1. DAGScheduler会把TaskSet通过submitTasks提交给TaskScheduler，在standalone的情况下也就是提交给了TaskSchedulerImpl。

**if** (tasks.size > 0) {  
 logInfo("Submitting " + tasks.size + " missing tasks from " + stage + " (" + stage.rdd + ")")  
 stage.*pendingPartitions* ++= tasks.map(\_.partitionId)  
 logDebug("New pending partitions: " + stage.*pendingPartitions*)  
 taskScheduler.submitTasks(**new** TaskSet(  
 tasks.toArray, stage.id, stage.latestInfo.attemptId, jobId, properties))  
 stage.latestInfo.*submissionTime* = *Some*(clock.getTimeMillis())

TaskScheduler里面只是定义了submitTasks方法，具体实现是在TaskSchedulerImpl

// Submit a sequence of tasks to run.  
**def** submitTasks(taskSet: TaskSet): Unit

TaskSchedulerImpl里面复写了submitTasks方法。

**override def** submitTasks(taskSet: TaskSet) {

TaskSchedulerImpl是TaskScheduler的具体实现。

**private**[spark] **class** TaskSchedulerImpl(  
 **val** sc: SparkContext,  
 **val** maxTaskFailures: Int,  
 isLocal: Boolean = **false**)  
 **extends** TaskScheduler **with** Logging

在submitTasks方法中创建了createTaskSetManager。

**override def** submitTasks(taskSet: TaskSet) {  
 **val** tasks = taskSet.tasks  
 logInfo("Adding task set " + taskSet.*id* + " with " + tasks.length + " tasks")  
 **this**.synchronized {  
 **val** manager = createTaskSetManager(taskSet, maxTaskFailures)

而createTaskSetManager中创建了TaskSetManager实例。

// Label as private[scheduler] to allow tests to swap in different task set managers if necessary  
**private**[scheduler] **def** createTaskSetManager(  
 taskSet: TaskSet,  
 maxTaskFailures: Int): TaskSetManager = {  
 **new** TaskSetManager(**this**, taskSet, maxTaskFailures)  
}

TaskSetManager里面的参数解析

this : 也就是TaskSchedulerImpl

taskSet：是DAGScheduler传过来的task

maxTaskFailures：最大的失败重试次数，默认情况下最大失败重试次数是4

**def this**(sc: SparkContext) = **this**(sc, sc.conf.getInt("spark.task.maxFailures", 4))

**小结一下：**

**DAGScheduler将TaskSet传给TaskScheduler，TaskScheduler是一个接口，TaskSchedulerImpl是他的具体实现，TaskSchedulerImpl里面复写了submitTasks方法来实现接收TaskSet。**

**但是TaskSchedulerImpl是在哪创建的呢？**

在SparkContext里面的createTaskScheduler中case匹配到了standalone的模式下，创建了

TaskSchedulerImpl。

**case** SPARK\_REGEX(sparkUrl) =>  
 **val** scheduler = **new** TaskSchedulerImpl(sc)  
 **val** masterUrls = sparkUrl.split(",").map("spark://" + \_)  
 **val** backend = **new** SparkDeploySchedulerBackend(scheduler, sc, masterUrls)  
 scheduler.initialize(backend)  
 (backend, scheduler)

**上述步骤完成之后，DAGScheduler就将TaskSet加入了TaskSetManager里面。**

**接下来应用程序的调度器就登场了。**

1. 其中schedulableBuilder是应用程序级别的调度器。

*schedulableBuilder*.addTaskSetManager(manager, manager.taskSet.properties)

schedulableBuilder在创建的时候就进行了实例化

**var** *schedulableBuilder*: SchedulableBuilder = **null**

在initialize的方法中对schedulableBuilder进行实例化，schedulableBuilder的调度分两种，具体两种调度详解请查看下面的补讲里面的内容。

**def** initialize(backend: SchedulerBackend) {  
 **this**.*backend* = backend  
 // temporarily set rootPool name to empty  
 *rootPool* = **new** Pool("", *schedulingMode*, 0, 0)  
 *schedulableBuilder* = {  
 *schedulingMode* **match** {  
 **case** SchedulingMode.*FIFO* =>  
 **new** FIFOSchedulableBuilder(*rootPool*)  
 **case** SchedulingMode.*FAIR* =>  
 **new** FairSchedulableBuilder(*rootPool*, *conf*)  
 }  
 }

**并且默认情况下是FIFO的方式：**

// default scheduler is FIFO  
**private val** *schedulingModeConf* = *conf*.get("spark.scheduler.mode", "FIFO")

schedulableBuilder是一个接口，里面定义了addTaskSetManager方法。

**private**[spark] **trait** SchedulableBuilder {  
 **def** rootPool: Pool  
  
 **def** buildPools()  
  
 **def** addTaskSetManager(manager: Schedulable, properties: Properties)  
}

**schedulableBuilder确定了TaskSetManager调度顺序。**

**知道了schedulableBuilder是咋回事之后，那么真正的调用就开始啦！**

然后按照TaskSetManager的locality aware来确定每个Task具体运行在哪个ExecutorBackend中；

1. CoarseGrainedSchedulerBackend.reviveOffers:给DriverEndpoint发送ReviveOffers。

*backend*.reviveOffers()

而scheduleBackend只是定义了reviveOffers方法。

**def** reviveOffers(): Unit

reviveOffers方法的具体实现是在：在CoarseGrainedSchedulerBackend实现，给DriverEndpoint发送ReviveOffers消息。

**override def** reviveOffers() {  
 *driverEndpoint*.send(ReviveOffers)  
}

ReviveOffers本身是一个空的case object对象，只是起到触发底层资源调度的作用，在有Task提交或者计算资源变动的时候会发送ReviveOffers这个消息作为触发器。

// Internal messages in driver  
**case object** ReviveOffers **extends** CoarseGrainedClusterMessage

1. 此时DriverEndpoint收到ReviveOffers后，路由到makeOffers中。

**case** ReviveOffers =>  
 makeOffers()

**首先会准备好所有可以用于计算的workOffers**(代表了所有可用ExecutorBackend中可以使用的Cores等信息),因为之前的资源已经分配好了，现在只需要关系有哪些cores可以用于Task计算。

// Make fake resource offers on all executors  
**private def** makeOffers() {  
 // Filter out executors under killing  
 **val** activeExecutors = *executorDataMap*.filterKeys(executorIsAlive)

//产生集合，里面包含executor的ID,freeCores  
 **val** workOffers = activeExecutors.map { **case** (id, executorData) =>  
 **new** WorkerOffer(id, executorData.executorHost, executorData.freeCores)  
 }.toSeq  
 launchTasks(scheduler.resourceOffers(workOffers))  
}

**将可用的计算资源准备好后，下面就可以为每个Task分配计算资源了。**

1. TaskSchedulerImpl.resourceOffers为每一个Task具体分配计算资源。输入是workOffers代表可用的资源，实质上是ExecutorBackend的列表。

launchTasks(scheduler.resourceOffers(workOffers))

输出值是：TaskDescription的二维数组

// Launch tasks returned by a set of resource offers  
**private def** launchTasks(tasks: Seq[Seq[TaskDescription]]) {

TaskDescription源码：

被TaskSetManager.resourceOffer创建的。**而TaskDescription是用来描述哪些要发送到executorbackend上计算的Task。也就是说TaskDescription此时描述的这个Task，是已经确定好了在哪个**ExecutorBackend上运行。而确定Task具体运行在哪个ExecutorBackend上的算法是由TaskSetManager的**resourceOffers**方法来定的。

*/\*\*  
 \* Description of a task that gets passed onto executors to be executed, usually created by  
 \** [[*TaskSetManager.resourceOffer*]]*.  
 \*/***private**[spark] **class** TaskDescription(  
 **val** taskId: Long,  
 **val** attemptNumber: Int,  
 **val** executorId: String,  
 **val** name: String,  
 **val** index: Int, // Index within this task's TaskSet  
 \_serializedTask: ByteBuffer)  
 **extends** Serializable {

**resourceOffers到底是如何确定Task具体运行在哪个ExecutorBackend上的呢？算法的实现具体如下：**

**具体到resourceOffers查看源码如下：**

1. 通过Random.shuffle打散的是executorBackend的计算资源，防止Task集中分布到某些机器上，为了负载均衡。

// Randomly shuffle offers to avoid always placing tasks on the same set of workers.  
**val** shuffledOffers = Random.shuffle(offers)

1. 根据每个ExecutorBackend的cores的个数声明类型为TaskDecription的ArrayBuffer数组。

// Build a list of tasks to assign to each worker.

//为每个worker创建了一个ArrayBuffer实例，

//每个executor上能放多少个TaskDescription就可以运行多少个Task。

//tasks的数组长度是由cores的多少决定的，cores也决定了worker上可以运行多少//个任务。

**val** tasks = shuffledOffers.map(o => **new** ArrayBuffer[TaskDescription](o.cores))  
**val** availableCpus = shuffledOffers.map(o => o.cores).toArray

**//** getSortedTaskSetQueue对TaskSetManager按照调度策略进行排序，将排序好的结//果赋值给sortedTaskSets  
**val** sortedTaskSets = *rootPool*.getSortedTaskSetQueue

1. 如果有新的ExecutorBackend分配给我们的Job，此时会调用executorAdd来获取最新的完整的可用计算的计算资源，因为在执行中集群中的资源可能会动态的改变的。

**for** (taskSet <- sortedTaskSets) {  
 logDebug("parentName: %s, name: %s, runningTasks: %s".format(  
 taskSet.*parent*.*name*, taskSet.*name*, taskSet.runningTasks))  
 **if** (newExecAvail) { //如果有可用的新的executor  
 taskSet.executorAdded()  
 }

1. 下面的增强for循环执行是这样的，每取出一个taskSet，maxLocality就会依次从PROCESS\_LOCAL, NODE\_LOCAL, NO\_PREF, RACK\_LOCAL, ANY遍历。从优先级高到低来遍历。追求最高级别的优先级本地性。maxLocality会传入resourceOfferSingleTaskSet.

// Take each TaskSet in our scheduling order, and then offer it each node in increasing order  
 // of locality levels so that it gets a chance to launch local tasks on all of them.  
 // NOTE: the preferredLocality order: PROCESS\_LOCAL, NODE\_LOCAL, NO\_PREF, RACK\_LOCAL, ANY  
 **var** launchedTask = **false  
 for** (taskSet <- sortedTaskSets; maxLocality <- taskSet.*myLocalityLevels*) {  
 **do** {  
 launchedTask = resourceOfferSingleTaskSet(  
 taskSet, maxLocality, shuffledOffers, availableCpus, tasks)  
 } **while** (launchedTask)  
 }  
  
 **if** (tasks.size > 0) {  
 *hasLaunchedTask* = **true** }  
 **return** tasks  
}

**下面具体看一下resourceOfferSingleTaskSet源码**

5． 此时的maxLocality就传入到了resourceOffer，通过调用TastSetManager的resourceOffer来确定Task应该运行在哪个ExecutorBackend的具体的Locality Level;

**for** (i <- 0 until shuffledOffers.size) {//循环遍历当前存在的executor  
 **val** execId = shuffledOffers(i).executorId //获取executor的ID  
 **val** host = shuffledOffers(i).host //executor的host名字  
 **if** (availableCpus(i) >= CPUS\_PER\_TASK) { //每台机器可用的计算资源  
 **try** {  
 **for** (task <- taskSet.resourceOffer(execId, host, maxLocality)) {  
 tasks(i) += task  
 **val** tid = task.taskId  
 *taskIdToTaskSetManager*(tid) = taskSet  
 *taskIdToExecutorId*(tid) = execId  
 *executorIdToTaskCount*(execId) += 1  
 *executorsByHost*(host) += execId  
 availableCpus(i) -= CPUS\_PER\_TASK  
 *assert*(availableCpus(i) >= 0)  
 launchedTask = **true** }

1. 确定好Task具体在哪个ExecutorBackend执行之后，通过luanchTasks把任务发送给ExecutorBackend去执行。

launchTasks(scheduler.resourceOffers(workOffers))

**补讲：**

1. Task默认的最大重试次数是4次：

**def this**(sc: SparkContext) = **this**(sc, sc.conf.getInt("spark.task.maxFailures", 4))

1. Spark应用程序目前支持两种调度器：FIFO、FAIR，可以通过spark-env.sh中spark.scheduler.mode进行具体的设置，默认情况下是FIFO的方式：

**private val** *schedulingModeConf* = *conf*.get("spark.scheduler.mode", "FIFO")  
**val** *schedulingMode*: SchedulingMode = **try** {  
 SchedulingMode.withName(*schedulingModeConf*.toUpperCase)

1. TaskScheduler中要负责为Task分配计算资源：此时程序已经具备集群中的计算资源了，根据计算本地性原则确定Task具体要运行在哪个ExecutorBackend中；
2. 数据本地优先级从高到底以此为：优先级高低排： PROCESS\_LOCAL, NODE\_LOCAL, NO\_PREF, RACK\_LOCAL, ANY，其中NO\_PREF是指机器本地性
3. **每个Task默认分配的core数为1**

// CPUs to request per task  
**val** *CPUS\_PER\_TASK* = *conf*.getInt("spark.task.cpus", 1)

1. **TaskSet类详解**

**TaskSet包含了一系列高层调度器交给底层调度器的任务的集合。**

*/\*\*  
 \* A set of tasks submitted together to the low-level TaskScheduler, usually representing  
 \* missing partitions of a particular stage.  
 \*/***private**[spark] **class** TaskSet(  
 **val** tasks: Array[Task[\_]],//任意类型的Task  
 **val** stageId: Int, //Task属于哪个Stage  
 **val** stageAttemptId: Int, //尝试的Id  
 **val** priority: Int, //优先级  
 **val** properties: Properties) {  
 **val** *id*: String = stageId + "." + stageAttemptId  
  
 **override def** toString: String = "TaskSet " + *id*}

调度的时候，底层是有一个pool调度池，这个调度池会规定Stage提交之后具体执行的优先级。

TaskSetManager（TaskSet的管理者）

实例化的时候要完成TaskSchedulerImpl工作的。

**private**[spark] **class** TaskSetManager(  
 sched: TaskSchedulerImpl,  
 **val** taskSet: TaskSet, //接收提交的任务的集合  
 **val** maxTaskFailures: Int,//最大失败提交次数  
 clock: Clock = **new** SystemClock())  
 **extends** Schedulable **with** Logging {  
  
 **val** *conf* = sched.sc.conf

1. DAGScheduler是从数据层面考虑preferedLocation的，确定数据在哪，而TaskScheduler是从具体计算Task角度考虑计算的本地性，在哪计算，优先考虑在内存中。
2. **Task进行广播时候的AKKAFrameSize大小为128MB，如果任务大于128MB-200K的时候，则Task会直接被丢弃掉。**

*/\*\* Returns the configured max frame size for Akka messages in bytes. \*/***def** maxFrameSizeBytes(conf: SparkConf): Int = {  
 **val** frameSizeInMB = conf.getInt("spark.akka.frameSize", 128)

**如果小于128 MB-200K的话会通过**CoarseGrainedSchedulerBackend去luanch到具体的ExecutorBackend上。executorEndpoint就会把当前的Task发送到要运行的executorBackend上。通过LaunchTask实现。

executorData.executorEndpoint.send(*LaunchTask*(**new** SerializableBuffer(serializedTask)))