一：任务调度与资源调度的区别

1. **任务调度：**是通过DAGScheduler，TaskScheduler，SchedulerBackend等进行的作业调度；
2. **资源调度：**是指应用程序如何获得资源；
3. **任务调度时在资源调度的基础上进行的，没有资源调度那么任务调度就成为了无源之水，无本之木。**

二：资源调度内幕天机揭秘

1. 因为Master负责资源管理和调度，所以资源调度的方法schedule位于Master.scala这个类中，当注册程序或者资源发生改变的时候都会导致schedule的调用，例如注册程序如下：

**case** RegisterApplication(description, driver) => {  
 // *TODO Prevent repeated registrations from some driver* **if** (*state* == RecoveryState.STANDBY) {  
 // ignore, don't send response  
 } **else** {  
 logInfo("Registering app " + description.name)  
 **val** app = createApplication(description, driver)  
 registerApplication(app)  
 logInfo("Registered app " + description.name + " with ID " + app.id)  
 *persistenceEngine*.addApplication(app)  
 driver.send(RegisteredApplication(app.id, self))  
 schedule()  
 }  
}

1. **schedule调用的时机**：每次有新的应用程序提交或者集群资源状况发生改变的时候(包括Executor增加或者减少，Worker增加或者减少等)；

*/\*\*  
 \* Schedule the currently available resources among waiting apps. This method will be called  
 \* every time a new app joins or resource availability changes.  
 \*/*

1. 当前Master必须是Alive的方式采用进行资源的调度，如果不是ALIVE的状态会直接返回，也就是说StandbyMaster不会进行Application的资源调度；

**if** (*state* != RecoveryState.ALIVE) { **return** }

1. 使用Random.shuffle把Master中保留的集群中所有Worker的信息随机打乱；为啥要打乱？为了负载均衡。

// Drivers take strict precedence over executors  
**val** shuffledWorkers = Random.shuffle(*workers*) // Randomization helps balance drivers

1. Shuffle源码详解：

将worker存入到ArrayBuffer中并赋值给buf.

swap函数: 是将索引位置上的Worker两两进行交换.

For循环： 从buf中最后一个元素开始循环，一直到索引为3，其中的nextInt是取0到n-1的随机数，然后调用swp()函数，将n-1和k进行交换，这样执行结束后，buf中的Worker顺序完全被打乱了。



Workers的源码是

**val** *workers* = **new** HashSet[WorkerInfo]

其算法内部是循环随机交换所有Worker在Master缓存数据结构中的位置；

1. 接下来要判断所有Worker中那些是ALIVE级别的Worker，ALIVE才能够参与资源的分配工作；

**for** (worker <- shuffledWorkers **if** worker.*state* == WorkerState.*ALIVE*) {

当SparkSubmit指定Driver在Cluster模式的情况下，此时Driver会加入waitingDriver等待列表中，在每个DriverInfo的DriverDescription中有要启动的Driver时候对Worker的内存及cores要求等内容。

Supervise: 如果是Cluster集群模式的话，SparkSubmit的时候，可以设置suprvise，Driver挂掉之后可以自动重启，但这个前提是Driver是在进群中的。

**private**[deploy] **case class** DriverDescription(  
 jarUrl: String,  
 mem: Int,  
 cores: Int,  
 supervise: Boolean,  
 command: Command) {  
  
 **override def** toString: String = s"DriverDescription (**$**{command.mainClass})"  
}

1. launchDriver源码：launch到worker中去了，而这个worker就是我们前面Shuffle之后打乱的Worker,此时就把Driver放到了Worker上。

**//判断Worker上的内存和可用的cores是否满足Driver的要求**

**if** (worker.memoryFree >= driver.desc.mem && worker.coresFree >= driver.desc.cores) {  
 launchDriver(worker, driver)

在符合资源要求的情况下，然后采用随机打乱后的一个Worker来启动Driver

**private def** launchDriver(worker: WorkerInfo, driver: DriverInfo) {  
//打印log日志信息

logInfo("Launching driver " + driver.id + " on worker " + worker.id)  
//worker:是workerInfo是对Worker的描述信息

//将Driver加入到Worker上的这个信息保存到WorkerInfo中，而WorkerInfo是master端持有的。

worker.addDriver(driver)

//同时，Driver也要将自己在那个Worker上面的信息加入到自己的描述信息里面//DriverInfo  
 driver.*worker* = *Some*(worker)  
 worker.endpoint.send(*LaunchDriver*(driver.id, driver.desc))  
 driver.*state* = DriverState.*RUNNING*}

Matser发指令给Worker，让远程的Worker启动Driver.

worker.endpoint.send(*LaunchDriver*(driver.id, driver.desc))

启动Driver之后，Driver的状态就变成了RUNNING。先启动Driver才会发生后续的一切的资源调度的模式。

driver.*state* = DriverState.*RUNNING*

1. startExecutorsOnWorkers()：为程序在Worker上启动Executor。

Spark默认为应用程序启动Executor的方式是FIFO的方式，也就是所有提交的应用程序都是放在调试的等待队列中的，先进先出，只有满足了前面应用程序的资源分配的基础上才能够满足下一个应用程序资源的分配；

Master调用了startExecutorsOnWorkers方法，但是在那个Worker上分配Executor，还不知道。

**private def** startExecutorsOnWorkers(): Unit = {  
 // Right now this is a very simple FIFO scheduler. We keep trying to fit in the first app  
 // in the queue, then the second app, etc.

为应用程序具体分配Executor之前要判断应用程序是否还需要分配Core,如果不需要则不会为应用程序分配Executor；

// Right now this is a very simple FIFO scheduler. We keep trying to fit in the first app  
// in the queue, then the second app, etc.  
**for** (app <- *waitingApps* **if** app.coresLeft > 0) {

coresLeft：

**private**[master] **def** coresLeft: Int = *requestedCores* - *coresGranted*

默认情况下一个executor分配一个core，coresPerExecutor是获得每个Executor上分配多少个cores。

**val** coresPerExecutor: Option[Int] = app.desc.coresPerExecutor

具体分配Executor之前要对要求Worker必须是ALIVE的状态且必须满足Application对每个Executor的内存和Cores的要求，并且在此基础上进行排序产生计算资源由大到小的usableWorkers(可用的Worker)数据结构：

// Filter out workers that don't have enough resources to launch an executor  
**val** usableWorkers = *workers*.toArray.filter(\_.*state* == WorkerState.*ALIVE*)  
 .filter(worker => worker.memoryFree >= app.desc.memoryPerExecutorMB &&  
 worker.coresFree >= coresPerExecutor.getOrElse(1))//至少大于一个  
//所有可用而且是符合条件的Worker进行排序，将cores多的最先选出

.sortBy(\_.coresFree).reverse// \_ => worker

scheduleExecutorsOnWorkers返回为每个Worker上分配的cores的数组。

然后将返回值复制给assignedCores。

**val** assignedCores = scheduleExecutorsOnWorkers(app, usableWorkers, spreadOutApps)

在FIFO的情况下默认是spreadOutApps来让应用程序尽可能多的运行在尽可能的Node上；

// As a temporary workaround before better ways of configuring memory, we allow users to set  
// a flag that will perform round-robin scheduling across the nodes (spreading out each app  
// among all the nodes) instead of trying to consolidate each app onto a small # of nodes.  
**private val** *spreadOutApps* = conf.getBoolean("spark.deploy.spreadOut", **true**)

为应用程序分配Executors有两种方式

第一种方式：是尽可能在集群的所有Worker上分配Executor，这样利于增大并发处理能力，这种方式往往会带来**潜在**的更好的数据本地性，资源分配的时候已经考虑到了最大化的本地性。

*\* Schedule executors to be launched on the workers.  
\* Returns an array containing number of cores assigned to each worker.  
\*  
\* There are two modes of launching executors. The first attempts to spread out an application's  
\* executors on as many workers as possible, while the second does the opposite (i.e. launch them  
\* on as few workers as possible). The former is usually better for data locality purposes and is  
\* the default.*

**private def** scheduleExecutorsOnWorkers(  
 app: ApplicationInfo,  
 usableWorkers: Array[WorkerInfo],  
 spreadOutApps: Boolean): Array[Int] = {  
 **val** coresPerExecutor = app.desc.coresPerExecutor

**//默认最小为每个executor分配一个core**  
**val** minCoresPerExecutor = coresPerExecutor.getOrElse(1)

**val** oneExecutorPerWorker = coresPerExecutor.isEmpty

**//每个Executor需要的memory**  
**val** memoryPerExecutor = app.desc.memoryPerExecutorMB

**//已经给每个Worker上分配的cores**

**val** assignedCores = **new** Array[Int](numUsable) // Number of cores to give to each worker  
**val** assignedExecutors = **new** Array[Int](numUsable) // Number of new executors on each worker  
**var** coresToAssign = math.*min*(app.coresLeft, usableWorkers.map(\_.coresFree).sum)

1. 具体在集群上分配Cores的时候会尽可能的满足我们的需求：

为啥要求最小值：因为可能我们的程序需要1000个cores，但是集群中只有100个cores。所以只能先分配100个cores，所以就要增加批次。

**//求最小值**

**var** coresToAssign = math.*min*(app.coresLeft, //程序需要的cores

//可用的Worker上面的free cores总和

usableWorkers.map(\_.coresFree).sum)

1. 判断Worker是否可以启动一个Executor。

*/\*\* Return whether the specified worker can launch an executor for this app. \*/*

*//筛选条件***def** canLaunchExecutor(pos: Int): Boolean = {

//必须要大于等于，因为如果默认一个cores都不能满足的话，无法启动了。  
 **val** keepScheduling = coresToAssign >= minCoresPerExecutor  
 **val** enoughCores = usableWorkers(pos).coresFree - assignedCores(pos) >= minCoresPerExecutor

// If we allow multiple executors per worker, then we can always launch new executors.  
 // Otherwise, if there is already an executor on this worker, just give it more cores.

//  
 **val** launchingNewExecutor = !oneExecutorPerWorker || assignedExecutors(pos) == 0  
 **if** (launchingNewExecutor) {

//具体executor上分配的内存  
 **val** assignedMemory = assignedExecutors(pos) \* memoryPerExecutor  
 **val** enoughMemory = usableWorkers(pos).memoryFree - assignedMemory >= memoryPerExecutor  
 **val** underLimit = assignedExecutors.sum + app.*executors*.size < app.*executorLimit* keepScheduling && enoughCores && enoughMemory && underLimit  
 } **else** {

//如果Worker上的executor已经存在，可用直接往executor上增加cores  
 // We're adding cores to an existing executor, so no need  
 // to check memory and executor limits  
 keepScheduling && enoughCores  
 }  
}

// Keep launching executors until no more workers can accommodate any  
// more executors, or if we have reached this application's limits

//根据filter就过滤出来满足在Worker上launchExecutor的条件  
**var** freeWorkers = (0 until numUsable).filter(canLaunchExecutor)

//可用的Worker不是空的话，就执行下面的循环。  
**while** (freeWorkers.nonEmpty) {  
 freeWorkers.foreach { pos =>  
 **var** keepScheduling = **true  
 while** (keepScheduling && canLaunchExecutor(pos)) {  
 coresToAssign -= minCoresPerExecutor  
 assignedCores(pos) += minCoresPerExecutor

如果是每个Worker下面只能够为当前的应用程序分配一个Executor的话，每次是分配一个Core!

如果是spreadOutApps(也是系统默认的情况下)的时候，会尽量使用集群中所有的executors. 每次都会给executor增加一个core。

如果不是spreadOutApps的时候，每次都会给executor增加一个core，会一直循环当前程序的executor上的freeCores,所以会占用本机器上的尽可能多的cores。

// If we are launching one executor per worker, then every iteration assigns 1 core  
// to the executor. Otherwise, every iteration assigns cores to a new executor.  
**if** (oneExecutorPerWorker) {  
 assignedExecutors(pos) = 1  
} **else** {  
 assignedExecutors(pos) += 1  
}

// Spreading out an application means spreading out its executors across as  
// many workers as possible. If we are not spreading out, then we should keep  
// scheduling executors on this worker until we use all of its resources.  
// Otherwise, just move on to the next worker.

//如果不是spreadOutApps的话，会尽可能用当前的机器去处理程序的一切的cores需求，也就是executor会占用尽可能多的cores。  
**if** (spreadOutApps) {  
 keepScheduling = **false**}

**至此已经决定了在那台分配多少个executor，每个executor上分配多少个core**

1. 下面就开始具体分配了。

// Now that we've decided how many cores to allocate on each worker, let's allocate them

//**前面是决定了每个worker上分配多少个cores，下面具体完成了。**  
**for** (pos <- 0 until usableWorkers.length **if** assignedCores(pos) > 0) {  
 allocateWorkerResourceToExecutors(  
 app, assignedCores(pos), coresPerExecutor, usableWorkers(pos))  
}

1. 具体看一下allocateWorkerResourceToExecutprs函数具体是如何实现的。

*/\*\*  
 \* Allocate a worker's resources to one or more executors.  
 \** ***@param app*** *the info of the application which the executors belong to  
 \** ***@param assignedCores*** *number of cores on this worker for this application  
 \** ***@param coresPerExecutor*** *number of cores per executor  
 \** ***@param worker*** *the worker info  
 \*/***private def** allocateWorkerResourceToExecutors(  
 app: ApplicationInfo,  
 assignedCores: Int,  
 coresPerExecutor: Option[Int],  
 worker: WorkerInfo): Unit = {  
 // If the number of cores per executor is specified, we divide the cores assigned  
 // to this worker evenly among the executors with no remainder.  
 // Otherwise, we launch a single executor that grabs all the assignedCores on this worker.  
 **val** numExecutors = coresPerExecutor.map { assignedCores / \_ }.getOrElse(1)  
 **val** coresToAssign = coresPerExecutor.getOrElse(assignedCores)  
 **for** (i <- 1 to numExecutors) {

*//将要分配executor的元数据信息加入到app,*  
 **val** exec = app.addExecutor(worker, coresToAssign)

*//luanchExecutor，分配executor*  
 launchExecutor(worker, exec)

*//分配executor之后，application就为RUNNING*  
 app.*state* = ApplicationState.*RUNNING* }  
}

1. addExecutor： 返回executor的描述信息。

**private**[master] **def** addExecutor(  
 worker: WorkerInfo,  
 cores: Int,  
 useID: Option[Int] = None): ExecutorDesc = {  
 **val** exec = **new** ExecutorDesc(newExecutorId(useID), **this**, worker, cores, desc.memoryPerExecutorMB)  
 *executors*(exec.id) = exec

**//在分配前就已经确定为executor分配多少cores。**  
 *coresGranted* += cores //为executor分配cores添加到executor描述信息中  
 exec  
}

1. 准备具体要为当前应用程序分配的Executor信息后，Master要通过远程通信发指令给Worker来具体启动ExecutorBackend进程；

**private def** launchExecutor(worker: WorkerInfo, exec: ExecutorDesc): Unit = {  
 logInfo("Launching executor " + exec.fullId + " on worker " + worker.id)  
 worker.addExecutor(exec)

**//让Worker接收到LaunchExecutor的指令。**  
 worker.endpoint.send(*LaunchExecutor*(*masterUrl*,  
 exec.application.id, exec.id, exec.application.desc, exec.cores, exec.memory))  
 exec.application.driver.send(  
 *ExecutorAdded*(exec.id, worker.id, worker.hostPort, exec.cores, exec.memory))  
}

1. 紧接着给我们应用程序的Driver发送一个ExecutorAdded的信息；

exec.application.driver.send(  
 *ExecutorAdded*(exec.id, worker.id, worker.hostPort, exec.cores, exec.memory))