# 1 环境配置

大数据环境配置，包括hadoop、spark、kafka、storm、flink、hive等

## 1.1 IDEA基础环境配置

### 1 mavena安装

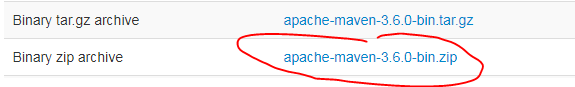
官方网址

<http://maven.apache.org/download.cgi>

参考文档：

<https://blog.csdn.net/westos_linux/article/details/78968012>

下载版本：选择binary zip格式文件（windows）（Linux下选择tar.gz格式）



#### 配置环境变量

新建环境变量

MAVEN\_HOME，赋值D:\Program Files\Apache\maven

编辑环境变量Path

追加 %MAVEN\_HOME%\bin\;

检查： mvn -v

#### 修改配置文件

在conf/settings.xml 文件下添加：

<!-- 配置本地仓库 -->

<localRepository>D:\studysoft\maven\repository</localRepository>

提前创建好文件夹D:\studysoft\maven\repository

备注：jdk最好配置jdk1.8版本

# 1 Storm

## 1.1 storm 基础

### 1 storm介绍

storm是Twitter开源的分布式大数据处理框架，实时版Hadoop.

Storm对于实时计算的意义类似于Hadoop对于批处理的意义。

hadoop Map Reduce

storm Spout Bolt

#### storm应用场景

1）流数据处理：Storm处理源源不断的消息，结果保存到持久化介质中。

2）分布式RPC：由于Storm的处理组件都是分布式的，而且处理延迟都极低，所以可以Storm可以做为一个通用的分布式RPC框架来使用。

\* 实际应用

推荐系统：实时推荐

金融系统

预警系统

网站统计：实时销量、流量统计

#### 特性

1 使用场景广

实时处理消息，更新DB.持续计算等

2 可伸缩性高

3 保证无数据丢失

4 异常健壮

5 容错性好

6 语言无关性

storm的topology和消息处理组件Bolt可以用任何语言定义.

#### Storm VS hadoop

hadoop 离线批处理

storm 实时处理

\* 处理过程

Hadoop：Map Reduce

Storm: Spout Bolt

\* 进程是否结束：

Hadoop -- 电梯 -- job结束，进程结束

Storm -- 扶梯 -- 进程Topology不会结束，除非人工stop

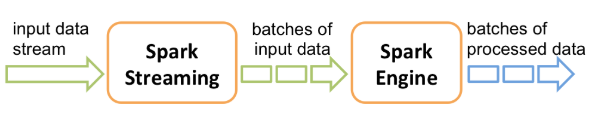
\* 处理速度

storm 实时在线处理，速度必须快

hadoop 离线处理，速度慢一点

#### Storm VS Spark Streaming

spark



spark Streaming并非是真正的实时处理系统，而是小批处理，近似实时；

但是spark streaming处理后的结果可以直接给机器学习、spark SQL操作，集成；便捷高效，而storm处理后的数据需要落地，相对要麻烦一点。

storm适合实时性很高，处理后就可以落地的场景

Storm优势

\* 编程模型

\* 扩展性

\* 可靠性

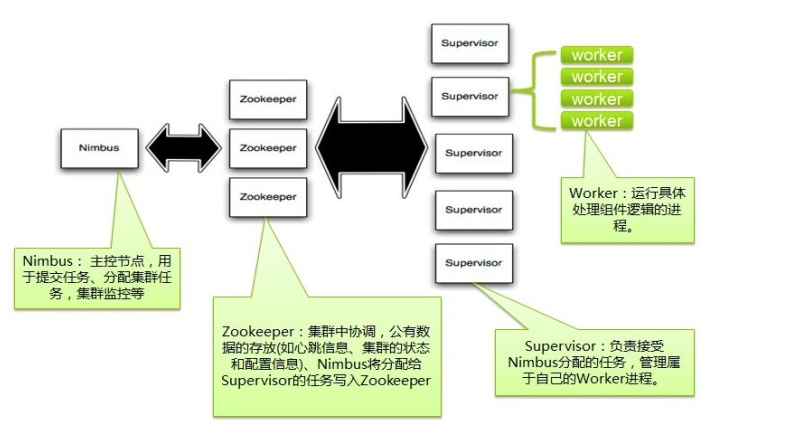
主节点挂了，启用备用节点

\* 容错性

\* 多语言

### 2 storm原理

#### 1 集群结构



Storm集群采用主从架构方式，主节点是Nimbus，从节点是Supervisor，有关调度相关的信息存储到ZooKeeper集群中

**\* Nimbus**

主控节点，用于提交任务、分配集群任务，集群监控等

**\* zookeeper**

集群中协调、共有数据的存放（如心跳信息、集群的状态和配置信息），Nimbus将分配给Supervisro的任务写进Zookeeper。

**\* Supervisor**

负责接受Nimbus分配的任务，管理属于自己的Worker进程。

Nimbus和Supervisor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。这也就意味着你可以用kill -9来杀死nimbus和supervisor进程， 然后再重启它们，它们可以继续工作， 就好像什么都没有发生过似的。这个设计使得storm不可思议的稳定。

storm集群和hadoop集群很像，但是hadoop运行MapReduce的Job，而storm运行Topology。

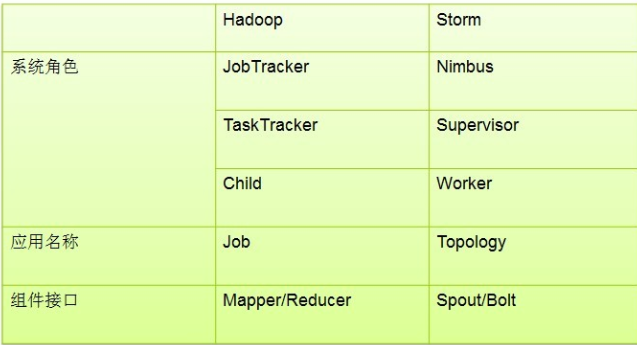
关键区别：一个MapReduce Job最终会结束， 而一个Topology运永远运行（除非你显式的杀掉他）。

在Storm的集群里面有两种节点： 控制节点(master node)和工作节点(worker node)。

master node：-->后台程序：Nimbus，类似Hadoop中的JobTracker。

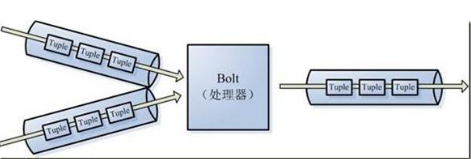
worker node:--> 运行Supervisor的节点（类似 TaskTracker）。每一个工作进程执行一个Topology（类似 Job）的一个子集；一个运行的Topology由运行在很多机器上的很多工作进程 Worker（类似 Child）组成。

**storm topology结构**



#### 2 Stream

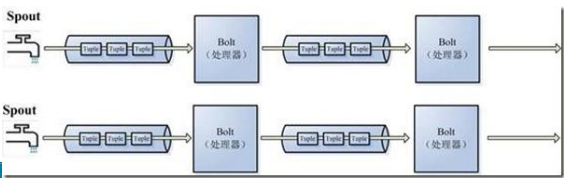
在Storm中有对于流stream的抽象，流是一个不间断的无界的连续tuple，注意Storm在建模事件流时，把流中的事件抽象为tuple即元组。



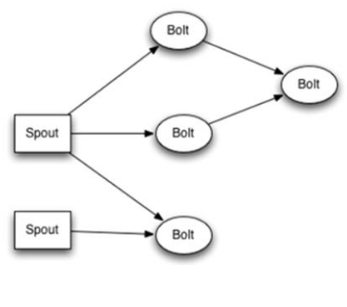
Storm认为每个stream都有一个源，也就是原始元组的源头，叫做Spout（管口）

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。



有向无环图 -->抽象为-->Topology(拓扑) -->storm的Job的抽象概念

图中每个节点是一个spout或者bolt，每个spout或者bolt发送元组到下一级组件，广播方式。

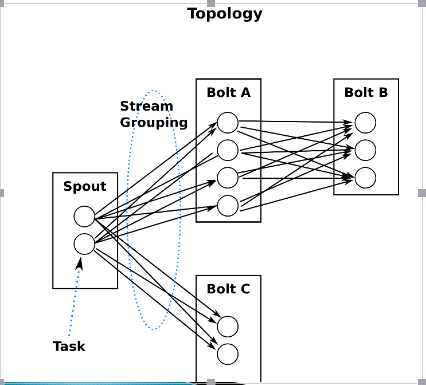
#### 3 Topology

storm:

流中的元素抽象为tuple，一个tuple --> 一个值列表value list

每个value有一个name，每个value可序列化

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。

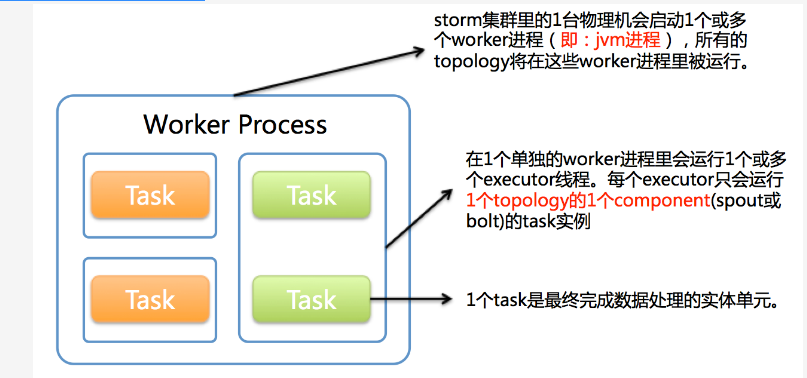


一个Storm在集群上运行一个Topology时，主要通过以下3个实体来完成Topology的执行工作：

(1). Worker（进程）

(2). Executor（线程）

(3). Task



#### 4 角色概念

Streams：消息流

消息流是一个没有边界的tuple序列。

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。



**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。

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。

#### 5 数据模型

storm使用tuple来作为它的数据模型。每个tuple是一堆值，每个值有一个名字，并且每个值可以是任何类型，总体来看，storm支持所有的基本类型、字符串以及字节数组作为tuple的值类型。也可以自定义类型， 只要实现序列化器(serializer)。

一个Tuple代表数据流中的一个基本的处理单元，例如一条cookie日志，它可以包含多个Field，每个Field表示一个属性。

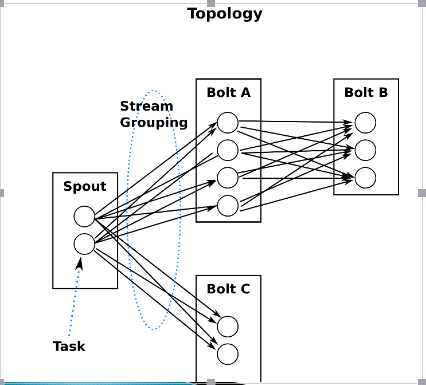


Tuple本来应该是一个Key-Value的Map，由于各个组件间传递的tuple的字段名称已经事先定义好了，所以Tuple只需要按序填入各个Value，所以就是一个Value List。

一个没有边界的、源源不断的、连续的Tuple序列就组成了Stream。

#### 6 Storm Streaming Grouping

Storm中最重要的抽象，应该就是Stream grouping了，它能够控制Spot/Bolt对应的Task以什么样的方式来分发Tuple，将Tuple发射到目的Spot/Bolt对应的Task



类型：

Shuffle Grouping ：随机分组，尽量均匀分布到下游Bolt中

将流分组定义为混排。这种混排分组意味着来自Spout的输入将混排，或随机分发给此Bolt中的任务。shuffle grouping对各个task的tuple分配的比较均匀。

Fields Grouping ：

按字段分组，按数据中field值进行分组；相同field值的Tuple被发 送到相同的Task。

这种grouping机制保证相同field值的tuple会去同一个task，这对于WordCount来说非常关键，如果同一个单词不去同一个task，那么统计出来的单词次数就不对了。“if the stream is grouped by the “user-id” field, tuples with the same “user-id” will alwaysGo to the same task”. —— 小示例

All grouping ：广播

广播发送， 对于每一个tuple将会复制到每一个bolt中处理。

Global grouping ：

全局分组，Tuple被分配到一个Bolt中的一个Task，实现事务性的Topology。

Stream中的所有的tuple都会发送给同一个bolt任务处理，所有的tuple将会发送给拥有最小task\_id的bolt任务处理。

None grouping ：不分组

不关注并行处理负载均衡策略时使用该方式，目前等同于shuffle grouping,另外storm将会把bolt任务和他的上游提供数据的任务安排在同一个线程下。

Direct grouping ：直接分组 指定分组

最常用：

Shuffle Grouping 、Fields Grouping、Direct grouping

### 3 编程模型

#### 3.1 常用接口类

**1 ISpout接口**

\* 概述

核心接口，负责将数据发送到topology中去处理。

storm会跟踪Spout发出去的tuple的DAG

\* 核心方法

open: 初始化操作

close: 资源释放操作

nextTuple: 发送数据 -- core

ack：tuple处理成功，storm会反馈给spout一个成功的消息

fail：tuple处理失败，storm会发送一个消息给Spout，处理失败

ack/fail/nextTuple是在同一个线程中执行，所以不用考虑线程安全问题

查找接口的实现类：

IDEA 风格 ctrl + alt +B

\* 实现类

public abstract class BaseRichSpout extends BaseComponent implements IRichSpout

public interface IRichSpout extends ISpout,IComponent

DRPCSpout

ShellSpout

**2 IComponent接口**

概述：

public interface IComponent extends Serializable

为topology中所有可能的组件提供公用的方法

void declareOutputFields(OutputFieldsDeclarer declarer);

用于声明当前Spout/Bolt发送的tuple的名称

使用OutputFieldsDeclarer配合使用

实现类：

public abstract class BaseComponent implements IComponent

**3 IBolt接口**

\* 概述

职责：接收tuple，并进行相应的处理(filter/join/....)

hold住tuple再处理

IBolt会在一个运行的机器上创建，使用Java序列化它，然后提交到主节点(nimbus)上去执行。

nimbus会启动worker来反序列化，调用prepare方法，然后才开始处理tuple处理

\* 方法

prepare：初始化

execute：处理一个tuple，tuple对象中包含了元数据信息

cleanup：shutdown之前的资源清理操作

\* 实现类：

public abstract class BaseRichBolt extends BaseComponent implements IRichBolt {}

public interface IRichBolt extends IBolt, IComponent{}

RichShellBolt

#### 3.2 实现累加案例

求和案例

需求：

实现 1+2+3+…… = ？？？

实现方案

Spout发送数字作为input

使用Blot来处理业务逻辑：求和

将结果输出到控制台

拓扑设计：

DataSourceSpout --> SumBolt

代码：

**import** org.apache.storm.utils.Utils;  
**import** org.apache.storm.Config;  
**import** org.apache.storm.LocalCluster;  
**import** org.apache.storm.spout.SpoutOutputCollector;  
**import** org.apache.storm.task.OutputCollector;  
**import** org.apache.storm.task.TopologyContext;  
**import** org.apache.storm.topology.OutputFieldsDeclarer;  
**import** org.apache.storm.topology.TopologyBuilder;  
**import** org.apache.storm.topology.base.BaseRichBolt;  
**import** org.apache.storm.topology.base.BaseRichSpout;  
**import** org.apache.storm.tuple.Fields;  
**import** org.apache.storm.tuple.Tuple;  
**import** org.apache.storm.tuple.Values;  
  
**import** java.util.Map;  
  
*/\*\*  
 \* storm实现累积求和操作  
 \*/***public class** LocalSumStormTopology {  
  
 */\*\*  
 \* Spout 需要继承BaseRichSpout  
 \*数据源需要产生数据并发射  
 \*/* **public static class** DataSourceSpout **extends** BaseRichSpout{  
 **private** SpoutOutputCollector **collector**;  
  
 */\*\*初始化方法，只会被调用一次  
 \** ***@param conf*** *配置参数  
 \** ***@param context*** *上下文 --理解为一个框  
 \** ***@param collector*** *数据发射器  
 \*/* **public void** open(Map conf, TopologyContext context, SpoutOutputCollector collector) {  
 **this**.**collector** = collector;  
 }  
  
 **int number** = 0;  
  
 */\*\*  
 \* 会产生数据，在生产上是从消息队列中获取数据  
 \* 这个方法是一个死循环，会一直不停往下执行  
 \*/* **public void** nextTuple() {  
 **this**.**collector**.emit(**new** Values(++**number**));  
 System.***out***.println(**"Spout:"** + **number**);  
 *//防止数据产生太快* Utils.*sleep*(1000);  
 }  
  
 */\*\*声明输出字段  
 \** ***@param declarer*** *\*/* **public void** declareOutputFields(OutputFieldsDeclarer declarer) {  
 declarer.declare(**new** Fields(**"num"**));  
 }  
 }  
  
 */\*\*  
 \* 数据累计求和Bolt：就收数据并处理  
 \*/* **public static class** SumBolt **extends** BaseRichBolt{  
  
 */\*\*初始化方法，会被执行一次  
 \** ***@param stormConf*** *\** ***@param context*** *\** ***@param collector*** *\*/* **public void** prepare(Map stormConf, TopologyContext context, OutputCollector collector) {  
 }  
 **int sum** = 1;  
 */\*\*其实也是一个死循环，职责：获取Spout发送过来的数据  
 \** ***@param input*** *\*/* **public void** execute(Tuple input) {  
 *//Bolt中获取值可以根据index获取，也可以根据上一个环节定义的field的名称获取* Integer value = input.getIntegerByField(**"num"**);  
 **sum** += value;  
 System.***out***.println(**"Bolt:sum = ["** + **sum** + **"]"**);  
 }  
  
 **public void** declareOutputFields(OutputFieldsDeclarer declarer) {  
  
 }  
 }  
  
 **public static void** main(String[] args) {  
 *//TopologyBuilder根据Spout和Bolt来构建出Topology  
 //storm中任何一个作业都是通过 Topology的方式进行提交的  
 //topology中需要制定Spout和Bolt的执行顺序* TopologyBuilder builder = **new** TopologyBuilder();  
 builder.setSpout(**"DataSourceSpout"**,**new** DataSourceSpout());  
 builder.setBolt(**"SumBolt"**,**new** SumBolt()).shuffleGrouping(**"DataSourceSpout"**);  
  
 *//创建一个本地Storm集群：本地模式运行，不需要搭建Storm集群* LocalCluster cluster = **new** LocalCluster();  
 cluster.submitTopology(**"LocalSumStormTopology"**,**new** Config(),builder.createTopology());  
  
 }  
}

总结：storm实现累计求和操作

\* Spout类--DataSourceSpout：数据源，产生数据并发射

\* 继承BaseRichSpout类

重写open(conf,context,collector)方法

conf配置参数

context 上下文

collector 数据发射器

重写nextTuple方法

死循环，产生数据

重写declareOutputFields方法

\* Bolt类--SumBolt：接收数据并处理

\* 继承BaseRichBolt方法

prepare方法 -- 为空

execute方法 -- 死循环，获取Spout发送过来的数据

declareOutputFields -- 为空

\* Main方法

构建Topology

#### 3.3 词频统计案例

需求：

读取指定目录的数据，并实现单词计数功能

实现方案：

Spout读取指定目录的数据源，作为后续Bolt处理的input

使用一个Bolt把input的数据，切割开，按照 逗号 , 进行分割

使用一个Bolt来进行最终单词次数的统计

并输出

拓扑设计：

DataSourceSpout --> SplitBolt --> CountBolt

代码如下：

*/\*\*  
 \* 使用Storm完成词频统计功能  
 \*/***public class** LocalWordCountStormTopology {  
 **public static class** DataSourceSpout **extends** BaseRichSpout {  
 **private** SpoutOutputCollector **collector**;  
 **public void** open(Map conf, TopologyContext context, SpoutOutputCollector collector) {  
 **this**.**collector** = collector;  
 }  
 */\*\*  
 \* 业务：  
 \* 1） 读取指定目录的文件夹下的数据:/Users/rocky/data/storm/wc  
 \* 2） 把每一行数据发射出去  
 \*/* **public void** nextTuple() {  
 *// 获取所有文件* Collection<File> files = FileUtils.*listFiles*(**new** File(**"F:\\test\\storm\\wc"**),  
 **new** String[]{**"txt"**},**true**);  
 **for**(File file : files) {  
 **try** {  
 *// 获取文件中的所有内容* List<String> lines = FileUtils.*readLines*(file);  
 *// 获取文件中的每行的内容* **for**(String line : lines) {  
 *// 发射出去* **this**.**collector**.emit(**new** Values(line));  
 }  
 *//* ***TODO... 数据处理完之后，改名，否则一直重复执行*** FileUtils.*moveFile*(file, **new** File(file.getAbsolutePath() + System.*currentTimeMillis*()));  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 **public void** declareOutputFields(OutputFieldsDeclarer declarer) {  
 declarer.declare(**new** Fields(**"line"**));  
 }  
 }  
 */\*\*  
 \* 对数据进行分割  
 \*/* **public static class** SplitBolt **extends** BaseRichBolt {  
 **private** OutputCollector **collector**;  
 **public void** prepare(Map stormConf, TopologyContext context, OutputCollector collector) {  
 **this**.**collector** = collector;  
 }  
 */\*\*  
 \* 业务逻辑：  
 \* line： 对line进行分割，按照逗号  
 \*/* **public void** execute(Tuple input) {  
 String line = input.getStringByField(**"line"**);  
 String[] words = line.split(**","**);  
 **for**(String word : words) {  
 **this**.**collector**.emit(**new** Values(word));  
 }  
 }  
 **public void** declareOutputFields(OutputFieldsDeclarer declarer) {  
 declarer.declare(**new** Fields(**"word"**));  
 }  
 }  
 */\*\*  
 \* 词频汇总Bolt  
 \*/* **public static class** CountBolt **extends** BaseRichBolt {  
 **public void** prepare(Map stormConf, TopologyContext context, OutputCollector collector) {  
 }  
 Map<String,Integer> **map** = **new** HashMap<String, Integer>();  
 */\*\*  
 \* 业务逻辑：  
 \* 1）获取每个单词  
 \* 2）对所有单词进行汇总  
 \* 3）输出  
 \*/* **public void** execute(Tuple input) {  
 *// 1）获取每个单词* String word = input.getStringByField(**"word"**);  
 Integer count = **map**.get(word);  
 **if**(count == **null**) {  
 count = 0;  
 }  
 count ++;  
 *// 2）对所有单词进行汇总* **map**.put(word, count);  
 *// 3）输出* System.***out***.println(**"~~~~~~~~~~~~~~~~~~~~~~"**);  
 Set<Map.Entry<String,Integer>> entrySet = **map**.entrySet();  
 **for**(Map.Entry<String,Integer> entry : entrySet) {  
 System.***out***.println(entry);  
 }  
 }  
  
 **public void** declareOutputFields(OutputFieldsDeclarer declarer) {  
 }  
 }  
 **public static void** main(String[] args) {  
 *// 通过TopologyBuilder根据Spout和Bolt构建Topology* TopologyBuilder builder = **new** TopologyBuilder();  
 builder.setSpout(**"DataSourceSpout"**, **new** DataSourceSpout());  
 builder.setBolt(**"SplitBolt"**, **new** SplitBolt()).shuffleGrouping(**"DataSourceSpout"**);  
 builder.setBolt(**"CountBolt"**, **new** CountBolt()).shuffleGrouping(**"SplitBolt"**);  
 *// 创建本地集群* LocalCluster cluster = **new** LocalCluster();  
 cluster.submitTopology(**"LocalWordCountStormTopology"**,  
 **new** Config(), builder.createTopology());  
 }  
}

总结：

1. DataSourceSpout类

open方法

nextTuple()方法：读取指定目录的数据，并把数据按行发送出去

declareOutputFields()方法：指定输出field名称

1. SplitBolt类：对数据进行分割

prepare方法

execute方法：对line进行分割，按照逗号

declareOutputFields()方法：指定输出fields方法

1. 词频汇总Bolt：获取每个单词；对所有单词汇总；输出

prepare方法为空

execute方法：最关键

declareOutputFields方法为空

1. main方法内通过TopologyBuilder根据Spout和Bolt构建Topology

#### 4 Storm周边框架

**1 logstash基础**

logstash是一个开源的服务器端数据处理管道，它可以从多个源中提取数据，对其进行转换，然后将其发送到需要的存储中去（elasticsearch）

# 解压后可使用

bin/logstash -e 'input { stdin { } } output { stdout {} }'

可在控制台输入：hadoop

以json格式输出

bin/logstash -e 'input { stdin { } } output { stdout {codec => json} }'

### 4 storm架构及部署

#### 1 Storm架构

类似于Hadoop的架构，主从(Master/Slave)

Nimbus: 主

集群的主节点，负责任务(task)的指派和分发、资源的分配

Supervisor: 从

可以启动多个Worker，具体几个呢？可以通过配置来指定

一个Topo可以运行在多个Worker之上，也可以通过配置来指定

集群的从节点，(负责干活的)，负责执行任务的具体部分

启动和停止自己管理的Worker进程

无状态，在他们上面的信息(元数据)会存储在ZK中

Worker: 运行具体组件逻辑(Spout/Bolt)的进程

=====================分割线===================

task：

Spout和Bolt

Worker中每一个Spout和Bolt的线程称为一个Task

executor： spout和bolt可能会共享一个线程

#### 2 storm安装

Storm部署的前置条件

jdk7+

python2.6.6+

storm版本：storm-1.2.2

$ tar -zxvf apache-storm-1.2.2.tar.gz

添加环境变量

$ sudo vim /etc/profile

# STORM\_HOME

export STORM\_HOME=/home/hadoop/app/storm

export export PATH=$STORM\_HOME/bin:$PATH

$ source /etc/profile

$ vim conf/storm-env.sh

export JAVA\_HOME=/home/hadoop/app/jdk

#### 3 storm单机启动

Storm启动

$STORM\_HOME/bin/storm 如何使用 执行storm就能看到很多详细的命令

dev-zookeeper 启动zk

storm dev-zookeeper 前台启动

nohup sh bin/storm dev-zookeeper & ##后台启动

jps ： dev\_zookeeper

nimbus 启动主节点

nohup sh storm nimbus &

supervisor 启动从节点

nohup sh storm supervisor &

ui 启动UI界面

nohup sh storm ui &

jps:core

网页登录：http://hadoop1:8080/

logviewer 启动日志查看服务

nohup sh storm logviewer &

storm list --> 查看运行中的任务

storm kill topology-name [-w wait-time-secs] --> 杀死名为topology-name的拓扑

storm drpc --> 启动一个DRPC守护进程

备注：config\_value表示正在启动

#### 4 storm作业提交

（1）提交作业到storm集群

storm jar jar包名称 类名 参数（可多），回车

storm jar /usr/local/storm-wordcount.jar com.kang.eshop.storm.WordCountTopology wordCountTopology

storm jar storm-wordcount.jar com.kang.eshop.storm.WordCountTopology wordCountTopology

（2）在storm ui上观察storm作业的运行

<http://139.199.10.125:8080/index.html>

问题：页面观察到3个executor,spout 1个，bolt 1个，还有一个呢？

如何修改将跑在本地的storm app改成运行在集群上的

StormSubmitter.submitTopology(topoName,new Config(), builder.createTopology());

#### 5 集群部署

1集群部署规划

hadoop1: zk nimbus supervisor

hadoop2: zk supervisor

hadoop2: zk supervisor

2 分发软件包

scp /home/hadoop/app/storm hadoop@hadoop2:/home/hadoop/app

scp /home/hadoop/app/storm hadoop@hadoop3:/home/hadoop/app

3 启动zookeeper

bin/zkServer.sh start

bin/zkServer.sh status

4 集群配置

Storm集群

$STORM\_HOME/conf/storm.yaml

storm.zookeeper.servers:

- "hadoop000"

- "hadoop001"

- "hadoop002"

storm.local.dir: "/home/hadoop/app/tmp/storm"

supervisor.slots.ports:

- 6700

- 6701

- 6702

- 6703

5 启动

启动

hadoop000: nimbus supervisor(ui,logviewer)

hadoop001: supervisor(logviewer)

hadoop002: supervisor(logviewer)

nimbus 启动主节点 -- 执行在hadoop1

nohup sh storm nimbus &

ui 启动UI界面 -- 执行在hadoop1

nohup sh storm ui &

supervisor 启动从节点 --执行在3台机器上

nohup sh storm supervisor &

logviewer 启动日志查看服务 --执行在3台机器上

nohup sh storm logviewer &

## 查看进程

[hadoop@hadoop1 bin]$ jps

7120 QuorumPeerMain

7220 nimbus

7623 logviewer

7495 Supervisor

7724 Jps

7373 core

[hadoop@hadoop2 bin]$ jps

6131 Jps

6036 logviewer

5769 QuorumPeerMain

5898 Supervisor

[hadoop@hadoop3 bin]$ jps

3970 Jps

3875 logviewer

3766 Supervisor

3631 QuorumPeerMain

提交任务到集群：

storm jar /home/hadoop/lib/storm-1.0.jar com.imooc.bigdata.ClusterSumStormTopology

### 5.2 深入理解storm

#### 1 并行度概念

storm的并行是由非常多的supervisor完成的。

storm的supervisor运行的是topology中的spout/bolt task。

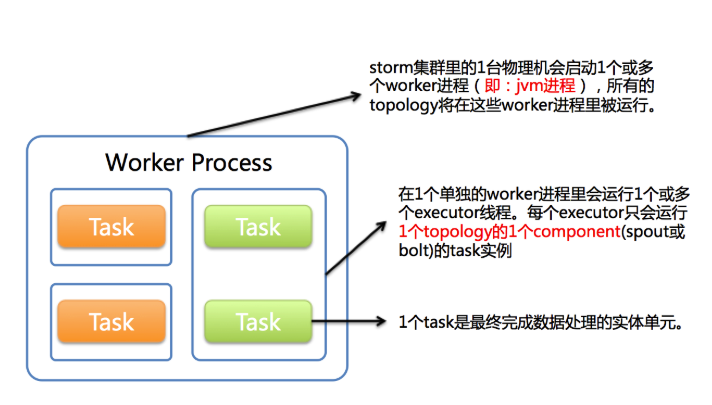
task是storm中进行计算的最小的运行单位，表示的是spout、bolt的运行实例。

程序执行的最大粒度的运行单位是进程。在supervisor中，运行task的进程称作worker。

supervisor节点上可以运行非常多的worker。

在worker中可以运行线程的，这些线程称作executor。在executor中，运行task。

总结一下，supervisor(节点)>worker(进程)>executor(线程)>task(实例)



#### 2 将集群设置为单机模式

在conf/storm.yaml 中修改

storm.zookeeper.servers:

- "hadoop1"

# - "hadoop2"

# - "hadoop3"

#### 3 并行度设置

supervisor:

在哪个节点上启动supervisor 则就会产生supervisor进程

worker

storm.yaml中，如下指定了 worker进程的端口，以及当前机器下能运行的work数量。

worker进程数量也可以通过config.setNumWorkers(workers)设置。(优先级更高)

executor设置：

builder.setSpout(id, spout, parallelism\_hint)： parallelism\_hint设置spout的数量

builder.setBolt(id, bolt, parallelism\_hint)设置的。parallelism\_hint设置bolt的数量

task的设置：

task是通过 spout/boltDeclarer.setNumTasks(num)设置对应spout/bolt的task个数。

默认情况下，每个supervisor启动4个worker，每个worker启动1个executor，每个executor中会有1个task。即每台机器启动4个进程，每个进程里面跑一个线程，每个线程来具体执行一个 bolt/spout的实例。

#### 4 实例

**public class** MyLocalStormTopology {  
 */\*\*  
 \* 组装火车 轨道 并让火车在轨道上行驶  
 \*  
 \** ***@throws*** *InterruptedException  
 \*/* **public static void** main(String[] args) **throws** Exception {  
  
 *// 祖品列车* TopologyBuilder topologyBuilder = **new** TopologyBuilder();  
 SpoutDeclarer spoutDeclarer = topologyBuilder.setSpout(**"1"**, **new** MySpout2()); *// 定义1号车厢  
 //spoutDeclarer.setNumTasks(val);// 设置task数量  
 //topologyBuilder.setSpout(id, spout, parallelism\_hint); // 动态设置 spout的executor数量* BoltDeclarer boltDeclarer = topologyBuilder.setBolt(**"2"**, **new** MyBolt1()).shuffleGrouping(**"1"**);*// 定义2号车厢 并和1号车厢连接起来  
 //boltDeclarer.setNumTasks(num); 动态设置 task数量  
 //topologyBuilder.setBolt(id, bolt, parallelism\_hint); 动态设置 bolt的executor数量  
  
 // 造出轨道 在集群中运行* StormSubmitter stormSubmitter = **new** StormSubmitter();*// storm集群执行  
 //HashMap conf = new HashMap();* Config config = **new** Config();  
 *//config.setNumWorkers(workers); 动态设置每台机器运行work进程数* stormSubmitter.*submitTopology*(MyLocalStormTopology.**class**.getSimpleName(), config, topologyBuilder.createTopology());  
  
 }  
}

6后续学习课程

hbase存储原理

<https://www.imooc.com/learn/996>

电商大数据应用之用户画像

<https://www.imooc.com/learn/460>