big\_day19

Storm01

**一.Strom概述**

**1.实时分析概念**

1)离线分析 通常是 需要一段时间的**数据积累** 积累到一定数量数据后 开始离线分析 无论数据量多大 离线分析 有开始 也有结束 最终得到一个处理的结果 这样的分析过程 得到的结果是有较大的**延迟**的

2)实时分析 通常 数据不停的到来 随着数据的到来 来进行增量的运算 立即得到新数据的处理结果 并没有一个数据积累的过程 有开始 但是没有明确的结束的时刻 **数据实时的进行运算 基本没有延迟**

**2.Strom概述**

Storm是一个**开源的分布式实时计算系统**，可以简单、可靠的处理大量的数据流。

Storm有很多**使用场景**：如实时分析，在线机器学习，持续计算，分布式RPC，ETL等等。

Storm**支持水平扩展**，具有高容错性，保证每个消息都会得到处理。

Storm性能优良，处理速度很快(在一个小集群中，每个结点每秒可以处理数以百万计的消息)。

Storm的部署和运维都很便捷，而且更为重要的是可以使用任意编程语言来开发应用。--最主要还是java语言.

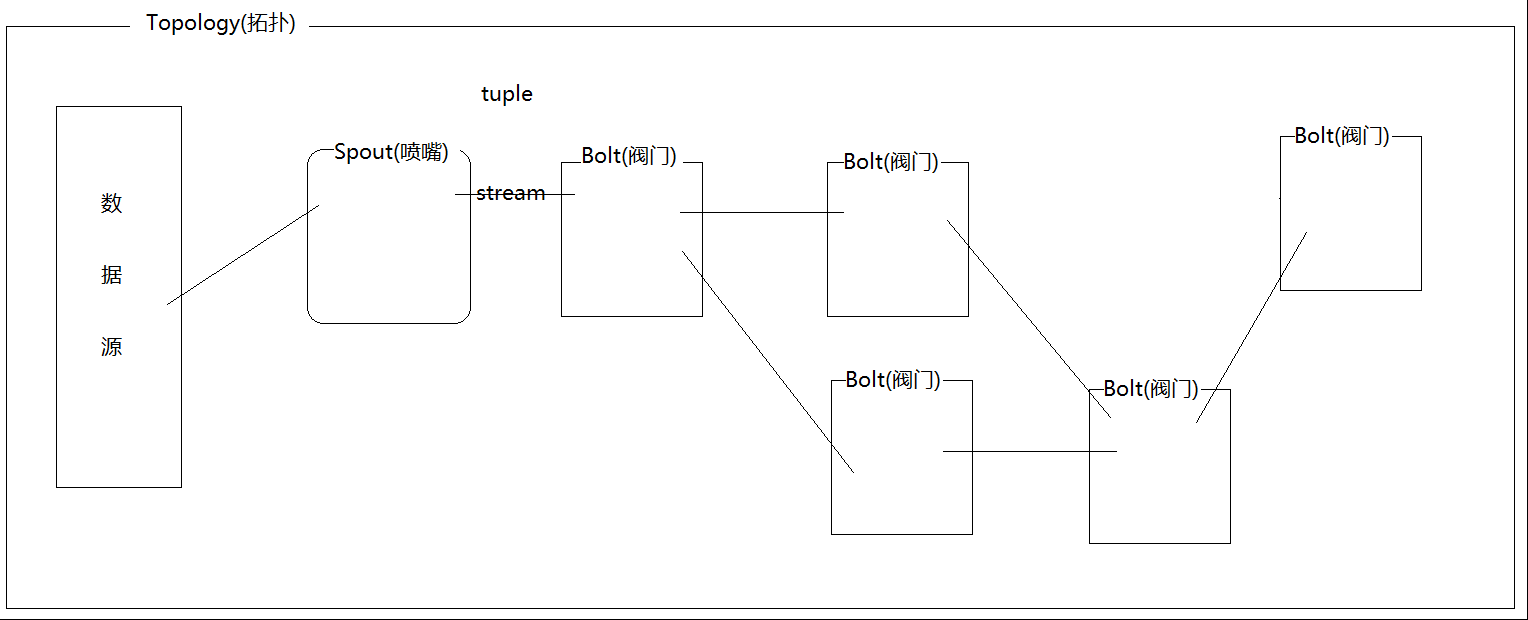
**3.Strom组件**

Storm将实时运算的过程 拆分为若干简单的步骤 再组装在一起完成复杂计算任务

由这些简单步骤组装起来的运算过程 称之为一个**Topology(拓扑)**,Topology由**Spout(喷嘴)** 和 **Bolt(阀门)**组成

**Spout**负责连接外部数据源 接收数据 并将数据 转换为**Tuple(元组)格式** 向后发送 可以发送给 一个或多个**Bolt**,Spout通常**只负责接收 转换 发送数据,不会进行任何的业务处理**

**Bolt**负责接收上游传入的Tuple数据,进行相应的**运算**,再将结果发送给后续的**零个或多个Bolt,**从而利用Spuot和Bolt组建起复杂的数据流处理网络 实现实时处理,在整个Topology中传递的Tuple就组成了Stream(数据流)

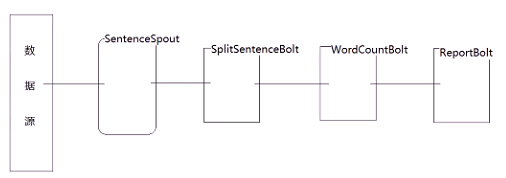


**二.Strom 入门案例 -wordcount**

**1)单机程序处理流程:**



**Topology设计**



**2)SentenceSpout开发**

**(1)SentenceSpout extends BaseRichSpout{**

/\*

\* 初始化的方法,在当前组件被初始化时调用,执行初始化相关信息

\* conf: 当前Topology的配置信息

\* context: 当前组件运行的上下文环境信息

\* collector: 可以用来发送tuple,可以在任意位置发送tuple,此对象线程安全,

\* 通常保存到类内部作为成员变量,方便在其他方法中访问

\*/

@Override

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

this.collector=collector;

}

/\*

\* strom会不停的调用此方法,要求发送tuple

\* 此方法不应该被阻塞 所以如果没有任何tuple要发送 直接返回即可

\* 在底层 其实就是一个单一的线程内循环不停的再调用此方法 所以如果真的没有任何tuple要

\* 发送 最好再这个方法中 睡眠一小段时间 以便于不至于浪费过多的cpu

\*/

@Override

public void nextTuple() {

if(index<sentences.length){

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

index++;

}else{

try {

Thread.sleep(1);

} catch (Exception e) {

e.printStackTrace();

}

return;

}

}

/\*

\* 用来声明当前组件输出的tuple的基本信息

\* declare:所有的组件如果想要发送tuple,都必须先声明再发送,

\* 此对象就是用来声明tuple的基本信息

\*/

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

declarer.declare(new Fields("sentence")); } }

**(2)SplitSentenceBolt extends BaseRichBolt{**

/\*

\* 初始化的方法

\* 在当前组件被初始化时被调用 执行初始化相关操作

\* conf: 当前Topology的配置信息

\* context: 当前组件运行的上下文环境信息

\* collector: 可以用来发送tuple,可以在任意位置发送tuple,此对象线程安全,

\* 通常保存到类内部作为成员变量,方便在其他方法中访问

\*/

@Override

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

this.collector=collector;

}

/\*

\* 处理上游发送的tuple,每个tuple都会进入一次此方法,进行处理

\* 处理过后可以发送两个或多个tuple给后续的组件

\* 如果需要,也可以不立即处理tuple,而是先持有tuple再在后续需要时处理

\* input:当前传入的要处理的tuple

\*/

@Override

public void execute(Tuple input) {

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

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

for(String word:words){

collector.emit(new Values(word));

}

}

/\*

\* 用来声明当前组件输出的tuple的基本信息

\* declare:所有的组件如果想要发送tuple,都必须先声明再发送,此对象就是用来声明

\* tuple的基本信息

\*/

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

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

}

**}**

**(3)WordCountBolt**

**(4)ReportBolt**

**三.Strom的并发机制**

**1.strom的并发级别** Storm可以在如下的四个级别上存在并发：

**1)Node服务器**

配置在Storm集群中的一个服务器，会执行Topology的一部分运算,一个Storm集群中包含一个或者多个Node

**2)Worker进程**

JVM虚拟机、进程。指一个Node上相互独立运作的JVM进程，每个Node可以配置运行一个或多个worker。一个Topology会分配到一个或者多个worker上运行。

**3)Executor线程**

指一个worker的jvm中运行的java线程。多个task可以指派给同一个executer来执行。除非是明确指定，Storm默认会给每个executor分配一个task。

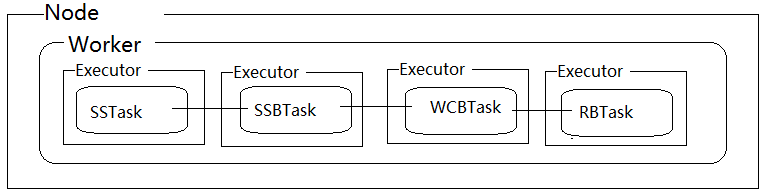
**4)Task任务**

bolt/spout实例。task是spout或bolt的实例，他们的nextTuple()和execute()方法会被executors线程调用执行。一个线程可以同时处理多个Task。

**2.storm的并发控制**

**1)默认并发控制**

如果不进行任何配置，默认并发度为**1**，也即，**默认情况下一个Node服务器上运行一个Worker进程，一个Worker进程里分配多个Executor线程，而每个Executor线程分别负责一个Task任务，此时唯一的并发处在线程级别。**



**2)人为指定并发度**

a.Node级别 Node级别的并发度无法通过代码控制，只能增加服务器硬件资源。

b.Worker级别 可以通过代码为指定的Topology配置多个Worker

Config conf = new Config();

conf.setNumWorkers(2);

\*\*单机测试模式下，设定Worker的数量没有意义，不会起效果。

c.Executor级别 可以通过代码指定Spout或Bolt需要几个线程

builder.setSpout(spout\_id,spout,2)

builder.setBolt(bolt\_id,bolt,2)

\*\* 如果只是指定线程数量，而未指定Task数量，则这个组件就会有指定数量的线程来处 理，而默认每个线程内都执行一个该组件的Task

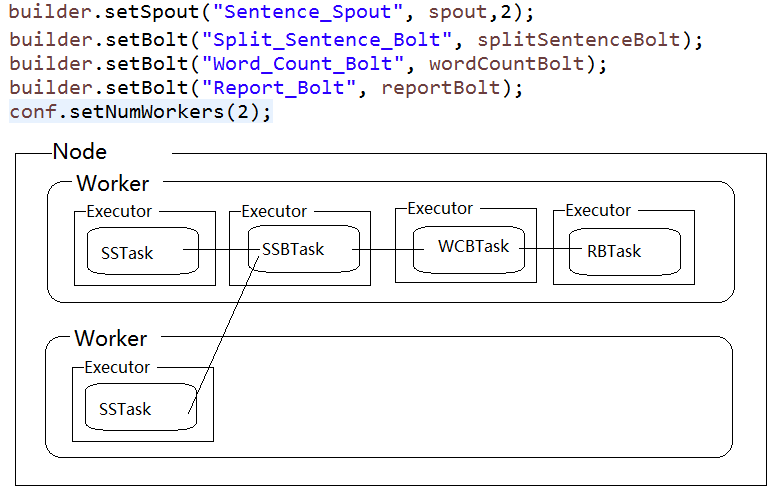
d.Task级别 可以通过代码指定Spout或Bolt需要并发几个Task

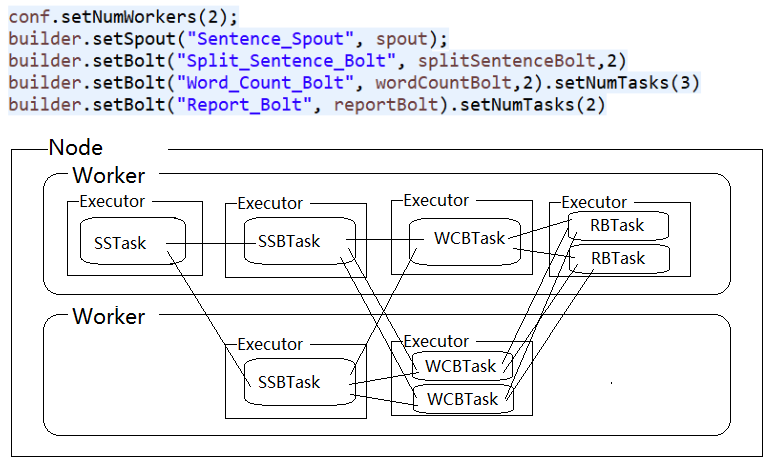
builder.setSpout(...).setNumTasks(2);

builder.setBolt(...).setNumTasks(2);

\*\*指定的Task会在线程内部执行，如果指定的Task的数量多于线程的数量，则这些Task自动分配到这些线程内执行。

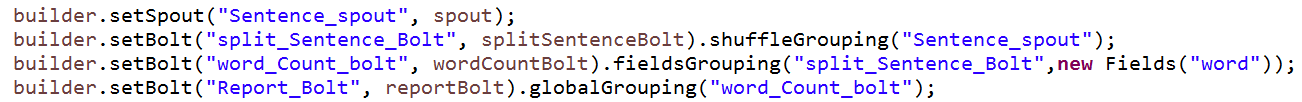
案例:单机,worker设置没有意义,spout会统计两次





**3)数据流分组方式**

在设定完并发度后，可以发现，组件向后发送tuple时，涉及到如何为后续组件的多个task分配这些tuple的问题，这是通过在连接组件时使用不同的方法，从而指定数据流分组方式来解决的。



storm**内置**了**七种数据流分组方式**：

(1)Shuffle Grouping(**随机分组**)

随机分发数据流中的tuple给bolt中的各个task，每个task接收到的tuple数量相同。

(2)Fields Grouping(按字段分组)

根据指定字段的值进行分组。指定字段具有相同值的tuple会路由到同一个bolt中的task中。

(3)All Grouping(全复制分组)

所有的tuple复制后分发给后续bolt的所有的task。

(4)Globle Grouping(全局分组)

这种分组方式将所有的tuple路由到唯一一个task上

Storm按照最小task id来选取接受数据的task

这种分组方式下配置bolt和task**的并发度没有意义。**

这种方式会导致所有tuple都发送到一个JVM实例上，可能会引起Strom集群中某个JVM或者服务器出现性能瓶颈或崩溃。

(5)None Grouping(不分组)

在功能上和随机分组相同，为将来预留。

(6)Direct Grouping(指向型分组)

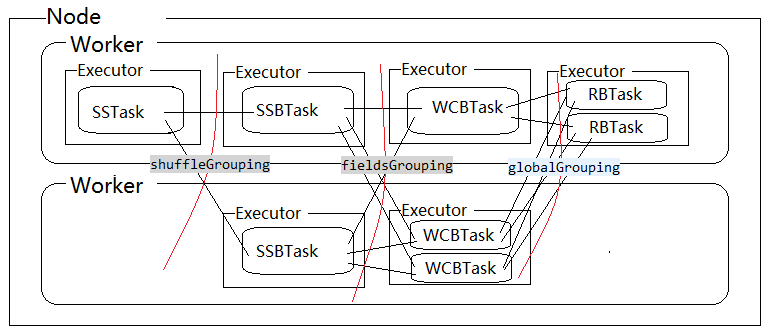
数据源会通过emitDirect()方法来判断一个tuple应该由哪个Strom组件来接受。

只能在声明为指向型的数据流上使用。

(7)Local or shuffle Grouping(本地或随机分组)

和随机分组类似，但是，会优先将tuple分发给同一个worker内的bolt的task，只有在同一个 Worker内没有目标bolt的task的情况下采用随机分组方式。

这种方式可以减少网络传输，从而提高topology的性能。



**四.Storm中的可靠性保证**

**1.storm可靠性概述**

Storm是分布式程序，进程之间是通过网络来通信的，此时可能因为网络的不稳定、节点宕机等原因造成数据可能存在丢失的情况，这就给Storm的实时处理带来了数据丢失的可能性，如何保障Storm处理数据时能够不丢数据 不多数据，使处理的结果就有可信度，这就是Storm可靠性相关的问题。

Storm中的可靠性分为三个级别：

至多一次 - 不多数据，但是可能会丢数据

至少一次 - 不丢数据，但是可能会多数据

恰好一次 - 不丢也不多，恰好完整的处理

这三种级别，越往下可靠性越高，但效率越低，越往上可靠性越低，但是效率越高，在真正的开发中应该根据业务需求，选择一个在满足的要求的可靠性的前提下，性能尽量好的可靠性级别。

**2.至多一次**

之前WC案例中，**没有任何并发控制**，此时就是 至多一次的并发状态，不会多数据，但是可能会因为网络延迟 程序出错 造成数据的丢失。

**3.至少一次**

在分布式的环境下，因为网络的不稳定性，数据丢失是无法彻底避免的，所以为了保证数据不丢失，应该检测数据丢失的情况，并重发丢失的数据。

这种方式下可以保证不丢数据，但是可能因为重发造成多数据。

1)对于Spout：

* 1. 持有已经发送的数据一段时间，以便于在数据丢失的情况下能够重发，直到该数据完成了整个Storm处理流程，才可以删除。
  2. 覆盖ack方法 和fail方法 ，storm会在某条数据完成整个处理环节过后，来调用ack方法，此时可以在ack方法中删除之前缓存的数据。而在某条数据处理失败时，storm会自动调用fail方法，此时可以在fail方法中实现重发的过程。

2)对于Bolt：

* 1. 需要在发送tuple的过程中实现锚定。

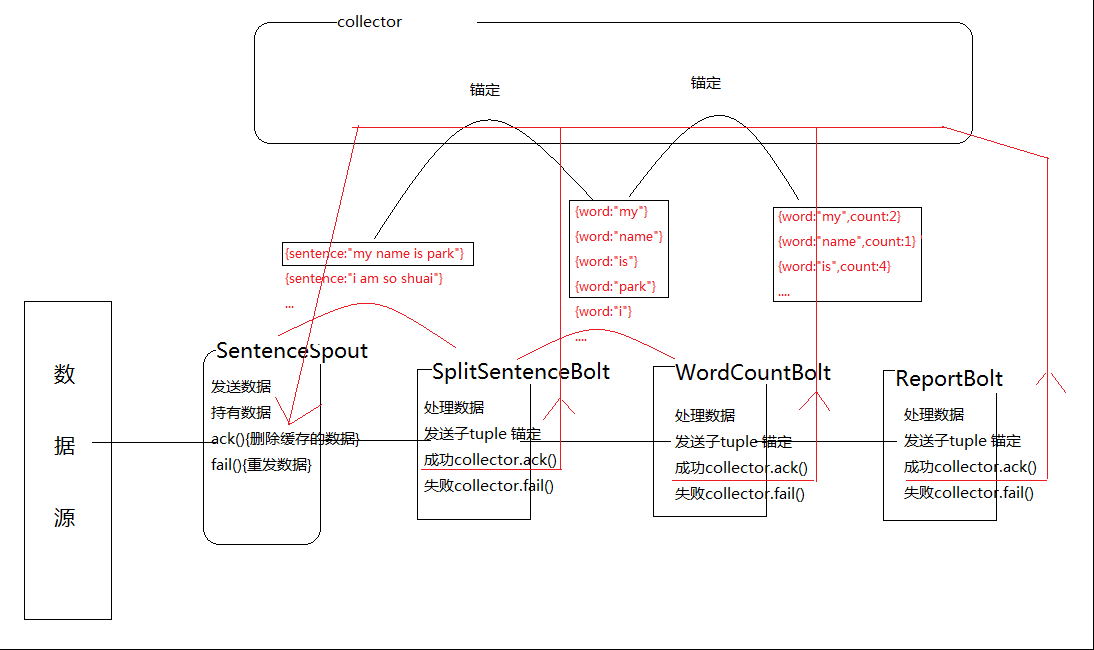
所谓的锚定就是将父tuple和子tuple的派生关系记录在collector中，以便于在后续tuple出错时，可以通过collector中维系的**锚定**关系，来**回溯**找到最初的tuple来实现重复

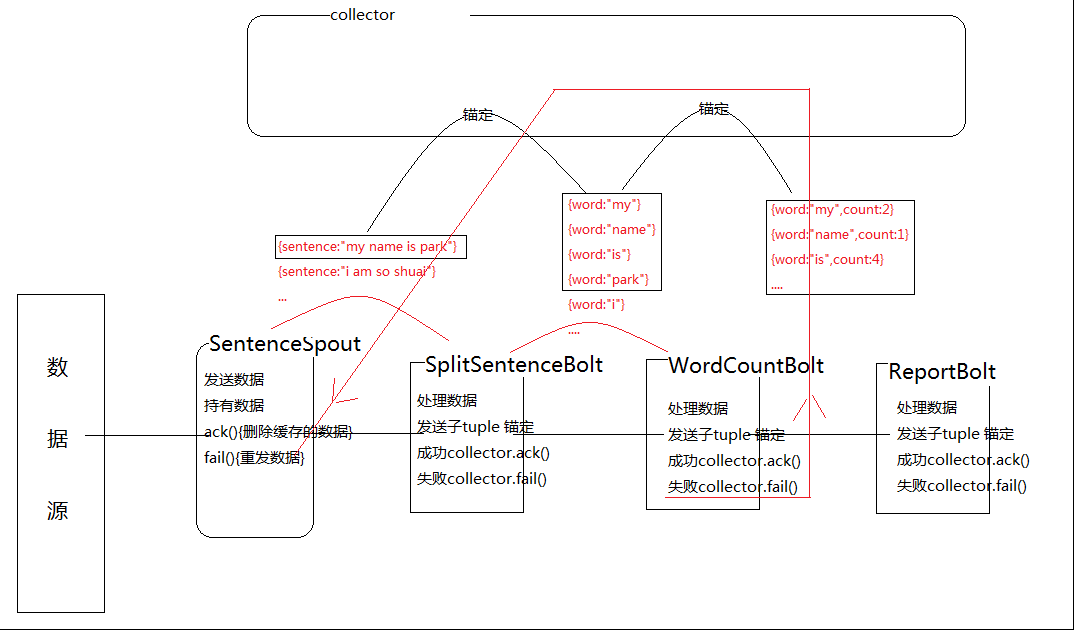
b.在整个bolt处理完一个tuple后调用collector提供的ack方法，或，在某个tuple处理过程中出错要调用collector的fail方法。

从而可以向collector告知，某个tuple在某个bolt中的处理状态，实现对tuple的处理的监控。

当对于某一个spout发送的tuple，整个拓扑中的所有bolt都处理成功，调用的是ack，则collector认为该tuple完成了整个处理流程会自动调用spout的ack方法

而任何一个bolt处理失败，调用的是fail方法，则collector认为该tuple处理失败，会自动调用spout的fail方法。





**4.恰好一次**

big\_day20

Storm02

**一.恰好一次**

至多一次 - 不多数据，但是可能会丢数据

至少一次 - 不丢数据，但是可能会多数据

恰好一次 - 不丢也不多，恰好完整的处理

4**.恰好一次**

在实现了至少一次语义之后，已经可以保证数据不丢，但是可能会多

想要实现恰好一次的语义，需要在以上的基础上，去除重复的数据

想要去除重复的数据，需要能够识别每条数据，并在处理过程中对已经处理过的数据的重发的数据进行抛弃

**方案1：**

想要唯一识别每条数据，可以为每条数据**指定唯一的编号**，这个编号是一个递增的数字，在重发时保持不变，通过这个编号唯一的识别tuple

后续处理过程中，记录所有已经处理过的tuple的编号，后续的tuple要处理之前，需要和这些已经处理过的tuple的编号做**比较**，**如果发现已经处理过，则抛弃，如果没有处理过就正常处理**

此方案，需要记录所有已经处理过的tuple的编号，且由于在任何时间点都无法确定之后会不会有指定tuple的重发，这些编号都不能删除，这样一来，随着storm的运行，**编号越记越多，浪费大量存储空间，且，比较的速度越来越慢**。在空间和时间上都无法接收。此方法不可行。

**方案2**：

想要解决方案1的问题，可以让所有的tuple的编号严格按照**顺序递增**，且要求整个topology严格**按照顺序处理这些tuple**，则只需要记录**最后一个处理过的tuple的编号**即可，后续再有tuple过来，如果编号小于等于记录中的tuple编号，则说明是之前已经处理过的tuple重发的数据，抛弃该数据。如果编号大于记录中的编号，则是正常数据，进行处理并更新记录中的最后一次处理的tuple编号即可。

此方案，需要严格保证tuple处理的顺序，可以要求所有的tuple**依次处理**，一个tuple处理时，其他tuple**排队等待**，可以保证顺序，但是**效率非常低下**。此方案不可行。

**方案3**：

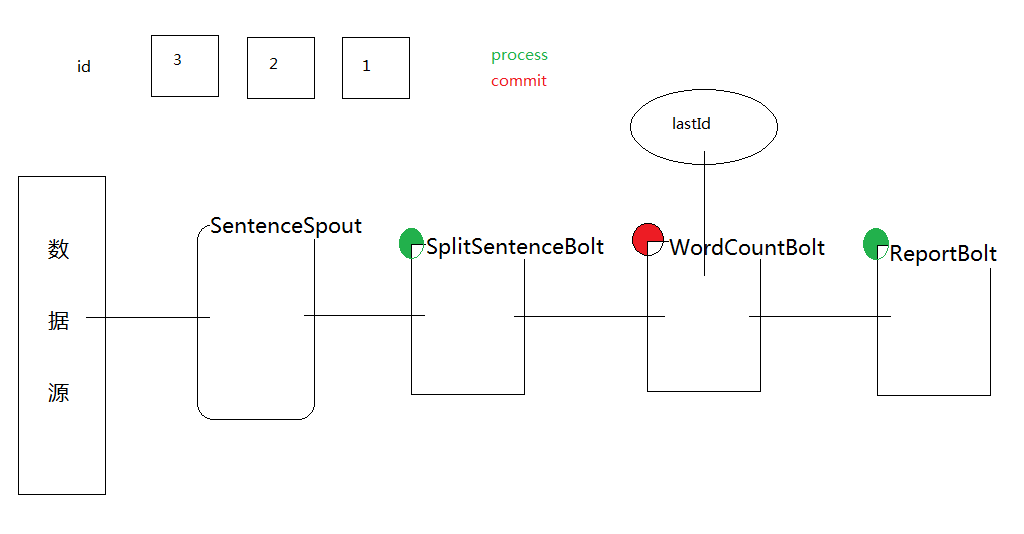
想要解决方案2的问题，则要求每次发一个批的数据，一个批中有若干tuple，每个批都被赋予一个递增的编号，批中任何一个tuple出错，整批重发，重发过程中编号不变。所有的批依次处理，一个批处理的过程中，其他批排队等待，保证顺序，批之间串行，但批内部有很多tuple，是可以并发的，这种方案，比起方案2，效率有了一定提升。

此方案，虽然比方案2效率提升了，但**是批和批之间仍然串行处理，性能依然受限**。

**方案4：**

通过分析发现，在整个Topology中，并不是所有的Bolt都需要严格保证一次且一次的语义，只要抓住最关键的一个或几个步骤，保证一次且一次的语义，就可以保障数据计算不出问题，基于这样的分析，可以将整个Topology中的Bolt分为**Process阶段 和 Commit阶段**两类。批在Porcess阶段的Bolt中随意并发保证性能，但一旦遇到Commit阶段的Bolt，则要严格按照顺序进入，保证顺序，再在Commit阶段的Bolt中记录最后一个处理过的批的编号，和后续批的编号做比较，进行去重操作，就可以保证一次且一次的语义。

此方案，即保证了一次且一次的语义，也在最大程度上，保证了程序的并发效率，Storm采用的正是这个方案。



**二.事务型拓扑-TransanctionTopology**

1.事务性拓扑

根据之前分析的恰好一次语义的实现方式，storm设计了事务型拓扑API来实现恰好一次的语义

即TransactionTopology API。

在TransactionTopology API中 存在TransactionSpout 和 TransactionBolt 来实现事务型拓扑。

2.TransactionSpout的开发

1)TransSentenceSpout extends BaseTransactionalSpout<SentenceMetaData>{}

2)SentenceMetaData implements Serializable {}

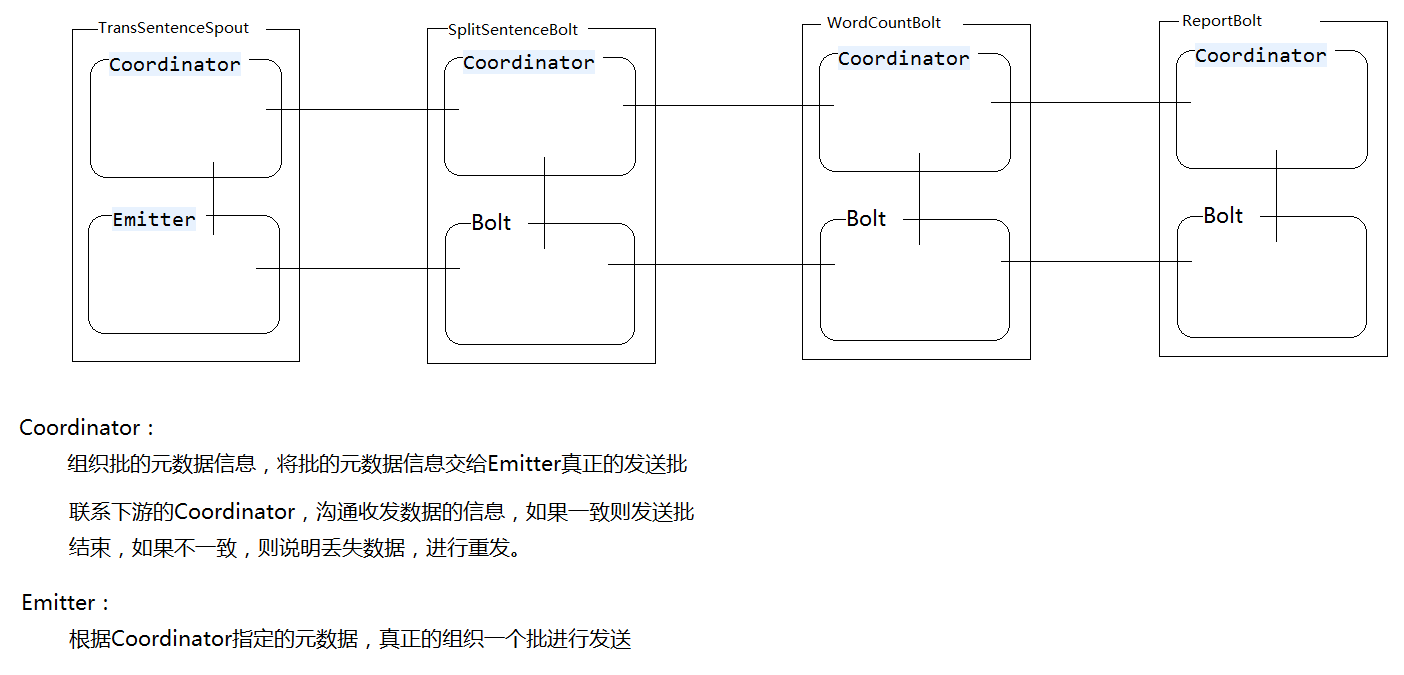
int begin

int end;

3)SentenceCoordinator implements Coordinator<SentenceMetaData>{}

4)SentenceDB{static String [] sentence}

5)SentenceEmitter implements Emitter<SentenceMetaData>{}



3.TransactionBolt的开发

1) TransSplitSentenceBolt extends BaseBatchBolt<TransactionAttempt>{

其中包含四个方法：

public void prepare(Map conf, TopologyContext context, BatchOutputCollector collector, TransactionAttempt id) {}

public void execute(Tuple tuple){}

public void finishBatch(){}

public void declareOutputFields(OutputFieldsDeclarer declarer) {} }

2)TransWordCountBolt

3)TransReportBolt

将Bolt设置为commit阶段Bolt 使Bolt实现Icommitter

在构建拓扑的过程中使用setCommitterBolt()方法提到setBolt()方法

 builder.setCommitterBolt("Trans\_Word\_Count\_Bolt",wordCountBolt).-TransWordCountTopology中

4.TransWordCountTopology开发

**三.Storm集群配置及任务提交**

1.Storm集群的基本概念

Storm集群遵**从主/从模式**

Storm的主节点称之为**nimbus** 从节点称之为**supervisor**

Storm通过**Zookeeper来实现集群协调**

Storm集群提供了**半容错机制**

2.Storm中的nimbus

nimbus守护进程主要的责任是**管理:协调和监控**在集群上运行的topology

包括topology的发布

任务的指派

事件处理失败时重新指派任务

将topology发布到Storm集群，将预先打包成jar文件的topology和配置信息提交到nimbus服务器上，一旦nimbus接收到topology的压缩包，会将jar包**分发到足够数量的supervisor节点上**，当supervisor节点接收到了topology压缩文件，nimbus就会指派task到每个supervisor并且发送信号指示supervisor生成足够的worker来执行指派的task

nimbus**记录所有的supervisor节点的状态和分配给他们的task**，如果nimbus发现某个supervisor没有上报心跳或者已经不可达了，他将会将故障supervisor分配的task重新分配到集群中的其他supervisor节点。

nimbus并不参与topology的数据处理过程，只是负责topology的初始化、任务分发和进程监控。因此，即使nimbus守护进程在topology运行时停止了，只要分配的supervisor和worker健康运行，topology会一直继续处理数据，所以称之为半容错机制。

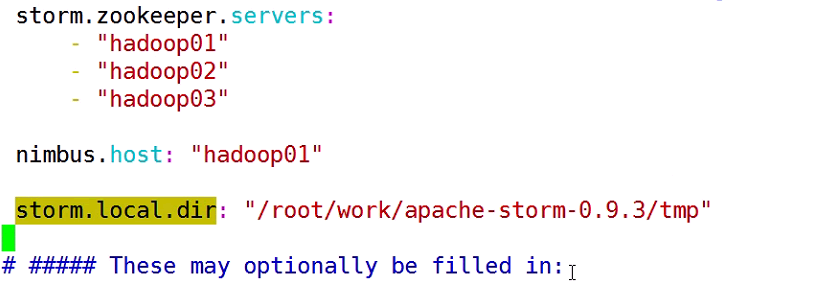
3.Storm中的supervisor节点

supervisor守护进程等待nimbus分配任务后生成并监控workers执行任务。

supervisor和worker都是运行在不同的JVM进程上，如果supervisor启动的worker进程因为错误异常退出，supervisor将会尝试重新生成新的worker进程。

4.Storm的安装配置 必须要有:JDK和Zookeeper

1)修改$STORM\_HOME/conf目录下的storm.yaml文件



必须修改的项：

(1)storm.zookeeper.services:配置zookeeper集群的主机名称。

(2)nimbus.host:指定了集群中nimbus的节点。

(3)supervisor.slots.ports:配置控制每个supervisor节点运行多少个worker进程。

这个配置定义为worker监听的端口的列表，监听端口的个数控制了supervisor节点上有多少个worker的插槽。默认的storm使用6700~6703端口，每个supervisor节点上有4个worker插槽。

(4)storm.local.dir:storm工作时产生的工作文件存放的位置，注意，要避免配置到/tmp下。

可选的常用修改项：

(1)nimbus.childopts(default: -Xms1024m):这项JVM配置会添加在启动nimbs守护进程的java命令行中

(2)ui.port(default:8080):这项配置指定了Storm UI的Web服务器监听的端口。

(3)ui.childopts(default:-Xms1024m):这项JVM配置会添加在StormUI服务启动的Java命令行中。

(4)supervisor.childopts(default:-Xms768m):这项JVM配置会添加Supervisor服务启动的Java命令行中。

(5)worker.childopts(default:-Xms768m):这项JVM配置会添加worker服务启动的Java命令行中。

(6)topology.message.timeout.secs(default:30):这个配置项定义了一个tuple树需要应答最大时间秒数限制，超过这个时间则认为超时失败。

(7)topology.max.spout.pending(default:null):在默认值null的情况下，spout每当产生新的tuple时会立即向后端发送，由于下游bolt执行可能具有延迟，可能导致topology过载，从而导致消息处理超时。如果手动将该值改为非null正整数时，会通过暂停spout发送数据来限制同时处理的tuple不能超过这个数，从而达到为Spout限速的作用。

(8)topology.enable.message.timeouts(default:true):这个选项用来锚定的tuple的超时时间。如果设置为false，则锚定的tuple不会超时。

将Storm复制到足够多的节点中scp -r xxxx xxx

5.管理Storm

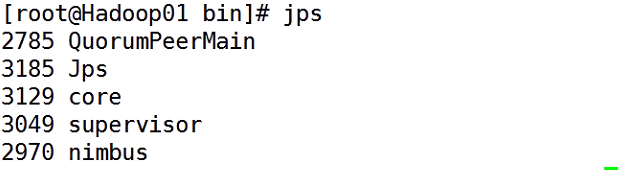
1).Storm的启动:

启动Zookeeper 三台

./Storm nimbus & 主机

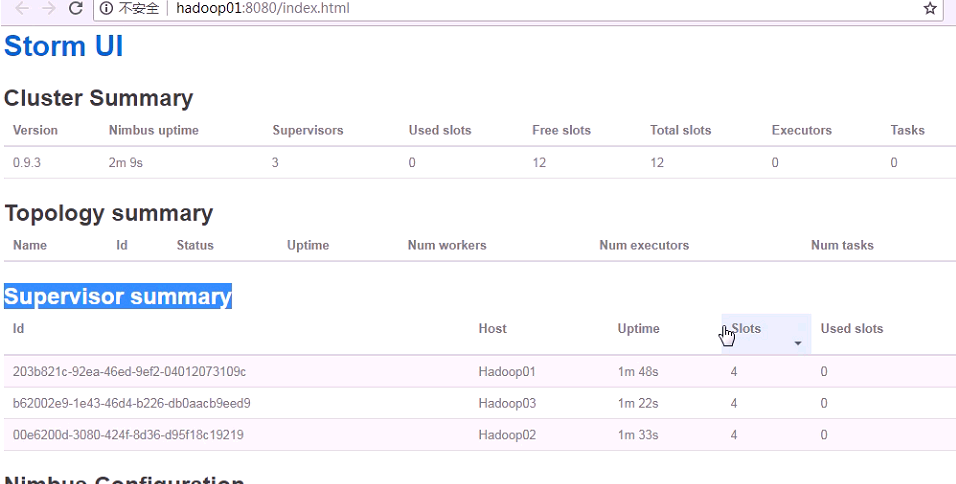
./Storm supervisor & 三台都要启动

./Storm ui & web的要启动



2).监控Storm:通过浏览器 访问ui进程所在的机器的ui监听端口(8080)

hadoop01:8080



3).管理命令:

a.storm jar topology\_jar topology\_class[arguments...]

向集群提交topology。它会使用指定的参数运行topology\_class中的main()方法，同时上传topology\_jar文件到nimbus以分发到整个集群。

提交后，Storm集群会激活并且开始运行topology。

topology中的main()方法需要调用StormSubmitter.submitTopology()方法，并且为topology提供集群内唯一的名称。

b.storm kill topology\_name[-w wait\_time] 用来关闭已经部署的topology。

c.storm deactivate topology\_name停止指定topology的spout发送tuple

d.storm activate topology\_name恢复指定topology的spout发送tuple。

e.storm rebalance topology\_name[-w wait\_time][-n worker\_count][-e component\_name=executor\_count]

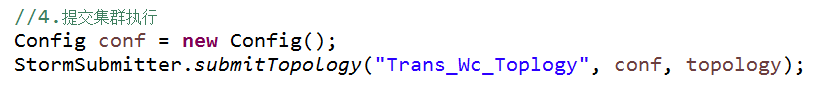
指定storm在集群的worker之间重新平均地分配任务，不需要关闭或者重新提交现有的topology。当执行rebalance命令时，Storm会先取消激活topology，等待配置的的时间使剩余的tuple处理完成，然后再supervisor节点中均匀的重新分配worker。重新分配后，Storm会将topology恢复到之前的激活状态。

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

6.提交Topology到集群中运行

修改提交topology的代码：

StormSubmitter.submitTopology("mywc", conf, topology);

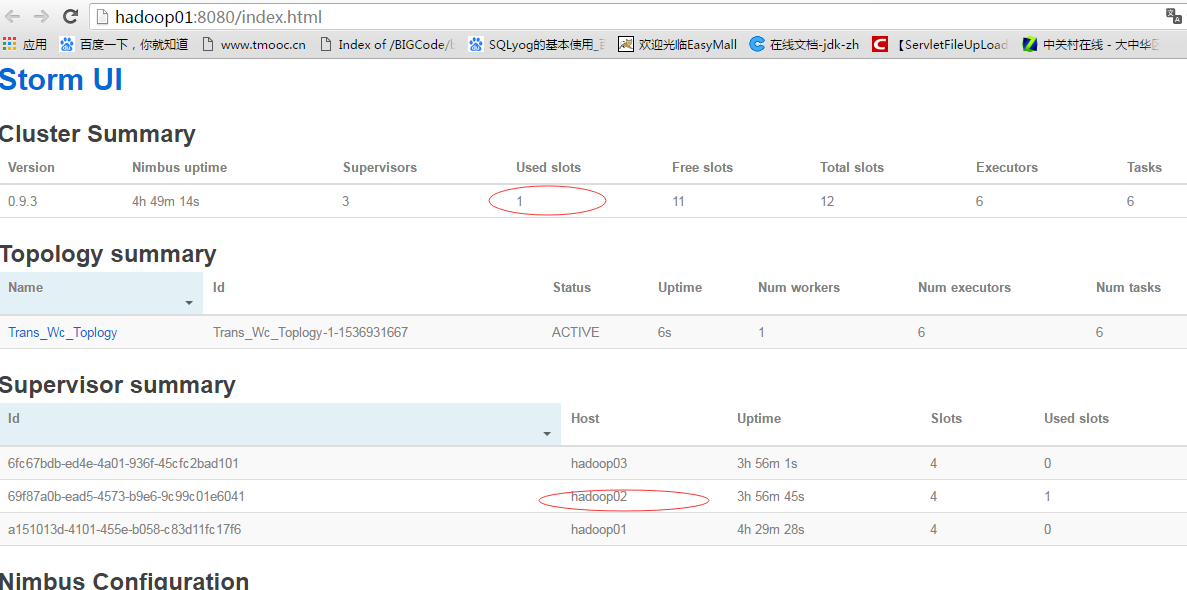


1)将程序打成jar包，同时设置jar包的主类

2)将jar包上传到集群中

3)通过命令执行jar包

bin# ./storm jar /home/software/transaction.jar cn.tedu.storm.wc3.transaction.TransWordCountTopology



system.out.printlin的打印信息在logs的worker...中

4)执行一段时间后，可以通过如果下命令关闭topology

bin]# ./storm kill Trans\_Wc\_Toplogy

