# storm从入门到精通

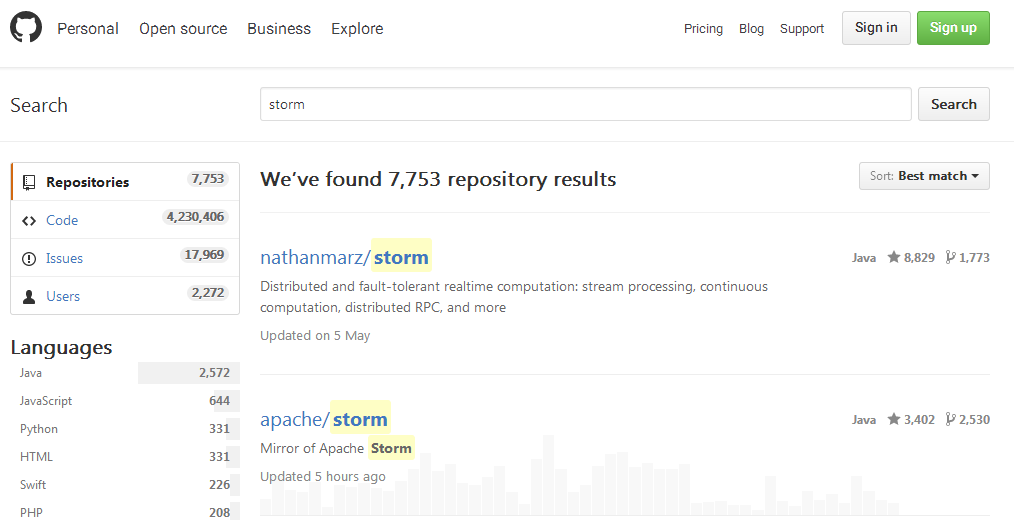
## storm是什么

### 背景-流式计算与storm

2011年在海量数据处理领域，Hadoop是人们津津乐道的技术，Hadoop不仅可以用来存储海量数据，还以用来计算海量数据。因为其高吞吐、高可靠等特点，很多互联网公司都已经使用Hadoop来构建数据仓库，高频使用并促进了Hadoop生态圈的各项技术的发展。一般来讲，根据业务需求，数据的处理可以分为离线处理和实时处理，在离线处理方面Hadoop提供了很好的解决方案，但是针对海量数据的实时处理却一直没有比较好的解决方案。

就在人们翘首以待的时间节点，storm横空出世，与生俱来的分布式、高可靠、高吞吐的特性，横扫市面上的一些流式计算框架，渐渐的成为了流式计算的首选框架。

如果庞麦郎在的话，他一定会说，这就是我要的滑板鞋！



在2013年，阿里巴巴开源了基于storm的设计思路使用java重现编写的流式计算框架jstorm。那jstorm是什么呢？

在jstorm早期的介绍中，一般会出现下面的语句：JStorm 比Storm更稳定，更强大，更快，Storm上跑的程序，一行代码不变可以运行在JStorm上。

在最新的介绍中，jstorm的团队是这样介绍的：JStorm 是一个类似Hadoop MapReduce的系统， 用户按照指定的接口实现一个任务，然后将这个任务递交给JStorm系统，Jstorm将这个任务跑起来，并且按7 \* 24小时运行起来，一旦中间一个Worker 发生意外故障， 调度器立即分配一个新的Worker替换这个失效的Worker。

因此，从应用的角度，JStorm 应用是一种遵守某种编程规范的分布式应用。

从系统角度， JStorm一套类似MapReduce的调度系统。 从数据的角度， 是一套基于流水线的消息处理机制。实时计算现在是大数据领域中最火爆的一个方向，因为人们对数据的要求越来越高，实时性要求也越来越快，传统的Hadoop MapReduce，逐渐满足不了需求，因此在这个领域需求不断。现在，Jstom在淘宝海量的数据和大量的业务场景的锤炼下，从开始的追随者，使用者慢慢的演变成了流式计算技术的领导者。当下，还有很多企业并不知道jstorm，他们的生产环境依然是storm，并且storm也在不断更新，在笔者成文的时间点上，storm发布了1.0的beta版。

鉴于大多数企业的生产环境还在使用storm，我们学习的目标还是切换到Apache基金会的storm上来。

### 背景-storm是为了解决什么问题

伴随着信息科技日新月异的发展，信息呈现出爆发式的膨胀，人们获取信息的途径也更加多样、更加便捷，同时对于信息的时效性要求也越来越高。

举个搜索场景中的例子，当一个卖家发布了一条宝贝信息时，他希望的当然是这个宝贝马上就可以被卖家搜索出来、点击、购买啦，相反，如果这个宝贝要等到第二天或者更久才可以被搜出来，估计这个大哥就要骂娘了。

再举一个推荐的例子，如果用户昨天在淘宝上买了一双袜子，今天想买一副泳镜去游泳，但是却发现系统在不遗余力地给他推荐袜子、鞋子，根本对他今天寻找泳镜的行为视而不见，估计这哥们心里就会想推荐你妹呀。其实稍微了解点背景知识的码农们都知道，这是因为后台系统做的是每天一次的全量处理，而且大多是在夜深人静之时做的，那么你今天白天做的事情当然要明天才能反映出来啦。

### 背景-实现实时计算系统需要解决哪些问题

如果让我们自己设计一个实时计算系统，我们要解决哪些问题。

1. 低延迟：都说了是实时计算系统了，延迟是一定要低的。
2. 高性能：性能不高就是浪费机器，浪费机器是要受批评的哦。
3. 分布式：系统都是为应用场景而生的，如果你的应用场景、你的数据和计算单机就能搞定，那么不用考虑这些复杂的问题了。我们所说的是单机搞不定的情况。
4. 可扩展：伴随着业务的发展，我们的数据量、计算量可能会越来越大，所以希望这个系统是可扩展的。
5. 容错：这是分布式系统中通用问题。一个节点挂了不能影响我的应用。
6. 通信：设计的系统需要应用程序开发人员考虑各个处理组件的分布、消息的传递吗？如果是，发人员可能会用不好，也不会想去用。
7. 消息不丢失：用户发布的一个宝贝消息不能在实时处理的时候给丢了，对吧？

### 离线计算是什么

离线计算：批量获取数据、批量传输数据、**周期性**批量计算数据、数据展示

代表技术：Sqoop批量导入数据、HDFS批量存储数据、MapReduce批量计算数据、Hive批量计算数据、\*\*\*任务调度

日常业务：

1，hivesql

2、调度平台

3、Hadoop集群运维

4、数据清洗（脚本语言）

5、元数据管理

6、数据稽查

7、数据仓库模型架构

### 流式计算是什么

流式计算：数据实时产生、数据实时传输、数据实时计算、实时展示

代表技术：Flume实时获取数据、Kafka/**metaq**实时数据存储、**Storm/JStorm**实时数据计算、Redis实时**结果**缓存、持久化存储(mysql)。

一句话总结：将源源不断产生的数据实时收集并实时计算，尽可能快的得到计算结果，用来支持决策。

### 离线计算与实时计算的区别

最大的区别：实时收集、实时计算、实时展示

离线计算，一次计算很多条数据

实时计算，数据被一条一条的计算

### storm是什么

Storm用来**实时处理数据**，特点：低延迟、高可用、分布式、可扩展、**数据不丢失**。提供简单容易理解的接口，便于开发。

Spout Bolt

数据输入 数据计算 数据输出 数据计算 数据输出

Spout Bolt 1.... Bolt N BoltN+1. BoltN....

### storm的应用场景

Storm处理数据的方式是基于消息的流水线处理， 因此特别适合无状态计算，也就是计算单元的依赖的数据全部在接受的消息中可以找到， 并且最好一个数据流不依赖另外一个数据流。

因此，常常用于

- 日志分析，从海量日志中分析出特定的数据，并将分析的结果存入外部存储器用来辅佐决策。

- 管道系统， 将一个数据从一个系统传输到另外一个系统， 比如将数据库同步到Hadoop

- 消息转化器， 将接受到的消息按照某种格式进行转化，存储到另外一个系统如消息中间件

- 统计分析器， 从日志或消息中，提炼出某个字段，然后做count或sum计算，最后将统计值存入外部存储器。中间处理过程可能更复杂。

**案例一:一淘-实时分析系统**

一淘-实时分析系统：实时分析用户的属性，并反馈给搜索引擎。最初，用户属性分析是通过每天在云梯上定时运行的MR job来完成的。为了满足实时性的要求，希望能够实时分析用户的行为日志，将最新的用户属性反馈给搜索引擎，能够为用户展现最贴近其当前需求的结果。

**案例二:携程-网站性能监控**

携程-网站性能监控：实时分析系统监控携程网的网站性能。利用HTML5获得可用的指标，并记录日志。Storm集群实时分析日志和入库。使用DRPC聚合成报表，通过历史数据对比等判断规则，触发预警事件。

**案例三:游戏实时运营**

一个游戏新版本上线，有一个实时分析系统，收集游戏中的数据，运营或者开发者可以在上线后几秒钟得到持续不断更新的游戏监控报告和分析结果，然后马上针对游戏的参数 和平衡性进行调整。这样就能够大大缩短游戏迭代周期，加强游戏的生命力。

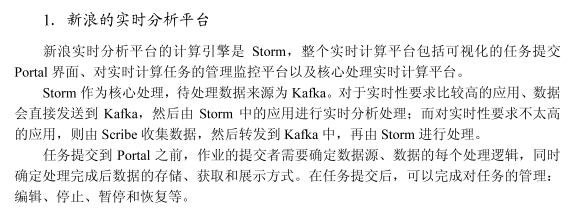
**案例四:实时计算在腾讯的应用**

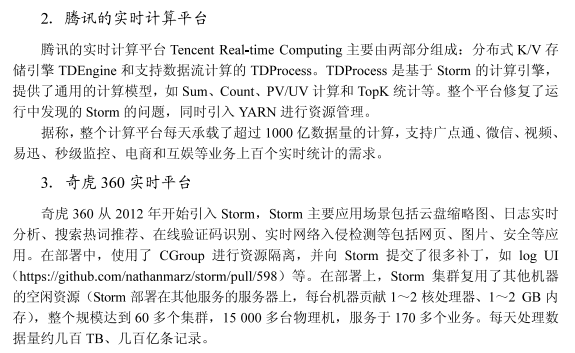
实时计算在腾讯的运用：精准推荐（广点通广告推荐、新闻推荐、视频推荐、游戏道具推荐）；实时分析（微信运营数据门户、效果统计、订单画像分析）；实时监控（实时监控平台、游戏内接口调用）

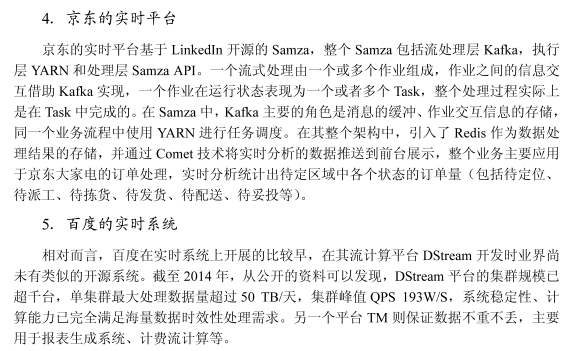
**案例五:实时计算在阿里的应用**

为了更加精准投放广告，阿里妈妈后台计算引擎需要维护每个用户的兴趣点（理想状态是，你对什么感兴趣，就向你投放哪类广告）。用户兴趣主要基于用户的历史行为、用户的实时查询、用户的实时点击、用户的地理信息而得，其中实时查询、实时点击等用户行为都是实时数据。考虑到系统的实时性，阿里妈妈使用Storm维护用户兴趣数据，并在此基础上进行受众定向的广告投放。

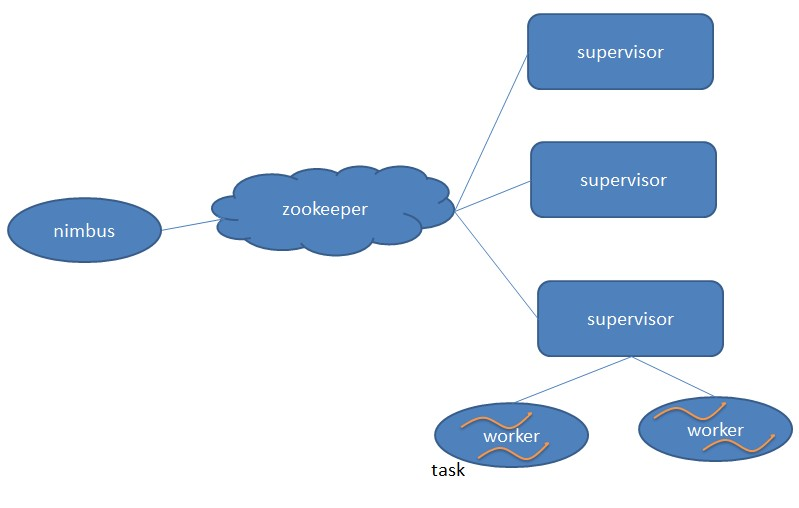
### storm在互联网公司







## storm架构图



Nimbus：负责资源分配和任务调度。

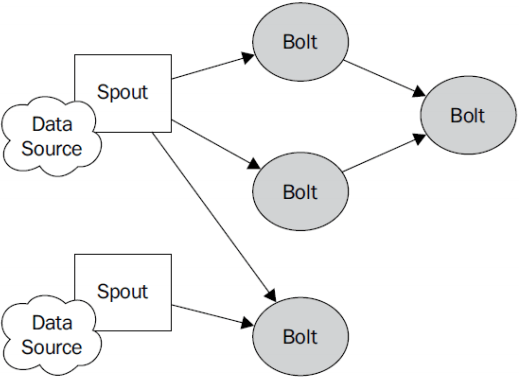
Supervisor：负责接受nimbus分配的任务，启动和停止属于自己管理的worker进程。

Worker：运行具体处理组件逻辑的进程。

Task：worker中每一个spout/bolt的线程称为一个task. 在storm0.8之后，task不再与物理线程对应，同一个spout/bolt的task可能会共享一个物理线程，该线程称为executor。



## storm编程模型



Topology：Storm中运行的一个实时应用程序，因为各个组件间的消息流动形成逻辑上的一个拓扑结构。

Spout：在一个topology中产生源数据流的组件。通常情况下spout会从外部数据源中读取数据，然后转换为topology内部的源数据。Spout是一个主动的角色，其接口中有个nextTuple()函数，storm框架会不停地调用此函数，用户只要在其中生成源数据即可。

Bolt：在一个topology中接受数据然后执行处理的组件。Bolt可以执行过滤、函数操作、合并、写数据库等任何操作。Bolt是一个被动的角色，其接口中有个execute(Tuple input)函数,在接受到消息后会调用此函数，用户可以在其中执行自己想要的操作。

Tuple：一次消息传递的基本单元。本来应该是一个key-value的map，但是由于各个组件间传递的tuple的字段名称已经事先定义好，所以tuple中只要按序填入各个value就行了，所以就是一个value list.

Stream：源源不断传递的tuple就组成了stream。

## wordcount案例分析

### 功能说明

设计一个topology，来实现对一个句子里面的单词出现的频率进行统计。

整个topology分为三个部分：

RandomSentenceSpout：数据源，在已知的英文句子中，随机发送一条句子出去。

SplitSentenceBolt:负责将单行文本记录（句子）切分成单词

WordCountBolt:负责对单词的频率进行累加

### TopologyMain 驱动类

|  |
| --- |
| **public static void** main(String[] args) **throws** Exception {  *// Storm框架支持多语言，在JAVA环境下创建一个拓扑，需要使用TopologyBuilder进行构建* TopologyBuilder builder = **new** TopologyBuilder();  *//RandomSentenceSpout类，在已知的英文句子中，随机发送一条句子出去。* builder.setSpout(**"spout1"**, **new** RandomSentenceSpout(), 3);  *// SplitSentenceBolt类，主要是将一行一行的文本内容切割成单词* builder.setBolt(**"split1"**, **new** SplitSentenceBolt(), 9).shuffleGrouping(**"spout1"**);  *// WordCountBolt类，对单词出现的次数进行统计* builder.setBolt(**"count2"**, **new** WordCountBolt(),3).fieldsGrouping(**"split1"**,**new** Fields(**"word"**)); *//启动topology的配置信息* Config conf = **new** Config();  *//TOPOLOGY\_DEBUG(setDebug), 当它被设置成true的话， storm会记录下每个组件所发射的每条消息。  //这在本地环境调试topology很有用， 但是在线上这么做的话会影响性能的。 // conf.setDebug(true);* conf.setDebug(**false**);  *//storm的运行有两种模式: 本地模式和分布式模式.* **if** (args != **null** && args.**length** > 0) {  *//定义你希望集群分配多少个工作进程给你来执行这个topology* conf.setNumWorkers(3);  *//向集群提交topology* StormSubmitter.*submitTopologyWithProgressBar*(args[0], conf, builder.createTopology());  } **else** {  conf.setMaxTaskParallelism(3);  LocalCluster cluster = **new** LocalCluster();  cluster.submitTopology(**"word-count"**, conf, builder.createTopology());  *//指定本地模式运行多长时间之后停止，如果不显式的关系程序将一直运行下去  //Utils.sleep(10000);  //cluster.shutdown();* }  } |

### RandomSentenceSpout

|  |
| --- |
| **public class** RandomSentenceSpout **extends** BaseRichSpout {  **private static final long *serialVersionUID*** = 5028304756439810609L;  *//用来收集Spout输出的tuple* SpoutOutputCollector **collector**;  Random **rand**;  *//该方法调用一次，主要由storm框架传入SpoutOutputCollector* **public void** open(Map conf, TopologyContext context,  SpoutOutputCollector collector) {  **this**.**collector** = collector;  **rand** = **new** Random();  //连接kafka mysql ，打开本地文件   }   */\*\*  \* 上帝之手  \* while(true){  \* spout.nexTuple()  \* }  \*/* **public void** nextTuple() {  String[] sentences = **new** String[] { **"the cow jumped over the moon"**,  **"the cow jumped over the moon"**,  **"the cow jumped over the moon"**,  **"the cow jumped over the moon"**, **"the cow jumped over the moon"** };  String sentence = sentences[**rand**.nextInt(sentences.**length**)];  **collector**.emit(**new** Values(sentence));  System.***out***.println(**"RandomSentenceSpout 发送数据："**+sentence);  }  *//消息源可以发射多条消息流stream。多条消息流可以理解为多中类型的数据。* **public void** declareOutputFields(OutputFieldsDeclarer declarer) {  declarer.declare(**new** Fields(**"sentence"**));  } } |

### SplitSentenceBolt

|  |
| --- |
| **public class** SplitSentenceBolt **extends** BaseBasicBolt {  **private static final long *serialVersionUID*** = -5283595260540124273L;  *//该方法只会被调用一次，用来初始化* **public void** prepare(Map stormConf, TopologyContext context) {  **super**.prepare(stormConf, context);  }  */\*\*  \* 接受的参数是RandomSentenceSpout发出的句子，即input的内容是句子 execute方法，将句子切割形成的单词发出  \*/* **public void** execute(Tuple input, BasicOutputCollector collector) {  String sentence = (String)input.getValueByField(**"sentence"**);  String[] words = sentence.split(**" "**);  **for** (String word : words) {  word = word.trim();  **if** (!word.isEmpty()) {  word = word.toLowerCase();  System.***out***.println(**"SplitSentenceBolt 切割单词："**+word);  collector.emit(**new** Values(word,1));  }  }  }  *//消息源可以发射多条消息流stream。多条消息流可以理解为多种类型的数据。* **public void** declareOutputFields(OutputFieldsDeclarer declarer) {  declarer.declare(**new** Fields(**"word"**,**"num"**));  } } |

### WordCountBolt

|  |
| --- |
| **public class** WordCountBolt **extends** BaseBasicBolt {  **private static final long *serialVersionUID*** = 5678586644899822142L;  *// 用来保存最后计算的结果key=单词，value=单词个数* Map<String, Integer> **counters** = **new** HashMap<String, Integer>();   *//该方法只会被调用一次，用来初始化* **public void** prepare(Map stormConf, TopologyContext context) {  **super**.prepare(stormConf, context);  }   */\*  \* 将collector中的元素存放在成员变量counters（Map）中.  \* 如果counters（Map）中已经存在该元素，getValule并对Value进行累加操作。  \*/* **public void** execute(Tuple input, BasicOutputCollector collector) {  String str = (String)input.getValueByField(**"word"**);  Integer num =input.getIntegerByField(**"num"**);  System.***out***.println(**"----------------"**+Thread.*currentThread*().getId()+**" "**+str);  **if** (!**counters**.containsKey(str)) {  **counters**.put(str, num);  } **else** {  Integer c = **counters**.get(str) + num;  **counters**.put(str, c);  }  System.***out***.println(**"WordCountBolt 统计单词："**+**counters**);  }   **public void** declareOutputFields(OutputFieldsDeclarer declarer) {  *//* ***TODO Auto-generated method stub*** } |

### pom依赖

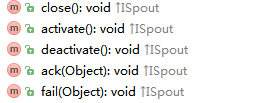
|  |
| --- |
| <**dependencies**>  *<!--<dependency>-->  <!--<groupId>org.apache.storm</groupId>-->  <!--<artifactId>storm-core</artifactId>-->  <!--<version>0.9.5</version>-->  <!--&lt;!&ndash;<scope>provided</scope>&ndash;&gt;-->  <!--</dependency>-->* <**dependency**>  <**groupId**>com.alibaba.jstorm</**groupId**>  <**artifactId**>jstorm-core</**artifactId**>  <**version**>2.1.1</**version**>  *<!-- 目前<scope>可以使用5个值：  \* compile，缺省值，适用于所有阶段，会随着项目一起发布。  \* provided，类似compile，期望JDK、容器或使用者会提供这个依赖。如servlet.jar。  \* runtime，只在运行时使用，如JDBC驱动，适用运行和测试阶段。  \* test，只在测试时使用，用于编译和运行测试代码。不会随项目发布。  \* system，类似provided，需要显式提供包含依赖的jar，Maven不会在Repository中查找它。 -->  <!--<scope>provided</scope>-->* </**dependency**> </**dependencies**> |

### 项目编译

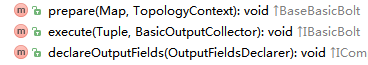
|  |
| --- |
| <**build**>  <**plugins**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-jar-plugin</**artifactId**>  <**configuration**>  <**archive**>  <**manifest**>  <**addClasspath**>true</**addClasspath**>  <**classpathPrefix**>lib/</**classpathPrefix**>  <**mainClass**>com.itcast.logmonitor.LogMonitorTopologyMain</**mainClass**>  </**manifest**>  </**archive**>  </**configuration**>  </**plugin**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-dependency-plugin</**artifactId**>  <**executions**>  <**execution**>  <**id**>copy</**id**>  <**phase**>package</**phase**>  <**goals**>  <**goal**>copy-dependencies</**goal**>  </**goals**>  <**configuration**>  <**outputDirectory**>target/lib</**outputDirectory**>  </**configuration**>  </**execution**>  </**executions**>  </**plugin**>  <**plugin**>  <**artifactId**>maven-assembly-plugin</**artifactId**>  <**configuration**>  <**descriptorRefs**>  <**descriptorRef**>jar-with-dependencies</**descriptorRef**>  </**descriptorRefs**>  <**archive**>  <**manifest**>  <**mainClass**>cn.itcast.bigdata.wc.WordCountTopologyMain</**mainClass**>  </**manifest**>  </**archive**>  </**configuration**>  <**executions**>  <**execution**>  <**id**>make-assembly</**id**>  <**phase**>package</**phase**>  <**goals**>  <**goal**>single</**goal**>  </**goals**>  </**execution**>  </**executions**>  </**plugin**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-compiler-plugin</**artifactId**>  <**configuration**>  <**source**>1.7</**source**>  <**target**>1.7</**target**>  </**configuration**>  </**plugin**>  </**plugins**> </**build**> |

### Component生命周期

#### Spout生命周期



#### Bolt生命周期



#### Bolt的两个抽象类

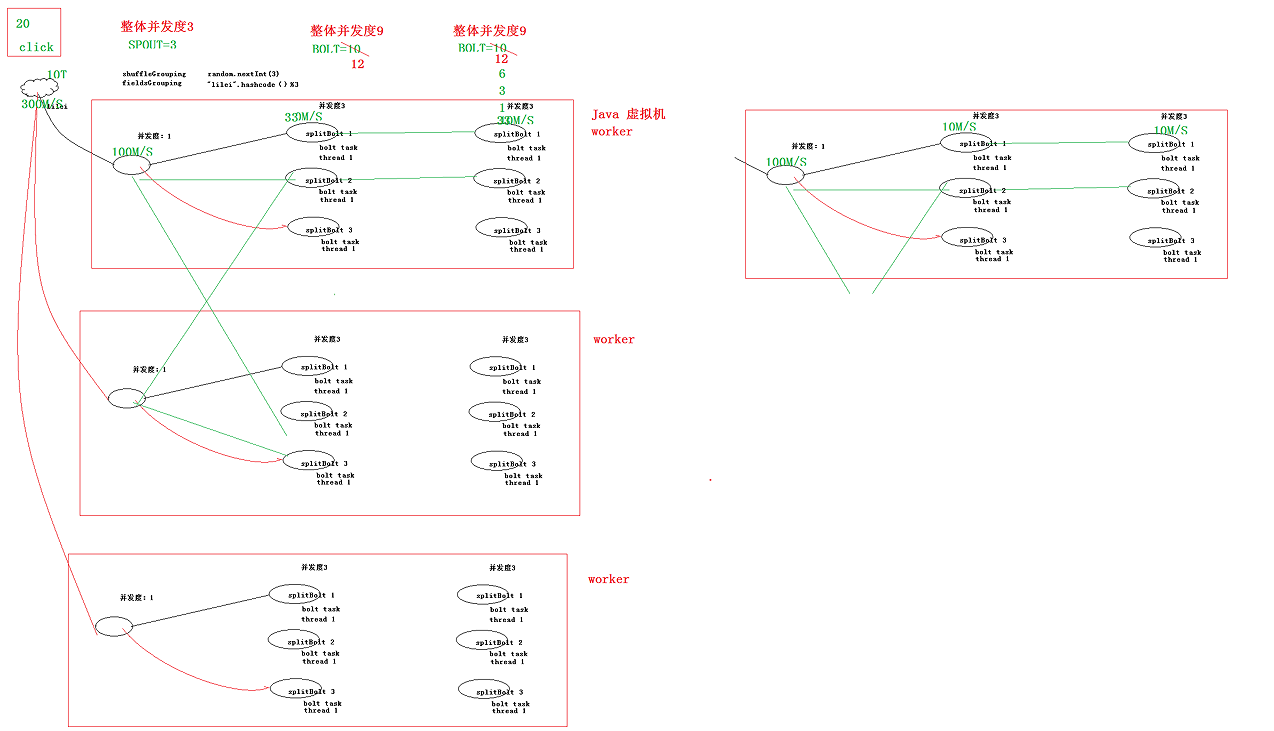
BaseRichBolt 需要手动调ack方法

BaseBasicBolt由storm框架自动调ack方法

### StreamGrouping

|  |
| --- |
| **public** Map<List<Integer>, List<MsgInfo>> grouperBatch(List<MsgInfo> batch) {  Map<List<Integer>, List<MsgInfo>> ret = **new** HashMap<List<Integer>, List<MsgInfo>>();  *//optimize fieldGrouping & customGrouping* **if** (GrouperType.***local\_or\_shuffle***.equals(**grouptype**)) {  ret.put(**local\_shuffer\_grouper**.grouper(**null**), batch);  } **else if** (GrouperType.***global***.equals(**grouptype**)) {  *// send to task which taskId is 0* ret.put(JStormUtils.*mk\_list*(**out\_tasks**.get(0)), batch);  } **else if** (GrouperType.***fields***.equals(**grouptype**)) {  **fields\_grouper**.batchGrouper(batch, ret);  } **else if** (GrouperType.***all***.equals(**grouptype**)) {  *// send to every task* ret.put(**out\_tasks**, batch);  } **else if** (GrouperType.***shuffle***.equals(**grouptype**)) {  *// random, but the random is different from none* ret.put(**shuffer**.grouper(**null**), batch);  } **else if** (GrouperType.***none***.equals(**grouptype**)) {  **int** rnd = Math.*abs*(**random**.nextInt() % **out\_tasks**.size());  ret.put(JStormUtils.*mk\_list*(**out\_tasks**.get(rnd)), batch);  } **else if** (GrouperType.***custom\_obj***.equals(**grouptype**) || GrouperType.***custom\_serialized***.equals(**grouptype**)) {  **for** (**int** i = 0; i < batch.size(); i++ ) {  MsgInfo msg = batch.get(i);  List<Integer> out = **custom\_grouper**.grouper(msg.**values**);  List<MsgInfo> customBatch = ret.get(out);  **if** (customBatch == **null**) {  customBatch = JStormUtils.*mk\_list*();  ret.put(out, customBatch);  }  customBatch.add(msg);  }  } **else if** (GrouperType.***localFirst***.equals(**grouptype**)) {  ret.put(**localFirst**.grouper(**null**), batch);  } **else** {  ***LOG***.warn(**"Unsupportted group type"**);  }  **return** ret; } |

## storm并行度



## storm分组

Stream grouping：即消息的partition方法。

Stream Grouping定义了一个流在Bolt任务间该如何被切分。这里有Storm提供的6个Stream Grouping类型：

1. 随机分组(Shuffle grouping)：随机分发tuple到Bolt的任务，保证每个任务获得相等数量的tuple。

2. 字段分组(Fields grouping)：根据指定字段分割数据流，并分组。例如，根据“user-id”字段，相同“user-id”的元组总是分发到同一个任务，不同“user-id”的元组可能分发到不同的任务。

3. 全部分组(All grouping)：tuple被复制到bolt的所有任务。这种类型需要谨慎使用。

4. 全局分组(Global grouping)：全部流都分配到bolt的同一个任务。明确地说，是分配给ID最小的那个task。

5. 无分组(None grouping)：你不需要关心流是如何分组。目前，无分组等效于随机分组。但最终，Storm将把无分组的Bolts放到Bolts或Spouts订阅它们的同一线程去执行(如果可能)。

6. 直接分组(Direct grouping)：这是一个特别的分组类型。元组生产者决定tuple由哪个元组处理者任务接收。

## storm程序的并行机制

### 概念

* Workers (JVMs): 在一个物理节点上可以运行一个或多个独立的JVM 进程。一个Topology可以包含一个或多个worker(并行的跑在不同的物理机上), 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**
* Executors (threads): 在一个worker JVM进程中运行着多个Java线程。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。一个worker可以包含一个或多个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。
* Tasks(bolt/spout instances)：Task就是具体的处理逻辑对象，每一个Spout和Bolt会被当作很多task在整个集群里面执行。每一个task对应到一个线程，而stream grouping则是定义怎么从一堆task发射tuple到另外一堆task。你可以调用TopologyBuilder.setSpout和TopBuilder.setBolt来设置并行度 — 也就是有多少个task。

### 配置并行度

* 对于并发度的配置, 在storm里面可以在多个地方进行配置, 优先级为：

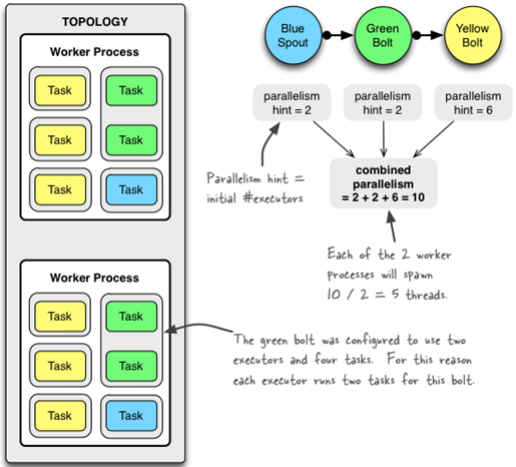
defaults.yaml < storm.yaml < topology-specific configuration

< internal component-specific configuration < external component-specific configuration

* worker processes的数目, 可以通过配置文件和代码中配置, worker就是执行进程, 所以考虑并发的效果, 数目至少应该大亍machines的数目
* executor的数目, component的并发线程数，只能在代码中配置(通过setBolt和setSpout的参数), 例如, setBolt("green-bolt", new GreenBolt(), 2)
* tasks的数目, 可以不配置, 默认和executor1:1, 也可以通过setNumTasks()配置

Topology的worker数通过config设置，即执行该topology的worker（java）进程数。它可以通过 storm rebalance 命令任意调整。

|  |
| --- |
| Config conf = newConfig();  conf.setNumWorkers(2); **//用2个worker**  topologyBuilder.setSpout("blue-spout", newBlueSpout(), 2); **//设置2个并发度**  topologyBuilder.setBolt("green-bolt", newGreenBolt(), 2).setNumTasks(4).shuffleGrouping("blue-spout"); **//设置2个并发度，4个任务**  topologyBuilder.setBolt("yellow-bolt", newYellowBolt(), 6).shuffleGrouping("green-bolt"); **//设置6个并发度**  StormSubmitter.submitTopology("mytopology", conf, topologyBuilder.createTopology()); |



3个组件的并发度加起来是10，就是说拓扑一共有10个executor，一共有2个worker，每个worker产生10 / 2 = 5条线程。

绿色的bolt配置成2个executor和4个task。为此每个executor为这个bolt运行2个task。

动态的改变并行度

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命令实现：

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

## storm+kafka+redis整合

步骤：

1. 将之前的Spout从本地读取数据改成Spout从kafka中读取数据
2. 将之前的Bolt2打印数据到控制台改成数据写入到redis中

## storm启动流程分析

------------程序员client------------------

1、客户端运行storm nimbus时，会调用storm的python脚本，该脚本中为每个命令编写一个方法，每个方法都可以生成一条相应的java命令。

命令格式如下：java -server xxxx.ClassName -args

nimbus---> Running: /export/servers/jdk/bin/java -server backtype.storm.daemon.nimbus

supervisor---> Running: /export/servers/jdk/bin/java -server backtype.storm.daemon.supervisor

--------------nimbus---------------------

2、nibums启动之后，接受客户端提交任务

命令格式：storm jar xxx.jar xxx驱动类 参数

Running: /export/servers/jdk/bin/java -client -Dstorm.jar=/export/servers/storm/examples/storm-starter/storm-starter-topologies-0.9.6.jar storm.starter.WordCountTopology wordcount-28

该命令会执行 storm-starter-topologies-0.9.6.jar 中的storm-starter-topologies-0.9.6.jar的main方法，main方法中会执行以下代码：

StormSubmitter.submitTopology("mywordcount",config,topologyBuilder.createTopology());

topologyBuilder.createTopology()，会将程序猿编写的spout对象和bolt对象进行序列化。

会将用户的jar上传到 nimbus物理节点的 /export/data/storm/workdir/nimbus/inbox目录下。并且改名，改名的规则是添加了一个UUID字符串。

在nimbus物理节点的 /export/data/storm/workdir/nimbus/stormdist目录下。有当前正在运行的topology的jar包和配置文件，序列化对象文件。

3、nimbus接受到任务之后，会将任务进行分配，分配会产生一个assignment对象，该对象会保存到zk中，目录是/storm/assignments ，该目录只保存正在运行的topology任务。

--------supervisor------------------

4、supervisor通过watch机制，感知到nimbus在zk上的任务分配信息，从zk上拉取任务信息，分辨出属于自己任务。

ResourceWorkerSlot[hostname=192.168.1.106,memSize=0,cpu=0,tasks=[1, 2, 3, 4, 5, 6, 7, 8],jvm=<null>,nodeId=61ce10a7-1e78-4c47-9fb3-c21f43a331ba,port=6900]

5、supervisor 根据自己的任务信息，启动自己的worker，并分配一个端口。

'/export/servers/jdk/bin/java' '-server' '-Xmx768m' export/data/storm/workdir/supervisor/stormdist/wordcount1-3-1461683066/stormjar.jar' 'backtype.storm.daemon.worker' 'wordcount1-3-1461683066' 'a69bb8fc-e08e-4d55-b51f-e539b066f90b' '6701' '9fac2805-7d2b-4e40-aabc-1c85c9856d64'

---------worker----------------------

6、worker启动之后，连接zk，拉取任务

ResourceWorkerSlot[hostname=192.168.1.106,memSize=0,cpu=0,tasks=[1, 2, 3, 4, 5, 6, 7, 8],jvm=<null>,nodeId=61ce10a7-1e78-4c47-9fb3-c21f43a331ba,port=6900]

假设任务信息：

1--->spout---type:spout

2--->bolt ---type:bolt

3--->acker---type:bolt

得到对象有几种方式？ new ClassName 创建对象、class.forName 反射对象、clone 克隆对象、序列化反序列化

worker通过反序列化，得到程序员自己定义的spout和bolt对象。

7、worker根据任务类型，分别执行spout任务或者bolt任务。

spout的声明周期是：open、nextTuple、outPutFiled

bolt的生命周期是：prepare、execute(tuple)、outPutFiled

## 代码流程

jstorm supervisor如何启动worker，worker如何启动task

1、下载Jstorm源码，在源码包下找到 daemon包，在这个包下有三个子包，分别是nimbus，supervisor，worker。

2、通过架构图，我们已知nimbus分配任务，并将任务信息写入到zk上，supervisor读取zk上的任务后启动自己的worker。所以我们分析supervisor如何启动worker，worker如何启动task。

3、supervisor如何启动worker。打开 com.alibaba.jstorm.daemon.supervisor.Supervisor 发现supervisor有几个方法，方法中有个mkSupervisor方法。

4、进去Supervisor中的mkSupervisor方法，在第144行有以下的代码，改代码创建了SyncSupervisorEvent 对象。

SyncSupervisorEvent syncSupervisorEvent =

new SyncSupervisorEvent(supervisorId, conf, syncSupEventManager, stormClusterState, localState, syncProcessEvent, hb);

5、SyncSupervisorEvent对象实现了RunnableCallback接口，该接口有个run方法会被定时执行。在run方法的191行，有代码如下，主要是要supervisor获取到任务信息，要开始准备启动worker了。

syncProcesses.run(zkAssignment, downloadFailedTopologyIds);

6、syncProcesses是com.alibaba.jstorm.daemon.supervisor.SyncProcessEvent的

引用变量，该类中有个自定义的run方法中有段代码如下，调用的startNewWorkers方法

startNewWorkers(keepPorts, localAssignments, downloadFailedTopologyIds);

7、SyncProcessEvent的startNewWorkers方法有代码片段如下，主要是根据集群模式启动不同模式下的worker。我们跟踪分布式集群模式下的worker启动。

for (Entry<Integer, LocalAssignment> entry : newWorkers.entrySet()) {

if (clusterMode.equals(“distributed”)) {

launchWorker(conf, sharedContext, assignment.getTopologyId(), supervisorId, port, workerId, assignment);

} else if (clusterMode.equals(“local”)) {

launchWorker(conf, sharedContext, assignment.getTopologyId(), supervisorId, port, workerId, workerThreadPids);

}

}

8、在分布式模式下worker启动最终会调用一个类似于java -server xxx.worker 启动worker。由于第7步中，有个for循环，该for循环会迭代出属于当前supervisor的所有worker任务并启动。

JStormUtils.launchProcess(cmd, environment, true);

9、java -server xxx.worker，命令执行之后，会执行Worker的mian方法。worker的main方法有代码如下，其实调用了worker自己内部的静态方法，叫做mk\_worker方法。

WorkerShutdown sd = mk\_worker(conf, null, topology\_id, supervisor\_id, Integer.parseInt(port\_str), worker\_id, jar\_path);

sd.join();

10、mk\_worker静态方法，会执行以下代码，创建一个worker的实例，并立即执行execute方法。

Worker w = new Worker(conf, context, topology\_id, supervisor\_id, port, worker\_id, jar\_path);

return w.execute();

11、execute方法会执行以下代码创建一个RefreshConnections 的实例。

RefreshConnections refreshConn = makeRefreshConnections();

12、makeRefreshConnections 方法会执行以下代码创建一个RefreshConnections 实例。

RefreshConnections refresh\_connections = new RefreshConnections(workerData);

13、RefreshConnections 是继承了 RunnableCallback，该实例的会有一个run方法会被定时执行。run方法中有以下代码，其中createTasks(addedTasks)方法用来创建Task任务。

shutdownTasks(removedTasks);

createTasks(addedTasks);

updateTasks(updatedTasks);

14、createTasks方法有代码如下，循环启动属于该worker的Task任务，启动Task任务主要调用Task.mk\_task(workerData, taskId);

for (Integer taskId : tasks) {

try {

TaskShutdownDameon shutdown = Task.mk\_task(workerData, taskId);

workerData.addShutdownTask(shutdown);

} catch (Exception e) {

LOG.error(“Failed to create task-” + taskId, e);

throw new RuntimeException(e);

}

}

15、Task.mk\_task(workerData, taskId)方法实现如下，创建一个Task对象并立即调用execute方法。

Task t = new Task(workerData, taskId);

return t.execute();

16、execute方法实现如下,用来初始化一个Executor，我们知道在默认情况下一个task等于一个executor。

RunnableCallback baseExecutor = prepareExecutor();

17、进入prepareExecutor()方法，代码如下，发现代码调用了mkExecutor方法。

final BaseExecutors baseExecutor = mkExecutor();

18、mkExecutor方法，代码如下，如果当前taskObj是Bolt就创建Bolt的executor，如果当前taskObj是Spout就创建相应的Spout executor。

public BaseExecutors mkExecutor() {

BaseExecutors baseExecutor = null;

if (taskObj instanceof IBolt) {

baseExecutor = new BoltExecutors(this);

} else if (taskObj instanceof ISpout) {

if (isSingleThread(stormConf) == true) {

baseExecutor = new SingleThreadSpoutExecutors(this);

} else {

baseExecutor = new MultipleThreadSpoutExecutors(this);

}

}

return baseExecutor;

}

19、创建完了executor，现在有两条线，分别是bolt executor和spout executor。以

bolt executor 为例，这个executor会实现Disruptor的EventHandler接口。 接口onevent方法需要实现，实现代码中会调用processTupleEvent()方法。下面节选onevent中的部分代码。

if (event instanceof Tuple) {

processControlEvent();

processTupleEvent((Tuple) event);

} else if (event instanceof BatchTuple) {

for (Tuple tuple : ((BatchTuple) event).getTuples()) {

processControlEvent();

processTupleEvent((Tuple) tuple);

}

}

20、进入processTupleEvent方法，发现有代码如下，其实最终是调用了bolt.execute()方法。

private void processTupleEvent(Tuple tuple) {

try {

if (xxx) {

backpressureTrigger.handle(tuple);

} else {

bolt.execute(tuple);

}

} catch (Throwable e) {

error = e;

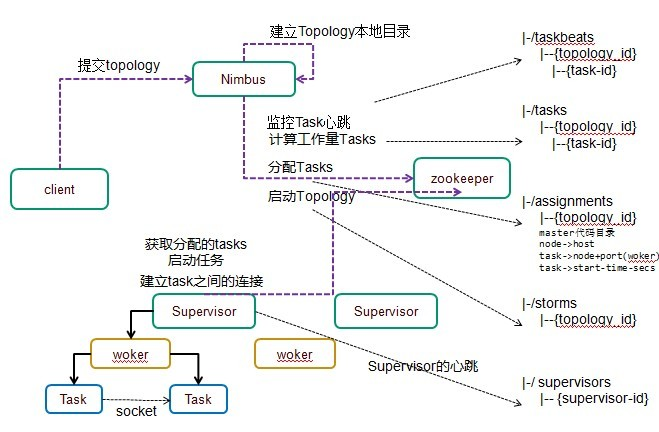
LOG.error(“bolt execute error “, e);

report\_error.report(e);

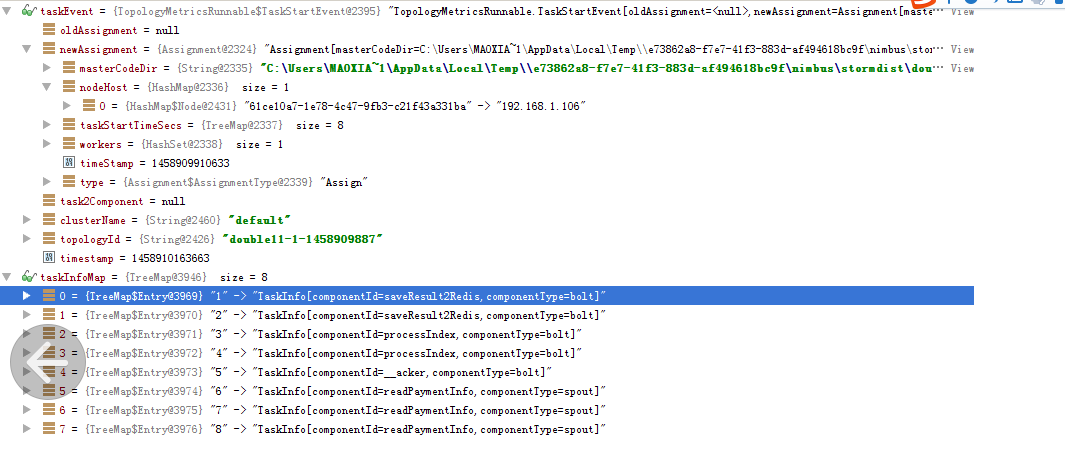
}

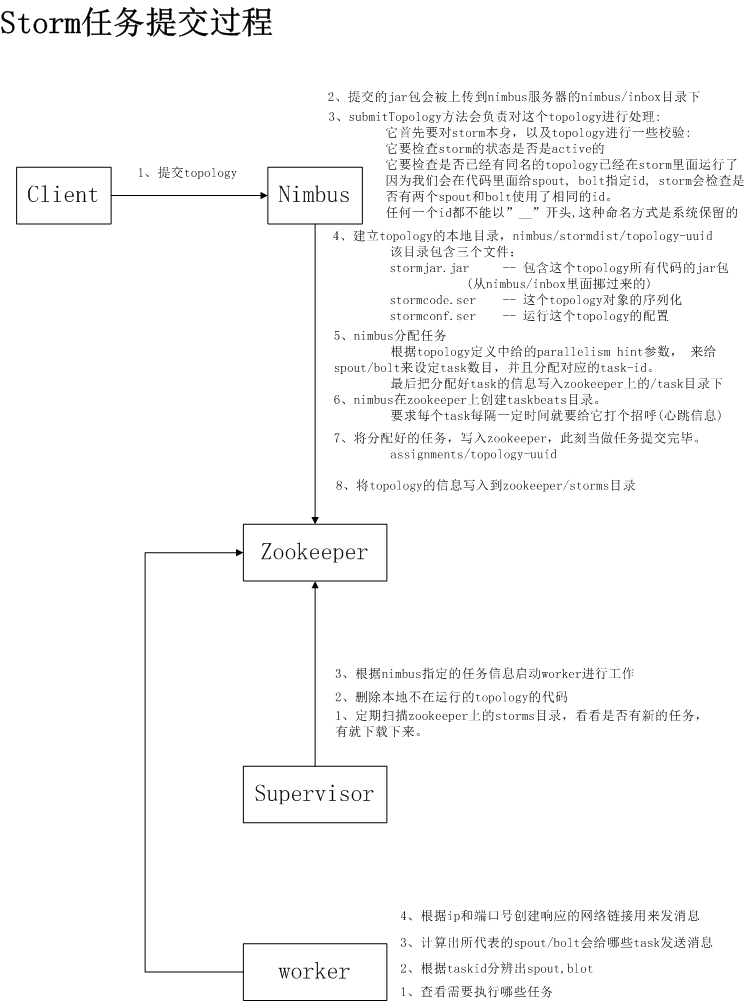
}

## storm任务提交过程



TopologyMetricsRunnable.TaskStartEvent[oldAssignment=<null>,newAssignment=Assignment[masterCodeDir=C:\Users\MAOXIA~1\AppData\Local\Temp\\e73862a8-f7e7-41f3-883d-af494618bc9f\nimbus\stormdist\double11-1-1458909887,nodeHost={61ce10a7-1e78-4c47-9fb3-c21f43a331ba=192.168.1.106},taskStartTimeSecs={1=1458909910, 2=1458909910, 3=1458909910, 4=1458909910, 5=1458909910, 6=1458909910, 7=1458909910, 8=1458909910},workers=[ResourceWorkerSlot[hostname=192.168.1.106,memSize=0,cpu=0,tasks=[1, 2, 3, 4, 5, 6, 7, 8],jvm=<null>,nodeId=61ce10a7-1e78-4c47-9fb3-c21f43a331ba,port=6900]],timeStamp=1458909910633,type=Assign],task2Component=<null>,clusterName=<null>,topologyId=double11-1-1458909887,timestamp=0]

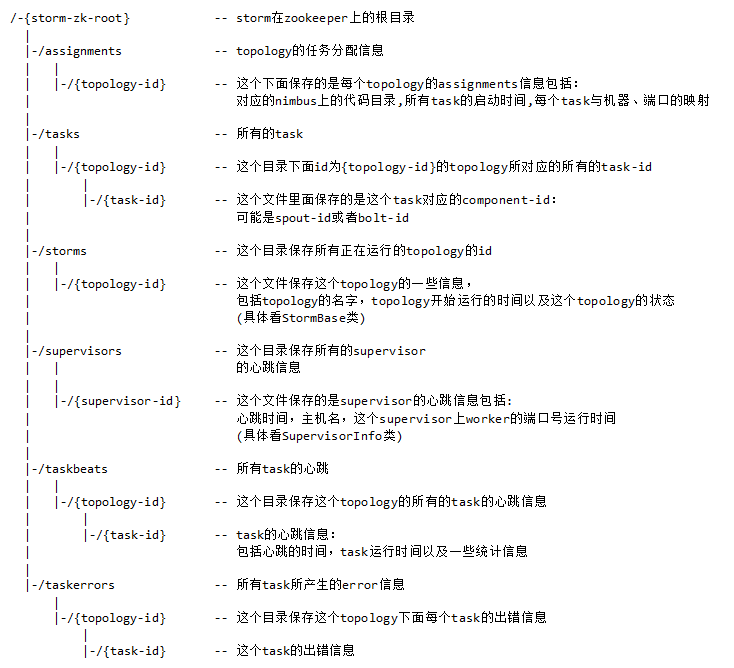




## storm组件本地树



## storm zookeeper树



## storm消息容错机制及源码分析

### ack是什么

ack 机制是storm整个技术体系中非常闪亮的一个创新点。

通过Ack机制，spout发送出去的每一条消息，都可以确定是被成功处理或失败处理， 从而可以让开发者采取动作。比如在Meta中，成功被处理，即可更新偏移量，当失败时，重复发送数据。

因此，通过Ack机制，很容易做到保证所有数据均被处理，一条都不漏。

**另外需要注意的，当spout触发fail动作时，不会自动重发失败的tuple，需要spout自己重新获取数据，手动重新再发送一次**

ack机制即， spout发送的每一条消息，

* 在规定的时间内，spout收到Acker的ack响应，即认为该tuple 被后续bolt成功处理
* 在规定的时间内，没有收到Acker的ack响应tuple，就触发fail动作，即认为该tuple处理失败，
* 或者收到Acker发送的fail响应tuple，也认为失败，触发fail动作

另外Ack机制还常用于限流作用： 为了避免spout发送数据太快，而bolt处理太慢，常常设置pending数，当spout有等于或超过pending数的tuple没有收到ack或fail响应时，跳过执行nextTuple， 从而限制spout发送数据。

通过conf.put(Config.TOPOLOGY\_MAX\_SPOUT\_PENDING, pending);设置spout pend数。

### 如何使用Ack机制

spout 在发送数据的时候带上msgid

设置acker数至少大于0；Config.setNumAckers(conf, ackerParal);

在bolt中完成处理tuple时，执行OutputCollector.ack(tuple), 当失败处理时，执行OutputCollector.fail(tuple);

推荐使用IBasicBolt， 因为IBasicBolt 自动封装了OutputCollector.ack(tuple), 处理失败时，请抛出FailedException，则自动执行OutputCollector.fail(tuple)

### 如何关闭Ack机制

有2种途径

spout发送数据是不带上msgid

设置acker数等于0

### 基本实现

Storm 系统中有一组叫做"acker"的特殊的任务，它们负责跟踪DAG（有向无环图）中的每个消息。

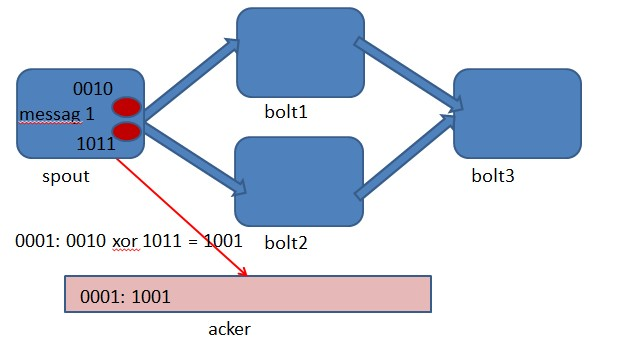
acker任务保存了spout id到一对值的映射。第一个值就是spout的任务id，通过这个id，acker就知道消息处理完成时该通知哪个spout任务。第二个值是一个64bit的数字，我们称之为"ack val"， 它是树中所有消息的随机id的异或计算结果。

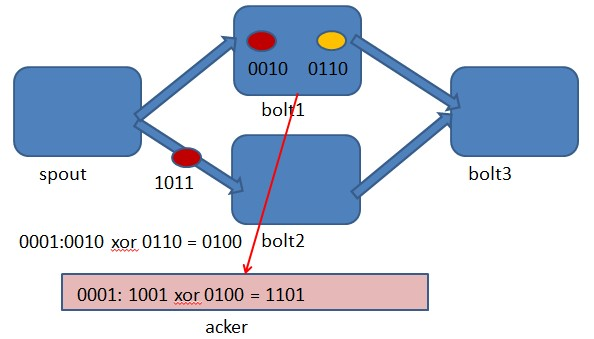
<TaskId,<RootId,ackValue>>

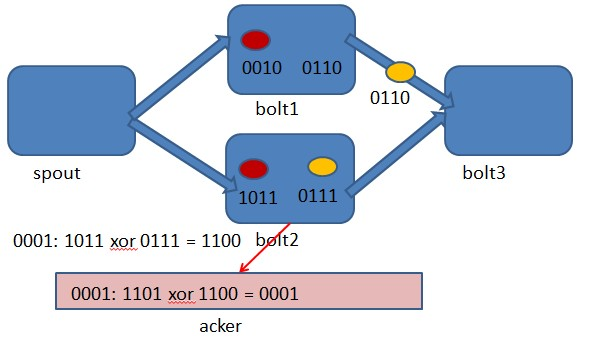
Spoutid,<系统生成的id,ackValue>

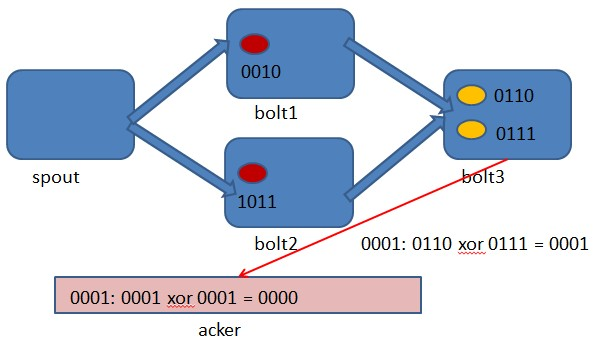
Task-0,64bit,0

ack val表示了整棵树的的状态，无论这棵树多大，只需要这个固定大小的数字就可以跟踪整棵树。当消息被创建和被应答的时候都会有相同的消息id发送过来做异或。 每当acker发现一棵树的ack val值为0的时候，它就知道这棵树已经被完全处理了









## storm通信机制

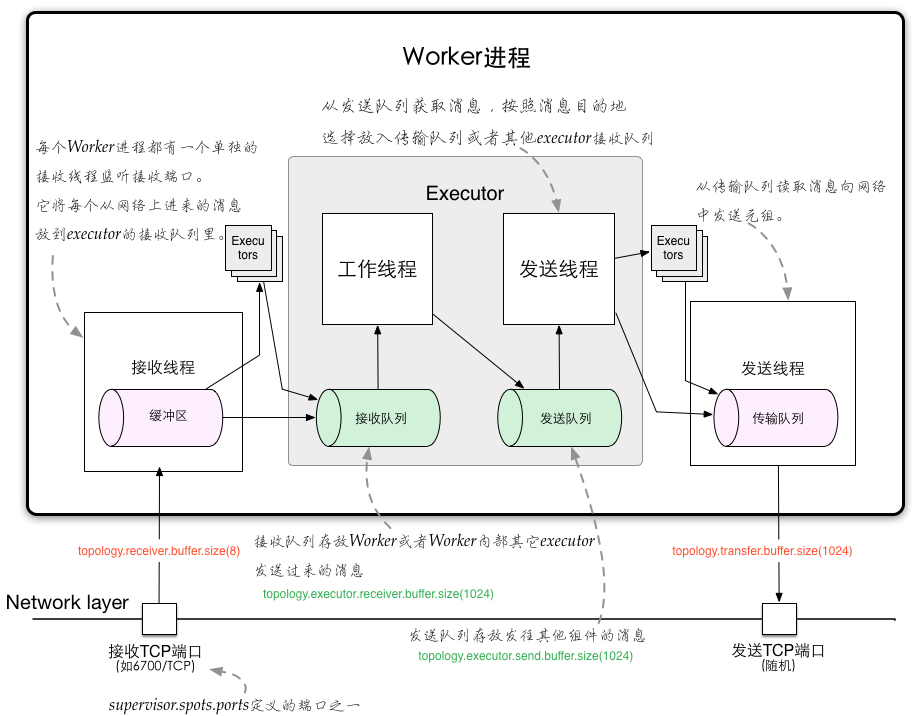
Worker间的通信经常需要通过网络跨节点进行，Storm使用ZeroMQ或Netty(0.9以后默认使用)作为进程间通信的消息框架。

Worker进程内部通信：**不同worker的thread通信使用LMAX Disruptor来完成。**

  不同topologey之间的通信，Storm不负责，需要自己想办法实现，例如使用kafka等；

### Worker进程间通信

worker进程间消息传递机制，消息的接收和处理的大概流程见下图



* + 对于worker进程来说，为了管理流入和传出的消息，每个worker进程有一个独立的接收线程(对配置的TCP端口supervisor.slots.ports进行监听);

对应Worker接收线程，每个worker存在一个独立的发送线程，它负责从worker的transfer-queue中读取消息，并通过网络发送给其他worker

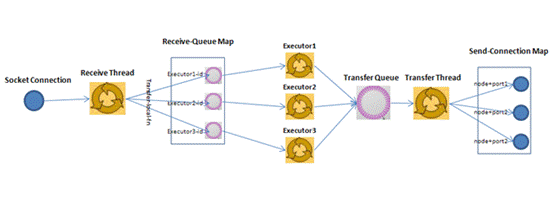
* + 每个executor有自己的incoming-queue和outgoing-queue。

Worker接收线程将收到的消息通过task编号传递给对应的executor(一个或多个)的incoming-queues;

每个executor有单独的线程分别来处理spout/bolt的业务逻辑，业务逻辑输出的中间数据会存放在outgoing-queue中，当executor的outgoing-queue中的tuple达到一定的阀值，executor的发送线程将批量获取outgoing-queue中的tuple,并发送到transfer-queue中。

* + 每个worker进程控制一个或多个executor线程，用户可在代码中进行配置。**其实就是我们在代码中设置的并发度个数。**

### Worker进程间通信分析



1. Worker接受线程通过网络接受数据，并根据Tuple中包含的taskId，匹配到对应的executor；然后根据executor找到对应的incoming-queue，将数据存发送到incoming-queue队列中。
2. 业务逻辑执行现成消费incoming-queue的数据，通过调用Bolt的execute(xxxx)方法，将Tuple作为参数传输给用户自定义的方法
3. 业务逻辑执行完毕之后，将计算的中间数据发送给outgoing-queue队列，当outgoing-queue中的tuple达到一定的阀值，executor的发送线程将批量获取outgoing-queue中的tuple,并发送到Worker的transfer-queue中
4. Worker发送线程消费transfer-queue中数据，计算Tuple的目的地，连接不同的node+port将数据通过网络传输的方式传送给另一个的Worker。
5. 另一个worker执行以上步骤1的操作。

### Worker进程间技术(Netty、ZeroMQ)

#### Netty

Netty是一个NIO client-server(客户端服务器)框架，使用Netty可以快速开发网络应用，例如服务器和客户端协议。Netty提供了一种新的方式来使开发网络应用程序，这种新的方式使得它很容易使用和有很强的扩展性。Netty的内部实现时很复杂的，但是Netty提供了简单易用的api从网络处理代码中解耦业务逻辑。Netty是完全基于NIO实现的，所以整个Netty都是异步的。

书籍：Netty权威指南

#### ZeroMQ

ZeroMQ是一种基于消息队列的多线程网络库，其对套接字类型、连接处理、帧、甚至路由的底层细节进行抽象，提供跨越多种传输协议的套接字。ZeroMQ是网络通信中新的一层，介于应用层和传输层之间（按照TCP/IP划分），其是一个可伸缩层，可并行运行，分散在分布式系统间。

ZeroMQ定位为：一个简单好用的传输层，像框架一样的一个socket library，他使得Socket编程更加简单、简洁和性能更高。是一个消息处理队列库，可在多个线程、内核和主机盒之间弹性伸缩。ZMQ的明确目标是“成为标准网络协议栈的一部分，之后进入Linux内核”。

### Worker 内部通信技术(Disruptor)

#### Disruptor的来历

* 一个公司的业务与技术的关系，一般可以分为三个阶段。第一个阶段就是跟着业务跑。第二个阶段是经历了几年的时间，才达到的驱动业务阶段。第三个阶段，技术引领业务的发展乃至企业的发展。所以我们在学习Disruptor这个技术时，不得不提LMAX这个机构，因为**Disruptor这门技术就是由LMAX公司开发并开源的**。
  + LMAX是在英国注册并受到FSA监管（监管号码为509778）的外汇黄金交易所。LMAX也是欧洲第一家也是唯一一家采用多边交易设施Multilateral Trading Facility（MTF）拥有交易所牌照和经纪商牌照的欧洲顶级金融公司
  + LAMX拥有最迅捷的交易平台，顶级技术支持。LMAX交易所使用“（MTF）分裂器Disruptor”技术，可以在极短时间内（一般在3百万秒之一内）处理订单，**在一个线程里每秒处理6百万订单**。所有订单均为撮合成交形式，无一例外。多边交易设施（MTF）曾经用来设计伦敦证券交易 所（london Stock Exchange）、德国证券及衍生工具交易所（Deutsche Borse）和欧洲证券交易所（Euronext）。
  + 2011年LMAX凭借该技术获得了金融行业技术评选大赛的最佳交易系统奖和甲骨文“公爵杯”创新编程框架奖。

#### Disruptor是什么

* 1. 简单理解：Disruptor是一个Queue。Disruptor是实现了“队列”的功能，而且是一个有界队列。而队列的应用场景自然就是“生产者-消费者”模型。
  2. 在JDK中Queue有很多实现类，包括不限于ArrayBlockingQueue、LinkBlockingQueue，这两个底层的数据结构分别是数组和链表。数组查询快，链表增删快，能够适应大多数应用场景。
  3. 但是ArrayBlockingQueue、LinkBlockingQueue都是线程安全的。涉及到线程安全，就会有synchronized、lock等关键字，这就意味着CPU会打架。
  4. Disruptor一种线程之间信息无锁的交换方式（使用CAS（Compare And Swap/Set）操作）。

#### Disruptor主要特点

1. 没有竞争=没有锁=非常快。
2. 所有访问者都记录自己的序号的实现方式，允许多个生产者与多个消费者共享相同的数据结构。
3. 在每个对象中都能跟踪序列号（ring buffer，claim Strategy，生产者和消费者），加上神奇的cache line padding，就意味着没有为伪共享和非预期的竞争。

#### Disruptor 核心技术点

Disruptor可以看成一个事件监听或消息机制，在队列中一边生产者放入消息，另外一边消费者并行取出处理.

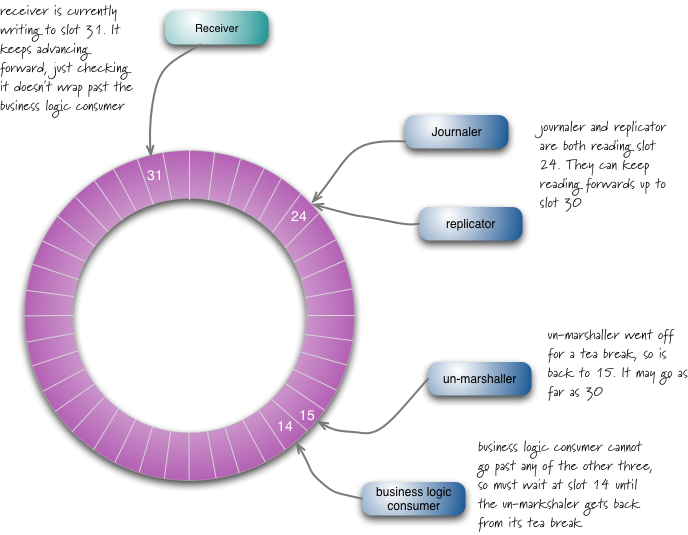
底层是单个数据结构：一个ring buffer。

每个生产者和消费者都有一个次序计算器，以显示当前缓冲工作方式。

每个生产者消费者能够操作自己的次序计数器的能够读取对方的计数器，生产者能够读取消费者的计算器确保其在没有锁的情况下是可写的。

**核心组件**

* Ring Buffer 环形的缓冲区，负责对通过 Disruptor 进行交换的数据（事件）进行存储和更新。
* Sequence 通过顺序递增的序号来编号管理通过其进行交换的数据（事件），对数据(事件)的处理过程总是沿着序号逐个递增处理。
* RingBuffer底层是个数组，次序计算器是一个64bit long 整数型，平滑增长。



1. 接受数据并写入到脚标31的位置，之后会沿着序号一直写入，但是不会绕过消费者所在的脚标。
2. Joumaler和replicator同时读到24的位置，他们可以批量读取数据到30

3、消费逻辑线程读到了14的位置，但是没法继续读下去，因为他的sequence暂停在15的位置上，需要等到他的sequence给他序号。如果sequence能正常工作，就能读取到30的数据。