bolts可以做很多事情： 过滤， 聚合， 查询数据库等。

Bolts可以简单的做消息流的传递。复杂的消息流处理往往需要很多步骤， 从而也就需要经过很多Bolts。第一级Bolt的输出可以作为下一级Bolt的输入。而Spout不能有下一级。

Bolts的主要方法是execute（死循环）连续处理传入的tuple，成功处理完每一个tuple调用OutputCollector的ack方法，以通知storm这个tuple被处理完成了。当处理失败时，可以调fail方法通知Spout端可以重新发送该tuple。

流程是： Bolts处理一个输入tuple, 然后调用ack通知storm自己已经处理过这个tuple了。storm提供了一个IBasicBolt会自动调用ack。

Bolts使用OutputCollector来发射tuple到下一级Blot。

# Storm安装部署与测试运行

官网：

http://storm.apache.org/

并不是hadoop生态系统的一个框架

版本：0.9.6

http://storm.apache.org/2015/11/05/storm096-released.html

<http://storm.apache.org/downloads.html>

http://storm.apache.org/releases/0.9.6/index.html

## 安装

### 下载安装包，上传服务器

apache-storm-0.9.6.tar.gz

### 安装要求、前提

zookeeper集群 python 2.6.6以上

### 解压安装

tar -zxvf apache-storm-0.9.6.tar.gz -C /opt/modules/

### 进入解压路径，修改配置文件

conf目录下 ： storm\_env.ini storm.yaml

storm\_evn.ini ：JAVA\_HOME:/opt/modules/jdk1.7.0\_67

storm.yaml: yaml 有特殊的文件格式要求

storm.zookeeper.servers:

- "hive-stu.ibeifeng.com"

nimbus.host: "nimbus" nimbus 主节点

storm.local.dir: " /usr/local/storm/workspace "

参数指定storm本地文件存放目录，存放任务jar的目录

supervisor.slots.ports:

- 6700

- 6701

- 6702

- 6703

指定supervisor节点启动一些工作进程 worker 所使用的默认端口

supervisor storm的从节点

ui.port ： storm web监控页面

**storm.zookeeper.servers：**这是一个为Storm集群配置的Zookeeper集群的主机列表

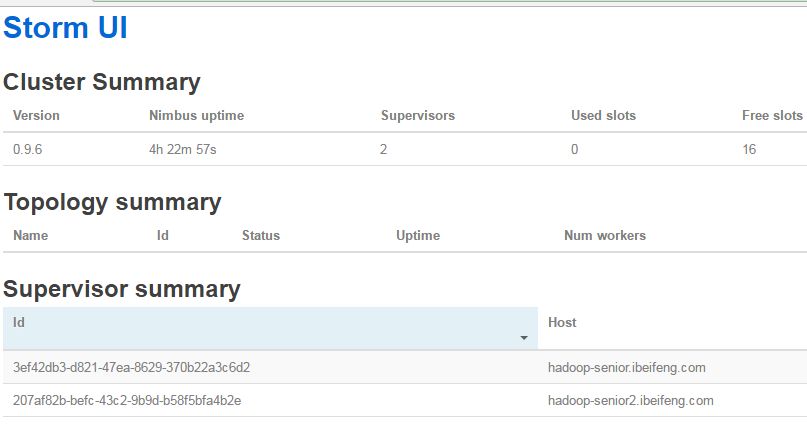
**storm.local.dir：**Nimbus和Supervisor守护程序需要一个本地磁盘目录存储小量状态（像jars，confs，其它），每台机器都创建这些目录，赋可写权限

**java.library.path：**这是Storm使用的本地库（ZeroMQ和JZMQ）载入路径。大多数安装，默认路径"/usr/local/lib:/opt/local/lib:/usr/lib"就行，所以你可能不需要配置它。

**supervisor.slots.ports:**每一台worker机器，你用这个配置来指定多少workers运行在那台机。每个worker使用单一端口接收消息，并且这个设置定义哪个端口是打开的且可以使用。如果你定义5个端口，那么Storm将在这台机分配5个worker运行。

### 测试运行

web 界面； <http://hadoop-senior.ibeifeng.com:9081/index.html>



### 启动Nimbus进程：

需要在配置文件中nimbus.host该参数指定的服务器上启动

\*\*\*\*\*bin/storm nimbus\*\*\*\*

$ nohup bin/storm nimbus > /dev/null 2>&1 &

jps

4309 config\_value 表示正在启动读取配置文件中

4184 nimbus

$ ps -ef | grep daemon.nimbus

### 启动Supervisor进程：

$ nohup bin/storm supervisor > /dev/null 2>&1 &

jps

4472 config\_value

4441 supervisor

$ ps -ef | grep daemon.supervisor

### 启动ui进程：一定是在Nimbus节点上启动

$ nohup bin/storm ui > /dev/null 2>&1 &

4593 config\_value

4592 core

$ ps -ef | grep ui.core

端口：$ netstat -tlnup | grep 8081

http://hive-stu.ibeifeng.com:8081/index.html

### 启动logviewer日志收集进程：

一定要在每个supervisor节点上启动

$ nohup bin/storm logviewer > /dev/null 2>&1 &

4718 config\_value

4717 logviewer

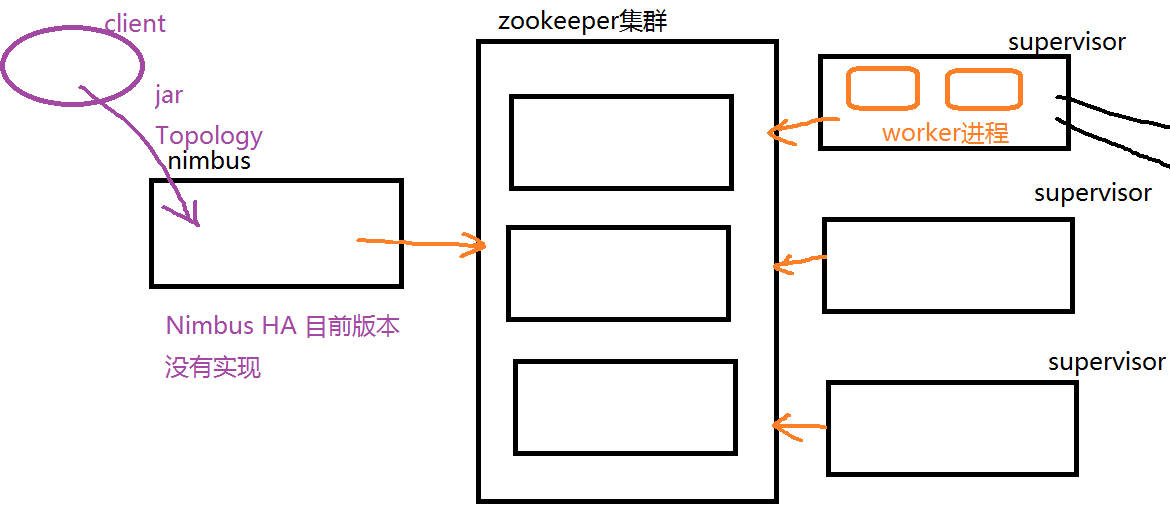
$ ps -ef | grep daemon.logviewer

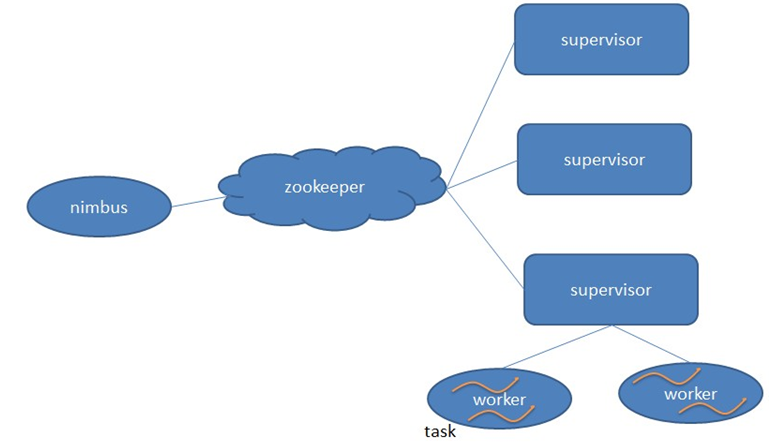
使用案例wordcount测试：

$ bin/storm jar examples/storm-starter/storm-starter-topologies-0.9.6.jar storm.starter.WordCountTopology wordcount

Storm编程模型

## Storm的安装部署架构





#### nimbus主节点：

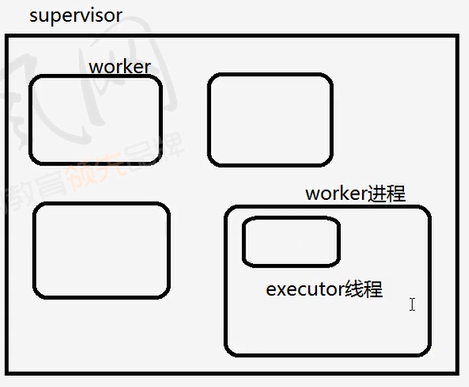
1）接收客户端提交的任务请求，任务由Nimbus进行分配，将分配信息提交到

Zookeeper集群（在Zookeeper相应的znode节点上写入任务分配信息，由supervisor查看这些znode上的任务分配信息，获取分配到的任务）

2）监控整个集群的状态（从Zookeeper集群中相应znode上读取supervisor、worker进程的状态信息数据）

3）容错：当任务在某些Supervisor节点上运行的时候由于Supervisor进程失效，重新将这些任务分配给其他supervisor运行

#### Supervisor：



1）需要定时将自己的运行状态信息（心跳信息）汇报给zookeeper（在Zookeeper相应的znode节点上写入心跳信息）

2）接收Nimbus分配给它的任务，负责启动、停止工作进程worker，其本身并不是执行任务的工作进程，worker的容错由Supervisor进程负责

#### worker进程：

并不是常驻进程，不能通过手动启动 真正执行任务的进程

worker进程启动后，也会定时将状态信息汇报给zookeeper

#### executor 线程：

worker进程来启动，executor负责执行客户提交到Storm集群上任务中Task(spout/bolt)

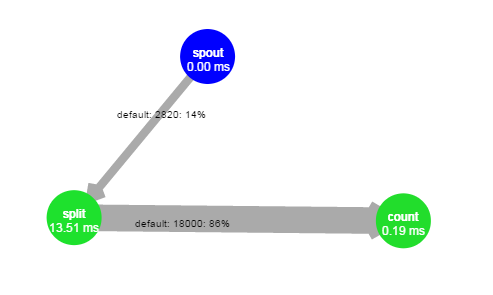
## Storm的任务

### 1、杀掉任务

$ bin/storm kill wordcount

### Topology : 提交到Storm集群上的任务

拓扑图 有向无环图 DAG



比较Storm与MapReduce：

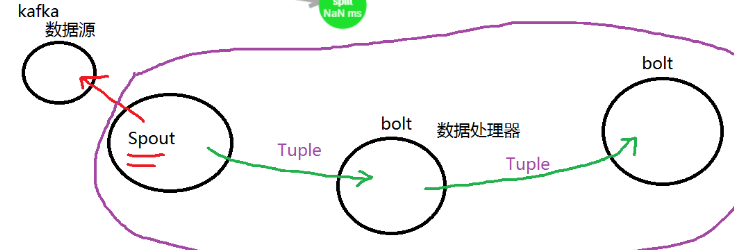
1）Topology 提交到Storm集群上运行，除非手动执行kill命令，否则将一直永远运行下

MapReduce任务把数据处理完就终止

2）实时处理 批处理

地铁站：自动扶梯 普通电梯

Topology：

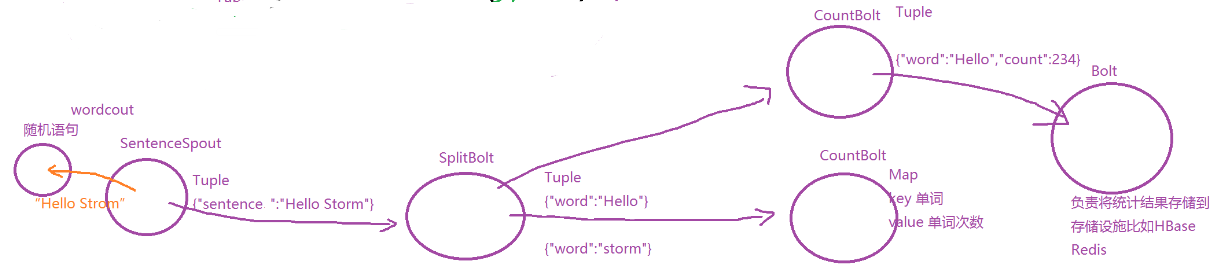


spout 数据采集器 ，由它负责从数据源上获取数据，转发给后面的bolt进行处理

bolt：数据处理器，在bolt里面实现数据的处理逻辑

stream：Tuple格式 keyvalue对的集合

{"name":"zhangsan","sex","M","age":26}



## zookeeper在Storm中的作用

znode

/storm

/workerbeats worker工作进程的状态信息

/errors topology在运行过程中出现异常的task信息，方便Nimbus将运行出错的任务进行重新分配

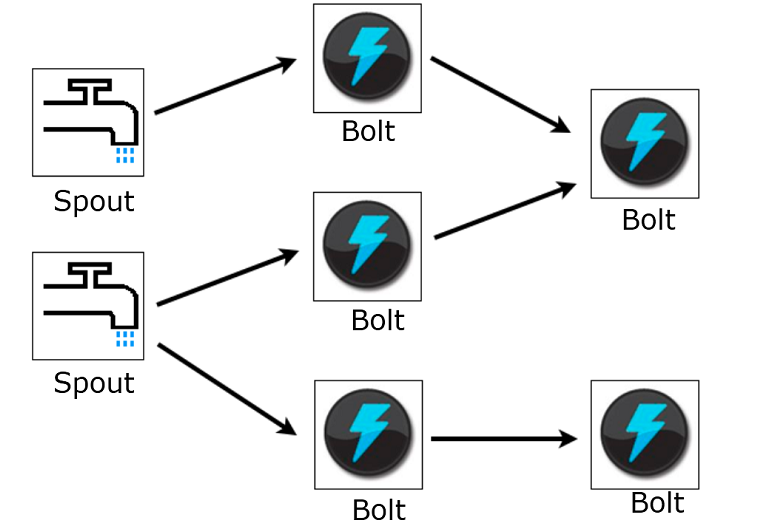
/supervisors Supervisor节点的状态信息

/storms Topology的基本配置信息

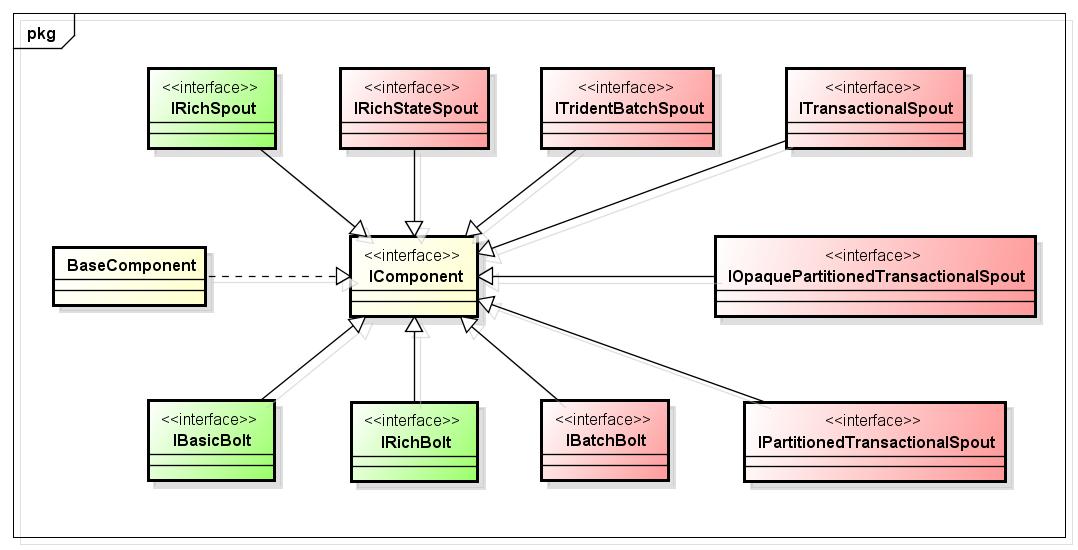
/assignments Topology任务的分配信息

/storm/workerbeats/wordcount-3-1471753943/各个worker对应的znode

# Storm编程模型

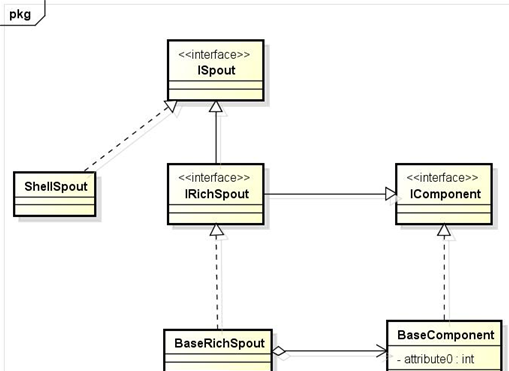


## API介绍



绿色是我们最常用的，红色是与事务相关的

BaseComponent 是Storm提供的“偷懒”的类，它及其子类都或多或少的实现了其接口定义的部分方法。这样我们在用的时候，而不是自己每次都写所有的方法。



Open()是初始化方法

nextTuple()循环发射数据

ack() 成功处理tuple回调方法

Fail()处理失败tuple回调方法

activate和deactivate ：spout可以被暂时激活和关闭

close方法在该spout关闭前执行，

但是并不能得到保证其一定被执行，kill -9时不执行，

Storm kill {topoName} 时执行

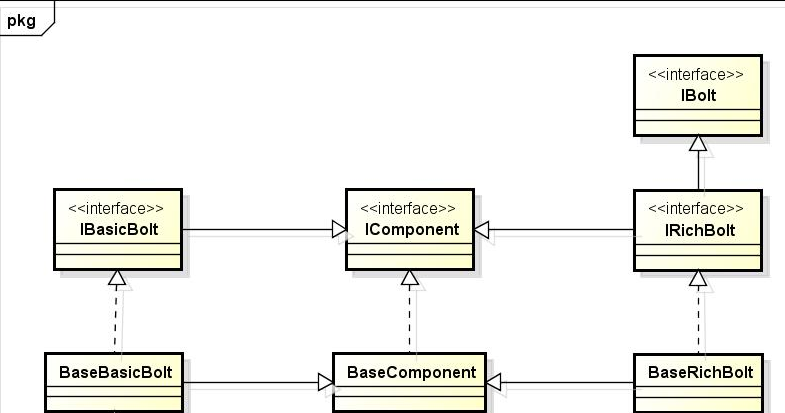
### Spout



原则：通常情况下（Shell和事务型的除外），实现一个Spout，可以直接实现接口**IRichSpout**，如果不想写多余的代码，可以直接继承BaseRichSpout。

Sport tail 特性

### Bolt





prepare方法进行初始化，传入当前执行的上下文

execute接受一个tuple进行处理，也可emit数据到下一级组件

cleanup 同ISpout的close方法，在关闭前调用，不保证其一定执行。

IBolt继承了Serializable，我们在nimbus上提交了topology以后，创建出来的bolt会序列化后发送到具体执行的worker(工作进程)上去。worker在执行该Bolt时，会先调用prepare方法传入当前执行的上下文.

execute接受一个tuple进行处理，并用prepare方法传入的OutputCollector的ack方法（表示成功）或fail（表示失败）来反馈处理结果.还可以通过OutputCollector的emit方法把结果发射到下一级组件。

IBasicBolt接口，实现该接口的Bolt不用在代码中提供反馈结果了，Storm内部会自动反馈成功。如果你确实要反馈失败，可以抛出FailedException。

（后续章节讲到的Trident均不用显性调用ack和fail，框架会自动调。）

*实现一个Bolt，可以实现IRichBolt接口或继承BaseRichBolt，如果不想自己处理结果反馈，可以实现 IBasicBolt接口或继承BaseBasicBolt，它实际上相当于自动做了prepare方法和 collector.emit.ack(inputTuple)。*

## stream grouping

用来定义一个stream应该如何分配给Bolts上面的多个Tasks。

storm里面有6种类型的stream grouping:

### Shuffle Grouping:

随机分组， 随机派发stream里面的tuple， 保证每个bolt接收到的tuple数目相同。轮询，平均分配。

### Fields Grouping：

按字段分组， 比如按userid来分组， 具有同样userid的tuple会被分到相同的Bolts， 而不同的userid则会被分配到不同的Bolts。

### All Grouping：

广播发送， 对于每一个tuple， 所有的Bolts都会收到。

### Global Grouping:

全局分组， 这个tuple被分配到storm中的一个bolt的其中一个task。再具体一点就是分配给id值最低的那个task。

### Non Grouping:

不分组， 这个分组的意思是说stream不关心到底谁会收到它的tuple。目前这种分组和Shuffle grouping是一样的效果，不平均分配。

### Direct Grouping:

直接分组， 这是一种比较特别的分组方法，用这种分组意味着消息的发送者决定由消息接收者的哪个task处理这个消息。 只有被声明为Direct Stream的消息流可以声明这种分组方法。而且这种消息tuple必须使用emitDirect方法来发射。消息处理者可以通过TopologyContext来或者处理它的消息的taskid (OutputCollector.emit方法也会返回taskid)

stream grouping就是用来定义一个stream应该如果**分配给Bolts上面的多个**

**Executors(多线程，并发度)**

注：不是一个spout或bolt emit到多个bolt（广播方式）。

storm里面有6种类型的stream grouping。

**单线程下均等同于All Grouping**

1. Shuffle Grouping

轮询，平均分配。随机派发stream里面的tuple，保证每个bolt接收到的tuple数目相同。

2. Non Grouping: 无分组， 这种分组和Shuffle grouping是一样的效果，多线程下不平均分配。

3. Fields Grouping：按Field分组，比如按word来分组， 具有同样word的tuple会被分到相同的Bolts， 而不同的word则会被分配到不同的Bolts。

作用：1、过滤，从源端（Spout或上一级Bolt）多输出Fields中选择某些Field

2、相同的tuple会分发给同一个Executer或task处理

典型场景： 去重操作、Join

## 并发度

一个Topology可以包含一个或多个**worker**(并行的跑在不同的machine上), 所以worker process就是执行一个topology的子集, 并且worker只能对应于一个topology

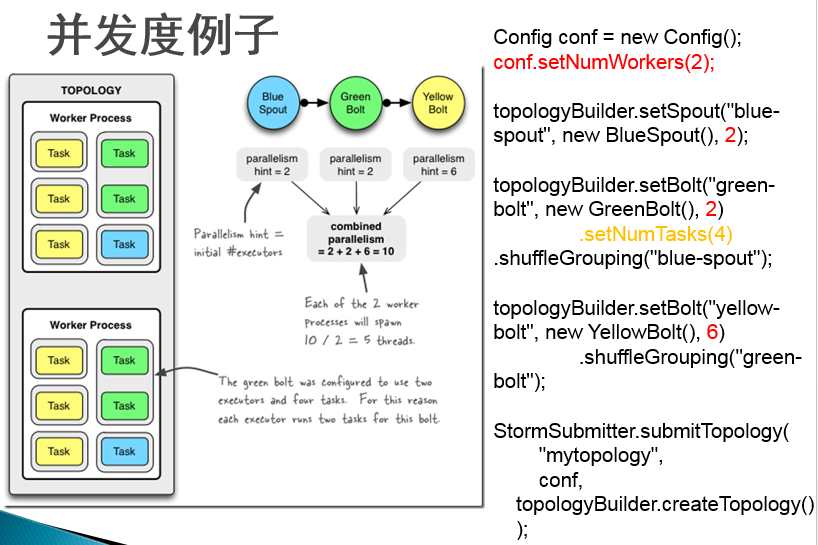
一个worker可用包含一个或多个**executor**, 每个component (spout或bolt)至少对应于一个executor, 所以可以说executor执行一个compenent的子集, 同时一个executor只能对应于一个component

**Task**就是具体的处理逻辑对象, 一个executor线程可以执行一个或多个tasks

但一般默认每个executor只执行一个task, 所以我们往往认为task就是执行线程, 其实不是。

task代表最大并发度, 一个component的task数是不会改变的, 但是一个componet的executer数目是会发生变化的（storm rebalance命令），

task数>=executor数, executor数代表实际并发数



通过setBolt和setSpout一共定义2+2+6=10个executor threads   
并且同setNumWorkers设置2个workers, 所以storm会平均在每个worker上run 5个executors   
而对于green-bolt, 定义了4个tasks, 所以每个executor中有2个tasks

场景分析：

单线程下：加减乘除，和任何处理类Operate，汇总

多线程下：

1、局部加减乘除

2、做处理类Operate，如split

3、持久化，如入DB

以WordCountTopology.java 为例讲解

**思考题：如何计算：word总数和word个数 ？并且在高并发下完成**

**前者是总行数，后者是去重word个数**

**类似企业场景：计算网站PV和UV**

### 动态设置并发度

Storm支持在不restart topology的情况下, 动态的改变(增减)worker processes的数目和executors的数目, 称为rebalancing.   
通过Storm web UI, 或者通过storm rebalance命令

# Reconfigure the topology "mytopology" to use 5 worker processes,

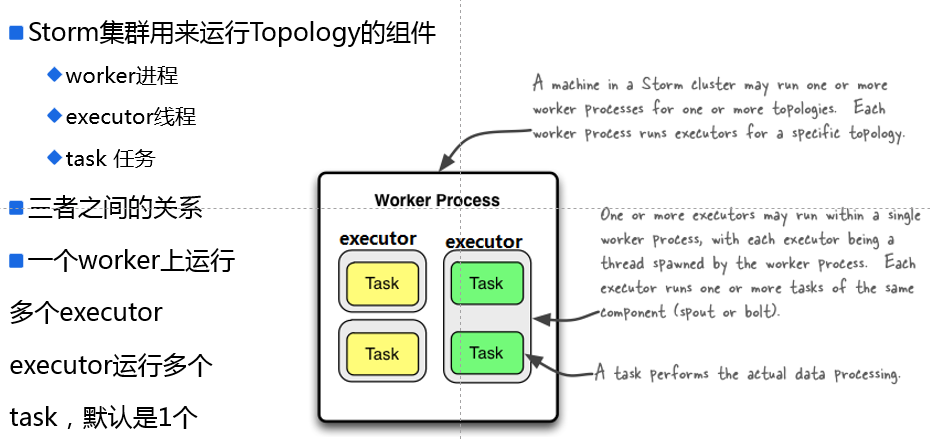
# the spout "blue-spout" to use 3 executors and

# the bolt "yellow-bolt" to use 10 executors.

storm rebalance mytopology -n 5 -e blue-spout=3 -e yellow-bolt=10

其中mytopology 是topo名，blue-spout对应setSpout("blue-spout“…

## Topology的构造



backtype.storm.topology.TopologyBuilder

## Spout组件的编写

实现接口 backtype.storm.topology.IRichSpout;

或者继承backtype.storm.topology.base.BaseRichSpout;

@Override

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

// TODO Auto-generated method stub

}

open 方法，是spout的组件初始化方法，而且Spout实例创建后首先被调用，只调用一次

@Override

public void close() {

// 对于资源的释放关闭，可以在该方法中实现

}

@Override

public void nextTuple() {

// 实现如何从数据源上获取数据的逻辑

// 以及向后面的组件bolt发射数据

}

nextTuple 循环调用

@Override

public void ack(Object msgId) {

// TODO Auto-generated method stub

}

Topology启用了消息可靠性保障机制，当某个Tuple在Topology上处理成功后，调用ack方法执行一些消息处理成功后该干的事情

@Override

public void fail(Object msgId) {

// Topology启用了消息可靠性保障机制，某个Tuple在后面处理失败，该干什么

// 比如重试，重试达到最大可重试就丢弃

}

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

// 声明向后面组件发射的Tuple keys依次是什么

}

@Override

public Map<String, Object> getComponentConfiguration() {

// 设置该组件Spout一些专用的参数

return null;

}

kafkaSpout 向后发射的Tuple {"str":"msg"}

注意点：

Topology中使用的一些类，最好都要实现序列化接口 java.io.Serializable

## Bolt组件

实现backtype.storm.topology.IRichBolt

或者继承backtype.storm.topology.base.BaseRichBolt

@Override

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

//类似于spout中open方法

}

SpoutOutputCollector spout组件中tuple的发射器

OutputCollector bolt组件中tuple发射器

@Override

public void execute(Tuple input) {

// TODO Auto-generated method stub

}

execute 类似于Spout的nextTuple方法

@Override

public void cleanup() {

// TODO Auto-generated method stub

}

类似于spout中close方法

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

// 声明向后面组件发射的Tuple keys依次是什么

}

@Override

public Map<String, Object> getComponentConfiguration() {

// 设置该组件Spout一些专用的参数

return null;

}

## 数据流分组 方式

shuffleGrouping 随机分配

fieldsGrouping 根据key分组进行分配

globalGrouping 全局分组 只会将tuple往后面组件中固定一个上发送

## 消息可靠性保障机制

KafkaSpout 重复消费消息出现的原因：

KafkaSpout启用了消息可靠性保障机制

SplitBot、CountBolt并没有启用消息可靠性保障机制

ack fail 方法

### Spout端：

1）发射器发射tuple时，需要指定一个msgID

collector.emit(new Values(sentence),mssageId );

2）使用缓存所发射的tuple，Map key=msgID,value = Values

private Map<Object,Values> tuples;

3）Spout接收到后面组件对tuple处理成功的确认信息，调用ack方法，ack方法内实现tuple处理成功所要做的事情，比如将tuple从内存map中移除

// 确认发射成功，将tuple从缓存中移除

tuples.remove(msgId);

4）fail方法失败，或者超时，spout调用fail方法

// 重试

Values values = tuples.get(msgId);

// 重新发射

collector.emit(values,msgId );

// 设置tuple的处理超时时间

conf.setMessageTimeoutSecs(30);

### Bolt端：

1）如果bolt端继续往后面组件发射，需要锚定前面的tuple

// 启用消息可靠性保障机制，需要锚定接收到tuple

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

2）处理完tuple后

// 确认处理结束

collector.ack(input);

try{

}catch{

// 处理失败

collector.fail(input);

}

### acker组件：

执行消息可靠性保障机制的任务

如果启用了消息可靠性保障机制，对Topology运行性能一定会造成下降影响，如果下降程度超过可接受范围，解决办法：增加acker组件的并发度（增加执行acker任务的executor线程个数）

conf.setNumAckers(4);

## Storm的并发

1）worker工作进程级别的并发设置

conf.setNumWorkers(2)

2）executor线程级别的并发

真正能够提高Topology性能的并发级别

3）Task任务级别的并发

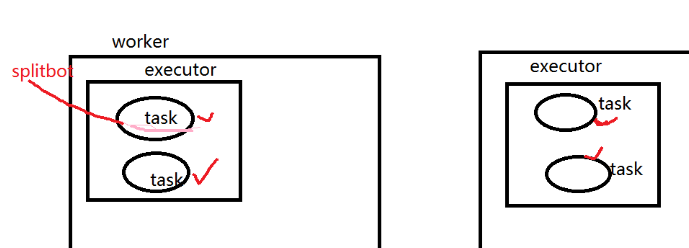
设置某个组件（spout、bolt）总任务数

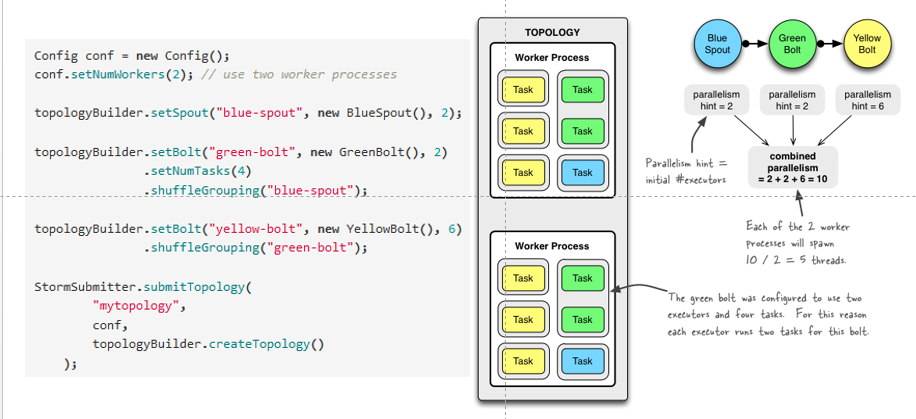
builder.setBolt(SPLIT\_BOLT, new SplitBolt(),2).setNumTasks(4)

指定启动两个executor线程来执行splitbolt任务，

setNumTasks(4)表示splitbolt总共有4个task，

每个executor执行2个task，这两个task并不是并发运行，而是轮流执行





#### worker工作进程的并发度：Topology由多少个worker来执行

Config conf = new Config();

conf.setNumWorkers(2);

#### executor线程的并发度：指Spout、bolt由多少个线程来执行

builder.setSpout(SPOUT\_ID, new SentenceSpout(),2);

builder.setBolt(SPLIT\_BOLT, new SplitBolt(),3).shuffleGrouping(SPOUT\_ID);

#### 直接指定task并发度：

builder.setBolt(SPLIT\_BOLT, new SplitBolt(),3).setNumTasks(12).shuffleGrouping(SPOUT\_ID);

bolt有12个Task来运行，12task由3个executor执行

每个executor线程要运行4个task，4个Task只能在executor线程上***轮流***执行

# 事务批处理

对于容错机制，Storm通过一个系统级别的组件acker，结合xor校验机制判断一个tuple是否发送成功，进而spout可以重发该tuple ，保证一个tuple在出错的情况下**至少被重发一次**。

但是在需要精确统计tuple的数量如销售金额场景时，希望每个tuple”**被且仅被处理一次**” 。Storm 0.7.0引入了Transactional Topology, 它可以保证每个tuple”被且仅被处理一次”, 这样我们就可以实现一种非常准确，且高度容错方式来实现计数类应用。

逐个处理单个tuple，增加很多开销，如写库、输出结果频率过高

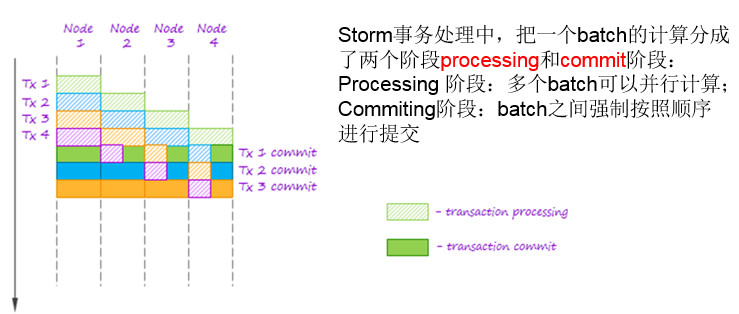
事务处理单个tuple效率比较低，因此storm中引入batch处理，

批处理是一次性处理一批（batch）tuple，事务可确保该批次要么全部处理成功，如果有处理失败的则全部不计，Storm会对失败的批次重新发送，且确保每个batch被且仅被处理一次。

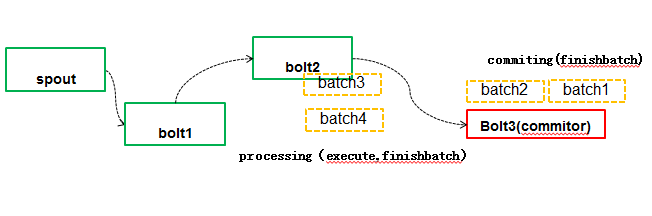
## 事务机制原理

对于只处理一次的需要，从原理上来讲，需要在发送tuple的时候带上txid，在需要事务处理的时候，根据该txid是否以前已经处理成功来决定是否进行处理，当然需要把txid和处理结果一起做保存。

在事务batch处理中，一批tuple赋予一个txid，为了提高batch之间处理的并行度，storm采用了pipeline（管道）处理模型，这样多个事务可以并行执行，但是commit的是严格按照顺序的。



### 事务Topoology



Processing 阶段：多个batch可以并行计算，上面例子中bolt2是普通的batchbolt(实现**IBatchBolt**)，那么多个batch在 bolt2的task之间可以并行执行.

Commiting阶段：batch之间强制按照顺序进行提交，上图中Bolt3实现IBatchBolt并且标记需要事务处理的(实现了ICommitter接口或者通过TransactionalTopologyBuilder的setCommitterBolt方法把BatchBolt添加到topology里面)，那么在Storm认为可以提交batch的时候调用 finishbatch，在finishBatch做txid的比较以及状态保存工作。例子中batch2必须等待batch1提交后，才可以进行提交。

## 事务Topologies

使用**Transactional Topologies**的时候， storm为你做下面这些事情：

**管理状态**: Storm把所有实现Transactional Topologies所必须的状态保存在 zookeeper里面，包括当前transaction id及定义每个batch的一些元数据。

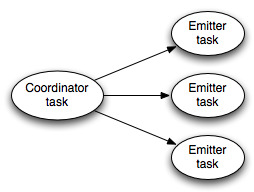
**协调事务**: Storm帮你管理所有事情，如帮你决定在任何一个时间点是该proccessing 还是该committing。

**错误检测**: Storm利用acking框架来高效地检测什么时候一个batch被成功处理了，被成功提交了，或者失败了。Storm然后会相应地replay对应的batch。你不需要自己手 动做任何**acking**或者**anchoring** (emit时发生的动作)。

**内置的批处理API**: Storm在普通bolt之上包装了一层API来提供对tuple的批处理支持。Storm管理所有的协调工作，包括决定什么时候一个bolt接收到一个特定transaction的所有tuple。Storm同时也会自动清理每个transaction所产生的中间数据。

## 事务Topology 实现

事务性的spout需要实现ITransactionalSpout，这个接口包含两个内部接口类Coordinator和Emitter。在topology运行的时候，事务性的spout内部包含一个子Topology，结构图如下：

这里面有两种类型的tuple，一种是事务性的tuple，一种是batch中的tuple；

coordinator 开启一个事务准备发射一个batch时候，进入一个事务的processing阶段，会发射一个事务性 tuple(transactionAttempt & metadata)到”batch emit”流

Emitter以all grouping(广播)的方式订阅coordinator的”batch emit”流，负责为每个batch实际发射tuple。发送的tuple都必须以TransactionAttempt作为第一个field，storm根据这个field来判断tuple属于哪一个batch。

coordinator只有一个，emitter根据并行度可以有多个实例

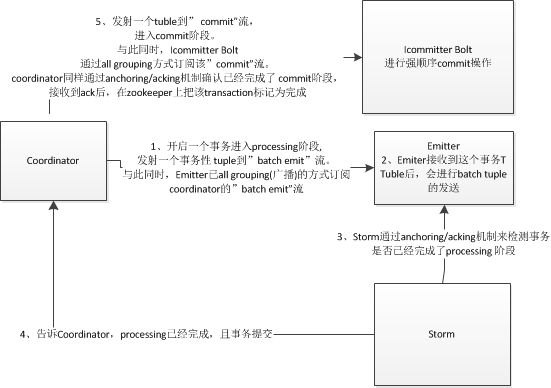
### TransactionAttempt 和 元数据

 TransactionAttempt包含两个值：一个transaction id，一个attempt id。transaction id的作用就是我们上面介绍的对于每个batch中的tuple是唯一的，而且不管这个batch    replay多少次都是一样的。

attempt id是对于每个batch唯一的一个id， 但是对于同一个batch，它replay之后的attempt id跟replay之前就不一样了，

       我们可以把attempt id理解成replay-times， storm利用这个id来区别一个batch发射的tuple的不同版本

metadata(元数据)中包含当前事务可以从哪个point进行重放数据，存放在zookeeper中的，spout可以通过Kryo从zookeeper中序列化和**反序列化**该元数据。

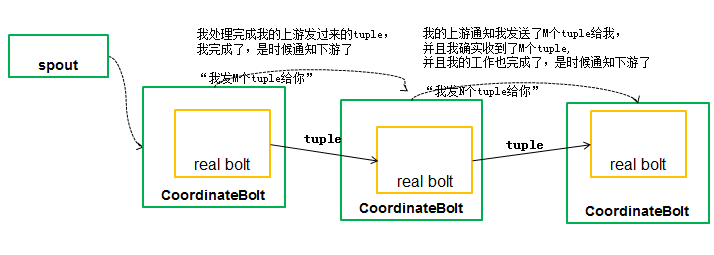


## 事务性Bolt

BaseTransactionalBolt

处理batch在一起的tuples，对于每一个tuple调用调用execute方 法，而在整个batch处理(processing)完成的时候调用finishBatch方法。如果BatchBolt被标记成Committer，则 只能在commit阶段调用finishBatch方法。一个batch的commit阶段由storm保证只在前一个batch成功提交之后才会执行。并且它会重试直到topology里面的所有bolt在commit完成提交。那么如何知道batch的processing完成了，也就是bolt是否接收处理了batch里面所有的tuple；在bolt内部，有一个 CoordinatedBolt的模型。

CoordinateBolt具体原理如下：



### CoordinateBolt

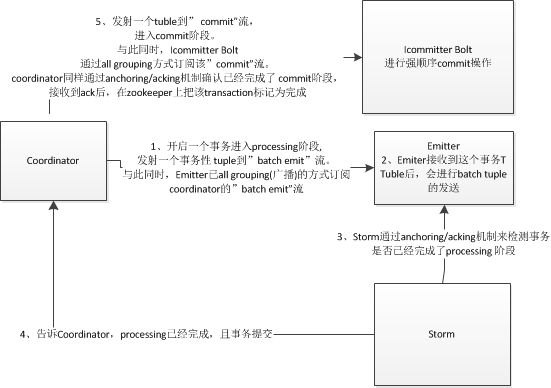
每个CoordinateBolt记录两个值：有哪些task给我发送了tuple（根据topology的grouping信息）；我要给哪些task发送信息（同样根据groping信息）。

等所有的tuple都发送完了之后，CoordinateBolt通过另外一个特殊的stream以emitDirect的方式告诉所有它发送过 tuple的task，它发送了多少tuple给这个task。下游task会将这个数字和自己已经接收到的tuple数量做对比，如果相等，则说明处理 完了所有的tuple。

下游CoordinateBolt会重复上面的步骤，通知其下游。

## 事务API

## 事务处理流程图



### Spout

[**ITransactionalSpout<T>**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/ITransactionalSpout.html)，同BaseTransactionalSpout<T>，普通事务Spout

[**IPartitionedTransactionalSpout<T>**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/partitioned/IPartitionedTransactionalSpout.html)，同BasePartitionedTransactionalSpout<T>，分区事务Spout

[**IOpaquePartitionedTransactionalSpout<T>**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/partitioned/IOpaquePartitionedTransactionalSpout.html)：同BaseOpaquePartitionedTransactionalSpout<T>，不透明分区事务Spout

#### [ITransactionalSpout<T>](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/ITransactionalSpout.html)：普通事务Spout

-- [**ITransactionalSpout.Coordinator<X>**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/ITransactionalSpout.Coordinator.html)

--initializeTransaction(BigInteger txid, X prevMetadata) ：

创建一个新的metadata，当[isReady](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/ITransactionalSpout.Coordinator.html)() 为true时，发射该metadata（事务tuple）到“batch emit”流

--[**isReady**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/ITransactionalSpout.Coordinator.html)() ：为true时启动新事务，需要时可以在此sleep

-- [**ITransactionalSpout.Emitter<X>**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/ITransactionalSpout.Emitter.html)

--[emitBatch](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/ITransactionalSpout.Emitter.html)([TransactionAttempt](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/TransactionAttempt.html) tx, [X](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/ITransactionalSpout.Emitter.html) coordinatorMeta, [BatchOutputCollector](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/coordination/BatchOutputCollector.html) collector) ：逐个发射batch的tuple

#### [IPartitionedTransactionalSpout<T>](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/partitioned/IPartitionedTransactionalSpout.html)：分区事务Spout

主流事务Spout，原因是目前主流Message Queue都支持分区，分区的作用是增加MQ的吞吐量（每个分区作为一个数据源发送点），主流MQ如Kafka、RocketMQ

-- [**IPartitionedTransactionalSpout.Coordinator**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/partitioned/IPartitionedTransactionalSpout.Coordinator.html)

-- [**isReady**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/partitioned/IPartitionedTransactionalSpout.Coordinator.html)() ：同上  
 -- [**numPartitions**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/partitioned/IPartitionedTransactionalSpout.Coordinator.html)() ：返回分区个数。当增加了数据源新分区，同时一个事务被replayed ，此时则不发射新分区的tuples，因为它知道该事务中有多少个分区。

-- [**IPartitionedTransactionalSpout.Emitter**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/partitioned/IPartitionedTransactionalSpout.Emitter.html)<[**X**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/partitioned/IPartitionedTransactionalSpout.Emitter.html)>   
 --[**emitPartitionBatchNew**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/partitioned/IPartitionedTransactionalSpout.Emitter.html)([TransactionAttempt](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/TransactionAttempt.html) tx, [BatchOutputCollector](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/coordination/BatchOutputCollector.html) collector, int partition, [X](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/partitioned/IPartitionedTransactionalSpout.Emitter.html) lastPartitionMeta) ：发射一个新的Batch，返回Metadata

--[**emitPartitionBatch**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/partitioned/IPartitionedTransactionalSpout.Emitter.html)([TransactionAttempt](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/TransactionAttempt.html) tx, [BatchOutputCollector](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/coordination/BatchOutputCollector.html) collector, int partition, [X](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/partitioned/IPartitionedTransactionalSpout.Emitter.html) partitionMeta) ：如果这批消息Bolt消费失败了，emitPartitionBatch负责重发这批消息

#### IOpaquePartitionedTransactionalSpout<T>：不透明分区事务Spout

--[**IOpaquePartitionedTransactionalSpout.Coordinator**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/partitioned/IOpaquePartitionedTransactionalSpout.Coordinator.html)

--[**isReady**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/partitioned/IOpaquePartitionedTransactionalSpout.Coordinator.html)() ：同上  
 --[**IOpaquePartitionedTransactionalSpout.Emitter**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/partitioned/IOpaquePartitionedTransactionalSpout.Emitter.html)<[**X**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/partitioned/IOpaquePartitionedTransactionalSpout.Emitter.html)>

-- [**emitPartitionBatch**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/partitioned/IOpaquePartitionedTransactionalSpout.Emitter.html)([TransactionAttempt](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/TransactionAttempt.html) tx, [BatchOutputCollector](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/coordination/BatchOutputCollector.html) collector, int partition, [X](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/partitioned/IOpaquePartitionedTransactionalSpout.Emitter.html) lastPartitionMeta)   
 -- [**numPartitions**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/transactional/partitioned/IOpaquePartitionedTransactionalSpout.Emitter.html)()

它不区分发新消息还是重发旧消息，全部用emitPartitionBatch搞定。虽然emitPartitionBatch返回的X应该是下一批次供自己使用的（emitPartitionBatch的第4个参数），但是只有一个批次成功以后X才会更新到ZooKeeper中，如果失败重发，emitPartitionBatch读取的X还是旧的。所以这时候自定义的X不需要记录当前批次的开始位置和下一批次的开始位置两个值，只需要记录下一批次开始位置一个值即可，例如：

public class BatchMeta {   
    public long  nextOffset; //下一批次的偏移量      
}

[IPartitionedTransactionalSpout](http://nathanmarz.github.io/storm/doc-0.8.1/backtype/storm/transactional/partitioned/IPartitionedTransactionalSpout.html)和[IOpaquePartitionedTransactionalSpout](http://nathanmarz.github.io/storm/doc-0.8.1/backtype/storm/transactional/partitioned/IOpaquePartitionedTransactionalSpout.html)都是把tuple封装成batch进行处理，同时可以保证每一个tuple都被完整地处理，都支持消息重发。为了支持事务性，它们为每一个批次（batch）提供一个唯一的事务ID（transaction id：txid），txid是顺序递增的，而且保证对批次的处理是强有序的，即必须完整处理完txid=1才能再接着处理txid=2。

二者的区别以及用法：

IPartitionedTransactionalSpout的每一个tuple都会绑定在固定的批次中。无论一个tuple重发多少次，它都在同一个批次里面，都有同样的事务ID；一个tuple不会出现在两个以上的批次里。一个批次无论重发多少次，它也只有一个唯一且相同的事务ID，不会改变。这也就是说，一个批次无论重发多少次，它所包含的内容都是完全一致的。

但是IPartitionedTransactionalSpout会有一个问题，虽然这种问题非常罕见：假设一批消息在被bolt消费过程中失败了，需要spout重发，此时如果正巧遇到消息发送中间件故障，例如某一个分区不可读，spout为了保证重发时每一批次包含的tuple一致，它只能等待消息中间件恢复，也就是卡在那里无法再继续发送给bolt消息了，直至消息中间件恢复。IOpaquePartitionedTransactionalSpout可以解决这个问题。

而IOpaquePartitionedTransactionalSpout为了解决这个问题，它不保证每次重发一个批次的消息所包含的tuple完全一致。也就是说某个tuple可能第一次在txid=2的批次中出现，后面有可能在txid=5的批次中出现。这种情况只出现在当某一批次消息消费失败需要重发且恰巧消息中间件故障时。这时，IOpaquePartitionedTransactionalSpout不是等待消息中间件故障恢复，而是先读取可读的partition。例如txid=2的批次在消费过程中失败了，需要重发，恰巧消息中间件的16个分区有1个分区(partition=3)因为故障不可读了。这时候IOpaquePartitionedTransactionalSpout会先读另外的15个分区，完成txid=2这个批次的发送，这时候同样的批次其实包含的tuple已经少了。假设在txid=5时消息中间件的故障恢复了，那之前在txid=2且在分区partition=3的tuple会重新发送，包含在txid=5的批次中。

在使用IOpaquePartitionedTransactionalSpout时，因为tuple与txid的对应关系有可能改变，因此与业务计算结果同时保存一个txid就无法保证事务性了。这时候解决方案会稍微复杂一些，除了保存业务计算结果以外，还要保存两个元素：前一批次的业务计算结果以及本批次的事务ID。

如：

{ value = 4,

prevValue = 1,

txid = 2

}

### Bolt

[**IBatchBolt<T>**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/coordination/IBatchBolt.html)：同BaseBatchBolt<T>，普通批处理

[**BaseTransactionalBolt**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/topology/base/BaseTransactionalBolt.html)：事务Bolt

接口Icommitter：标识IBatchBolt 或[**BaseTransactionalBolt**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/topology/base/BaseTransactionalBolt.html)是否是一个committer  
[**CoordinatedBolt**](mk:@MSITStore:D:\javadoc\storm-0.9.0.1\Storm_0.9.chm::/backtype/storm/coordination/CoordinatedBolt.html)

# DPRC

分布式RPC（distributed RPC，DRPC）用于对Storm上大量的函数调用进行并行计算。

对于每一次函数调用，Storm集群上运行的拓扑接收调用函数的参数信息作为输入流，并将计算结果作为输出流发射出去。

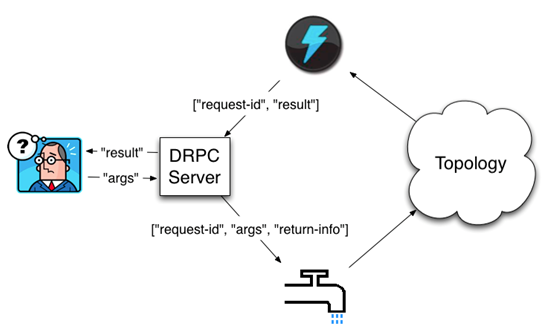
一句话概括：Storm进行计算，根据客户端提交的请求参数，而返回Storm计算的结果。

DRPC通过DRPC Server来实现，DRPC Server的整体工作过程如下：

接收到一个RPC调用请求；发送请求到Storm上的拓扑；从Storm上接收计算结果；将计算结果返回给客户端。

注：在client客户端看来，一个DRPC调用看起来和一般的RPC调用没什么区别。

## 工作流程



Client向DRPC Server发送被调用执行的DRPC函数名称及参数；

Storm上的topology通过DRPCSpout实现这一函数，从DPRC Server接收到函数调用流；

DRPC Server会为每次函数调用生成唯一的id；

Storm上运行的topology开始计算结果，最后通过一个**ReturnResults**的Bolt连接到DRPC Server，发送指定id的计算结果；

DRPC Server通过使用之前为每个函数调用生成的id，将结果关联到对应的发起调用的client，将计算结果返回给client。

## LinearDRPCTopologyBuilder

Storm提供了一个topology builder——[LinearDRPCTopologyBuilder](http://nathanmarz.github.com/storm/doc/backtype/storm/drpc/LinearDRPCTopologyBuilder.html)，

它可以自动完成几乎所有的DRPC步骤。

包括： 构建spout；向DRPC Server返回结果；为Bolt提供函数用于对tuples进行处理。

## 远程模式DRPC

在实际的Storm集群上运行DRPC也一样很简单。只需完成以下步骤：

启动DRPC Server(s)；

配置DRPC Server(s)地址；

向Storm集群提交DRPC拓扑。

首先，通过storm脚本启动DRPC Server：

storm drpc

然后，在Storm集群中配置DRPC Server地址，这就是DRPCSpout读取函数调用请求的地方。这一步的配置可以通过storm.yaml文件或者拓扑的配置来完成。通过storm.yaml文件的配置方式如下：

drpc.servers:

- "drpc1.foo.com"

- "drpc2.foo.com"

## 客户端访问

DRPCClient client = new DRPCClient("drpc-host", 3772);

String result = client.execute("reach", "http://twitter.com");

# Trident

## 概念

1）Storm高层次的抽象

2）在Trident中保留了Spout，但是不再有Bolt组件，将之前Storm在Bolt组件中所实现的数据处理逻辑抽象成一系列的Operation，比如过滤、函数、分组统计等等。

3）Trident封装好了消息可靠性保障机制

4）Trident 批次概念

a.将固定条数的Tuple划分为一个批次

b.给每个批次一个编号

c.更新统计结果状态，要严格按照批次顺序进行更新

5）事务控制

3个层次：

a.NON-Transactional：非事务控制

允许同一个批次内的 部分处理成功

失败的Tuple，可以在其他批次内进行重试，有可能导致被成功处理多次（在后面的多个批次内被重试）

也有可能不进行重试，被成功处理零次

b.Transactional：严格的事务控制

要求批次内处理失败的Tuple只能在本批次内进行重试（以相同的批次号进行重试）

如果tuple一直重试不成功，就会将整个任务程序挂起，不会进行下个批次的处理

没有容错

c.Opaque-Transactional：透明事务控制

批次内的tuple处理完后，先把成功的更新掉

失败的tuple允许在其他批次内进行重试，只会有一次成功处理

有容错

## Trident的编码开发

ITridentSpout:最通用的Spout，可以支持事务或者不透明事务语义。

    IBatchSpout: 一个非事务spout 。

    IPartitionedTridentSpout: 分区事务spout，从数据源(比如一个Kafka集群)读分区数据

    IOpaquePartitionedTridentSpout:不透明分区事务spout，从数据源读分区数据

### 构造topology

storm.trident.TridentTopology

storm.trident.Stream

trident中的Spout：从数据源上获取数据

将获取到的数据封装到一个个批次内，而且给每个批次指定一个批次号

### each方法：

>>>>>>>> 过滤操作Filter： 对满足条件的tuple进行保留，不满足的丢弃

isKeep方法：实现tuple是否继续保留在Stream的逻辑

在Trident编码中，要注意的：

Stream流在经过各种操作后 Tuple的演变

Tuple中keyvalue对的演变（keyvalue保留、丢弃、追加、替换）

比如：Filter 仅仅只是在Stream保留或者丢弃tuple，而不会对Tuple中的keyvalue对进行更改

### Functions函数操作

实现storm.trident.operation.Function

execute方法

Stream经过函数操作后，将新产生的keyvalue对追加到原来的Tuple中，注意的是，如果没有新产生keyvalue对，那么相应的tuple将会被丢弃掉

### Stream.project方法

指定保留Tuple中的哪些keyvalue 、丢弃哪些keyvalue

## topology提交storm集群测试

// 当args没有值，运行过程中没有指定参数

if(args == null || args.length <= 0){

// 本地测试

LocalCluster localCluter = new LocalCluster();

localCluter.submitTopology("wordcountTrident", config, topology.build());

}else{

// 提交集群运行

try {

StormSubmitter.submitTopology(args[0], config, topology.build());

} catch (AlreadyAliveException e) {

e.printStackTrace();

} catch (InvalidTopologyException e) {

e.printStackTrace();

}

}

maven打包

maven-assembly-plugin 插件

$ bin/storm jar /home/ibeifeng/storm-test-0.0.1-SNAPSHOT-jar-with-dependencies.jar com.ibeifeng.storm.trident.WordCountTrident wordcountTrident

## Trident的并发度

.each(new Fields("str"), new SplitFunction(),new Fields("word"))

//设置2个executor来执行splitfunction操作

.parallelismHint(2)

在某个操作调用后，stream调用 parallelismHint，设置前面这个操作的并发度

## Operation的特性

是否同一分区内、是否需要跨网络

分区：指的就是一个在excutor线程中运行的task

Filter Function project 分区内的操作，只是对本分区内的Tuple进行操作

## 函数

public class AddAndMultiply extends BaseFunction

{

public void execute(TridentTuple tuple, TridentCollector collector)

{

int i1 = tuple.getInteger(0);

int i2 = tuple.getInteger(1);

collector.emit(new Values(i1 + i2, i1 \* i2));

}

}

stream.each(new Fields("x", "y"), new AddAndMultiply(), new Fields("added", "multiplied"));

输出的功能字段被添加到输入tuple中。因此每个tuple中将会有4个字段"x", "y", "added", 和 "multiplied".

其中 "added" 和"multiplied"对应于AddAndMultiply输出的第一和第二个字段。

## 聚合

Trident有aggregate和persistentAggregate函数对流做聚合。Aggregate在每个批次上独立运行，persistentAggregate聚合流的所有的批次并将结果存储到State中。

在一个流上做全局的聚合，可以使用reducecerAggregator或者aggretator，这个流先被分成一个分区，然后聚合函数在这个分区上 运行。如果使用CombinerAggreator，Trident先在每个分区上做一个局部的汇总，然后重分区为一个分区，在网络传输结束后完成聚合。

stream**.**aggregate**(new** Fields**(**"val2"**),** **new** Sum**(),** **new** Fields**(**"sum"**))**

output stream将会只包含一个叫做“sum”的字段，这个sum字段就是“val2”的累积和。

### persistentAggregate持久化聚合

Trident有另外一种更新State的方法叫做persistentAggregate，在之前的word count例子中我们已经见过了。

persistentAggregate是在partitionPersist之上的另外一层抽象, 它知道怎么去使用一个Trident 聚合器来更新State。

在word count 例子中，因为这是一个group好的stream，Trident会期待你提供的state是实现了**MapState**接口，用来进行group的字段以key的形式存在于State当中，聚合后的结果会以value的形式存储在State当中。

MapState接口看上去如下所示：  
public interface MapState<T> extends State {

    List<T> multiGet(List<List<Object>> keys);

    List<T> multiUpdate(List<List<Object>> keys, List<ValueUpdater> updaters);

    void multiPut(List<List<Object>> keys, List<T> vals);

}   
当你在一个未经过group的stream上面进行聚合的话，Trident会期待你的state实现Snapshottable接口：

public interface Snapshottable<T> extends State {

    T get();

    T update(ValueUpdater updater);

    void set(T o);

}

## Group By

在group之后的stream上，输出将会是被group的字段以及聚合器输出的字段,如：

**stream.groupBy(new Fields("val1")) .aggregate(new Fields("val2"), new Sum(), new Fields("sum"))**

在这个例子中，**输出将包含字段"val1" 和 "sum".**

GroupBy操作根据特殊的字段对流进行重分区，分组字段相同的元组（tuple）被分到同一个分区。如果对分组的流进行聚合，聚会会对每个组聚合而不是这个批次聚合。（和关系型数据库的groupby相同）。

## State

Trident在读写有状态的数据源方面是有着一流的抽象封装的。状态即可以保留在topology的内部，如内存。也可以放到外部存储当中，如HDFS， Memcached（内存级数据库）或**no-sql库**。这些都是使用同一套Trident API。

Trident以一种容错的方式来管理状态（状态指结果数据），以至于当你在更新状态的时候你不需要去考虑错误以及重试的情况。这种保证每个消息被处理有且只有一次的原理会让你更放心的使用Trident的topology。

Opaque transactional state有着最为强大的容错性。但是这是以存储更多的信息作为代价的。Transactional states 需要存储较少的状态信息，但是仅能和 transactional spouts协同工作. Finally, non-transactional state所需要存储的信息最少，但是却不能实现有且只有一次被成功处理的语义。

**State和Spout类型的选择其实是一种在容错性和存储消耗之间的权衡，根据应用的需要会进行选择。**

Trident把所有容错相关的逻辑都放在了State里面。

 作为一个用户，你并不需要自己去处理复杂的txid，存储多余的信息到数据库中，或者是任何其他类似的事情。

## 如何开发一个State

假如用数据库来存储用户Location，并且你想要在Trident中去访问这个数据。你的state的实现应该有用户信息的set、get方法。

public class **LocationDB** implements State {

    public void beginCommit(Long txid) {

    }

    public void commit(Long txid) {

    }

    public void setLocation(long userId, String location) {

      // code to access database and set location

    }

    public String getLocation(long userId) {

      // code to get location from database

    }

}

然后你还需要提供给Trident一个StateFactory来在Trident中创建你的State对象。StateFactory可以如下：

public class **LocationDBFactory** implements StateFactory {

   public State **makeState**(Map conf, int partitionIndex, int numPartitions) {

      return new **LocationDB**();

   }

}

Trident提供了一个**QueryFunction**接口用来实现Trident中在一个source state上查询的功能，如MapGet类。还提供StateUpdater来实现Trident中更新source state的功能。

例如：写一个查询地址的操作，来查询LocationDB找到用户的Location。

假定这个topology会接受一个用户id作为输入数据流，如：

TridentTopology topology = new TridentTopology();

TridentState**locations** = topology.newStaticState(new **LocationDBFactory**());

topology.newStream("myspout", spout)

        .stateQuery(**locations**, new Fields("userid"), new **QueryLocation**(), new Fields("location"))

QueryLocation 的实现:

public class QueryLocation extends **BaseQueryFunction**<**LocationDB, String**> {

    public List<String> **batchRetrieve**(LocationDB state, List<TridentTuple> inputs) {

        List<String> ret = new ArrayList();

        for(TridentTuple input: inputs) {

            ret.add(**state.getLocation**(input.getLong(0)));

        }

        return ret;

    }

    public void execute(TridentTuple tuple, String **location**, TridentCollector collector) {

        collector.emit(new Values(location));

    }

}

QueryFunction的执行分为两部分。

batchRetrieve 收集一个batch的read操作结果。

execute 方法负责把结果emit出去。

你可以看到，这段代码为每个输入tuple去查询了一次LocationDB，没利用Batch优势。所以一种更好的操作LocationDB方式应该是这样的：

public class LocationDB implements State {

    public void beginCommit(Long txid) {

    }

    public void commit(Long txid) {

    }

    public void setLocationsBulk(List<Long> userIds, List<String> locations) {

      // set locations in bulk

    }

    public List<String> bulkGetLocations(List<Long> userIds) {

      // get locations in bulk

    }

}

## StateUpdater接口

若要更新State，你需要使用StateUpdater接口。一个StateUpdater的例子用来将新的地址信息更新到LocationDB当中。

public class LocationUpdater extends BaseStateUpdater<LocationDB> {

    public void updateState(LocationDB state, List<TridentTuple> tuples, TridentCollector collector) {

        List<Long> ids = new ArrayList<Long>();

        List<String> locations = new ArrayList<String>();

        for(TridentTuple t: tuples) {

            ids.add(t.getLong(0));

            locations.add(t.getString(1));

        }

**state.setLocationsBulk(ids, locations);**

    }

}

如何在Trident topology中使用上面声明的LocationUpdater？

TridentTopology topology = new TridentTopology();

TridentState locations =  topology.newStream("locations", locationsSpout)

   .**partitionPersist**(new **LocationDBFactory**(), new Fields("userid", "location"),

new LocationUpdater());

partitionPersist 会更新State。其内部是将 State和一批更新的tuple交给StateUpdater，由StateUpdater完成相应的更新操作。

在这段代码中，只是简单的从输入的tuple中提取处userid和对应的location，并一起更新到State中。

## 如何实现Map Stages?

在Trident中实现MapState是非常简单的，它几乎帮你做了所有的事情。OpaqueMap, TransactionalMap, 和 NonTransactionalMap 类实现了所有相关的逻辑，包括容错的逻辑。你只需要将一个IBackingMap 的实现提供给这些类就可以了。IBackingMap接口看上去如下所示：

public interface IBackingMap<T> {

    List<T> multiGet(List<List<Object>> keys);

    void multiPut(List<List<Object>> keys, List<T> vals);

}

OpaqueMap会用OpaqueValue的value来调用multiPut方法，TransactionalMap's会提供TransactionalValue中的value，而NonTransactionalMaps只是简单的把从Topology获取的object传递给multiPut。

## 重分区操作 数据流分组

partitionBy 类似于 mapreduce分区

例如：.partitionBy(new Fields("word"))

取Tuple，key名称为word的keyvalue的value值 求hashcode ，然后根据哈希值 % 分区数进行取模

0 1 2

特点： **相同的keyvalue对 进入同一个分区**

同一个分区内的keyvalue对是否都相同？不一定相同

groupBy：// MapReduce 分区 + 分组

例如：.groupBy(new Fields("word"))

根据指定的keyvalue对进行分组

实质上进行了两步操作：

1）partitionBy

2）在将同一个分区内的相同的keyvalue对分为一组

groupBy：

1）同一批次内各个分区的分组统计（局部，不需要跨网络传输）

partitionAggregate

.chainedAgg()

.chainEnd()

构造聚合链

### 全局统计 ： 需要跨网络传输 persistentAggregate

分组统计：

1）直接进行全局统计（跨网络数据传输）

2）先在同一批次各分区内进行局部统计（不需要进行跨网络传输），然后对局部统计好的结果进行全局统计

group by

partitionAggregate 局部统计

persistentAggregate 全局统计

### shuffle 随机分配

前面组件产生的Tuple随机分给后面组件的各个分区 类似于shuffleGrouping

\*\*\*\*\*\* partitionIndex=2 , str=kafka yarn kafka

\*\*\*\*\*\* partitionIndex=0 , str=kafka yarn kafka

### partitionBy效果(用的比较少) ：

partitionIndex=0---->hadoop

partitionIndex=0---->storm

partitionIndex=0---->spark

{"word":"spark"}

partitionIndex=1---->mapreduce

partitionIndex=1---->kafka

partitionIndex=1---->mahout

partitionIndex=2---->flume

partitionIndex=3---->yarn

### global 类似于gloablGrouping

所有的Tuple只会进入到后面一个分区内 （慎重使用）

vvfieldsGrouping === partitionBy

### batchGlobal：

同一批次内的tuple只会进入到后面一个分区内，不同批次内的Tuple可能进入不同的分区内

### broadcast 广播

将tuple复制给后面的所有分区，一般结合drpc使用

## DRPC

分布式远程过程调用 请求-响应

需求：查询当前统计结果

\*统计结果数据存入到HBase、Redis

TridentState 统计结果状态

TridentTopology.newDRPCStream

### 集群上使用drpc

要启用drpc server进程 需要先修改storm.yaml配置文件添加

drpc.servers

drpc.port

重启storm集群

添加启动 drpc服务进程

$ nohup bin/storm drpc > /dev/null 2>&1 &

# storm 批启动脚本

统一写在节点nimbus服务器

## zookeeper批启动脚本：

zoo.cfg

server.1=hostname:2888:388

zookeeper.sh 内容：

#!/bin/bash

if [ $# -ne 1 ];then

echo "Usage: bin/zkServer.sh {start|start-foreground|stop|restart|status|upgrade|print-cmd}"

exit 4

fi

source /etc/profile

ZK\_HOME=/opt/modules/zookeeper-3.4.5-cdh5.3.6

for node in hive-stu.ibeifeng.com

do

echo "$1 zookeeper in $node"

ssh $node "source /etc/profile && $ZK\_HOME/bin/zkServer.sh $1"

done

## Storm停止脚本

主节点/nimbus ui/从节点/logviewer/supervisor

在storm的主节点上添加一个指定supervisor节点的文件

supervisors

/opt/modules/apache-storm-0.9.6/conf/supervisors

内容 一行一个Supervisor节点服务器主机名

$ kill -9 `ps -ef | grep daemon.nimbus | grep –v grep|awk '{print $2}' | head -n 1`

$ kill -9 `ps -ef | grep daemon.supervisor | awk '{print $2}' | head -n 1`

$ kill -9 `ps -ef | grep ui.core | awk '{print $2}' | head -n 1`

$ kill -9 `ps -ef | grep daemon.logviewer | awk '{print $2}' | head -n 1`

stop-storm.sh

#!/bin/bash

source /etc/profile

STORM\_HOME=/opt/modules/apache-storm-0.9.6

#先在主节点上停止nimbus和ui进程

kill -9 `ps -ef | grep daemon.nimbus | awk '{print $2}' | head -n 1`

kill -9 `ps -ef | grep ui.core | awk '{print $2}' | head -n 1`

#在从节点上停止logviewer和supervisor

SUPERVISORS=$(cat $STORM\_HOME/conf/supervisors)

for supervisor in $SUPERVISORS

do

echo "stop supervisor and logviewer in $supervisor"

ssh $supervisor kill -9 `ssh $supervisor ps -ef | grep daemon.supervisor | awk '{print $2}'|head -n 1`

ssh $supervisor kill -9 `ssh $supervisor ps -ef | grep daemon.logviewer | awk '{print $2}'|head -n 1`

done

### 批启动脚本

start-storm.sh

内容：

#!/bin/bash

source /etc/profile

STORM\_HOME=/opt/modules/apache-storm-0.9.6

#先在主节点上启动nimbus和ui进程

$STORM\_HOME/bin/storm nimbus >/dev/null 2>&1 &

$STORM\_HOME/bin/storm ui >/dev/null 2>&1 &

#在从节点上启动logviewer和supervisor

SUPERVISORS=$(cat $STORM\_HOME/conf/supervisors)

for supervisor in $SUPERVISORS

do

echo "start supervisor and logviewer in $supervisor"

ssh $supervisor "source /etc/profile && nohup $STORM\_HOME/bin/storm supervisor >/dev/null 2>&1" >/dev/null 2>&1 &

ssh $supervisor "source /etc/profile && nohup $STORM\_HOME/bin/storm logviewer >/dev/null 2>&1" >/dev/null 2>&1 &

done