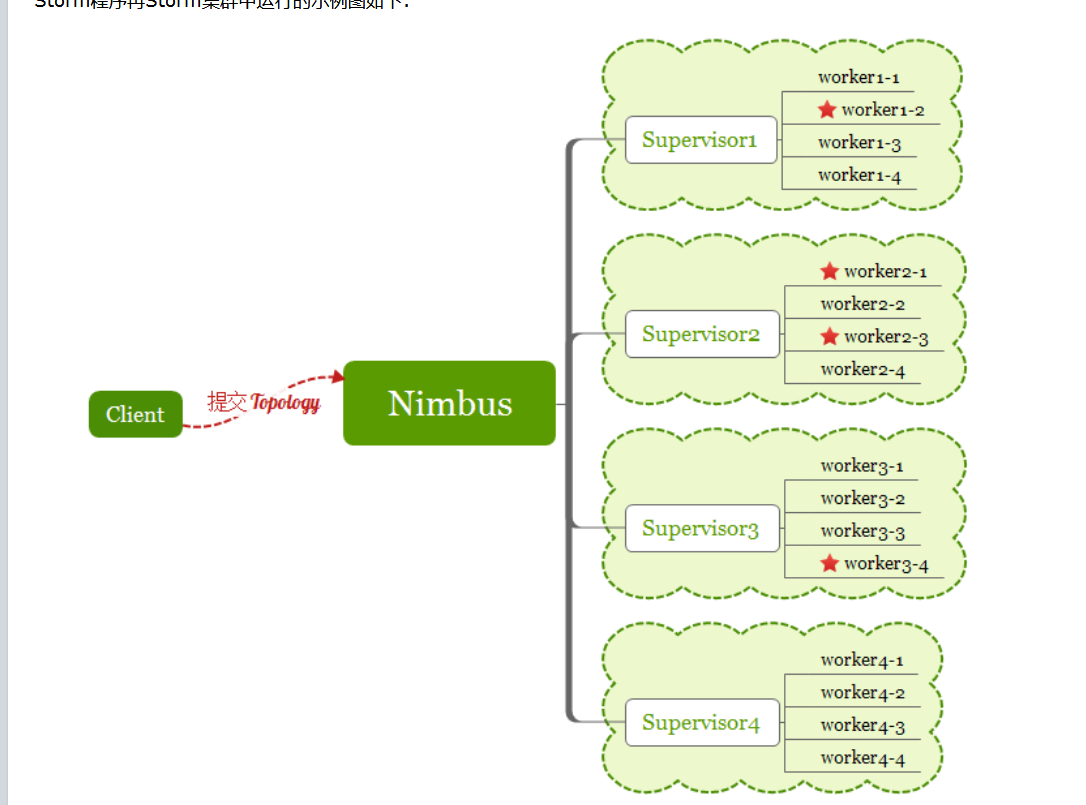
**Storm**

## 一、基础知识

**1 Storm含义**

**（1）Storm是Twitter开源的分布式实时大数据处理框架**，最早开源于github，从0.9.1版本之后，归于Apache社区，被业界称为实时版Hadoop。随着越来越多的场景对Hadoop的MapReduce高延迟无法容忍，比如**网站统计、推荐系统、预警系统、金融系统(高频交易、股票)**等等，大数据实时处理解决方案（流计算）的应用日趋广泛，目前已是分布式技术领域最新爆发点，而Storm更是流计算技术中的佼佼者和主流。



1. **Storm是一个开源免费的分布式实时计算系统**，storm可以可靠地处理无限的数据流（大数据），可以实时的处理 Hadoop 所处理的事项。Storm 是一个分布式的，可靠的，容错的数据流处理系统。它会把工作任务委托给不同类型的组件，每个组件负责处理一项简单特定的任务。Storm 集群的输入流由一个被称作 spout 的组件管理，spout 把数据传递给 bolt， bolt 要么把数据保存到某种存储器，要么把数据传递给其它的 bolt。你可以想象一下，**一个 Storm 集群就是在一连串的 bolt 之间转换 spout 传过来的数据**。这里用一个简单的例子来说明这个概念。昨晚我在新闻节目里看到主持人在谈论政治人物和他们对于各种政治话题的立场。他们一直重复着不同的名字，而我开始考虑这些名字是否被提到了相同的次数，以及不同次数之间的偏差。想像播音员读的字幕作为你的数据输入流。你可以用一个 spout 读取一个文件（或者 socket，通过 HTTP，或者别的方法）。文本行被 spout 传给一个 bolt，再被 bolt 按单词切割。单词流又被传给另一个 bolt，在这里每个单词与一张政治人名列表比较。每遇到一个匹配的名字，第二个 bolt 为这个名字在数据库的计数加1。你可以随时查询数据库查看结果， 而且这些计数是随着数据到达实时更新的。
2. **Storm有很多优势**

 1）它使用很简单，多种语言均可以使用它，它也有很多使用者；

2）它处理数据非常快，基本每秒可以处理一百万组单元数据；

3）它伸缩性很强，容错性很好，可以保证数据一直被处理

4）它很容易设置以及操作

**（4）Storm使用场景**

1）实时分析

2）机器算法

3）持续计算

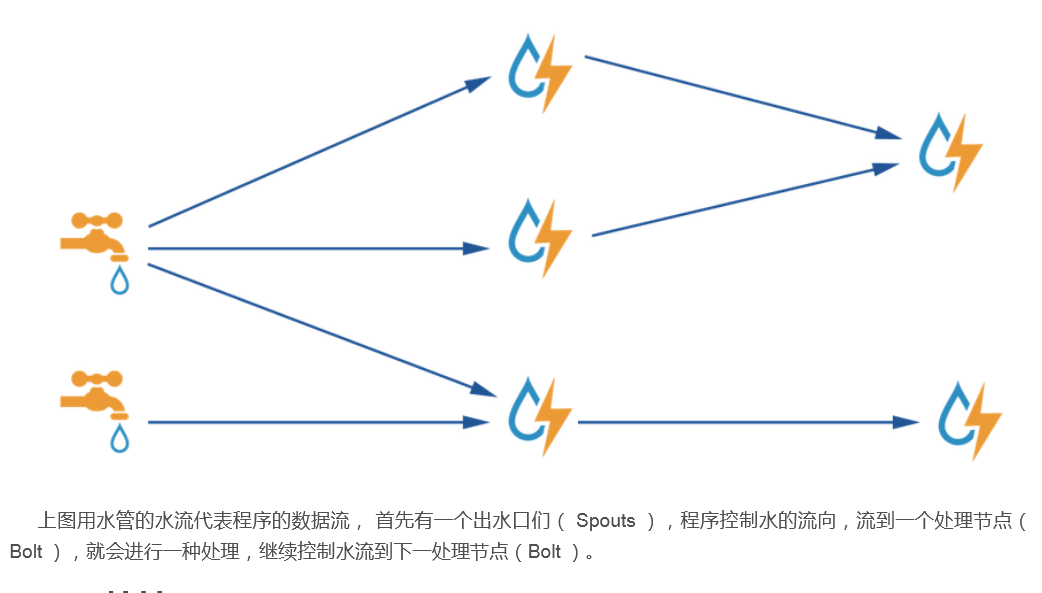
4）分布式远程服务调用

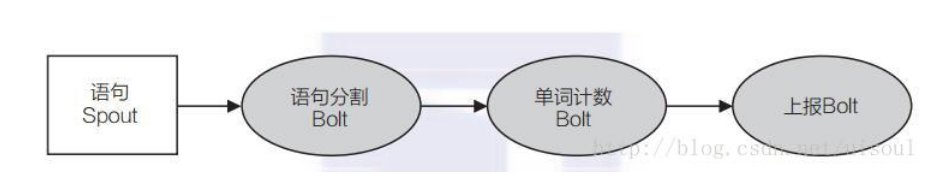
5）数据仓库技术

6）还有更多功能待挖掘

storm将会集成消息与数据库技术，storm拓扑在数据流和更复杂的处理流处理中，可以实现每个流计算的需要。

**（5）storm的工作流程**





**2 Storm的核心组件**

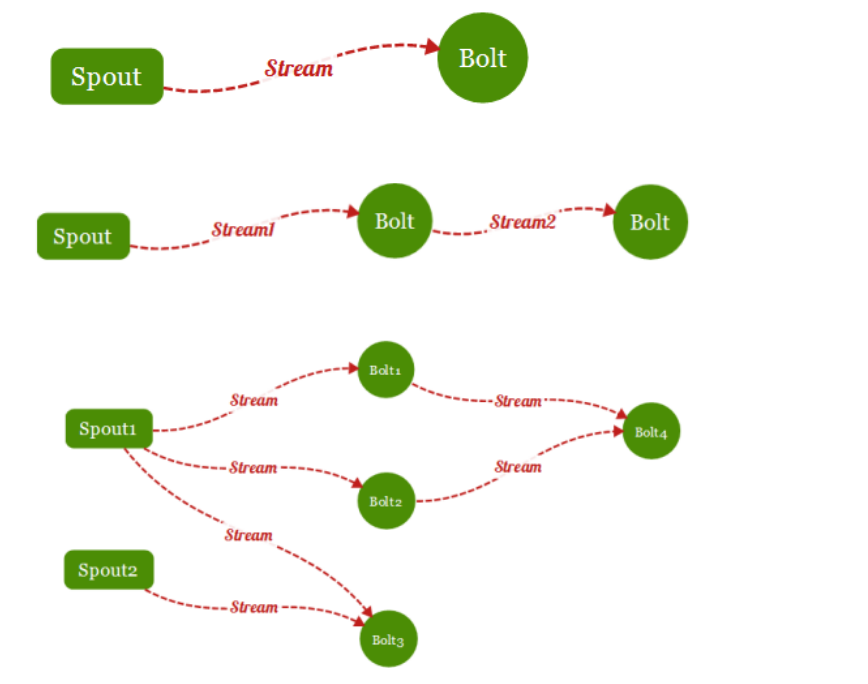
1. **Nimbus：**即Storm的Master，负责资源分配和任务调度。一个Storm集群只有一个Nimbus。

**（2）Supervisor**：即Storm的Slave，负责接收Nimbus分配的任务，管理所有Worker，一个Supervisor节点中包含多个Worker进程。

**（3）Worker：**工作进程，每个工作进程中都有多个Task。

**（4）Task：**任务，在 Storm 集群中每个 Spout 和 Bolt 都由若干个任务（tasks）来执行。每个任务都与一个执行线程相对应。

**（5）Topology：**计算拓扑，Storm 的拓扑是对实时计算应用逻辑的封装，它的作用与 MapReduce 的任务（Job）很相似，区别在于 MapReduce 的一个 Job 在得到结果之后总会结束，而拓扑会一直在集群中运行，直到你手动去终止它。拓扑还可以理解成由一系列通过数据流（Stream Grouping）相互关联的 Spout 和 Bolt 组成的的拓扑结构。Topology和MapReduce很相像。MapReduce是Map进行获取数据，Reduce进行处理数据。而Topology则是使用Spout获取数据，Bolt来进行计算。**总的来说就是一个Topology由一个或者多个的Spout和Bolt组成。**



**图片有三种模式，解释如下:**

**第一种比较简单**，就是由一个Spout获取数据，然后交给一个Bolt进行处理;

**第二种稍微复杂点**，由一个Spout获取数据，然后交给一个Bolt进行处理一部分，然后在交给下一个Bolt进行处理其他部分。

**第三种则比较复杂**，一个Spout可以同时发送数据到多个Bolt，而一个Bolt也可以接受多个Spout或多个Bolt，最终形成多个数据流。但是这种数据流必须是有方向的，有起点和终点，不然会造成死循环，数据永远也处理不完。就是Spout发给Bolt1，Bolt1发给Bolt2，Bolt2又发给了Bolt1,最终形成了一个环状。

1. **Stream：**数据流（Streams）是 Storm 中最核心的抽象概念。一个数据流指的是在分布式环境中并行创建、处理的一组元组（tuple）的无界序列。数据流可以由一种能够表述数据流中元组的域（fields）的模式来定义。
2. **Spout：**数据源（Spout）是拓扑中数据流的来源。一般 Spout 会从一个外部的数据源读取元组然后将他们发送到拓扑中。根据需求的不同，Spout 既可以定义为可靠的数据源，也可以定义为不可靠的数据源。一个可靠的 Spout能够在它发送的元组处理失败时重新发送该元组，以确保所有的元组都能得到正确的处理；相对应的，不可靠的 Spout 就不会在元组发送之后对元组进行任何其他的处理。一个 Spout可以发送多个数据流。
3. **Bolt：**拓扑中所有的数据处理均是由 Bolt 完成的。通过数据过滤（filtering）、函数处理（functions）、聚合（aggregations）、联结（joins）、数据库交互等功能，Bolt 几乎能够完成任何一种数据处理需求。一个 Bolt 可以实现简单的数据流转换，而更复杂的数据流变换通常需要使用多个 Bolt 并通过多个步骤完成。
4. **Stream grouping：**为拓扑中的每个 Bolt 的确定输入数据流是定义一个拓扑的重要环节。数据流分组定义了在 Bolt 的不同任务（tasks）中划分数据流的方式。在 Storm 中有八种内置的数据流分组方式。

**（10）Reliability：**可靠性。Storm 可以通过拓扑来确保每个发送的元组都能得到正确处理。通过跟踪由 Spout 发出的每个元组构成的元组树可以确定元组是否已经完成处理。每个拓扑都有一个“消息延时”参数，如果 Storm 在延时时间内没有检测到元组是否处理完成，就会将该元组标记为处理失败，并会在稍后重新发送该元组。

**3 Storm特性**

**（1）简单方便**

storm拥有简单方便的API，当用storm编程时，开发者可以很巧妙的处理和转换元组的数据流，一个元组就是一个有名字的列表中的一个值 ，元组里面可以包含任何对象或者类型，当你想用某一个类型的时候，storm可以很轻松为那个类型序列化。 在storm中仅有三个抽象的概念： **spouts（发射器）, bolts（闪电），topologies （拓扑）。**

**（2）可伸缩**

storm拓扑是并行的运行在集群当中，拓扑不同的部分可以通过调整他们的并行性进行单独扩展，在storm客户端通过 rebalance  命令可以动态调整运行中的拓扑的并行（平衡度），storm与生俱来的并行性意味着它可以低延迟的处理大吞吐量的消息，通过测试得知，在处理器： 2x Intel E5645@2.4Ghz   ，内存：24G 的服务器环境下，storm可以完成 每节点一百万每秒100字节的信息。

**（3）高容错（自救）**

storm有很高的容错性，当处理任务死亡时，storm将会自动重新启动他们，如果一个节点服务器死亡时，storm将会自动把处理任务在其他节点服务器运行。 storm的守护进程、矢量图和监测者均被设计为无状态模式和快速死亡，所以，当他们死亡时，他们会自动重启，就像什么都没发生过似的，这意味着如果你强制实行 kill -9命令杀死storm进程，也不会影响你的拓扑正常运行。

**（4）高可用（零容错）**

storm可以保证每个元组均被正常执行，storm的核心机制之一，就是它可以追踪到每一个元组关系，这是保证每个元组不丢失的有效方法。 storm提供了一个“至少一次”处理的机制，当使用一个队列系统时，你将使用同一个机制，当他们失败时，消息就会被重新发送。 Trident是storm更高级别的抽象语义，它可以保证消息只被执行一次。

**（5）跨语言**

storm可以在任何编程语言中使用，storm的核心是通过Thrift规范来定义和提交拓扑，Thrift是可以跨语言调用，所以拓扑也可以任何编程语言中定义或者提交。 同样的， spouts（发射器）和bolts（闪电） 也可以在任何编程语言中定义或者提交，无虚拟机的spouts（发射器）和bolts（闪电）的输入/输出将会基于json的基础协议进行通讯。通讯适配器用 Ruby, Python, Javascript, Perl 这些语言的协议进行了实现。

**（6）集群易搭建**

 storm只需要很少的配置和设置就可以运行起来，storm生产配置开箱即用，EC2可以一键发布、配置storm项目集群。另外，storm发布的时候操作很简单，storm的设计很强壮，集群可以日复一日的运行下去。

**4 Storm架构**

（1）Nimbus负责代码分发、任务发布、监控集群状态。

（2）Supervisor负责执行分派到的任务，启动或停止工作进程。

（3）UI是一个能看到集群信息的网页。

（4）ZooKeeper是Hadoop的正式子项目，它是一个针对大型分布式系统的可靠协调系统，提供的功能包括：配置维护、名字服务、分布式同步、组服务等。ZooKeeper的目标就是封装好复杂易出错的关键服务，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

（5）Storm和Zookeeper的关系：，Storm依靠ZooKeeper集群实现节点间的信息交换

（6）Storm、ZooKeeper和ZeroMQ的关系

Nimbus通过Zookeeper发布任务、管理节点、监控集群状态、统计信息。

Supervisor通过ZooKeeper注册节点状态、领取任务。

Nimbus和Supervisor之间通过ZeroMQ传递消息和数据。

（7）Spouts：数据源，是消息生产者，他会从一个外部源读取数据并向topology里面面发出消息（tuple）。

（8）Stream：数据流，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地创建和处理。

（9）Bolts：消息处理者，所有的消息处理逻辑被封装在bolts里面，处理输入的数据流并产生输出的新数据流,可执行过滤，聚合，查询数据库等操作。

**5 Storm和 Spark 对比**

|  |  |  |
| --- | --- | --- |
| **对比点** | **Storm** | **Spark Streaming** |
| **实时计算模型** | 纯实时，来一条数据，处理一条数据 | 准实时，对一个时间段内的数据收集起来，作为一个RDD，再处理 |
| **实时计算延迟度** | 毫秒级 | 秒级 |
| **吞吐量** | 低 | 高 |
| **事务机制** | 支持完善 | 支持，但不够完善 |
| **健壮性 / 容错性** | ZooKeeper，Acker，非常强 | Checkpoint，WAL，一般 |
| **动态调整并行度** | 支持 | 不支持 |

**6 Spark Streaming与Storm的应用场景**

**对于Storm来说：**  
1）、建议在那种需要纯实时，不能忍受1秒以上延迟的场景下使用，比如实时金融系统，要求纯实时进行金融交易和分析  
2）、此外，如果对于实时计算的功能中，要求可靠的事务机制和可靠性机制，即数据的处理完全精准，一条也不能多，一条也不能少，也可以考虑使用Storm  
3）、如果还需要针对高峰低峰时间段，动态调整实时计算程序的并行度，以最大限度利用集群资源（通常是在小型公司，集群资源紧张的情况），也可以考虑用Storm  
4）、如果一个大数据应用系统，它就是纯粹的实时计算，不需要在中间执行SQL交互式查询、复杂的transformation算子等，那么用Storm是比较好的选择

**对于Spark Streaming来说：**  
1）、如果对上述适用于Storm的三点，一条都不满足的实时场景，即，不要求纯实时，不要求强大可靠的事务机制，不要求动态调整并行度，那么可以考虑使用Spark Streaming  
2）、考虑使用Spark Streaming最主要的一个因素，应该是针对整个项目进行宏观的考虑，即，如果一个项目除了实时计算之外，还包括了离线批处理、交互式查询等业务功能，而且实时计算中，可能还会牵扯到高延迟批处理、交互式查询等功能，那么就应该首选Spark生态，用Spark Core开发离线批处理，用Spark SQL开发交互式查询，用Spark Streaming开发实时计算，三者可以无缝整合，给系统提供非常高的可扩展性

**7 Spark Streaming与Storm的优劣分析**

（1）事实上，Spark Streaming绝对谈不上比Storm优秀。这两个框架在实时计算领域中，都很优秀，只是擅长的细分场景并不相同。

（2）**Spark Streaming仅仅在吞吐量上比Storm要优秀，而吞吐量这一点，也是历来挺Spark Streaming**，贬Storm的人着重强调的。但是问题是，是不是在所有的实时计算场景下，都那么注重吞吐量？不尽然。因此，通过吞吐量说Spark Streaming强于Storm，不靠谱。事实上，Storm在实时延迟度上，比Spark Streaming就好多了，前者是纯实时，后者是准实时。而且，**Storm的事务机制、健壮性 / 容错性、动态调整并行度等特性，都要比Spark Streaming更加优秀。**

（3）Spark Streaming，有一点是Storm绝对比不上的，就是：它位于Spark生态技术栈中，因此Spark Streaming可以和Spark Core、Spark SQL无缝整合，也就意味着，我们可以对实时处理出来的中间数据，立即在程序中无缝进行延迟批处理、交互式查询等操作。这个特点大大增强了Spark Streaming的优势和功能。

**8 Storm事务**

**（1）Spout**

ITransactionalSpout<T>，同BaseTransactionalSpout<T>，普通事务Spout

IPartitionedTransactionalSpout<T>，同BasePartitionedTransactionalSpout<T>，分区事务Spout

IOpaquePartitionedTransactionalSpout<T>：同BaseOpaquePartitionedTransactionalSpout<T>，不透明分区事务Spout

**（2）Bolt**

IBatchBolt<T>：同BaseBatchBolt<T>，普通批处理

BaseTransactionalBolt：事务Bolt

接口Icommitter：标识IBatchBolt 或BaseTransactionalBolt是否是一个committer CoordinatedBolt

**（3）传统事务**

ITransactionalSpout<T>：普通事务Spout

-- ITransactionalSpout.Coordinator<X>

--initializeTransaction(BigInteger txid, XprevMetadata) ：

创建一个新的metadata，当isReady() 为true时，发射该metadata（事务tuple）到“batchemit”流

--isReady() ：为true时启动新事务，需要时可以在此sleep

-- ITransactionalSpout.Emitter<X>

--emitBatch(TransactionAttempt tx,X coordinatorMeta, BatchOutputCollector collector) ：逐个发射batch的tuple

**（4）分区事务**

IPartitionedTransactionalSpout<T>：分区事务Spout，主流事务Spout，原因是目前主流MessageQueue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

--isReady()：同上

--numPartitions()：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<X>

--emitPartitionBatchNew(TransactionAttempt tx,BatchOutputCollector collector,int partition,X lastPartitionMeta)：发射一个新的Batch，返回Metadata

--emitPartitionBatch(TransactionAttempt tx,BatchOutputCollector collector,int partition,X partitionMeta)：如果这批消息Bolt消费失败了，emitPartitionBatch负责重发这批消息

**（5）不透明分区事务**

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

--IOpaquePartitionedTransactionalSpout.Coordinator

--isReady()：同上

--IOpaquePartitionedTransactionalSpout.Emitter<X>

-- emitPartitionBatch(TransactionAttempt tx,BatchOutputCollector collector,int partition,X lastPartitionMeta)

-- numPartitions()

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），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

}

**（6）IPartitionedTransactionalSpout和IOpaquePartitionedTransactionalSpout的区别与联系**

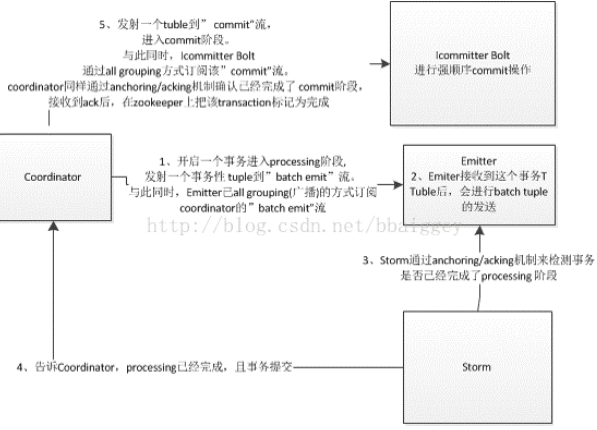
1）IPartitionedTransactionalSpout和IOpaquePartitionedTransactionalSpout都是把tuple封装成batch进行处理，同时可以保证每一个tuple都被完整地处理，都支持消息重发。为了支持事务性，它们为每一个批次（batch）提供一个唯一的事务ID（transaction id：txid），txid是顺序递增的，而且保证对批次的处理是强有序的，即必须完整处理完txid=1才能再接着处理txid=2。

2）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。

1. 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。

4）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。



**9 Storm并行度详解**

**（1）Storm并行度相关的概念**

Storm集群有很多节点，按照类型分为nimbus（主节点）、supervisor（从节点），在conf/storm.yaml中配置了一个supervisor,有多个槽（supervisor.slots.ports），每个槽就是一个JVM，就是一个worker(一个节点，运行一个worker)，在每个worker里面可以运行多个线程叫做executor，在executor里运行一个topology的一个component（spout、bolt）叫做task。task 是storm中进行计算的最小的运行单位，表示是spout或者bolt的运行实例。

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

程序执行的最大粒度的运行单位是进程，刚才说的task也是需要有进程来运行它的，在supervisor中，运行task的进程称为worker，Supervisor节点上可以运行非常多的worker进程，一般在一个进程中是可以启动多个线程的，所以我们可以在worker中运行多个线程，这些线程称为executor，在executor中运行task。

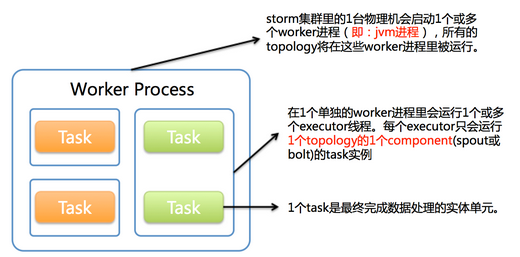
**（2）提高storm的并行度，可 考虑如下几点：**

worker(进程)>executor(线程)>task(实例)

增加work进程，增加executor线程，增加task实例

**（3）看下面的图：**

这表示是一个work进程，其实就是一个jvm虚拟机进程，在这个work进程里面有多个executor线程，每个executor线程会运行一个或多个task实例。一个task是最终完成数据处理的实体单元。(默认情况下一个executor运行一个task).



**（4）worker,executor,task解释**

**1个worker进程执行的是1个topology的子集**（注：不会出现1个worker为多个topology服务）。1个worker进程会启动1个或多个executor线程来执行1个topology的component(spout或bolt)。因此，1个运行中的topology就是由集群中多台物理机上的多个worker进程组成的。

**executor是1个被worker进程启动的单独线程。**每个executor只会运行1个topology的1个component(spout或bolt)的task（注：task可以是1个或多个，storm默认是1个component只生成1个task，executor线程里会在每次循环里顺序调用所有task实例）。

t**ask是最终运行spout或bolt中代码的单元**（注：1个task即为spout或bolt的1个实例，executor线程在执行期间会调用该task的nextTuple或execute方法）。topology启动后，1个component(spout或bolt)的task数目是固定不变的，但该component使用的executor线程数可以动态调整（例如：1个executor线程可以执行该component的1个或多个task实例）。这意味着，对于1个component存在这样的条件：#threads<=#tasks（即：线程数小于等于task数目）。默认情况下task的数目等于executor线程数目，即1个executor线程只运行1个task。

**（5）worker(进程)>executor(线程)>task(实例) 是如何设置的**

**1）worker(进程)：**这个worker进程数量是在集群启动之前配置好的，在哪配置的呢？是在storm/conf/storm.yaml文件中，参数是supervisor.slots.port，如果我们不在这进行配置的话，这个参数也是有默认值的，在strom-0.9.3的压缩包中的lib目录下，有一个strom-core.jar，打开这个jar文件，在里面有一个defaults.yaml文件中是有一些默认配置的。**默认情况下一个storm项目只使用一个work进程**，也可以通过代码进行修改，通过config.setNumWorkers(workers)设置。(最好一台机器上的一个topology只使用一个worker,主要原因时减少了worker之间的数据传输)

**2） executor(线程)：**默认情况下一个executor运行一个task，可以通过在代码中设置**builder.setSpout(id,spout, parallelism\_hint);或者builder.setBolt(id,bolt,parallelism\_hint);**来提高线程数的。

**3） task(实例)：**通过boltDeclarer.setNumTasks(num);来设置实例的个数

默认情况下，**一个supervisor节点会启动4个worker进程。每个worker进程会启动1个executor，每个executor启动1个task。**

**（6）举个例子看一下对这些配置修改之后的效果**

**1） worker(进程)**，通过在代码中设置，可以在ui界面上查看worker的总数，并且还可以在linux服务器上执行jps查看work进程。在代码中设置使用3个worker，查看ui界面，发现workers是3个，executors使用了5个，为什么呢？因为每一个worker默认都会占用一个executor（这个executor会启动一个acker任务），这样就会占用三个，剩下的两个是spout和bolt实例占用了。如果使用5个worker，executor会使用7个，因为worker本身就会占用5个，spout和bolt占用两个。

**2） executor(线程)**，在spout和bolt中设置线程数，都设置为2个，查看ui界面使用的executor和tasks就是7个了，因为worker本身使用3个，spout和bolt分别使用2个。

**3） task(实例)**，在sum中设置实例个数为5，查看ui界面发现显示的tasks是10，因为spout占用2个，bolt占用5个，剩下的3个由acker任务占用

注意：虽然在这设置了多个task实例，但是并行度并没有很大提高，因为只有两个线程去运行这些实例，只有设置足够多的线程和实例才可以真正的提高并行度。在这设置多个实例主要是为了下面执行rebalance的时候用到，因为rebalance不需要修改代码，就可以动态修改topology的并行度，这样的话就必须提前配置好多个实例，在rebalance的时候主要是对之前设置多余的任务实例分配线程去执行。

**（7）在命令行动态修改并行度**

除了使用代码进行调整，还可以在shell命令行下对并行度进行调整。

**storm rebalance mytopology -w 10 -n 2 -e spout=2 -e bolt=2**

表示 10秒之后对mytopology进行并行度调整。把spout调整为2个executor，把bolt调整为2个executor。

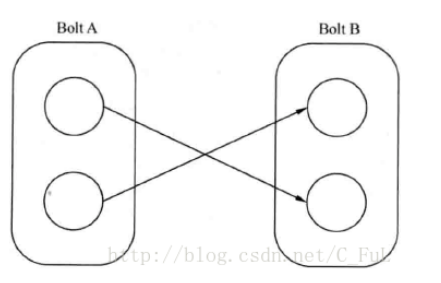
注意：并行度主要就是调整executor的数量，但是调整之后的executor的数量必须小于等于task的数量，如果分配的executor的线程数比task数量多的话也只能分配和task数量相等的executor。

**10 Storm分组方式**

在Storm中有7中内置的分组方式，也可以通过实现CustomStreamGrouping接口来自定义分组。

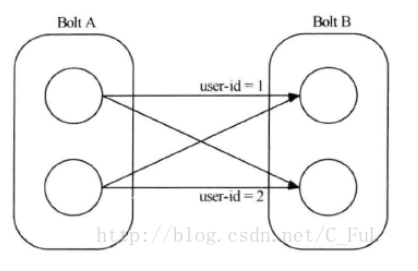
**（1）Shuffle 分组**

Task中的数据随机分配，可以保证同一级Bolt上的每个Task处理的Tuple数量一致



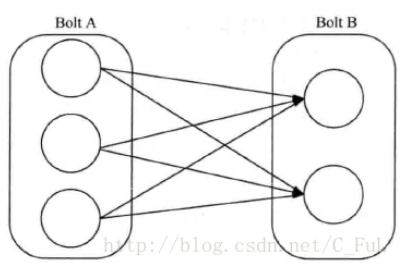
**（2）Fields 分组**

根据Tuple中的某一个Field或多个Field的值来划分。比如Stream根据user-id的值来分组，具有相同user-id值的Tuple会被分发到相同的Task中。如下图（具有不同user-id值的Tuple可能会被分发到其它Task中。比如user-id为1的Tuple都会分发给Task1，userid-2的Tuple可能在Task1上也可能在Task2上，但是同时只能在一个Task上）



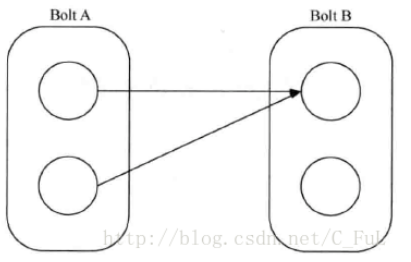
**（3）All分组**

所有的Tuple都会分发到所有的Task上



**（4）Global 分组**

整个Storm会选择一个Task作为分发的目的地，通常是具有最新ID的Task



**（5）None 分组**

也就是你不关系如何在Task中做Stream的分发，目前等同于Shuffle分组

**（6）Direct 分组**

这是一种特殊的分组方式，也就是产生数据的Spout/Blot自己明确决定这个Tuple被Bolt的哪些Task所消费，如果使用Direct分组，需要使用OutputColletor的emitDirect方法来实现。

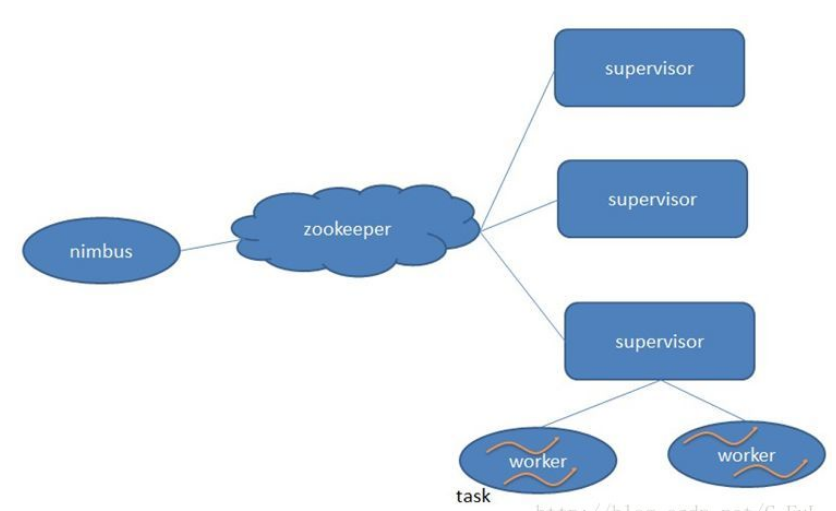
**（7）Local or Shuffle 分组**

如果目标Bolt中的一个或多个Task和当前产生数据的Task在同一个Worker进程中，那么就走内部的线程通信，将Tuple直接发给当前Worker进程中的目的Task。否则，通Shuffle分组

**11 Storm 核心组件**

1. Storm 的集群表面上看和Hadoop的非常像，但是在Hadoop上运行的是MapReduce的作业（job），而在Storm上运行的是是Topology，Storm和Hadoop一个非常关键的区别是Hadoop的MapReduce作业最终会结束，而Storm的Topology会一直运行（除非显示的杀掉它）如果说批处理的Hadoop需要一桶一桶地搬走水，那么Storm 就好比自来水水管，只要预先接好水管，然后打开水龙头，水就源源不断地流出来了，即消息就会被实时地处理。

（2）在Storm 集群中有两种节点：主节点（Master Node）Nimbus和工作节点（Worker Node）Supervisor。

@Nimbus：负责资源分配和任务调度。

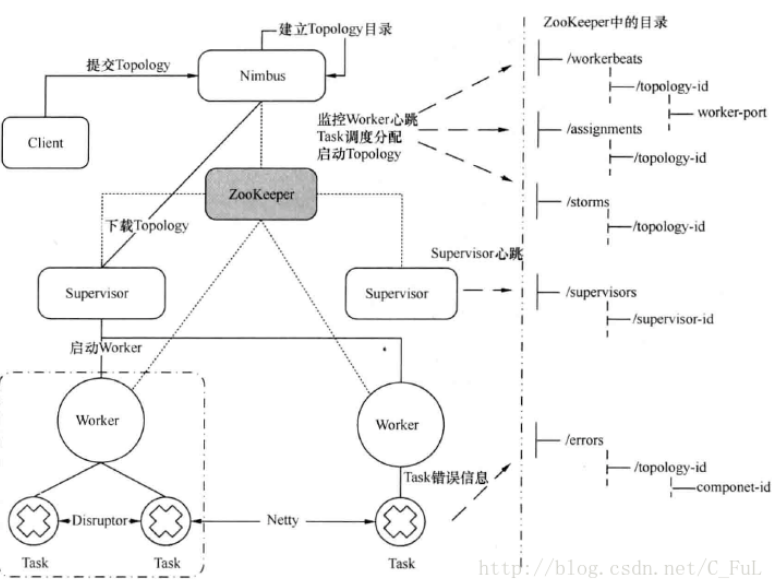
@Supervisor：负责接受Nimbus分配的任务，启动和停止属于自己管理的worker进程。通过配置文件设置当前supervisor上启动多少个worker。

@Worker：运行具体处理组件逻辑的进程（执行任务的具体组件）。Worker运行的任务类型只有两种，一种是Spout任务，一种是Bolt任务。

@Task：每一个Spout/Bolt具体要做的工作，也是各个节点之间进行分组的单位。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，不同spout/bolt的task可能会共享一个物理线程，该线程称为Executor。

@Zookeeper：保存任务分配的信息、心跳信息、元数据信息

**12 Storm 总体架构图**

(1). 客户端提交Topology到Nimbus

(2). Nimbus 针对该Topology建立本地目录

(3). Nimbus中的调度器根据Topology的配置计算Task，并把Task分配到不同的Worker上，调度的结果写入Zookeeper

(4). ZooKeeper上建立assignments节点，存储Task和Supervisor中Worker的对应关系。

(5). 在ZooKeeper上创建workerbeats节点来监控Worker的心跳。

(6). Supervisor 去 ZooKeeper上获取分配的Task信息，启动一个或多个Worker执行。每个Worker上运行多个Task，Task由Executor来具体执行。

(7) .Worker根据Topology信息初始化建立Task之间的连接，相同Worker内的Task通过DisrupterQueue来通信，不同Worker间默认采用Netty通信。

(8). 然后整个Topology就运行起来了

**13 Storm运行模式**

（1）本地模式(Local Mode)： 即Topology（相当于一个任务，后续会详细讲解） 运行在本地机器的单一JVM上，这个模式主要用来开发、调试。

（2）远程模式(Remote Mode):在这个模式，我们把我们的Topology提交到集群，在这个模式中，Storm的所有组件都是线程安全的，因为它们都会运行在不同的Jvm或物理机器上，这个模式就是正式的生产模式。

## 二、java实例

**1 环境准备**

在进行代码开发之前，首先得做好相关的准备。

本项目是使用Maven构建的，使用Storm的版本为1.1.1。

Maven的相关依赖如下:

<!--storm相关jar -->

<dependency>

<groupId>org.apache.storm</groupId>

<artifactId>storm-core</artifactId>

<version>1.1.1</version>

<scope>provided</scope>

</dependency>

**2 具体流程**

在写代码的时候，我们先来明确要用Storm做什么。

那么第一个程序，就简单的输出下信息。具体步骤如下

（1）启动topology，设置好Spout和Bolt。

（2）将Spout获取的数据传递给Bolt。

（3）Bolt接受Spout的数据进行打印。

**3 Spout**

那么首先开始编写Spout类。一般是实现 IRichSpout 或继承BaseRichSpout该类，然后实现该方法。

这里我们继承BaseRichSpout这个类,该类需要实现这几个主要的方法:

**（1）open**

open()方法中是在ISpout接口中定义，在Spout组件初始化时被调用。

有三个参数，它们的作用分别是:

Storm配置的Map;

topology中组件的信息;

发射tuple的方法;

代码示例:

@Override

public void open(Map map, TopologyContext arg1, SpoutOutputCollector collector) {

System.out.println("open:"+map.get("test"));

this.collector = collector;

}

**（2）nextTuple**

nextTuple()方法是Spout实现的核心。

也就是主要执行方法，用于输出信息,通过collector.emit方法发射。

这里我们的数据信息已经写死了，所以这里我们就直接将数据进行发送。

这里设置只发送两次。

代码示例:

@Override

public void nextTuple() {

if(count<=2){

System.out.println("第"+count+"次开始发送数据...");

this.collector.emit(new Values(message));

}

count++;

}

**（3）declareOutputFields**

declareOutputFields是在IComponent接口中定义，用于声明数据格式。

即输出的一个Tuple中，包含几个字段。

因为这里我们只发射一个，所以就指定一个。如果是多个，则用逗号隔开。

代码示例:

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

System.out.println("定义格式...");

declarer.declare(new Fields(field));

}

**（4）ack**

ack是在ISpout接口中定义，用于表示Tuple处理成功。

代码示例:

@Override

public void ack(Object obj) {

System.out.println("ack:"+obj);

}

**（5）fail**

fail是在ISpout接口中定义，用于表示Tuple处理失败。

代码示例:

@Override

public void fail(Object obj) {

System.out.println("失败:"+obj);

}

**（6）close**

close是在ISpout接口中定义，用于表示Topology停止。

代码示例:

@Override

public void close() {

System.out.println("关闭...");

}

至于还有其他的，这里就不在一一列举了。

**4 Bolt**

Bolt是用于处理数据的组件，主要是由execute方法来进行实现。一般来说需要实现 IRichBolt 或继承BaseRichBolt该类，然后实现其方法。

需要实现方法如下:

**（1）prepare**

在Bolt启动前执行，提供Bolt启动环境配置的入口。

参数基本和Sqout一样。

一般对于不可序列化的对象进行实例化。

这里的我们就简单的打印下

@Override

public void prepare(Map map, TopologyContext arg1, OutputCollector collector) {

System.out.println("prepare:"+map.get("test"));

this.collector=collector;

}

注:如果是可以序列化的对象，那么最好是使用构造函数。

**（2）execute**

execute()方法是Bolt实现的核心。

也就是执行方法，每次Bolt从流接收一个订阅的tuple，都会调用这个方法。

从tuple中获取消息可以使用 tuple.getString()和tuple.getStringByField();这两个方法。个人推荐第二种，可以通过field来指定接收的消息。

注:如果继承的是IRichBolt，则需要手动ack。这里就不用了,BaseRichBolt会自动帮我们应答。

代码示例:

@Override

public void execute(Tuple tuple) {

// String msg=tuple.getString(0);

String msg=tuple.getStringByField("test");

//这里我们就不做消息的处理，只打印

System.out.println("Bolt第"+count+"接受的消息:"+msg);

count++;

/\*\*

\*

\* 没次调用处理一个输入的tuple，所有的tuple都必须在一定时间内应答。

\* 可以是ack或者fail。否则，spout就会重发tuple。

\*/

// collector.ack(tuple);

}

**（3）declareOutputFields**

和Spout的一样。

因为到了这里就不再输出了，所以就什么都没写。

@Override

public void declareOutputFields(OutputFieldsDeclarer arg0) {

}

**（4）cleanup**

cleanup是IBolt接口中定义,用于释放bolt占用的资源。

Storm在终止一个bolt之前会调用这个方法。

因为这里没有什么资源需要释放，所以就简单的打印一句就行了。

@Override

public void cleanup() {

System.out.println("资源释放");

}

**（5）Topology**

这里我们就是用main方法进行提交topology。

不过在提交topology之前，需要进行相应的设置。

这里我就不一一细说了，代码的注释已经很详细了。

代码示例:

import org.apache.storm.Config;

import org.apache.storm.LocalCluster;

import org.apache.storm.StormSubmitter;

import org.apache.storm.topology.TopologyBuilder;

public class App {

private static final String str1="test1";

private static final String str2="test2";

public static void main(String[] args) {

// TODO Auto-generated method stub

//定义一个拓扑

TopologyBuilder builder=new TopologyBuilder();

//设置一个Executeor(线程)，默认一个

builder.setSpout(str1, new TestSpout());

//设置一个Executeor(线程)，和一个task

builder.setBolt(str2, new TestBolt(),1).setNumTasks(1).shuffleGrouping(str1);

Config conf = new Config();

conf.put("test", "test");

try{

//运行拓扑

if(args !=null&&args.length>0){ //有参数时，表示向集群提交作业，并把第一个参数当做topology名称

System.out.println("远程模式");

StormSubmitter.submitTopology(args[0], conf, builder.createTopology());

} else{//没有参数时，本地提交

//启动本地模式

System.out.println("本地模式");

LocalCluster cluster = new LocalCluster();

cluster.submitTopology("111" ,conf, builder.createTopology() );

Thread.sleep(10000);

// 关闭本地集群

cluster.shutdown();

}

}catch (Exception e){

e.printStackTrace();

}

}

}

运行该方法，输出结果如下:

本地模式

定义格式...

open:test

第1次开始发送数据...

第2次开始发送数据...

prepare:test

Bolt第1接受的消息:这是个测试消息!

Bolt第2接受的消息:这是个测试消息!

资源释放

关闭...

到这里，是不是基本上对Storm的运作有些了解了呢。

这个demo达到了上述的三种模式图中的第一种，一个Spout传输数据， 一个Bolt处理数据。

**5 第二种模式做法**

假如我们想统计下在一段文本中的单词出现频率的话，我们只需执行一下步骤就可以了。

（1）.首先将Spout中的message消息进行更改为数组，并依次将消息发送到TestBolt。

（2）.然后TestBolt将获取的数据进行分割，将分割的数据发送到TestBolt2。

（3）.TestBolt2对数据进行统计，在程序关闭的时候进行打印。

（4）.Topology成功配置并且启动之后，等待20秒左右，关闭程序，然后得到输出的结果。

**6 Spout代码示例用于发送消息。**

import java.util.Map;

import org.apache.storm.spout.SpoutOutputCollector;

import org.apache.storm.task.TopologyContext;

import org.apache.storm.topology.OutputFieldsDeclarer;

import org.apache.storm.topology.base.BaseRichSpout;

import org.apache.storm.tuple.Fields;

import org.apache.storm.tuple.Values;

public class TestSpout extends BaseRichSpout{

private static final long serialVersionUID = 225243592780939490L;

private SpoutOutputCollector collector;

private static final String field="word";

private int count=1;

private String[] message = {

"My nickname is xuwujing",

"My blog address is http://www.panchengming.com/",

"My interest is playing games"

};

/\*\*

\* open()方法中是在ISpout接口中定义，在Spout组件初始化时被调用。

\* 有三个参数:

\* 1.Storm配置的Map;

\* 2.topology中组件的信息;

\* 3.发射tuple的方法;

\*/

@Override

public void open(Map map, TopologyContext arg1, SpoutOutputCollector collector) {

System.out.println("open:"+map.get("test"));

this.collector = collector;

}

/\*\*

\* nextTuple()方法是Spout实现的核心。

\* 也就是主要执行方法，用于输出信息,通过collector.emit方法发射。

\*/

@Override

public void nextTuple() {

if(count<=message.length){

System.out.println("第"+count +"次开始发送数据...");

this.collector.emit(new Values(message[count-1]));

}

count++;

}

/\*\*

\* declareOutputFields是在IComponent接口中定义，用于声明数据格式。

\* 即输出的一个Tuple中，包含几个字段。

\*/

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

System.out.println("定义格式...");

declarer.declare(new Fields(field));

}

/\*\*

\* 当一个Tuple处理成功时，会调用这个方法

\*/

@Override

public void ack(Object obj) {

System.out.println("ack:"+obj);

}

/\*\*

\* 当Topology停止时，会调用这个方法

\*/

@Override

public void close() {

System.out.println("关闭...");

}

/\*\*

\* 当一个Tuple处理失败时，会调用这个方法

\*/

@Override

public void fail(Object obj) {

System.out.println("失败:"+obj);

}

}

**7 TestBolt用于分割单词。**

import java.util.Map;

import org.apache.storm.task.OutputCollector;

import org.apache.storm.task.TopologyContext;

import org.apache.storm.topology.OutputFieldsDeclarer;

import org.apache.storm.topology.base.BaseRichBolt;

import org.apache.storm.tuple.Fields;

import org.apache.storm.tuple.Tuple;

import org.apache.storm.tuple.Values;

public class TestBolt extends BaseRichBolt{

/\*\*

\*

\*/

private static final long serialVersionUID = 4743224635827696343L;

private OutputCollector collector;

/\*\*

\* 在Bolt启动前执行，提供Bolt启动环境配置的入口

\* 一般对于不可序列化的对象进行实例化。

\* 注:如果是可以序列化的对象，那么最好是使用构造函数。

\*/

@Override

public void prepare(Map map, TopologyContext arg1, OutputCollector collector) {

System.out.println("prepare:"+map.get("test"));

this.collector=collector;

}

/\*\*

\* execute()方法是Bolt实现的核心。

\* 也就是执行方法，每次Bolt从流接收一个订阅的tuple，都会调用这个方法。

\*/

@Override

public void execute(Tuple tuple) {

String msg=tuple.getStringByField("word");

System.out.println("开始分割单词:"+msg);

String[] words = msg.toLowerCase().split(" ");

for (String word : words) {

this.collector.emit(new Values(word));//向下一个bolt发射数据

}

}

/\*\*

\* 声明数据格式

\*/

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

declarer.declare(new Fields("count"));

}

/\*\*

\* cleanup是IBolt接口中定义,用于释放bolt占用的资源。

\* Storm在终止一个bolt之前会调用这个方法。

\*/

@Override

public void cleanup() {

System.out.println("TestBolt的资源释放");

}

}

**8 Test2Bolt用于统计单词出现次数。**

import java.util.HashMap;

import java.util.Map;

import org.apache.storm.task.OutputCollector;

import org.apache.storm.task.TopologyContext;

import org.apache.storm.topology.OutputFieldsDeclarer;

import org.apache.storm.topology.base.BaseRichBolt;

import org.apache.storm.tuple.Tuple;

public class Test2Bolt extends BaseRichBolt{

/\*\*

\*

\*/

private static final long serialVersionUID = 4743224635827696343L;

/\*\*

\* 保存单词和对应的计数

\*/

private HashMap<String, Integer> counts = null;

private long count=1;

/\*\*

\* 在Bolt启动前执行，提供Bolt启动环境配置的入口

\* 一般对于不可序列化的对象进行实例化。

\* 注:如果是可以序列化的对象，那么最好是使用构造函数。

\*/

@Override

public void prepare(Map map, TopologyContext arg1, OutputCollector collector) {

System.out.println("prepare:"+map.get("test"));

this.counts=new HashMap<String, Integer>();

}

/\*\*

\* execute()方法是Bolt实现的核心。

\* 也就是执行方法，每次Bolt从流接收一个订阅的tuple，都会调用这个方法。

\*

\*/

@Override

public void execute(Tuple tuple) {

String msg=tuple.getStringByField("count");

System.out.println("第"+count+"次统计单词出现的次数");

/\*\*

\* 如果不包含该单词，说明在该map是第一次出现

\* 否则进行加1

\*/

if (!counts.containsKey(msg)) {

counts.put(msg, 1);

} else {

counts.put(msg, counts.get(msg)+1);

}

count++;

}

/\*\*

\* cleanup是IBolt接口中定义,用于释放bolt占用的资源。

\* Storm在终止一个bolt之前会调用这个方法。

\*/

@Override

public void cleanup() {

System.out.println("===========开始显示单词数量============");

for (Map.Entry<String, Integer> entry : counts.entrySet()) {

System.out.println(entry.getKey() + ": " + entry.getValue());

}

System.out.println("===========结束============");

System.out.println("Test2Bolt的资源释放");

}

/\*\*

\* 声明数据格式

\*/

@Override

public void declareOutputFields(OutputFieldsDeclarer arg0) {

}

}

**9 Topology主程序入口。**

import org.apache.storm.Config;

import org.apache.storm.LocalCluster;

import org.apache.storm.StormSubmitter;

import org.apache.storm.topology.TopologyBuilder;

import org.apache.storm.tuple.Fields;

/\*\*

\*

\* Title: App

\* Description:

\* storm测试

\* Version:1.0.0

\* @author pancm

\* @date 2018年3月6日

\*/

public class App {

private static final String test\_spout="test\_spout";

private static final String test\_bolt="test\_bolt";

private static final String test2\_bolt="test2\_bolt";

public static void main(String[] args) {

//定义一个拓扑

TopologyBuilder builder=new TopologyBuilder();

//设置一个Executeor(线程)，默认一个

builder.setSpout(test\_spout, new TestSpout(),1);

//shuffleGrouping:表示是随机分组

//设置一个Executeor(线程)，和一个task

builder.setBolt(test\_bolt, new TestBolt(),1).setNumTasks(1).shuffleGrouping(test\_spout);

//fieldsGrouping:表示是按字段分组

//设置一个Executeor(线程)，和一个task

builder.setBolt(test2\_bolt, new Test2Bolt(),1).setNumTasks(1).fieldsGrouping(test\_bolt, new Fields("count"));

Config conf = new Config();

conf.put("test", "test");

try{

//运行拓扑

if(args !=null&&args.length>0){ //有参数时，表示向集群提交作业，并把第一个参数当做topology名称

System.out.println("运行远程模式");

StormSubmitter.submitTopology(args[0], conf, builder.createTopology());

} else{//没有参数时，本地提交

//启动本地模式

System.out.println("运行本地模式");

LocalCluster cluster = new LocalCluster();

cluster.submitTopology("Word-counts" ,conf, builder.createTopology() );

Thread.sleep(20000);

// //关闭本地集群

cluster.shutdown();

}

}catch (Exception e){

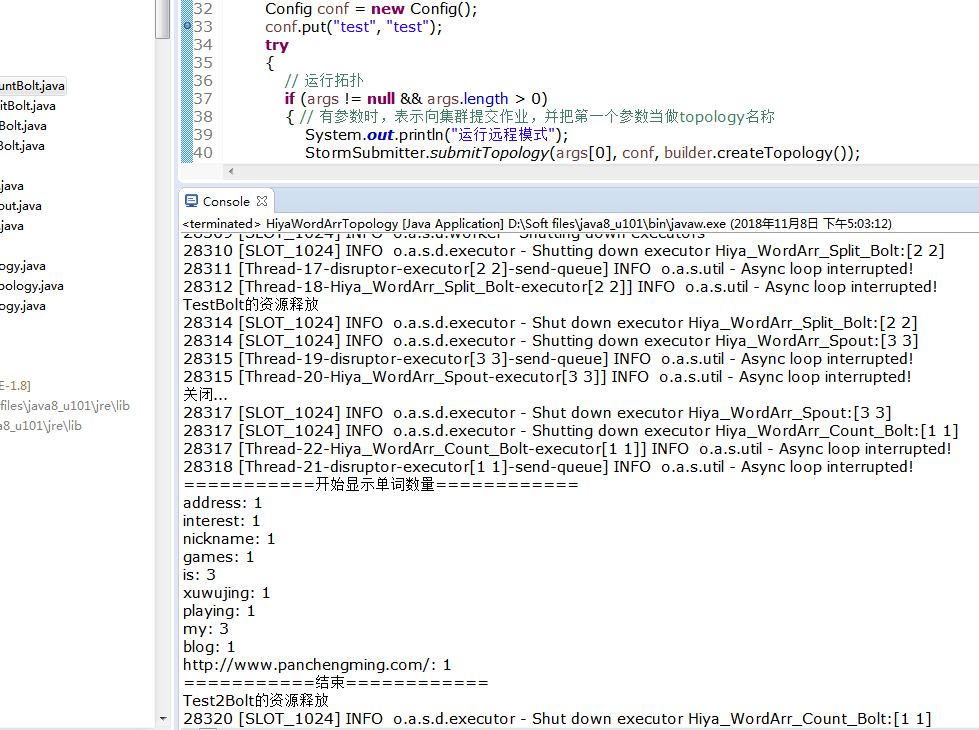
e.printStackTrace();

}

}

}

**10 输出结果:**



**11 操作模式**  
**（1）本地模式**   
在本地模式下，Storm拓扑结构运行在本地计算机的单一JVM进程上。这个模式用于开发、测试以及调试，因为这是观察所有组件如何协同工作的最简单方法。在这种模式下，我们可以调整参数，观察我们的拓扑结构如何在不同的Storm配置环境下运行。要在本地模式下运行，我们要下载Storm开发依赖，以便用来开发并测试我们的拓扑结构。我们创建了第一个Storm工程以后，很快就会明白如何使用本地模式了。   
NOTE: 在本地模式下，跟在集群环境运行很像。不过很有必要确认一下所有组件都是线程安全的，因为当把它们部署到远程模式时它们可能会运行在不同的JVM进程甚至不同的物理机上，这个时候它们之间没有直接的通讯或共享内存。   
我们要在本地模式运行本章的所有例子。

**（2）远程模式**   
在远程模式下，我们向Storm集群提交拓扑，它通常由许多运行在不同机器上的流程组成。远程模式不会出现调试信息， 因此它也称作生产模式。不过在单一开发机上建立一个Storm集群是一个好主意，可以在部署到生产环境之前，用来确认拓扑在集群环境下没有任何问题。

## 三、复杂实例

**1 新建类SentenceSpout.java（数据流生成者）**

import java.util.Map;

import org.apache.storm.spout.SpoutOutputCollector;

import org.apache.storm.task.TopologyContext;

import org.apache.storm.topology.OutputFieldsDeclarer;

import org.apache.storm.topology.base.BaseRichSpout;

import org.apache.storm.tuple.Fields;

import org.apache.storm.tuple.Values;

import org.apache.storm.utils.Utils;

/\*\*

\* 向后端发射tuple数据流

\* @author soul

\*

\*/

public class SentenceSpout extends BaseRichSpout {

//BaseRichSpout是ISpout接口和IComponent接口的简单实现，接口对用不到的方法提供了默认的实现

private SpoutOutputCollector collector;

private String[] sentences = {

"my name is soul",

"im a boy",

"i have a dog",

"my dog has fleas",

"my girl friend is beautiful"

};

private int index=0;

/\*\*

\* open()方法中是ISpout接口中定义，在Spout组件初始化时被调用。

\* open()接受三个参数:一个包含Storm配置的Map,一个TopologyContext对象，提供了topology中组件的信息,SpoutOutputCollector对象提供发射tuple的方法。

\* 在这个例子中,我们不需要执行初始化,只是简单的存储在一个SpoutOutputCollector实例变量。

\*/

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

// TODO Auto-generated method stub

this.collector = collector;

}

/\*\*

\* nextTuple()方法是任何Spout实现的核心。

\* Storm调用这个方法，向输出的collector发出tuple。

\* 在这里,我们只是发出当前索引的句子，并增加该索引准备发射下一个句子。

\*/

public void nextTuple() {

//collector.emit(new Values("hello world this is a test"));

// TODO Auto-generated method stub

this.collector.emit(new Values(sentences[index]));

index++;

if (index>=sentences.length) {

index=0;

}

Utils.sleep(1);

}

/\*\*

\* declareOutputFields是在IComponent接口中定义的，所有Storm的组件（spout和bolt）都必须实现这个接口

\* 用于告诉Storm流组件将会发出那些数据流，每个流的tuple将包含的字段

\*/

public void declareOutputFields(OutputFieldsDeclarer declarer) {

// TODO Auto-generated method stub

declarer.declare(new Fields("sentence"));//告诉组件发出数据流包含sentence字段

}

}

**2 新建类SplitSentenceBolt.java（单词分割器）**

import java.util.Map;

import org.apache.storm.task.OutputCollector;

import org.apache.storm.task.TopologyContext;

import org.apache.storm.topology.OutputFieldsDeclarer;

import org.apache.storm.topology.base.BaseRichBolt;

import org.apache.storm.tuple.Fields;

import org.apache.storm.tuple.Tuple;

import org.apache.storm.tuple.Values;

/\*\*

\* 订阅sentence spout发射的tuple流，实现分割单词

\* @author soul

\*

\*/

public class SplitSentenceBolt extends BaseRichBolt {

//BaseRichBolt是IComponent和IBolt接口的实现

//继承这个类，就不用去实现本例不关心的方法

private OutputCollector collector;

/\*\*

\* prepare()方法类似于ISpout 的open()方法。

\* 这个方法在blot初始化时调用，可以用来准备bolt用到的资源,比如数据库连接。

\* 本例子和SentenceSpout类一样,SplitSentenceBolt类不需要太多额外的初始化,

\* 所以prepare()方法只保存OutputCollector对象的引用。

\*/

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

// TODO Auto-generated method stub

this.collector=collector;

}

/\*\*

\* SplitSentenceBolt核心功能是在类IBolt定义execute()方法，这个方法是IBolt接口中定义。

\* 每次Bolt从流接收一个订阅的tuple，都会调用这个方法。

\* 本例中,收到的元组中查找“sentence”的值,

\* 并将该值拆分成单个的词,然后按单词发出新的tuple。

\*/

public void execute(Tuple input) {

// TODO Auto-generated method stub

String sentence = input.getStringByField("sentence");

String[] words = sentence.split(" ");

for (String word : words) {

this.collector.emit(new Values(word));//向下一个bolt发射数据

}

}

/\*\*

\* plitSentenceBolt类定义一个元组流,每个包含一个字段(“word”)。

\*/

public void declareOutputFields(OutputFieldsDeclarer declarer) {

// TODO Auto-generated method stub

declarer.declare(new Fields("word"));

}

}

**3 新建类WordCountBolt.java（单词计数器）**

import java.util.HashMap;

import java.util.Map;

import org.apache.storm.task.OutputCollector;

import org.apache.storm.task.TopologyContext;

import org.apache.storm.topology.OutputFieldsDeclarer;

import org.apache.storm.topology.base.BaseRichBolt;

import org.apache.storm.tuple.Fields;

import org.apache.storm.tuple.Tuple;

import org.apache.storm.tuple.Values;

/\*\*

\* 订阅 split sentence bolt的输出流，实现单词计数，并发送当前计数给下一个bolt

\* @author soul

\*

\*/

public class WordCountBolt extends BaseRichBolt {

private OutputCollector collector;

//存储单词和对应的计数

private HashMap<String, Long> counts = null;//注：不可序列化对象需在prepare中实例化

/\*\*

\* 大部分实例变量通常是在prepare()中进行实例化，这个设计模式是由topology的部署方式决定的

\* 因为在部署拓扑时,组件spout和bolt是在网络上发送的序列化的实例变量。

\* 如果spout或bolt有任何non-serializable实例变量在序列化之前被实例化(例如,在构造函数中创建)

\* 会抛出NotSerializableException并且拓扑将无法发布。

\* 本例中因为HashMap 是可序列化的,所以可以安全地在构造函数中实例化。

\* 但是，通常情况下最好是在构造函数中对基本数据类型和可序列化的对象进行复制和实例化

\* 而在prepare()方法中对不可序列化的对象进行实例化。

\*/

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

// TODO Auto-generated method stub

this.collector = collector;

this.counts = new HashMap<String, Long>();

}

/\*\*

\* 在execute()方法中,我们查找的收到的单词的计数(如果不存在，初始化为0)

\* 然后增加计数并存储,发出一个新的词和当前计数组成的二元组。

\* 发射计数作为流允许拓扑的其他bolt订阅和执行额外的处理。

\*/

public void execute(Tuple input) {

// TODO Auto-generated method stub

String word = input.getStringByField("word");

Long count = this.counts.get(word);

if (count == null) {

count = 0L;//如果不存在，初始化为0

}

count++;//增加计数

this.counts.put(word, count);//存储计数

this.collector.emit(new Values(word,count));

}

/\*\*

\*

\*/

public void declareOutputFields(OutputFieldsDeclarer declarer) {

// TODO Auto-generated method stub

//声明一个输出流，其中tuple包括了单词和对应的计数，向后发射

//其他bolt可以订阅这个数据流进一步处理

declarer.declare(new Fields("word","count"));

}

}

**4 新建类ReportBolt.java（报告生成器）**

import java.util.ArrayList;

import java.util.Collections;

import java.util.HashMap;

import java.util.Map;

import org.apache.storm.task.OutputCollector;

import org.apache.storm.task.TopologyContext;

import org.apache.storm.topology.OutputFieldsDeclarer;

import org.apache.storm.topology.base.BaseRichBolt;

import org.apache.storm.tuple.Tuple;

/\*\*

\* 生成一份报告

\* @author soul

\*

\*/

public class ReportBolt extends BaseRichBolt {

private HashMap<String, Long> counts = null;//保存单词和对应的计数

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

// TODO Auto-generated method stub

this.counts = new HashMap<String, Long>();

}

public void execute(Tuple input) {

// TODO Auto-generated method stub

String word = input.getStringByField("word");

Long count = input.getLongByField("count");

this.counts.put(word, count);

//实时输出

System.out.println("结果:"+this.counts);

}

public void declareOutputFields(OutputFieldsDeclarer declarer) {

// TODO Auto-generated method stub

//这里是末端bolt，不需要发射数据流，这里无需定义

}

/\*\*

\* cleanup是IBolt接口中定义

\* Storm在终止一个bolt之前会调用这个方法

\* 本例我们利用cleanup()方法在topology关闭时输出最终的计数结果

\* 通常情况下，cleanup()方法用来释放bolt占用的资源，如打开的文件句柄或数据库连接

\* 但是当Storm拓扑在一个集群上运行，IBolt.cleanup()方法不能保证执行（这里是开发模式，生产环境不要这样做）。

\*/

public void cleanup(){

System.out.println("---------- FINAL COUNTS -----------");

ArrayList<String> keys = new ArrayList<String>();

keys.addAll(this.counts.keySet());

Collections.sort(keys);

for(String key : keys){

System.out.println(key + " : " + this.counts.get(key));

}

System.out.println("----------------------------");

}

}

**5 修改程序主入口App.java**

import org.apache.storm.Config;

import org.apache.storm.LocalCluster;

import org.apache.storm.topology.TopologyBuilder;

import org.apache.storm.tuple.Fields;

import org.apache.storm.utils.Utils;

/\*\*

\* 实现单词计数topology

\*

\*/

public class App

{

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 Exception

{

//System.out.println( "Hello World!" );

//实例化spout和bolt

SentenceSpout spout = new SentenceSpout();

SplitSentenceBolt splitBolt = new SplitSentenceBolt();

WordCountBolt countBolt = new WordCountBolt();

ReportBolt reportBolt = new ReportBolt();

TopologyBuilder builder = new TopologyBuilder();//创建了一个TopologyBuilder实例

//TopologyBuilder提供流式风格的API来定义topology组件之间的数据流

//builder.setSpout(SENTENCE\_SPOUT\_ID, spout);//注册一个sentence spout

//设置两个Executeor(线程)，默认一个

builder.setSpout(SENTENCE\_SPOUT\_ID, spout,2);

// SentenceSpout --> SplitSentenceBolt

//注册一个bolt并订阅sentence发射出的数据流，shuffleGrouping方法告诉Storm要将SentenceSpout发射的tuple随机均匀的分发给SplitSentenceBolt的实例

//builder.setBolt(SPLIT\_BOLT\_ID, splitBolt).shuffleGrouping(SENTENCE\_SPOUT\_ID);

//SplitSentenceBolt单词分割器设置4个Task，2个Executeor(线程)

builder.setBolt(SPLIT\_BOLT\_ID, splitBolt,2).setNumTasks(4).shuffleGrouping(SENTENCE\_SPOUT\_ID);

// SplitSentenceBolt --> WordCountBolt

//fieldsGrouping将含有特定数据的tuple路由到特殊的bolt实例中

//这里fieldsGrouping()方法保证所有“word”字段相同的tuuple会被路由到同一个WordCountBolt实例中

//builder.setBolt(COUNT\_BOLT\_ID, countBolt).fieldsGrouping( SPLIT\_BOLT\_ID, new Fields("word"));

//WordCountBolt单词计数器设置4个Executeor(线程)

builder.setBolt(COUNT\_BOLT\_ID, countBolt,4).fieldsGrouping( SPLIT\_BOLT\_ID, new Fields("word"));

// WordCountBolt --> ReportBolt

//globalGrouping是把WordCountBolt发射的所有tuple路由到唯一的ReportBolt

builder.setBolt(REPORT\_BOLT\_ID, reportBolt).globalGrouping(COUNT\_BOLT\_ID);

Config config = new Config();//Config类是一个HashMap<String,Object>的子类，用来配置topology运行时的行为

//设置worker数量

//config.setNumWorkers(2);

LocalCluster cluster = new LocalCluster();

//本地提交

cluster.submitTopology(TOPOLOGY\_NAME, config, builder.createTopology());

Utils.sleep(10000);

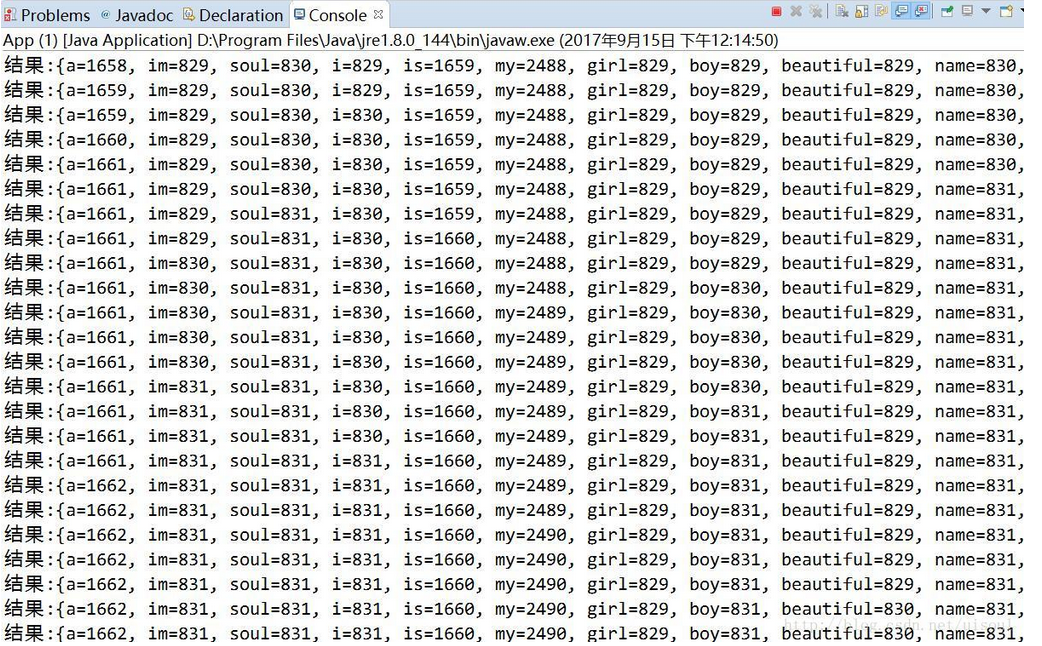
cluster.killTopology(TOPOLOGY\_NAME);

cluster.shutdown();

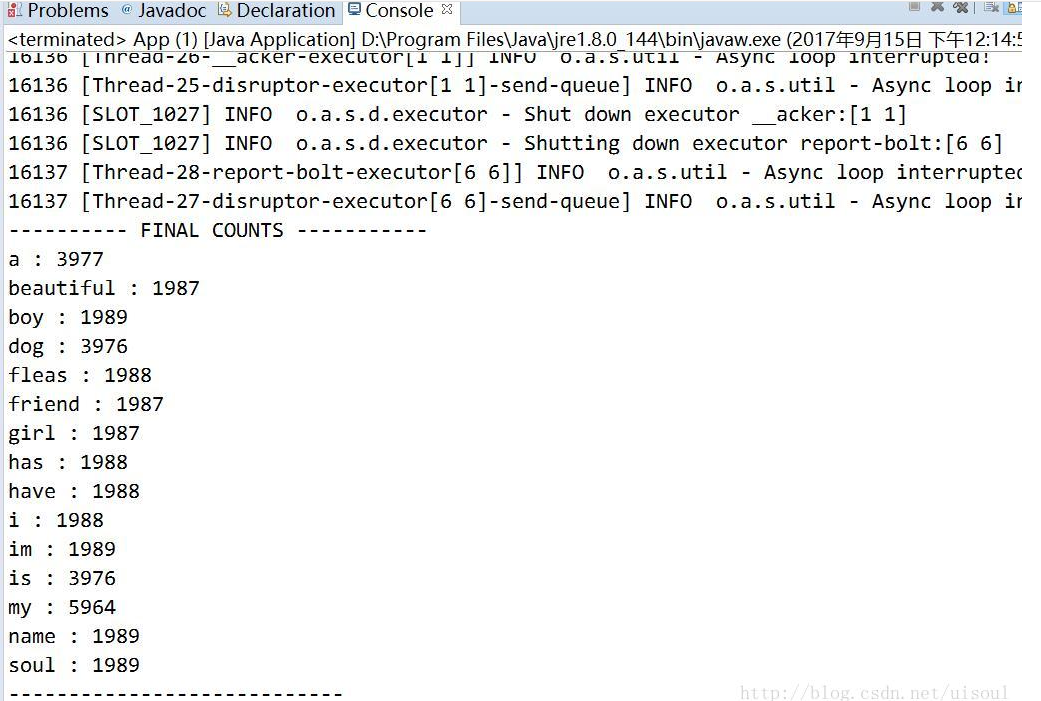
}

}

**6 运行程序，可看到单词计数实时输出效果**



运行10秒后生成报告



## 四、storm作业提交到集群上运行

**1 maven的pom.xml**

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.sid.bigdata</groupId>

<artifactId>storm</artifactId>

<version>0.0.1</version>

<packaging>jar</packaging>

<name>storm</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<storm.version>1.1.1</storm.version>

</properties>

<dependencies>

<dependency>

<groupId>org.apache.storm</groupId>

<artifactId>storm-core</artifactId>

<version>${storm.version}</version>

</dependency>

</dependencies>

</project>

**2 代码**

package com.sid.bigdata.storm.sum;

import java.util.Map;

import org.apache.storm.Config;

import org.apache.storm.StormSubmitter;

import org.apache.storm.generated.AlreadyAliveException;

import org.apache.storm.generated.AuthorizationException;

import org.apache.storm.generated.InvalidTopologyException;

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 org.apache.storm.utils.Utils;

public class ClusterSumStormTopology {

public static class DataSourceSpout extends BaseRichSpout{

private SpoutOutputCollector collector;

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

this.collector = collector;

}

int number = 0;

public void nextTuple() {

this.collector.emit(new Values(++number));

System.out.println("spout发出："+number);

Utils.sleep(1000);

}

public void declareOutputFields(OutputFieldsDeclarer declarer) {

declarer.declare(new Fields("num"));

}

}

public static class SumBolt extends BaseRichBolt{

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

int sum=0;

public void execute(Tuple input) {

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Topology

//storm中任何一个作业都是通过Topology方式进行提交的

//Topology中需要指定spout和bolt的执行顺序

TopologyBuilder tb = new TopologyBuilder();

tb.setSpout("DataSourceSpout", new DataSourceSpout());

//SumBolt以随机分组的方式从DataSourceSpout中接收数据

tb.setBolt("SumBolt", new SumBolt()).shuffleGrouping("DataSourceSpout");

//代码提交到storm集群上运行

try {

StormSubmitter.submitTopology("ClusterSumStormTopology", new Config(), tb.createTopology());

} catch (AlreadyAliveException e) {

e.printStackTrace();

} catch (InvalidTopologyException e) {

e.printStackTrace();

} catch (AuthorizationException e) {

e.printStackTrace();

}

}

}

**3 打包**

对项目名单机右键选择run as --> maven install

完成后Jar包在项目中target下

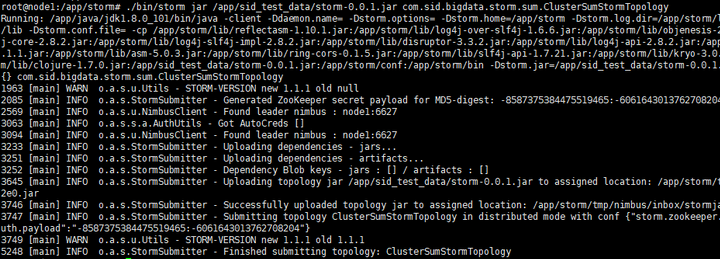
将这个jar包上传到nimbus所在的节点上

IMG_256

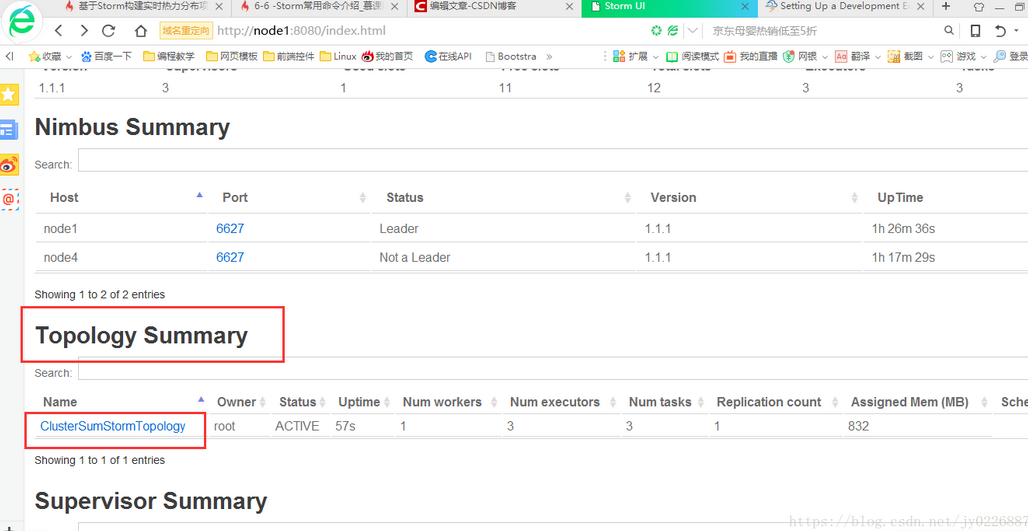
**4 启动storm**

**5 运行jar包**

./bin/storm jar /app/sid\_test\_data/storm-0.0.1.jar com.sid.bigdata.storm.sum.ClusterSumStormTopology



**6 在storm的UI中查看日志可以看到代码中system.out.print的输出**



**7 storm常用命令介绍**

1. **storm list 显示正在运行的作业的状态。**

IMG_256

**（2）storm kill ClusterSumStormTopology名字停掉作业。**

