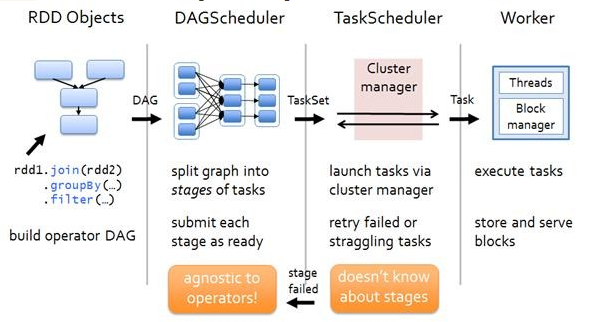
Spark作业执行原理

Spark的作业和任务调度系统是核心，它能够有效的进行调度根本原因是对任务划分DAG和容错，使得它对底层到底层的各个模块之间的调用和处理显得游刃有余，相关术语：

1. 作业（Job），RDD中由Action操作所生成的一个或多个调度阶段
2. 调度阶段（Stage），每个作业会因为RDD之间的依赖关系拆分成多组任务集合，称为调度阶段，也叫作任务集（TaskSet）。调度阶段的划分是由DAGSheduler来划分的，调度阶段有Shuffle Map Stage和Result Stage两种。
3. 任务（Task），分发到Executor上的工作任务，是Spark实际执行应用的最小单元
4. DAGScheduler，面向调度阶段的任务调度器，负责接收Spark应用提交的作业，根据RDD的依赖关系划分调度阶段，并提交调度阶段给TaskScheduler。
5. TaskScheduler，面向任务的调度器，它接受DAGScheduler提交过来的调度阶段，然后把任务分发到Worker节点运行，由Worker节点的Executor来运行该任务。

Spark的作业调度主要是基于RDD的一系列操作构成一个作业，然后在Executor中执行。这些操作算法主要分为转换操作和行动操作，对于转换操作的计算是Lazy级别，也就是延迟执行，只有出现了Action操作才触发作业的提交。在Spark调度中最重要的是DAGScheduler和TaskScheduler两个调度器，其中DAGScheduler负责任务的逻辑调度，将作业拆分成不同阶段具有依赖关系的任务集，而TaskScheduler则负责具体任务的调度执行，下图是Spark作业和任务调度系统：



Spark的作业和任务调度系统

1. Spark应用程序进行各种转换操作，通过action操作触发Job的运行。提交之后根据RDD之间的依赖关系构建DAG图，然后提交个DAGScheduler进行解析
2. DAGScheduler是面向调度阶段的高层次的调度器，DAGScheduler把DAG拆分为相互依赖的调度阶段，以RDD的依赖是否否宽依赖，当遇到宽依赖进行划分为新的调度阶段。每个调度阶段包含一个或多个任务，这些任务形成任务集，提交给底层调度TaskScheduler进行调度进行。另外，DAGScheduler记录哪些RDD被存入磁盘等物化阶段，同时要需求任务的最优化调度，例如数据本地性等。DAGScheduler监控运行调度阶段过程，如果某个调度阶段失败，则需要重新提交给调度阶段。
3. 每个TaskScheduler只为一个SparkContext实例服务，TaskScheduler接收来自DAGScheduler发送过来的任务集，TaskScheduler收到任务集后负责把任务集以任务的形式一个个分发到集群的Worker节点的Executor中去运行。如果某个任务执行失败，TaskScheduler负责重试。另外如果TaskScheduler发现某个任务一直未完成，就可能启动同样的任务运行同一个任务，那个任务先运行完就用哪个任务的结果。
4. Worker中的Executor收到TaskScheduler发送过来的任务后，以多线程的方式运行，每个线程负责一个任务。任务运行结束后返回给TaskScheduler，不同类型的任务，返回的方式也不同。ShuffleMapTask返回的是一个MapStatus对象，而不是结果本身。ResultTask根据结果大小的不同，返回的方式也不同。

Spark作业执行的实现类图如下所示：



# 1.DAGScheduler

DSGScheduler的相关类图如下所示：

下面介绍TaskScheduler的相关概念：

1. ActiveJob，Job是以ActiveJob类为代表，根据finalStage分为两种：a result Job（对应ResultStage）或者a map-stage Job(对应ShuffleMapStage，主要用在查询计划上)

*private[spark] class ActiveJob(*

*val jobId: Int,*

*val finalStage: Stage,*

*val callSite: CallSite,*

*val listener: JobListener,*

*val properties: Properties) {*

*val numPartitions = finalStage match {*

*case r: ResultStage => r.partitions.length*

*case m: ShuffleMapStage => m.rdd.partitions.length*

*}*

*val finished = Array.fill[Boolean](numPartitions)(false)*

*var numFinished = 0*

*}*

2）Stage，就是一组并行的task，每个stage之间以Shuffle为边界进行划分。Stage分为两种：

ShuffleMapStage和ResultStage，以下是Stage类：

*private[scheduler] abstract class Stage(*

*val id: Int,*

*val rdd: RDD[\_],*

*val numTasks: Int,*

*val parents: List[Stage],*

*val firstJobId: Int,*

*val callSite: CallSite)*

*extends Logging {*

*val numPartitions = rdd.partitions.length*

*val jobIds = new HashSet[Int]*

*private var nextAttemptId: Int = 0*

*val name: String = callSite.shortForm*

*val details: String = callSite.longForm*

*private var \_latestInfo: StageInfo = StageInfo.fromStage(this, nextAttemptId)*

*val fetchFailedAttemptIds = new HashSet[Int]*

*private[scheduler] def clearFailures() : Unit = {*

*fetchFailedAttemptIds.clear()*

*}*

*def makeNewStageAttempt(*

*numPartitionsToCompute: Int,*

*taskLocalityPreferences: Seq[Seq[TaskLocation]] = Seq.empty): Unit = {*

*val metrics = new TaskMetrics*

*metrics.register(rdd.sparkContext)*

*\_latestInfo = StageInfo.fromStage(*

*this, nextAttemptId, Some(numPartitionsToCompute), metrics, taskLocalityPreferences)*

*nextAttemptId += 1*

*}*

*def latestInfo: StageInfo = \_latestInfo*

*override final def hashCode(): Int = id*

*override final def equals(other: Any): Boolean = other match {*

*case stage: Stage => stage != null && stage.id == id*

*case \_ => false*

*}*

*def findMissingPartitions(): Seq[Int]*

*}*

3）Task，对应两个类，ShuffleMapTask和ResultTask，其中前者执行任务并将输出写入分区；后者执行任务将输出发送到Driver Application中。

4）DAGScheduler内部维护了各种task/stage/job之间的映射关系

# 2.Spark提交作业过程

作业调度的两个主要入口是submitJob和runJob，两者的区别在于前者返回一个Jobwaiter，可以用在异步调用中，用于判断作业完成或者取消作业，本质上runJob在内部调用submitJob,阻塞知道作业完成或者失败。

用户代码都是基于RDD的一系列计算操作，当RDD触发Action算子后，Spark往Cluster上提交作业，例如count,collect等，这些操作自动触发。通过SparkContext创建一个RDD:

*scala> val textFile = sc.textFile("README.md")*

*textFile: org.apache.spark.rdd.RDD[String] = README.md MapPartitionsRDD[7] at textFile*

RDD的count 操作会自动触发runJob提交作业，不需要用户显示的提交作业：

*scala> textFile.count()*

*res3: Long = 103*

RDD.count,在类内部隐性调动runJob方法，提交作业：

*def count(): Long = sc.runJob(this, Utils.getIteratorSize \_).sum*

SparkContext.runJob中根据依赖关系形成DAG,然后交给DAGScheduler处理，其runJob方法中内部调用submitJob:

*def runJob[T, U](*

*rdd: RDD[T],*

*func: (TaskContext, Iterator[T]) => U,*

*partitions: Seq[Int],*

*callSite: CallSite,*

*resultHandler: (Int, U) => Unit,*

*properties: Properties): Unit = {*

*……*

*val waiter = submitJob(rdd, func, partitions, callSite, resultHandler, properties)*

*ThreadUtils.awaitReady(waiter.completionFuture, Duration.Inf)*

*waiter.completionFuture.value.get match {*

*case scala.util.Success(\_) =>*

*logInfo("Job %d finished: %s, took %f s".format*

*(waiter.jobId, callSite.shortForm, (System.nanoTime - start) / 1e9))*

*case scala.util.Failure(exception) =>*

*logInfo("Job %d failed: %s, took %f s".format*

*(waiter.jobId, callSite.shortForm, (System.nanoTime - start) / 1e9))*

*// SPARK-8644: Include user stack trace in exceptions coming from DAGScheduler.*

*val callerStackTrace = Thread.currentThread().getStackTrace.tail*

*exception.setStackTrace(exception.getStackTrace ++ callerStackTrace)*

*throw exception*

*}*

*}*

在runJob中调用submitJob：

*def submitJob[T, U](*

*rdd: RDD[T],*

*func: (TaskContext, Iterator[T]) => U,*

*partitions: Seq[Int],*

*callSite: CallSite,*

*resultHandler: (Int, U) => Unit,*

*properties: Properties): JobWaiter[U] = {*

*// Check to make sure we are not launching a task on a partition that does not exist.*

*val maxPartitions = rdd.partitions.length*

*partitions.find(p => p >= maxPartitions || p < 0).foreach { p =>*

*throw new IllegalArgumentException(*

*"Attempting to access a non-existent partition: " + p + ". " +*

*"Total number of partitions: " + maxPartitions)*

*}*

*val jobId = nextJobId.getAndIncrement()*

*if (partitions.size == 0) {*

*return new JobWaiter[U](this, jobId, 0, resultHandler)*

*}*

*assert(partitions.size > 0)*

*val func2 = func.asInstanceOf[(TaskContext, Iterator[\_]) => \_]*

*val waiter = new JobWaiter(this, jobId, partitions.size, resultHandler)*

*eventProcessLoop.post(JobSubmitted(*

*jobId, rdd, func2, partitions.toArray, callSite, waiter,*

*SerializationUtils.clone(properties)))*

*waiter*

*}*

DAGScheduler在submitJob方法中处理JobSubmitted事件，调用方法handleJobSubmitted。在该方法中，DAGScheduler为Job分割Stage，同时决定最佳的调度方案，生成TaskSet后，将其教给TaskScheduler，当遇到Stage Lost会重新提交Task。

在Spark应用程序中，会拆分多个作业，对于多个作业的调度提供了两种调度策略：FIFO（默认）和FAIR模式。

# 3.Spark划分调度Stage

Spark调度阶段的划分是由DAGScheduler实现的，DAGScheduler从最后一个RDD出发广度优先遍历整个依赖树，从而划分调度阶段（以操作是否为宽依赖，ShuffleDependency进行），当某个RDD的操作时Shuffle时，以该Shuffle操作为界限划分前后两个调度阶段。

代码的实现是在DAGScheduler的handleJobSubmitted方法中根据最后一个RDD生成ResultStage开始，具体的方式是从finalRDD使用getParentStages方法中找出其依赖的parents RDD是否存在shuffle操作，如果没有存在Shuffle操作，则本次作业仅有一个ResultStage，该ResultStage不存在父调度节点。如果存在Shuffle操作，则本次作业存在一个ResultStage和至少一个ShuffleMapStage,则该ResultStage存在父调度节点，代码如下：

*private[scheduler] def handleJobSubmitted(jobId: Int,*

*finalRDD: RDD[\_],*

*func: (TaskContext, Iterator[\_]) => \_,*

*partitions: Array[Int],*

*callSite: CallSite,*

*listener: JobListener,*

*properties: Properties) {*

*var finalStage: ResultStage = null*

*try {*

*finalStage = createResultStage(finalRDD, func, partitions, jobId, callSite)*

*}*

*val job = new ActiveJob(jobId, finalStage, callSite, listener, properties)*

*clearCacheLocs()*

*val jobSubmissionTime = clock.getTimeMillis()*

*jobIdToActiveJob(jobId) = job*

*activeJobs += job*

*finalStage.setActiveJob(job)*

*val stageIds = jobIdToStageIds(jobId).toArray*

*val stageInfos = stageIds.flatMap(id => stageIdToStage.get(id).map(\_.latestInfo))*

*listenerBus.post(*

*SparkListenerJobStart(job.jobId, jobSubmissionTime, stageInfos, properties))*

*submitStage(finalStage)}*

在上面的代码中，把final RDD传入createResultStage方法中，该方法调用getOrCreateParent

Stages，生成finalStage，

*private def getOrCreateParentStages(rdd: RDD[\_], firstJobId: Int): List[Stage] = {*

*getShuffleDependencies(rdd).map { shuffleDep =>*

*getOrCreateShuffleMapStage(shuffleDep, firstJobId)*

*}.toList*

*}*

当final RDD存在父调度阶段，需要从Shuffle操作的RDD往前遍历，找到所有的ShuffleMapStage，这是Stage划分的最关键部分，先调用getShuffleDependencies，如下：

*private[scheduler] def getShuffleDependencies(*

*rdd: RDD[\_]): HashSet[ShuffleDependency[\_, \_, \_]] = {*

*val parents = new HashSet[ShuffleDependency[\_, \_, \_]]*

*val visited = new HashSet[RDD[\_]]*

*val waitingForVisit = new ArrayStack[RDD[\_]]*

*waitingForVisit.push(rdd)*

*while (waitingForVisit.nonEmpty) {*

*val toVisit = waitingForVisit.pop()*

*if (!visited(toVisit)) {*

*visited += toVisit*

*toVisit.dependencies.foreach {*

*case shuffleDep: ShuffleDependency[\_, \_, \_] =>*

*parents += shuffleDep*

*case dependency =>*

*waitingForVisit.push(dependency.rdd)*

*}*

*}*

*}*

*parents*

*}*

获取RDD的所有ShuffleDependencies，然后根据ShuffleDependencies，调用getOrCreateShuffleMapStage找到所有的ShuffleStage：

*private def getOrCreateShuffleMapStage(*

*shuffleDep: ShuffleDependency[\_, \_, \_],*

*firstJobId: Int): ShuffleMapStage = {*

*shuffleIdToMapStage.get(shuffleDep.shuffleId) match {*

*case Some(stage) =>*

*stage*

*case None =>*

*getMissingAncestorShuffleDependencies(shuffleDep.rdd).foreach { dep =>*

*if (!shuffleIdToMapStage.contains(dep.shuffleId)) {*

*createShuffleMapStage(dep, firstJobId)*

*}*

*}*

*createShuffleMapStage(shuffleDep, firstJobId)*

*}*

*}*

获取到ShuffleDependecy后，调用createShuffleMapStage创建ShuffleMapStage：

*def createShuffleMapStage(shuffleDep: ShuffleDependency[\_, \_, \_], jobId: Int): ShuffleMapStage = {*

*val rdd = shuffleDep.rdd*

*val numTasks = rdd.partitions.length*

*val parents = getOrCreateParentStages(rdd, jobId)*

*val id = nextStageId.getAndIncrement()*

*val stage = new ShuffleMapStage(*

*id, rdd, numTasks, parents, jobId, rdd.creationSite, shuffleDep, mapOutputTracker)*

*stageIdToStage(id) = stage*

*shuffleIdToMapStage(shuffleDep.shuffleId) = stage*

*updateJobIdStageIdMaps(jobId, stage)*

*if (!mapOutputTracker.containsShuffle(shuffleDep.shuffleId)) {*

*mapOutputTracker.registerShuffle(shuffleDep.shuffleId, rdd.partitions.length)*

*}*

*stage*

*}*

当所有的调度阶段划分以后，这些调度节点建立了依赖关系。该依赖关系通过调度Stage其中的属性parents:List[Stage]来定义，通过该属性可以获取当前阶段所有祖先阶段，可以根据这些信息按提交调度阶段进行。

下图为Stage划分的例子，有7个RDD，分布式rddA-rddG，之间有5个操作，划分调度的详细步骤如下：



1. 在SparkContext中提交运行时，会调用DAGScheduler的handleJobSubmitted进行处理，在该方法中会找到最后一个RDD(即rddG)，并调用getParentStages方法
2. 在getParentStages方法判断rddG的依赖树是否存在Shuffle操作，在该例子中发现join操作为Shuffle操作，即获取进行该操作的RDD为rddB和rddF
3. 使用getAncestorShuffleDepenencies方法，从rddB向前遍历，发现该依赖分支上没有宽依赖，调用newOrUsedShuffleStage方法生成调度阶段ShuffleMapStage0
4. 使用getAncestorShuffleDependecies方法，从rddF向前遍历，寻找该依赖分支存在宽依赖操作groupBy，以此分界划分rddD和rddE为ShuffleMapStage1，rddE和rddF为ShuffleMapStage2
5. 最后生成rddG的ResultStage3，在该划分调度阶段中，共划分4个调度阶段，分别为ShuffleMapStage0-3

# 4.提交调度Stage

在DAGScheduler的handleJobSubmitted方法中，生成finalStage的同时建立所有调度阶段的依赖关系，然后通过finalStage生成一个Job实例，在该作业实例中按照顺序提交调度阶段进行执行，在执行过程中通过监听总线获取作业及Stage阶段执行情况。

在Job提交调度Stage开始时，在submitStage方法中调用getMissingParentStages方法获取finalStage父调度阶段，如果不存在父调度阶段，则使用submitMissingTasks方法提交执行；如果存在父调度Stage，则把该调度Stage放到waitingStage列表，同时递归调用submitStage。通过该算法把存在父调度Stage的等待调度阶段放入waitingStages列表，不存在父调度阶段的调度阶段作为作业运行的入口，代码如下：

*private def submitStage(stage: Stage) {*

*val jobId = activeJobForStage(stage)*

*if (jobId.isDefined) {*

*logDebug("submitStage(" + stage + ")")*

*if (!waitingStages(stage) && !runningStages(stage) && !failedStages(stage)) {*

*val missing = getMissingParentStages(stage).sortBy(\_.id)*

*logDebug("missing: " + missing)*

*if (missing.isEmpty) {*

*logInfo("Submitting " + stage + " (" + stage.rdd + "), which has no missing parents")*

*submitMissingTasks(stage, jobId.get)*

*} else {*

*for (parent <- missing) {*

*submitStage(parent)*

*}*

*waitingStages += stage*

*}*

*}*

*} else {*

*abortStage(stage, "No active job for stage " + stage.id, None)*

*}*

*}*

当入口的调度阶段运行完成后相继提交后续调度阶段，在调度前先判断该调度节点所依赖的父调度阶段的结果是否可用（即运行是否成功）。如果结果可用，则提交该调度节点；如果存在不可用的情况，则尝试提交结果不可用的父调度阶段。对于调度阶段是否可用的判断是在ShuffleMapTask完成时进行，DAGScheduler会检查调度阶段所有任务是否都完成。如果存在执行失败的任务，则重新提交该调度阶段；如果所有任务都完成，则扫描等待运行调度阶段列表，检查它们的父调度阶段是否存在未完成，如果不存在则表明该调度阶段准备就绪，生成实例并提交运行。

代码参见DAGScheduler的handleTaskCompletion方法，该方法在Executor.run任务执行完成时发送消息，通知DAGScheduler调度器更新状态，如下所示：

*case smt: ShuffleMapTask =>*

*val shuffleStage = stage.asInstanceOf[ShuffleMapStage]*

*val status = event.result.asInstanceOf[MapStatus]*

*val execId = status.location.executorId*

*logDebug("ShuffleMapTask finished on " + execId)*

*if (stageIdToStage(task.stageId).latestInfo.attemptId == task.stageAttemptId) {*

*shuffleStage.pendingPartitions -= task.partitionId*

*}*

*if (failedEpoch.contains(execId) && smt.epoch <= failedEpoch(execId)) {*

*logInfo(s"Ignoring possibly bogus $smt completion from executor $execId")*

*} else {*

*mapOutputTracker.registerMapOutput(*

*shuffleStage.pendingPartitions -= task.partitionId*

*}*

*if (runningStages.contains(shuffleStage) && shuffleStage.pendingPartitions.isEmpty) {*

*markStageAsFinished(shuffleStage)*

*mapOutputTracker.incrementEpoch()*

*clearCacheLocs()*

*if (!shuffleStage.isAvailable) {*

*submitStage(shuffleStage)*

*} else {*

*// Mark any map-stage jobs waiting on this stage as finished*

*if (shuffleStage.mapStageJobs.nonEmpty) {*

*val stats = mapOutputTracker.getStatistics(shuffleStage.shuffleDep)*

*for (job <- shuffleStage.mapStageJobs) {*

*markMapStageJobAsFinished(job, stats)*

*}*

*}*

*submitWaitingChildStages(shuffleStage)*

*}*

*}*

*}*

上一例子上对作业划分调度阶段，在本节中将按顺序提交调度Stage运行，具体步骤如下：



提交调度Stage运行顺序

1. 在handleJobSubmitted方法中获取final Stage ResultStage3，通过submitStage方法提交运行该调度Stage
2. 在submitStage方法中，先创建作业实例，然后判断该调度阶段是否存在父调度阶段，由于ResultStage3有两个父调度阶段ShuffleMapStage0和ShuffleMapStage2，所以并不能立即提交给调度阶段运行，把ResultStage3加入到waitingStages中
3. 递归调用submitStage方法可以知道ShuffleMapStage0不存在父调度阶段，而ShuffleMapStage2存在父调度阶段ShuffleMapStage1，这样ShuffleMapStage2加入到WaitingStages，而ShuffleMapStage1和ShuffleMapStage0两个调度阶段作为第一次调度使用submitMissingTasks方法提交运行
4. Executor任务执行完成时发送消息，通知DAGScheduler等调度器更新状态时，检查调度阶段运行情况，如果存在执行失败的任务，则重新提交该调度阶段；如果所有任务完成，则继续提交调度阶段运行。由于ResultStage3的父调度阶段没有，在第二次调度只提交ShuffleMapStage2运行
5. 当ShuffleMapStage2运行完毕后，此时ResultStage3的父调度阶段全部完成，提交该调度Stage运行

# 5.提交任务

当调度阶段提交运行后，在DAGScheduler的submitMissingTasks方法中，会根据调度阶段的Partition个数拆分对应个数的任务，这些任务组成一个任务集提交到TaskScheduler进行处理。对于ResultStage(作用中的final stage)生成ResultTask，对于ShuffleMapStage生成ShuffleMapTask。对于每一个任务集包含了对应调度阶段的所有任务，这些任务处理逻辑完全一样，不同的是对应处理的数据，而这些数据是其对应的数据分片（Partition）。DAGScheduler的submitMissingTask方法的具体代码如下：

*private def submitMissingTasks(stage: Stage, jobId: Int) {*

*......*

*val tasks: Seq[Task[\_]] = try {*

*val serializedTaskMetrics = closureSerializer.serialize(stage.latestInfo.taskMetrics).array()*

*stage match {*

*case stage: ShuffleMapStage =>*

*stage.pendingPartitions.clear()*

*partitionsToCompute.map { id =>*

*val locs = taskIdToLocations(id)*

*val part = stage.rdd.partitions(id)*

*stage.pendingPartitions += id*

*new ShuffleMapTask(stage.id, stage.latestInfo.attemptId,*

*taskBinary, part, locs, properties, serializedTaskMetrics, Option(jobId),*

*Option(sc.applicationId), sc.applicationAttemptId)*

*}*

*case stage: ResultStage =>*

*partitionsToCompute.map { id =>*

*val p: Int = stage.partitions(id)*

*val part = stage.rdd.partitions(p)*

*val locs = taskIdToLocations(id)*

*new ResultTask(stage.id, stage.latestInfo.attemptId,*

*taskBinary, part, locs, id, properties, serializedTaskMetrics,*

*Option(jobId), Option(sc.applicationId), sc.applicationAttemptId)*

*}*

*}*

*} catch {}*

*if (tasks.size > 0) {*

*taskScheduler.submitTasks(new TaskSet(*

*tasks.toArray, stage.id, stage.latestInfo.attemptId, jobId, properties))*

*} else {*

*markStageAsFinished(stage, None)*

*submitWaitingChildStages(stage)*

*}*

*}*

当TaskScheduler收到发送过来的任务集时，在submitTasks方法中（在TaskSchedulerImpl类中进行实现）构建一个TaskSetManager的实例，用于管理这个任务集的生命周期，而TaskSetManager会加入系统的调度池中，根据系统设置的调度算法进行调度，TaskSchedulerImpl#submitTasks方法代码如下：

*override def submitTasks(taskSet: TaskSet) {*

*val tasks = taskSet.tasks*

*this.synchronized {*

*val manager = createTaskSetManager(taskSet, maxTaskFailures)*

*val stage = taskSet.stageId*

*val stageTaskSets =*

*taskSetsByStageIdAndAttempt.getOrElseUpdate(stage, new HashMap[Int, TaskSetManager])*

*stageTaskSets(taskSet.stageAttemptId) = manager*

*val conflictingTaskSet = stageTaskSets.exists { case (\_, ts) =>*

*ts.taskSet != taskSet && !ts.isZombie*

*}*

*......*

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

*.....*

*backend.reviveOffers()*

*}*

SparkDeploySchedulerBackend的receiveOffers方法是继承于父类

CoarseGrainedSchedulerBackend，该方法会向DriverEndpoint终点发送消息，调用其makeOffers方法。在该方法中会先获取集群中可用的Executor，然后发送到TaskSchedulerImpl中对任务集的任务分配运行资源，最后提交到launchTasks方法中，其中CoarseGrainedSchedul

backEnd.makeOffers代码如下：

*private def makeOffers() {*

*val taskDescs = CoarseGrainedSchedulerBackend.this.synchronized {*

*val activeExecutors = executorDataMap.filterKeys(executorIsAlive)*

*val workOffers = activeExecutors.map {*

*case (id, executorData) =>*

*new WorkerOffer(id, executorData.executorHost, executorData.freeCores)*

*}.toIndexedSeq*

*scheduler.resourceOffers(workOffers)*

*}*

*if (!taskDescs.isEmpty) {*

*launchTasks(taskDescs)*

*}*

*}*

在TaskSchedulerImpl的resourceOffers方法中进行资源分配，在分配的过程中会根据调度策略对TaskSetManager进行排序，然后依次对这些TaskSetManager按照就近原则进行分配资源，顺序依次为：PROCESS\_LOCAL,NODE\_LOCAL,NO\_PREF,RACK\_LOCAL和ANY，具体的实现代码：

*def resourceOffers(offers: IndexedSeq[WorkerOffer]): Seq[Seq[TaskDescription]] = synchronized {*

*var newExecAvail = false*

*for (o <- offers) {*

*if (!hostToExecutors.contains(o.host)) {*

*hostToExecutors(o.host) = new HashSet[String]()*

*}*

*if (!executorIdToRunningTaskIds.contains(o.executorId)) {*

*hostToExecutors(o.host) += o.executorId*

*executorAdded(o.executorId, o.host)*

*executorIdToHost(o.executorId) = o.host*

*executorIdToRunningTaskIds(o.executorId) = HashSet[Long]()*

*newExecAvail = true*

*}*

*for (rack <- getRackForHost(o.host)) {*

*hostsByRack.getOrElseUpdate(rack, new HashSet[String]()) += o.host*

*}*

*}*

*blacklistTrackerOpt.foreach(\_.applyBlacklistTimeout())*

*val filteredOffers = blacklistTrackerOpt.map { blacklistTracker =>*

*offers.filter { offer =>*

*!blacklistTracker.isNodeBlacklisted(offer.host) &&*

*!blacklistTracker.isExecutorBlacklisted(offer.executorId)*

*}*

*}.getOrElse(offers)*

*val shuffledOffers = shuffleOffers(filteredOffers)*

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

*val tasks = shuffledOffers.map(o => new ArrayBuffer[TaskDescription](o.cores / CPUS\_PER\_TASK))*

*val availableCpus = shuffledOffers.map(o => o.cores).toArray*

*val sortedTaskSets = rootPool.getSortedTaskSetQueue*

*for (taskSet <- sortedTaskSets) {*

*logDebug("parentName: %s, name: %s, runningTasks: %s".format(*

*taskSet.parent.name, taskSet.name, taskSet.runningTasks))*

*if (newExecAvail) {*

*taskSet.executorAdded()*

*}*

*}*

*for (taskSet <- sortedTaskSets) {*

*var launchedAnyTask = false*

*var launchedTaskAtCurrentMaxLocality = false*

*for (currentMaxLocality <- taskSet.myLocalityLevels) {*

*do {*

*launchedTaskAtCurrentMaxLocality = resourceOfferSingleTaskSet(*

*taskSet, currentMaxLocality, shuffledOffers, availableCpus, tasks)*

*launchedAnyTask |= launchedTaskAtCurrentMaxLocality*

*} while (launchedTaskAtCurrentMaxLocality)*

*}*

*if (!launchedAnyTask) {*

*taskSet.abortIfCompletelyBlacklisted(hostToExecutors)*

*}*

*}*

*if (tasks.size > 0) {*

*hasLaunchedTask = true*

*}*

*return tasks*

*}*

分配好的资源的任务提交到CoarseGrainedSchedulerBackend的launchTask方法中，在该方法中会把任务一个个发送到Worker节点上的CoarseGrainedExecutorBackend，然后通过其内部的Executor来执行任务，具体的代码如下：

*private def launchTasks(tasks: Seq[Seq[TaskDescription]]) {*

*for (task <- tasks.flatten) {*

*val serializedTask = TaskDescription.encode(task)*

*if (serializedTask.limit >= maxRpcMessageSize) {*

*scheduler.taskIdToTaskSetManager.get(task.taskId).foreach { taskSetMgr =>*

*try {*

*var msg = "Serialized task %s:%d was %d bytes, which exceeds max allowed: " +*

*"spark.rpc.message.maxSize (%d bytes). Consider increasing " +*

*"spark.rpc.message.maxSize or using broadcast variables for large values."*

*msg = msg.format(task.taskId, task.index, serializedTask.limit, maxRpcMessageSize)*

*taskSetMgr.abort(msg)*

*} catch {*

*case e: Exception => logError("Exception in error callback", e)*

*}*

*}*

*}*

*else {*

*val executorData = executorDataMap(task.executorId)*

*executorData.freeCores -= scheduler.CPUS\_PER\_TASK*

*logDebug(s"Launching task ${task.taskId} on executor id: ${task.executorId} hostname: " +*

*s"${executorData.executorHost}.")*

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

*}*

*}*

*}*

前一节示例的顺序提交调度阶段，拆分成任务集，然后对这些任务分配资源，并提交Executor运行，具体步骤：



1. 在提交调度阶段中，第一次调度是ShuffleMapStage0和ShuffleMapStage1，这两个调度阶段在DAGScheduler的submitMissingTasks方法中，会根据Partition个数拆分任务。由于这两个调度阶段分布有两个分片，ShuffleMapStage0分别拆分成ShuffleMapStage(0,0)和ShuffleMapStage(0,1)两个任务，这两个任务集TaskSet0，而ShuffleMapStage1分别拆分成ShuffleMapStage(1,0)和ShuffleMapStage(1,1)两个任务，这两个任务组成任务集TaskSet1。
2. TaskScheduler收到发送过来的任务集TaskSet0和TaskSet1，在submitTasks方法中分别构建TaskSetManager的实例TaskSetManager0和TaskSetManager1进行管理，这两个TaskSetManager放到系统的调度池中，根据系统设置的调度算法进行调度，当前仅有FIFOSchedulableBuilder和FAIRSchedulabeBuilder。
3. 在TaskSchedulerImpl的resouceOffers方法中按照就近原则进行资源分配，到该步骤每个任务均分配运行代码、数据分片和处理资源等，使用CoarseGrainedSchedulerBackend的launchTasks方法把任务发送到Worker节点上的CoarseGrainedExecutorBackend，调用其Executor来执行任务，executor.launchTask(taskDesc)。
4. 当ShuffleMapStage0和ShuffleMapStage1执行完毕后，随后的ShuffleMapStage2和ResultStage3按按照步骤1~3运行，不同的是，ResultStage3生成的任务类型是ResultTask

# 6.执行任务

当CoarseGrainedExecutorBackend接收到LaunchTask消息时，会调用Executor的launchTask方法进行处理，先初始化一个TaskRunner来封装任务，它用于管理任务运行时的细节，再把TaskRunner对象放入到ThreadPool中去执行。

在TaskRunner的run方法里，首先会对发送过来的Task本身及它所依赖的Jar等文件的反序列，然后对反序列化的任务调用Task的runTask方法。由于Task本身是一个抽象类，具体的runTask方法是由它的两个子类ShuffleMapTask和ResultTask来实现，TaskRunner的

Run方法，代码如下：

*override def run(): Unit = {*

*//taskMemoryManager用于任务运行期间的内存管理*

*val taskMemoryManager = new TaskMemoryManager(env.memoryManager, taskId)*

*......*

*//向Driver终端点发送任务运行开始消息*

*execBackend.statusUpdate(taskId, TaskState.RUNNING, EMPTY\_BYTE\_BUFFER)*

*.....*

*try {*

*//对任务运行时所需要的文，jar包和代码进行反序列化*

*Executor.taskDeserializationProps.set(taskDescription.properties)*

*updateDependencies(taskDescription.addedFiles, taskDescription.addedJars)*

*task = ser.deserialize[Task[Any]](*

*taskDescription.serializedTask, Thread.currentThread.getContextClassLoader)*

*task.localProperties = taskDescription.properties*

*task.setTaskMemoryManager(taskMemoryManager)*

*if (!isLocal) {*

*env.mapOutputTracker.asInstanceOf[MapOutputTrackerWorker].updateEpoch(task.epoch)*

*}*

*//调用Task的run方法，由于Task是抽象类，具体的run方法由其子类ShuffleMapTask和ResultTask定义*

*val value = try {*

*val res = task.run(*

*taskAttemptId = taskId,*

*attemptNumber = taskDescription.attemptNumber,*

*metricsSystem = env.metricsSystem)*

*threwException = false*

*res*

对于ShuffleMapTask而言，它的计算结果会写到BlockManager之中，最终返回给DAGScheduler的一个MapStatus对象。该对象中管理了ShuffleMapTask的运算结果存储到BlockManager里的相关存储信息，而不是计算结果本身，这些存储信息会将成为下一阶段的任务需要获得输入数据时的一句。其中ShuffleMapTask.runTask代码如下：

*override def runTask(context: TaskContext): MapStatus = {*

*// Deserialize the RDD using the broadcast variable.*

*val threadMXBean = ManagementFactory.getThreadMXBean*

*val deserializeStartTime = System.currentTimeMillis()*

*val deserializeStartCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {*

*threadMXBean.getCurrentThreadCpuTime*

*} else 0L*

*val ser = SparkEnv.get.closureSerializer.newInstance()*

*val (rdd, dep) = ser.deserialize[(RDD[\_], ShuffleDependency[\_, \_, \_])](*

*ByteBuffer.wrap(taskBinary.value), Thread.currentThread.getContextClassLoader)*

*\_executorDeserializeTime = System.currentTimeMillis() - deserializeStartTime*

*\_executorDeserializeCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {*

*threadMXBean.getCurrentThreadCpuTime - deserializeStartCpuTime*

*} else 0L*

*var writer: ShuffleWriter[Any, Any] = null*

*try {*

*val manager = SparkEnv.get.shuffleManager*

*writer = manager.getWriter[Any, Any](dep.shuffleHandle, partitionId, context)*

*writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[\_ <: Product2[Any, Any]]])*

*writer.stop(success = true).get*

*} catch {......}*

*}*

*}*

对于ResultTask的runTask而言，它的最终返回的是func函数的计算结果：

*override def runTask(context: TaskContext): U = {*

*// Deserialize the RDD and the func using the broadcast variables.*

*val threadMXBean = ManagementFactory.getThreadMXBean*

*val deserializeStartTime = System.currentTimeMillis()*

*val deserializeStartCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {*

*threadMXBean.getCurrentThreadCpuTime*

*} else 0L*

*val ser = SparkEnv.get.closureSerializer.newInstance()*

*val (rdd, func) = ser.deserialize[(RDD[T], (TaskContext, Iterator[T]) => U)](*

*ByteBuffer.wrap(taskBinary.value), Thread.currentThread.getContextClassLoader)*

*\_executorDeserializeTime = System.currentTimeMillis() - deserializeStartTime*

*\_executorDeserializeCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {*

*threadMXBean.getCurrentThreadCpuTime - deserializeStartCpuTime*

*} else 0L*

*func(context, rdd.iterator(partition, context))*

*}*

# 7.获取执行结果

对于Executor的计算结果，会根据结果的大小有不同的策略

1. 生成结果大于spark.driver.maxResultSize，例如]，结果直接丢弃
2. 生成结果在]，结果大于128MB-200KB时，会把该结果以taskId为编号存入BlockManager中，然后把该编号通过Netty发送给Driver终端点，该阈值是Netty框架传输的最大值spark.akka.frameSize，默认128MB和Netty预留空间reservedSizeBytes(200KB)差值
3. 生成结果大小在（120MB-200KB,0），通过Netty直接发送到Driver终端点

具体任务执行在TaskRunner的run方法后半部分实现，代码如下：

*private def run(): Unit = {*

*……*

*val directResult = new DirectTaskResult(valueBytes, accumUpdates)*

*val serializedDirectResult = ser.serialize(directResult)*

*val resultSize = serializedDirectResult.limit*

*val serializedResult: ByteBuffer = {*

*if (maxResultSize > 0 && resultSize > maxResultSize) {*

*//生成结果序列化大于最大值，直接丢掉*

*ser.serialize(new IndirectTaskResult[Any](TaskResultBlockId(taskId), resultSize))*

*} else if (resultSize > maxDirectResultSize) {*

*//序列化结果在[1G，128MB-200KB]，存到BlockManager中，然后把编号通过Netty发送给Driver终端点*

*val blockId = TaskResultBlockId(taskId)*

*env.blockManager.putBytes(*

*blockId,*

*new ChunkedByteBuffer(serializedDirectResult.duplicate()),*

*StorageLevel.MEMORY\_AND\_DISK\_SER)*

*ser.serialize(new IndirectTaskResult[Any](blockId, resultSize))*

*} else {*

*//通过Netty直接发送给Driver终端点*

*serializedDirectResult*

*}*

*}*

*execBackend.statusUpdate(taskId, TaskState.FINISHED, serializedResult)*

*}*

任务执行完毕后,TaskRunner将任务的执行结果发送给DriverEndpoint终端点。该终端点会转给TaskSchedulerImpl的statusUpdate方法进行处理，在该方法中对于不同的任务状态有不同的处理：

1. 如果类型是TaskState.FINISHED，那么调用TaskResultGetter的enqueueSuccessfulTask方法进行处理，enqueueSuccessfulTask的方法逻辑比较简单，就是如果是IndirectTaskResult，那么需要通过blockid来拉取结果：

sparkEnv.blockManager.getRemoteBytes(blockId)，如果是DirectTaskResult，那么结果就无需远程获取

1. 如果类型是TaskState.FAILED/ KILLED或者LOST，调用TaskResultGetter的

enqueueFailedTask进行处理，对于TaskState.LOST，还需要将其所在的Executor标记failed，并且根据更新后的Executor重新调度

TaskSchedulerImpl的handleSuccesssfulTask方法中连续调用，最终调用DAGScheduler的handleTaskCompletion方法：



Spart任务运行类调用关系图

如果任务是ShuffleMapTask，需要将结果通过某种机制告诉下游的调度阶段，以便其可以作为后续调度阶段的输入，使用的结果为MapStatus，其序列化后存入DirectTaskResult或者IndirectTaskResult中。而DAGScheduler的handleTaskCompletion方法获取这个结果，并把这个MapStatus注册到MapOutputTrackerMaster中，从而完成ShuffleMapTask处理：

*case smt: ShuffleMapTask =>*

*val shuffleStage = stage.asInstanceOf[ShuffleMapStage]*

*val status = event.result.asInstanceOf[MapStatus]*

*val execId = status.location.executorId*

*logDebug("ShuffleMapTask finished on " + execId)*

*if (stageIdToStage(task.stageId).latestInfo.attemptId == task.stageAttemptId) {*

*shuffleStage.pendingPartitions -= task.partitionId*

*}*

*if (failedEpoch.contains(execId) && smt.epoch <= failedEpoch(execId)) {*

*logInfo(s"Ignoring possibly bogus $smt completion from executor $execId")*

*} else {*

*mapOutputTracker.registerMapOutput(*

*shuffleStage.shuffleDep.shuffleId, smt.partitionId, status)*

*shuffleStage.pendingPartitions -= task.partitionId*

*}*

如果是ResultTask，判断作业是否完成，如果完成，则标记作业已经完成，释放作业依赖的资源并发送消息给系统监听中心告知作业执行完毕：

*case rt: ResultTask[\_, \_] =>*

*val resultStage = stage.asInstanceOf[ResultStage]*

*resultStage.activeJob match {*

*case Some(job) =>*

*if (!job.finished(rt.outputId)) {*

*job.finished(rt.outputId) = true*

*job.numFinished += 1*

*// If the whole job has finished, remove it*

*if (job.numFinished == job.numPartitions) {*

*markStageAsFinished(resultStage)*

*cleanupStateForJobAndIndependentStages(job)*

*listenerBus.post(*

*SparkListenerJobEnd(job.jobId, clock.getTimeMillis(), JobSucceeded))*

*}*

*try {*

*job.listener.taskSucceeded(rt.outputId, event.result)*

*} catch {}*

*case None =>*

*logInfo("Ignoring result from " + rt + " because its job has finished")*

*}*

Dependency

Task

RDD

Stage划分实例：<http://www.cnblogs.com/bonelee/p/6039469.html>

<https://catalog.data.gov/dataset/baby-names-from-social-security-card-applications-national-level-data>

<http://blog.csdn.net/pmp4561705/article/details/53212196>

http://www.cnblogs.com/arachis/p/Spark\_TaskScheduler\_1.html