1、jstorm源码跟踪

* Nimbus的启动流程

1、通过bin/storm脚本启动nimbus，调用了backtype.storm.daemon.nimbus

2、从而调用了nimbus.clj这个clojure脚本

3、在脚本中读取了所有的配置文件

4、在storm中，通信中间件是thrift，是个RPC，所有定义的接口在storm.thrift中

5、nimbus的启动入口在NimbusServer.java类中

1)、根据配置文件初始化Context数据；

2)、与Zookeeper数据同步；

3)、初始化RPC服务处理类ServiceHandler；

4)、启动任务分配策略线程；

5)、启动Task的Heartbeat监控线程；

6)、启动thriftserver服务；

7)、其他初始化工作。

* wordcount的任务提交流程

1、在main方法中，我们创建TopologyBuilder，用来设置和提交任务

2、在设置spout时，将所有的spout的设置封装到了TopologyBuilder的一个map中

3、在设置bolt时，将所有的bolt设置到了TopologyBuilder的一个map中

4、同时，将spout、bolt的依赖关系grouping设置到了另一个map中，以供三个map

5、在创建topology的时候，将参数什么的都封装到了json中

6、在提交任务的时候，是通过thrift的client调用的RPC接口，将任务提交给了nimbus，存储在了本地目录中，通过RPC向Nimbus提交Topology DAG及配置信息

7、提交任务后，由Nimbus.java的实现类ServiceHandler.java进行设置任务

8、任务提交的相关细节在ServiceHandler.java

1）Nimbus端接收到Client提交上来的Topology计算逻辑代码jar包后如前面所述将jar包暂存在目录$jstorm\_local\_dir/nimbus/inbox/stormjar-{$randomid}.jar；

2）Nimbus端接收到Client提交上来的Topology DAG和配置信息后：

3)简单合法性检查；主要检查是否存在相同TopologyName的Topology，如果存在，拒绝Topology提交。

4）生成topologyid；生成规则：TopologyName-counter-currenttime；

5）序列化配置文件和Topology代码；

6）Nimbus本地准备运行时所需数据；

7）向Zookeeper注册Topology和Task；

8）将Tasks压入分配队列等待TopologyAssign分配；

* 任务提交之后，会启动调度线程

任务调度需要解决的问题是：如何将Topology DAG中各个计算节点和集群资源匹配，才能发挥高效的逻辑处理。策略是：1、将集群中的资源排序：按照空闲worker数从小到大的顺序重排节点，节点内部按照端口大小顺序排列；2、Topology中需要分配的任务（重新分配的Topology时大多任务不再需要分配）逐个映射到上述排好序的资源里。

1、任务调度的过程在TopologyAssign.java中

2、在init方法中启动默认的调度器DefaultTopologyScheduler，然后启动TopologyAssign线程

3、在TopologyAssign的run方法中执行主要的业务逻辑

a）从队列中取出已经提交的任务

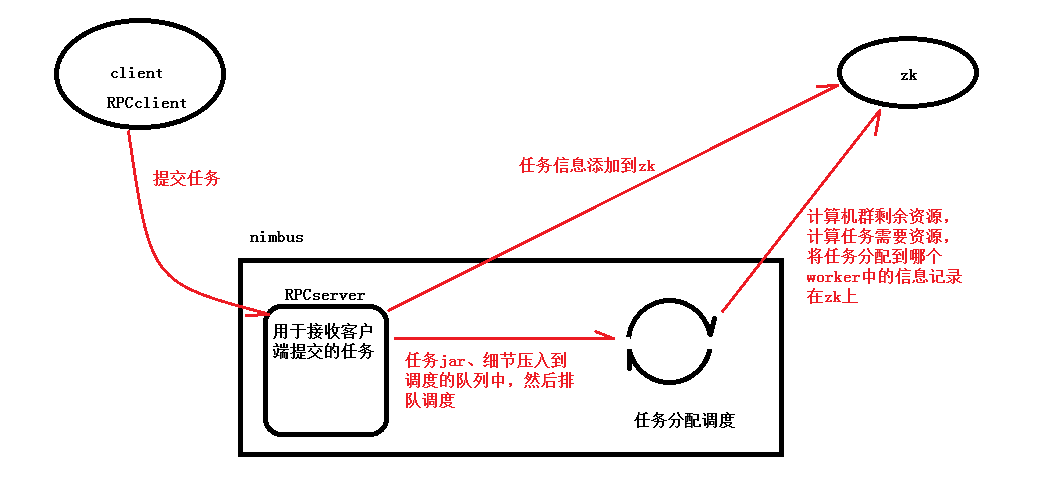
b）在doTopologyAssignment中进行任务的分配，就是计算集群中的资源总量，然后减去任务需要的资源总量，重新记录资源总量，然后将分配情况写到zookeeper

c）任务调度这里逻辑相对简单，但是代码相对复杂，想了解的请看我的源码注释

* 任务监控

初始化Nimbus时后台会随之启动一个称为MonitorRunnable的线程，该线程的作用是定期检查所有运行Topology的任务Tasks是否存在Dead的状态。一旦发现Topology中存在Dead的任务Task，MonitorRunnable将该Topology置为StatusType.monitor，等待任务分配线程对该Topology中的Dead任务进行重新分配。

MonitorRunnable线程默认10s执行一次检查



* supervisor的启动流程

1、在通过bin/storm脚本启动supervisor，调用了backtype.storm.daemon，最终调用启动类Supervisor.java的main方法

2、在main方法中调用了run方法

3、然后创建了supervisor进程的pid

4、然后调用mkSupervisor方法进程初始化，重要的逻辑就在这个方法中

a）清理本地tmp文件夹

b）创建zk实例

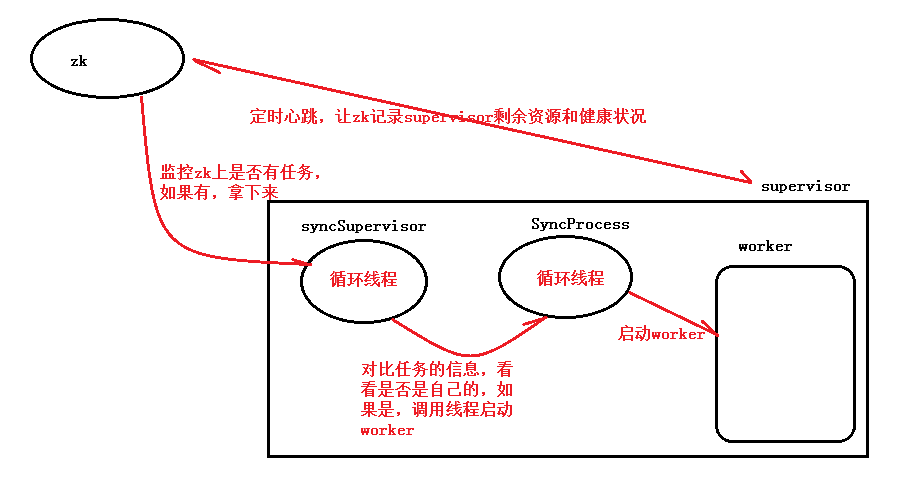
c）创建本地文件夹localstate记录本地状态

d）创建定时线程向zk发送心跳

e）启动syncSupervisor线程，定时的在zk上拿到最新的任务，对比任务是否是自己的

f）启动SyncProcess线程，如果上一步中任务是自己的，就在SyncProcess的run方法中启动一个worker进程

g）启动worker的方式是拼接java -server这样的命令，然后用win32的CreateProcess启动worker类



* worker启动流程

1、在main方法中，现将占用了需要分配的这个端口的worker杀死

2、然后封装启动参数，调用mk\_worker方法启动worker

3、在mk\_work方法中的执行逻辑如下：

a）封装传过来的id、端口、jar包

b）创建worker实例，将数据封装到workerdata中

c）初始化Tuple序列化功能和数据发送管道，此时初始化了nettyserver用来接收其他worker发送过来的数据，还创建了nettyclient。

d）在mk\_worker方法中还初始化了一大堆的调度线程，也就是每隔一段时间执行的线程。

e）最重要的是调用了createTasks方法创建分配到当前Worker的Tasks

4、在createTasks方法中的逻辑如下：

a）循环出所有的taskids

b）循环创建task的线程

c）然后运行task的run方法

5、在Task的run方法中的逻辑如下：

a）调用excute方法

b）先发送tuplt（“startup”）给系统的bolt

c）创建线程，从zeroMQ中获取获取tuple，然后发送消息到bolt/spout

d）在prepareExecutor方法中，创建发送线程、发送队列、序列化队列，并设置定时循环线程

1）通过mkExecutor方法创建boltexecutor和spoutexecutor，用于发送消息

2）通过mkTaskReceiver创建就收队列和线程

e）在mkTaskReceiver方法中，创建接收线程、接收队列、反序列化队列，并设置定时循环线程

d）创建守护线程，用于停止task

* executor执行流程

1、初始化SingleThreadSpoutExecutors线程，这个线程会被循环调用

2、在线程中调用我们实现的myspout的netTuple方法

3、在netTuple发放中emit数据，然后调用SpoutCollector的emit方法发送数据

4、计算哪些task使我们需要发送的目的地

5、将消息发送到worker的innermap<taskid,Q>中的对应Q中，所有内部组件都会订阅这个Q，然后把消息发送到序列化Q，所有外部消息会通过nettyclient发送出去。

6、然后计算生成ackSeq（异或的那些值），并计算异或结果

7、给acker发送out\_stream\_id, values, message\_id, root\_id, ackSeq, needAck

8、在sendMsgToAck方法中，将rootid、异或结果、taskid发送给acker

9、然后调用我们定义的myspout的ack方法

10、当acker接收到消息后，会初始化RotatingMap<Object, AckObject> pending这个变量，将我们的spoutId设置到当前的类中，以便于消息ack和fail时使用。

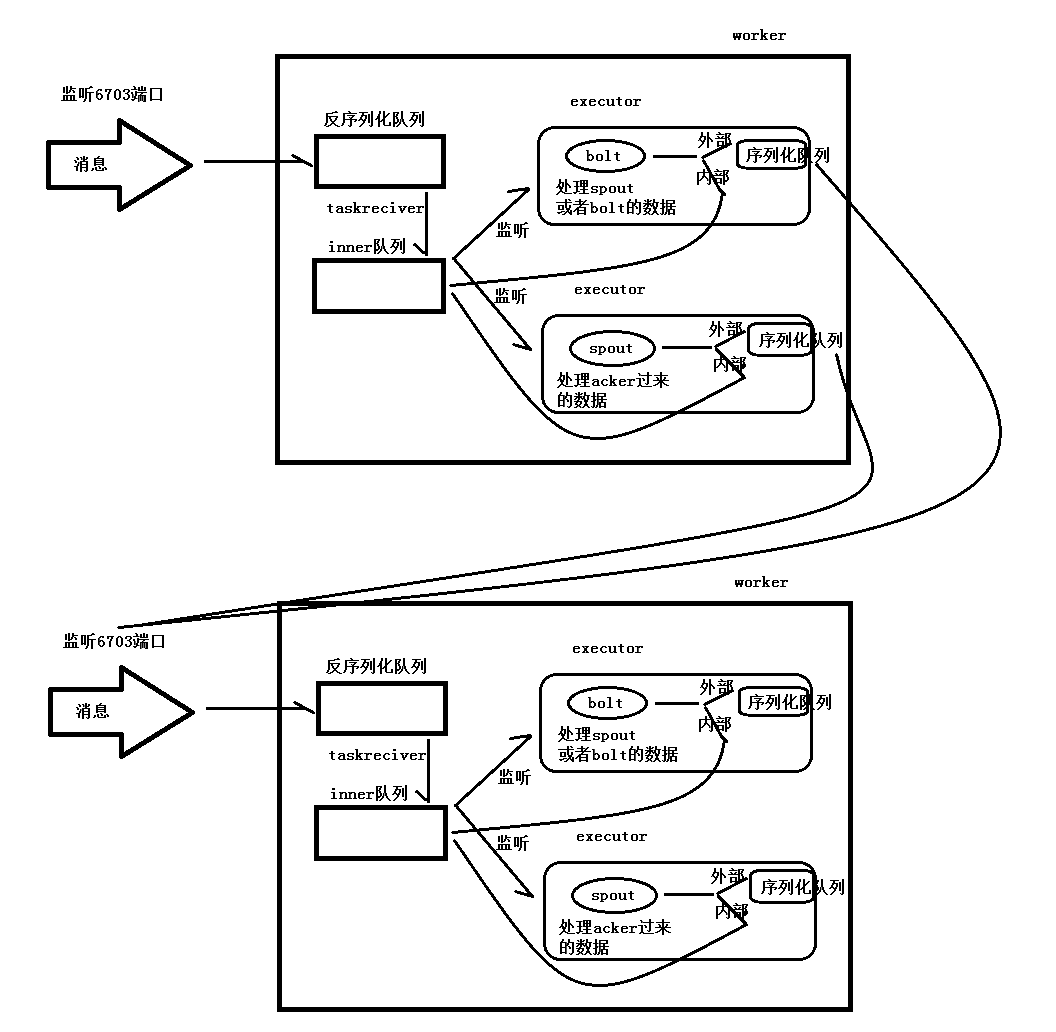
11、在Acker中执行execute方法，acker本身也是一个bolt。

12、在execute方法中，会根据不同类型（init、ack、fail）进行消息的不同处理，还会计算异或值

13、当异或值为0 的时候，会调用spout的ack方法。

14、当spout的消息发送成功后，BoltExecutors这个线程会启动，执行的逻辑和spoutexecutor的逻辑大致相同，不在敷衍。

消息的通信流程如下：

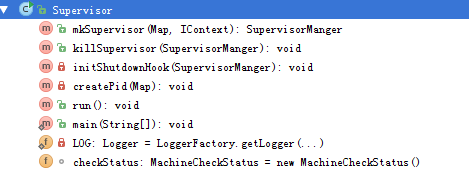


JStorm组件启动流程分析

、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);  
 }  
}