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. **任务调度是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：资源调度内幕天机解密

1. 因为Master负责资源管理和调度，所以资源调度的方法shedule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，例如注册程序的时候：

**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）；
2. 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Standby Master不会进行Application的资源调用！

**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；

**val** shuffledWorkers = Random.shuffle(*workers*) // Randomization helps balance drivers

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：

**def** shuffle[T, CC[X] <: TraversableOnce[X]](xs: CC[T])(**implicit** bf: CanBuildFrom[CC[T], T, CC[T]]): CC[T] = {  
 **val** buf = **new** ArrayBuffer[T] ++= xs  
  
 **def** swap(i1: Int, i2: Int) {  
 **val** tmp = buf(i1)  
 buf(i1) = buf(i2)  
 buf(i2) = tmp  
 }  
  
 **for** (n <- buf.length to 2 by -1) {  
 **val** k = nextInt(n)  
 swap(n - 1, k)  
 }  
  
 (bf(xs) ++= buf).result  
}

1. 接下来要判断所有Worker中哪些是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*) {

1. 当SparkSubmit指定Driver在Cluster模式的情况下，此时Driver会加入waitingDrivers等待列表中，在每个DriverInfo的DriverDescription中有要启动Driver时候对Worker的内存及Cores是要求等内容：

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

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) {  
 logInfo("Launching driver " + driver.id + " on worker " + worker.id)  
 worker.addDriver(driver)  
 driver.*worker* = *Some*(worker)  
 worker.endpoint.send(*LaunchDriver*(driver.id, driver.desc))  
 driver.*state* = DriverState.*RUNNING*}

Master发指令给Worker，让远程的Worker启动Driver：

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

1. 先启动Driver才会发生后续的一切的资源调度的模式。
2. 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度的等待队列中的，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；
3. 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；
4. 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数据结构：

**val** usableWorkers = workers.toArray.filter(\_.state == WorkerState.ALIVE)  
 .filter(worker => worker.memoryFree >= app.desc.memoryPerExecutorMB &&  
 worker.coresFree >= coresPerExecutor.getOrElse(1))  
 .sortBy(\_.coresFree).reverse

在FIFO的情况下默认是spreadOutApps来让应用程序尽可能多的运行在所有的Node上：

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

1. 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，这种方式往往会带来潜在的更好的数据本地性；
2. 具体在集群上分配Cores的时候会尽可能的满足我们的要求：

**var** coresToAssign = math.*min*(app.coresLeft, usableWorkers.map(\_.coresFree).sum)

1. 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！

// 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  
}

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：

worker.endpoint.send(*LaunchExecutor*(*masterUrl*,  
 exec.application.id, exec.id, exec.application.desc, 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))