# storm安装

## 1.1 单机版

### 1.1.1 上传解压

tar -xvzf apache-storm-0.9.6.tar.gz

mv apache-storm-0.9.6/ /opt/storm/

### 1.1.2配置storm.yaml

vim conf/storm.yaml

找到storm.zookeeper.servers:配置项，这里原来是配置所有zookeeper服务集群的，这里改成本机即可：

storm.zookeeper.servers:

- "hadoop"

# - "server2"

然后配置nimbus监控节点nimbus.host选项，当然还是本地地址：

#

nimbus.host: "hadoop"

#

然后在最后手动添加storm.local.dir相关文件存放的本地目录和supervisor.slots.ports执行worker作业进程的端口：

storm.local.dir: "/opt/storm"

supervisor.slots.ports:

- 6700

### 1.1.3 启动

到这里storm的基本配置就完成了，单机环境暂时不用配置drpc服务了，这个时候保存并退出配置文件

　　然后执行命令启动nimbus主节点： nohup bin/storm nimbus >> /dev/null &  稍等一会屏幕滚动，然后按回车就回到了命令行

　　继续启动supervisor从节点： nohup bin/storm supervisor >> /dev/null &

　　都启动完毕之后，启动strom ui管理界面： bin/storm ui &

现在，执行 jps 命令可以查看到nimbus、supervisor、core这三个进程，打开浏览器访问对应主机的地址，这里是http://192.168.28.133:8080，访问就可以进入Storm UI了

注意：

如果出现nohup: ignoring input and redirecting stderr to stdout错误

将“&”改为“2>&1 &”就OK了：

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

其中 2> 表示把标准错误(stderr)重定向，尖括号后面可以跟文件名，或者是&1, &2，分别表示重定向到标准输出和标准错误

## 1.2 集群版

部分需要依赖Python，红帽系列Linux默认Python版本是2.6.6，可以满足要求；Linux可以安装多个版本Python共存，生产过程中建议Python版本为2.7.x

### 1.2.1 上传解压

tar -xvzf apache-storm-0.9.5.tar.gz

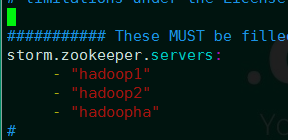
cd /usr/apache-storm-0.9.5

### 1.2.2 配置storm.yaml

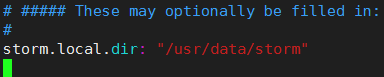
|  |
| --- |
| ***必须修改的项：***  storm.zookeeper.services:配置zookeeper集群的主机名称。  nimbus.host:指定了集群中nimbus的节点。  supervisor.slots.ports:配置控制每个supervisor节点运行多少个worker进程。这个配置定义为worker监听的端口的列表，监听端口的个数控制了supervisor节点上有多少个worker的插槽。默认的storm使用6700~6703端口，每个supervisor节点上有4个worker插槽。  storm.local.dir:storm工作时产生的工作文件存放的位置，注意，要避免配置到/tmp下。    ***可选的常用修改项：***  nimbus.childopts(default: -Xms1024m):这项JVM配置会添加在启动nimbs守护进程的java命令行中。  ui.port(default:8080):这项配置指定了Storm UI的Web服务器监听的端口。  ui.childopts(default:-Xms1024m):这项JVM配置会添加在StormUI服务启动的Java命令行中。  supervisor.childopts(default:-Xms768m):这项JVM配置会添加Supervisor服务启动的Java命令行中。  worker.childopts(default:-Xms768m):这项JVM配置会添加worker服务启动的Java命令行中。  topology.message.timeout.secs(default:30):这个配置项定义了一个tuple树需要应答最大时间秒数限制，超过这个时间则认为超时失败。  topology.max.spout.pending(default:null):在默认值null的情况下，spout每当产生新的tuple时会立即向后端发送，由于下游bolt执行可能具有延迟，可能导致topology过载，从而导致消息处理超时。如果手动将该值改为非null正整数时，会通过暂停spout发送数据来限制同时处理的tuple不能超过这个数，从而达到为Spout限速的作用。  topology.enable.message.timeouts(default:true):这个选项用来锚定的tuple的超时时间。如果设置为false，则锚定的tuple不会超时。 |

接下来需要修改配置文件storm.yaml，执行vim conf/storm.yaml打开文件:

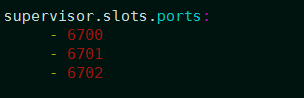
1.去掉storm.zookeeper.servers:前面的注释，修改为集群中所有部署zookeeper的主机，当然都可以自己手动添加:



2.增加storm.local.dir选项，指定nimbus，supervisor进程用于存储少量的状态数据，比如jar包，配置文件等:(需要手动建立)



1. 下面指定supervisor工作节点，需要配置该节点可以运行的worker数量，每个worker占用一个端口用于接收消息，最多分配5个；默认情况下每个节点可以运行4个worker，分别在6700、6701、6702、6703端口，这里定义3个端口，代表最多运行3个worker：



1. 下面设置集群主机，让集群中所有的节点可以从主机下载拓扑以及配置文件，主机上运行的就是nimbus，而其他节点就是supervisor进程，这里hadoopha为nimbus，而hadoop1和hadoop2为supervisor，所以配置如下：

IMG_256

1. 下面配置storm集群的drpc地址，这里就是hadoopha，实际中可以自己定义：

IMG_256

1. 最后配置storm进程的分配内存，默认情况下Storm启动worker进程时，JVM的最大内存是768M，由于在使用过程中，Bolt中加载大量数据，768M内存无法满足要求，会导致内存溢出，应该根据实际情况进行修改，这里设置为2G：

IMG_256

1. 然后在3台主机分别创建上面设置的数据目录，必须都要创建：

mkdir -p /usr/data/storm

　　上面的配置是在hadoopha上配置的，接下来要把storm目录发送到hadoop1和hadoop2：

$ scp -r apache-storm-0.9.5 hadoop1:/usr/

$ scp -r apache-storm-0.9.5 hadoop2:/usr/

### 1.2.3 启动守护进程

启动Nimbus，在hadoopha上执行：nohup bin/storm nimbus >> /dev/null &

在hadoop1，hadoop2节点都启动Supervisor服务：

nohup bin/storm supervisor >> /dev/null &

然后在配置drpc的主机hadoopha，drpc是一种后台服务，用于执行和storm相同的计算，但是比较节省资源，一般和nimbus使用同一台主机即可；执行以下命令启动drpc服务：

nohup bin/storm drpc >> /dev/null &

最后在nimbus节点启动UI服务：nohup bin/storm ui >> /dev/null &

### 1.24 storm管理命令

    1 、JAR：storm jar topology\_jar topology\_class [arguments...]

jar命令是用于提交一个集群拓扑.它运行指定参数的topology\_class中的main()方法，上传topology\_jar到nimbus， 由nimbus发布到集群中。一旦提交，storm将激活拓扑并开始处理topology\_class 中的main()方法，main()方法负责调用StormSubmitter.submitTopology()方法，并提供一个唯一的拓扑(集群)的 名。如果一个拥有该名称的拓扑已经存在于集群中，jar命令将会失败。常见的做法是在使用命令行参数来指定拓扑名称，以便拓扑在提交的时候被命名。

例： storm jar /root/work/stormwc.jar cn.tedu.storm.wc.WordCountTopology

执行一段时间后，可以通过如果下命令关闭topology

storm kill mywc

2、KILL：storm kill topology\_name [-w wait\_time]   
    杀死一个拓扑，可以使用kill命令。它会以一种安全的方式销毁一个拓扑，首先停用拓扑，在等待拓扑消息的时间段内允许拓扑完成当前的数据流。执行 kill命令时可以通过-w [等待秒数]指定拓扑停用以后的等待时间。也可以在Storm UI 界面上实现同样的功能

3、Deactivate：storm deactivate topology\_name   
    停用拓扑时，所有已分发的元组都会得到处理，spouts的nextTuple方法将不会被调用。也可以在Storm UI 界面上实现同样的功能

 4、Activate：storm activate topology\_name   
    启动一个停用的拓扑。也可以在Storm UI 界面上实现同样的功能

 5、Rebalance：storm rebalance topology\_name [-w wait\_time] [-n worker\_count] [-e component\_name=executer\_count]...   
    rebalance使你重新分配集群任务。这是个很强大的命令。比如，你向一个运行中的集群增加了节点。rebalance命令将会停用拓扑，然后在相应超时时间之后重分配worker，并重启拓扑   
例：storm rebalance wordcount-topology -w 15 -n 5 -e sentence-spout=4 -e split-bolt=8   
    还有其他管理命令，如：Remoteconfvalue、REPL、Classpath等

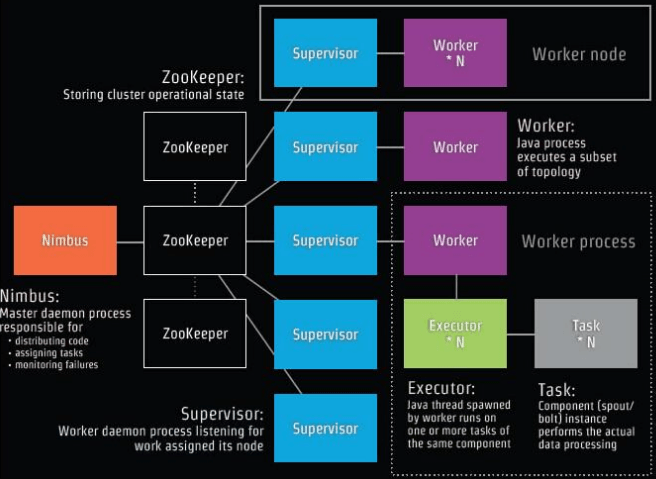
storm remoteconfvalue conf-name 用来查看远程集群中的配置参数值

# 2 Storm概述

## 1.结构topology

storm结构称为topology(拓扑)，由stream(数据流),spout(喷嘴-数据流的生成者),bolt(阀门-数据流运算者)组成(参考图:Storm组成结构)。

不同于Hadoop中的job,Storm中的topology会一直运行下去，除非进程被杀死或取消部署。  Storm集群中有两种节点，一种是控制节点(Nimbus节点)，另一种是工作节点(Supervisor节点)。所有Topology任务的 提交必须在Storm客户端节点上进行(需要配置 storm.yaml文件)，由Nimbus节点分配给其他Supervisor节点进行处理。 Nimbus节点首先将提交的Topology进行分片，分成一个个的Task，并将Task和Supervisor相关的信息提交到 zookeeper集群上，Supervisor会去zookeeper集群上认领自己的Task，通知自己的Worker进程进行Task的处理。   
    和同样是计算框架的MapReduce相比，MapReduce集群上运行的是Job，而Storm集群上运行的是Topology。但是Job在运行结束之后会自行结束，Topology却只能被手动的kill掉，否则会一直运行下去   
    Storm不处理计算结果的保存，这是应用代码需要负责的事情，如果数据不大，你可以简单地保存在内存里，也可以每次都更新数据库，也可以采用NoSQL存储。这部分事情完全交给用户。   
    数据存储之后的展现，也是你需要自己处理的，storm UI 只提供对topology的监控和统计。



## 2.Stream

Storm的核心数据结构是tuple(元组)，本质上是包含了一个或多个键值对的列表。Stream是由无限制的tuple组成的序列。

## 3.spout

spout连接到数据源，将数据转化为一个个的tuple，并将tuple作为数据流进行发射。开发一个spout的主要工作就是利用API编写代码从数据源消费数据流。

 nextTuple() -- 这是Spout类中最重要的一个方法。发射一个Tuple到Topology都是通过这个方法来实现的。调用此方法时，storm向spout发出请求， 让spout发出元组（tuple）到输出器（ouput collector）。这种方法应该是非阻塞的，所以spout如果没有元组发出，这个方法应该返回。nextTuple、ack 和fail 都在spout任务的同一个线程中被循环调用。 当没有元组的发射时，应该让nextTuple睡眠一个很短的时间（如一毫秒），以免浪费太多的CPU。

spout的数据源可以有很多种来源：

web或者移动程序的点击流

社交网络的信息

传感器收集到的数据

应用程序产生的日志信息

spout通常只负责转换数据、发射数据，通常不会用于处理业务逻辑，从而可以很方便的实现spout的复用。

## 4.bolt

bolt主要负责数据的运算，将接收到的数据实施运算后，选择性的输出一个或多个数据流。

一个bolt可以接收多个由spout或其他bolt发射的数据流，从而可以组建出复杂的数据转换和处理的网络拓扑结构。

execute方法 -- 这是Bolt中最关键的一个方法，对于Tuple的处理都可以放到此方法中进行。具体的发送是通过emit方法来完成的。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。

bolt常见的典型功能：

过滤 连接和聚合 计算 数据库的读写

## 5 Topology运行流程

1. Storm提交后，会把代码首先存放到Nimbus节点的inbox目录下，之后，会把当前Storm运行的配置生成一个 stormconf.ser文件放到Nimbus节点的stormdist目录中，在此目录中同时还有序列化之后的Topology代码文件   
       (2) 在设定Topology所关联的Spouts和Bolts时，可以同时设置当前Spout和Bolt的executor数目和task数目，默认情况下， 一个Topology的task的总和是和executor的总和一致的。之后，系统根据worker的数目，尽量平均的分配这些task的执行。 worker在哪个supervisor节点上运行是由storm本身决定的   
       (3)任务分配好之后，Nimbus节点会将任务的信息提交到zookeeper集群，同时在zookeeper集群中会有workerbeats节点，这里存储了当前Topology的所有worker进程的心跳信息   
       (4)Supervisor 节点会不断的轮询zookeeper集群，在zookeeper的assignments节点中保存了所有Topology的任务分配信息、代码存储目 录、任务之间的关联关系等，Supervisor通过轮询此节点的内容，来领取自己的任务，启动worker进程运行   
       (5)一个Topology运行之后，就会不断的通过Spouts来发送Stream流，通过Bolts来不断的处理接收到的Stream流，Stream流是无界的。   
       最后一步会不间断的执行，除非手动结束Topology。

# 实例

## 1 wordcount

spoutOut

|  |
| --- |
| **public class** SentenceSpout **extends** BaseRichSpout{  **private** String[] **data**={**"i am wu kong"**,  **"500 years ago"**,  **"i am da nao tian gong"**,  **"but now"**,  **"i am behend wu zhi shan"**};  SpoutOutputCollector **collector**=**null**;  **public void** declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {  outputFieldsDeclarer.declare(**new** Fields(**"sentence"**));  }   **public void** open(Map map, TopologyContext topologyContext, SpoutOutputCollector spoutOutputCollector) {  **this**.**collector**=spoutOutputCollector;  }  **private int index** = 0;  **public void** nextTuple() { *// String str = data[index]; // collector.emit(new Values(str),index); // index = index == data.length - 1 ? 0 : index+1;* **if**(**index**<**data**.**length**){  String str = **data**[**index**];  **collector**.emit(**new** Values(str),**index**);  **index**++;  }  **try** {  睡眠 避免cpu浪费  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }   @Override  **public void** ack(Object msgId) {  System.***out***.println(msgId+**"发送成功"**);  **collector**.emit(**new** Values(**data**[(**int**) msgId]),msgId);  }   @Override  **public void** fail(Object msgId) {  **super**.fail(msgId+**"发送失败,尝试重发"**);  } } |

splitBolt

|  |
| --- |
| **public class** SpiltBolt **extends** BaseRichBolt{  @Override  **public void** cleanup() {  **super**.cleanup();  }   **private** OutputCollector **outputCollector**=**null**;   **public void** prepare(Map map, TopologyContext topologyContext, OutputCollector outputCollector) {  **this**.**outputCollector**=outputCollector;  }  **private int count**=0;  **public void** execute(Tuple tuple) {  **try** {  String sentence = tuple.getStringByField(**"sentence"**);  **if**(**"i am behend wu zhi shan"**.equals(sentence) && **count** < 3){  **count** ++;  **throw new** RuntimeException(**"发生错误，发送不出去~~~~"**);  }  String words[] = sentence.split(**" "**);  **for** (String word : words) {  **outputCollector**.emit(tuple, **new** Values(word));  }  **outputCollector**.ack(tuple);  }**catch**(Exception e){  e.printStackTrace();  **outputCollector**.fail(tuple);  }  }   **public void** declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {  outputFieldsDeclarer.declare(**new** Fields(**"word"**));  } } |

wordCount

|  |
| --- |
| **public class** WordCountBolt **extends** BaseRichBolt {  **private** Map<String,Integer> **map**=**new** HashMap<String, Integer>();  OutputCollector **collector**=**null**;  **public void** prepare(Map map, TopologyContext topologyContext, OutputCollector outputCollector) {  **this**.**collector**=outputCollector;  }   **public void** execute(Tuple tuple) {  **try** {  String word = tuple.getStringByField(**"word"**);  **map**.put(word, **map**.containsKey(word) ? **map**.get(word) + 1 : 1);  **collector**.emit(tuple, **new** Values(word, **map**.get(word)));  **collector**.ack(tuple);  }**catch** (Exception e){  **collector**.fail(tuple);  }  }   **public void** declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {  outputFieldsDeclarer.declare(**new** Fields(**"words"**,**"count"**));  } } |

reportBolt

|  |
| --- |
| **public class** ReportBolt **extends** BaseRichBolt {  Map<String,Integer> **map**=**new** HashMap<String, Integer>();  OutputCollector **collector**=**null**;  **public void** prepare(Map map, TopologyContext topologyContext, OutputCollector outputCollector) {  **this**.**collector**=outputCollector;  }   **public void** execute(Tuple tuple) {  **try** {  String word = tuple.getStringByField(**"words"**);  **int** count = tuple.getIntegerByField(**"count"**); *// System.out.println("单词" + word + "出现" + count + "次");* System.***out***.println(**"################单词"**+word+**"已经出现了"**+count+**"次"**);  **map**.put(word,count);  **collector**.ack(tuple);  }**catch** (Exception e){  **collector**.fail(tuple);  }  }   **public void** declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {   }   @Override  **public void** cleanup() {  System.***out***.println(**"aaaa——————————————————————————————————————————————————"**);  **for**(Map.Entry<String,Integer> entry:**map**.entrySet()){  System.***out***.println(**"单词"**+entry.getKey()+**"出现了---------"**+entry.getValue());  } System.***out***.println(**"bbbb—————————————————————————————————————————————————"**);  } } |

Main

|  |
| --- |
| **public class** StormMain {  **private static final** String ***SENTENCE\_SPOUT\_ID*** = **"sentence-spout"**;  **private static final** String ***SPLIT\_BOLT\_ID*** = **"split-bolt"**;  **private static final** String ***COUNT\_BOLT\_ID*** = **"count-bolt"**;  **private static final** String ***REPORT\_BOLT\_ID*** = **"report-bolt"**;  **private static final** String ***TOPOLOGY\_NAME*** = **"word-count-topology"**;   **public static void** main(String[] args) **throws** InterruptedException {  SentenceSpout spout=**new** SentenceSpout();  SpiltBolt spiltBolt=**new** SpiltBolt();  WordCountBolt wordCountBolt=**new** WordCountBolt();  ReportBolt reportBolt=**new** ReportBolt();   TopologyBuilder builder=**new** TopologyBuilder();  builder.setSpout(***SENTENCE\_SPOUT\_ID***,spout);  builder.setBolt(***SPLIT\_BOLT\_ID***,spiltBolt).shuffleGrouping(***SENTENCE\_SPOUT\_ID***);  builder.setBolt(***COUNT\_BOLT\_ID***,wordCountBolt).fieldsGrouping(***SPLIT\_BOLT\_ID***,**new** Fields(**"word"**));  builder.setBolt(***REPORT\_BOLT\_ID***,reportBolt).globalGrouping(***COUNT\_BOLT\_ID***);   Config config=**new** Config(); *// StormSubmitter.submitTopology(TOPOLOGY\_NAME,config,builder.createTopology());* LocalCluster cluster=**new** LocalCluster();  cluster.submitTopology(***TOPOLOGY\_NAME***,config,builder.createTopology());   Thread.*sleep*(10000);  cluster.killTopology(***TOPOLOGY\_NAME***);  cluster.shutdown();   } } |

# 3 Storm 可靠性和DRPC

Storm提供了数据流处理时的可靠性，所谓的可靠性是指spout发送的每个tuple都能够执行完整的处理过程。

## 1.spout的可靠性

spout需要记录它发射出去的tuple，当下游bolt处理tuple或子tuple失败时，spout能够重新发射该tuple。

bolt在处理tuple或子tuple时，无论是否成功都需要向上游节点进行报告或者报错。

而在ISpout接口中定义了三个可靠性相关的API:

nextTuple

ack

fail

每个bolt都要向上一级负责报告自己的处理结果，如果spout的直接子bolt，都向spout进行了确认应答，表明后续处理都完成，则spout会调用ack方法来表明该消息已经完全处理了。如果任何一个bolt处理tuple报错，或者处理超时，spout会调用fail方法。

## 2.bolt的可靠性

在发射衍生的tuple时，需要锚定下一级的tuple。

子bolt处理消息成功或者失败时分别发送确认应答或者报错给父tuple或父spout。

锚定的意思是建立tuple和衍生出的tuple之间的对应关系，这样下游的bolt可以通过应答确认/报错或超时来加入到tuple树结构中来。

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

\*\*此方法具有重载的方法，不锚定子节点，这种方式不会进行锚定操作，非锚定的tuple不会对数据流的可靠性起作用，如果一个非锚定的tuple在下游处理失败，原始的根tuple不会重新发送。

collector.emit(new Values(word));

当处理完成或者发送了新tuple之后，可靠数据流中的bolt需要应答读入的tuple:

this.collector.ack(tuple);

如果处理失败，则spout必须发射tuple,bolt就要明确的处理失败的tuple报错:

this.collector.fail(tuple);

如果因为超时的原因，或者显示调用OutputCollector.fail()方法，spout都会重新发送原始tuple。

## 3 drpc

|  |
| --- |
| \*\*LinearDRPCTopologyBuilder 已经过时 被Trident替代 以下内容暂缓  Storm里面引入DRPC主要是利用storm的实时计算能力来并行化CPU密集型（CPU intensive）的计算任务  DRPC其实不能算是storm本身的一个特性， 它是通过组合storm的原语stream、spout、bolt、 topology而成的一种模式(pattern)。  DRPC的storm topology以函数的参数流作为输入，而把这些函数调用的返回值作为topology的输出流(参考DRPC概述.jpg)。  Distributed RPC是由一个”DPRC服务器”协调(storm自带了一个实现)。DRPC服务器协调：① 接收一个RPC请求 ② 发送请求到storm topology ③ 从storm topology接收结果 ④ 把结果发回给等待的客户端。从客户端的角度来看一个DRPC调用跟一个普通的RPC调用没有任何区别。  案例：改造如上单词计数案例，通过DRPC机制实现单词结果查询  */\*\*  \* 在执行DRPC的过程中，execute方法接受的tuple中具有n+1个值，  \* 第一个为request-id即请求的编号，后n个字段是请求的参数  同时要求我们topology的最后一个bolt发送一个形如[id, result]的二维tuple：  第一个field是request-id，第二个field是这个函数的结果。最后所有中间tuple的第一个field必须是request-id。  \* Created by gao on 2016/12/23.  \*/* **public class** ExclaimBolt **extends** BaseRichBolt {  OutputCollector **collector**=**null**;  @Override  **public void** prepare(Map map, TopologyContext topologyContext, OutputCollector outputCollector) {  **this**.**collector**=outputCollector;  }  @Override  **public void** execute(Tuple tuple) {  String str=tuple.getString(1);  **collector**.emit(**new** Values(tuple.getValue(0),tuple+**"!"**));  }  @Override  **public void** declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {  outputFieldsDeclarer.declare(**new** Fields(**"id"**,**"result"**));  } }  //服务端：  **public class** DRPCDriver {  **public static void** main(String[] args) **throws** AlreadyAliveException, InvalidTopologyException {  *//--指定一个被调用方法的名字* LinearDRPCTopologyBuilder builder=**new** LinearDRPCTopologyBuilder(**"exec"**);  *//--设置要被调用topology中的执行bolt，可以设置多个* builder.addBolt(**new** ExclaimBolt());  Config config=**new** Config();  *//提交drpcTopology到服务器* StormSubmitter.*submitTopology*(**"drpc\_topo"**,config,builder.createRemoteTopology());  *//本地模拟远程调用 // LocalDRPC localDRPC=new LocalDRPC(); // LocalCluster cluster=new LocalCluster(); // cluster.submitTopology("drpc\_topology",config,builder.createLocalTopology(localDRPC)); // String result=localDRPC.execute("exec","hello"); // System.out.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_----------------------result"+result); // cluster.shutdown(); // localDRPC.shutdown();* } }  客户端：  **public class** DrpcClient {  **public static void** main(String[] args) **throws** TException, DRPCExecutionException {  DRPCClient client=**new** DRPCClient(**"192.168.8.107"**,3772);  String result=client.execute(**"exec"**,**"aaa"**);  System.***out***.println(result);  } } |

# 4 Trident