Spark Executor

 一、引子

  在Worker Actor中，每次LaunchExecutor会创建一个CoarseGrainedExecutorBackend进程，Executor和CoarseGrainedExecutorBackend是1对1的关系。也就是说集群里启动多少Executor实例就有多少CoarseGrainedExecutorBackend进程。

  那么到底是如何分配Executor的呢？怎么控制调节Executor的个数呢？

 二、Driver和Executor资源调度

   下面主要介绍一下[Spark](http://lib.csdn.net/base/spark" \o "Apache Spark知识库" \t "http://blog.csdn.net/oopsoom/article/details/_blank) Executor分配策略：

   我们仅看，当Application提交注册到Master后，Master会返回RegisteredApplication，之后便会调用schedule（）这个方法，来分配Driver的资源，和启动Executor的资源。

schedule()方法是来调度当前可用资源的调度方法，它管理还在排队等待的Apps资源的分配，这个方法是每次在集群资源发生变动的时候都会调用，根据当前集群最新的资源来进行Apps的资源分配。

Driver资源调度：

  随机的将Driver分配到空闲的Worker上去，详细流程请看我写的注释 ：）

1. // First schedule drivers, they take strict precedence over applications
2. val shuffledWorkers = Random.shuffle(workers) // 把当前workers这个HashSet的顺序随机打乱
3. **for** (worker <- shuffledWorkers **if** worker.state == WorkerState.ALIVE) { //遍历活着的workers
4. **for** (driver <- waitingDrivers) { //在等待队列中的Driver们会进行资源分配
5. **if** (worker.memoryFree >= driver.desc.mem && worker.coresFree >= driver.desc.cores) { //当前的worker内存和cpu均大于当前driver请求的mem和cpu，则启动
6. launchDriver(worker, driver) //启动Driver 内部实现是发送启动Driver命令给指定Worker，Worker来启动Driver。
7. waitingDrivers -= driver //把启动过的Driver从队列移除
8. }
9. }
10. }

Executor资源调度：

 Spark默认提供了一种在各个节点进行round-robin的调度，用户可以自己设置这个flag

1. val spreadOutApps = conf.getBoolean("spark.deploy.spreadOut", **true**)

在介绍之前我们先介绍一个概念，

可用的Worker：什么是可用，可用就是资源空闲足够且满足一定的规则来启动当前App的Executor。

Spark定义了一个canUse方法：这个方法接受一个ApplicationInfo的描述信息和当前Worker的描述信息。

1、当前worker的空闲内存 比 该app在每个slave要占用的内存 （executor.memory默认512M）大

2、当前app从未在此worker启动过App

总结： 从这点看出，要满足：该Worker的当前可用最小内存要比配置的executor内存大，并且对于同一个App只能在一个Worker里启动一个Exeutor，如果要启动第二个Executor，那么请到其它Worker里。这样的才算是对App可用的Worker。

1. /\*\*
2. \* Can an app use the given worker? True if the worker has enough memory and we haven't already
3. \* launched an executor for the app on it (right now the standalone backend doesn't like having
4. \* two executors on the same worker).
5. \*/
6. def canUse(app: ApplicationInfo, worker: WorkerInfo): Boolean = {
7. worker.memoryFree >= app.desc.memoryPerSlave && !worker.hasExecutor(app)
8. }

SpreadOut分配策略：

SpreadOut分配策略是一种以round-robin方式遍历集群所有可用Worker，分配Worker资源，来启动创建Executor的策略，好处是尽可能的将cores分配到各个节点，最大化负载均衡和高并行。

下面看看，默认的spreadOutApps模式启动App的过程：

 1、等待分配资源的apps队列默认是FIFO的。

 2、app.coresLeft表示的是该app还有cpu资源没申请到：  app.coresLeft  = 当前app申请的maxcpus - granted的cpus

 3、遍历未分配完全的apps，继续给它们分配资源，

 4、usableWorkers =  从当前ALIVE的Workers中过滤找出上文描述的可用Worker，然后根据cpus的资源空闲，从大到小给Workers排序。

 5、当toAssign（即将要分配的的core数>0，就找到可以的Worker持续分配）

 6、当可用Worker的free cores 大于 目前该Worker已经分配的core时，再给它分配1个core，这样分配是很平均的方法。

 7、round-robin轮询可用的Worker循环

 8、toAssign=0时结束循环，开始根据分配策略去真正的启动Executor。

举例： 1个APP申请了6个core, 现在有2个Worker可用。

      那么： toAssign = 6，assigned = 2

 那么就会在assigned(1)和assigned(0)中轮询平均分配cores，以+1 core的方式，最终每个Worker分到3个core，即每个Worker的启动一个Executor，每个Executor获得3个cores。

1. // Right now this is a very simple FIFO scheduler. We keep trying to fit in the first app
2. // in the queue, then the second app, etc.
3. **if** (spreadOutApps) {
4. // Try to spread out each app among all the nodes, until it has all its cores
5. **for** (app <- waitingApps **if** app.coresLeft > 0) { //对还未被完全分配资源的apps处理
6. val usableWorkers = workers.toArray.filter(\_.state == WorkerState.ALIVE)
7. .filter(canUse(app, \_)).sortBy(\_.coresFree).reverse //根据core Free对可用Worker进行降序排序。
8. val numUsable = usableWorkers.length //可用worker的个数 eg:可用5个worker
9. val assigned = **new** Array[Int](numUsable) //候选Worker，每个Worker一个下标，是一个数组，初始化默认都是0
10. var toAssign = math.min(app.coresLeft, usableWorkers.map(\_.coresFree).sum)//还要分配的cores = 集群中可用Worker的可用cores总和（10）， 当前未分配core（5）中找最小的
11. var pos = 0
12. **while** (toAssign > 0) {
13. **if** (usableWorkers(pos).coresFree - assigned(pos) > 0) { //以round robin方式在所有可用Worker里判断当前worker空闲cpu是否大于当前数组已经分配core值
14. toAssign -= 1
15. assigned(pos) += 1 //当前下标pos的Worker分配1个core +1
16. }
17. pos = (pos + 1) % numUsable //round-robin轮询寻找有资源的Worker
18. }
19. // Now that we've decided how many cores to give on each node, let's actually give them
20. **for** (pos <- 0 until numUsable) {
21. **if** (assigned(pos) > 0) { //如果assigned数组中的值>0，将启动一个executor在，指定下标的机器上。
22. val exec = app.addExecutor(usableWorkers(pos), assigned(pos)) //更新app里的Executor信息
23. launchExecutor(usableWorkers(pos), exec)  //通知可用Worker去启动Executor
24. app.state = ApplicationState.RUNNING
25. }
26. }
27. }
28. } **else** {

非SpreadOut分配策略：

非SpreadOut策略，该策略：会尽可能的根据每个Worker的剩余资源来启动Executor，这样启动的Executor可能只在集群的一小部分机器的Worker上。这样做对node较少的集群还可以，集群规模大了，Executor的并行度和机器负载均衡就不能够保证了。

当用户设定了参数spark.deploy.spreadOut 为false时，触发此游戏分支IMG_256，跑个题，有些困了。。

1、遍历可用Workers

2、且遍历Apps

3、比较当前Worker的可用core和app还需要分配的core，取最小值当做还需要分配的core

4、如果coreToUse大于0，则直接拿可用的core来启动Executor。。奉献当前Worker全部资源。（Ps：挨个榨干每个Worker的剩余资源。。。。）

举例： App申请12个core，3个Worker，Worker1剩余1个core, Worke2r剩7个core, Worker3剩余4个core.

这样会启动3个Executor，Executor1 占用1个core, Executor2占用7个core, Executor3占用4个core.

总结：这样是尽可能的满足App，让其尽快执行，而忽略了其并行效率和负载均衡。

[java] [view plain](http://blog.csdn.net/oopsoom/article/details/38763985" \o "view plain) [copy](http://blog.csdn.net/oopsoom/article/details/38763985" \o "copy)

1. } **else** {
2. // Pack each app into as few nodes as possible until we've assigned all its cores
3. **for** (worker <- workers **if** worker.coresFree > 0 && worker.state == WorkerState.ALIVE) {
4. **for** (app <- waitingApps **if** app.coresLeft > 0) {
5. **if** (canUse(app, worker)) { //直接问当前worker是有空闲的core
6. val coresToUse = math.min(worker.coresFree, app.coresLeft) //有则取，不管多少
7. **if** (coresToUse > 0) { //有
8. val exec = app.addExecutor(worker, coresToUse) //直接启动
9. launchExecutor(worker, exec)
10. app.state = ApplicationState.RUNNING
11. }
12. }
13. }
14. }
15. }
16. }

一、概述：

1、 在Worker Actor中，每次LaunchExecutor会创建一个CoarseGrainedExecutorBackend进程，一个Executor对应一个CoarseGrainedExecutorBackend

2、针对同一个App，每个Worker里只能有一个针对该App的Executor存在，切记。如果想让整个App的Executor变多，设置SPARK\_WORKER\_INSTANCES，让Worker变多。

3、Executor的资源分配有2种策略：

3.1、SpreadOut ：一种以round-robin方式遍历集群所有可用Worker，分配Worker资源，来启动创建Executor的策略，好处是尽可能的将cores分配到各个节点，最大化负载均衡和高并行。

3.2、非SpreadOut：会尽可能的根据每个Worker的剩余资源来启动Executor，这样启动的Executor可能只在集群的一小部分机器的Worker上。这样做对node较少的集群还可以，集群规模大了，Executor的并行度和机器负载均衡就不能够保证了。



本资源来自互联网：