**深入理解Spark 2.1 Core （二）：DAG调度器的原理与源码分析**

原创 2016年12月28日 22:11:02

* 上一篇[《深入理解Spark 2.0 （一）：RDD实现及源码分析 》](http://blog.csdn.net/u011239443/article/details/53894611)的5.2 Spark任务调度器我们省略过去了，这篇我们就来讲讲Spark的调度器。

**概述**

上一篇《深入理解Spark（一）：RDD实现及源码分析 》提到：

定义RDD之后，程序员就可以在动作（注：即action操作）中使用RDD了。动作是向应用程序返回值，或向存储系统导出数据的那些操作，例如，count（返回RDD中的元素个数），collect（返回元素本身），save（将RDD输出到存储系统）。在Spark中，只有在动作第一次使用RDD时，才会计算RDD（即延迟计算）。这样在构建RDD的时候，运行时通过管道的方式传输多个转换。

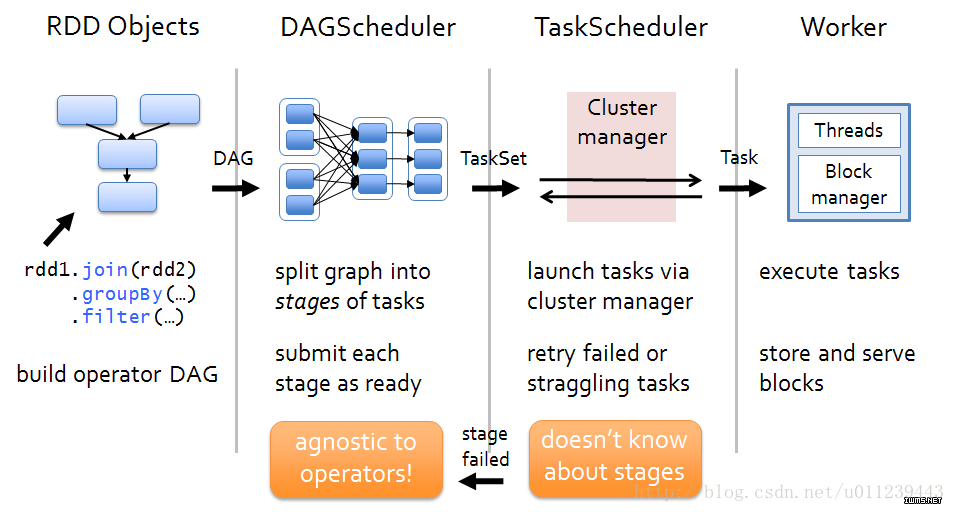
一次action操作会触发RDD的延迟计算，我们把这样的一次计算称作一个Job。我们还提到了窄依赖和宽依赖的概念：

窄依赖指的是：每个parent RDD 的 partition 最多被 child RDD的一个partition使用   
宽依赖指的是：每个parent RDD 的 partition 被多个 child RDD的partition使用

窄依赖每个child RDD 的partition的生成操作都是可以并行的，而宽依赖则需要所有的parent partition shuffle结果得到后再进行。

由于在RDD的一系类转换中，若其中一些连续的转换都是窄依赖，那么它们是可以并行的，而有宽依赖则不行。所有，Spark将宽依赖为划分界限，将Job换分为多个Stage。而一个Stage里面的转换任务，我们可以把它抽象成TaskSet。一个TaskSet中有很多个Task，它们的转换操作都是相同的，不同只是操作的对象是对数据集中的不同子数据集。

接下来，Spark就可以提交这些任务了。但是，如何对这些任务进行调度和资源分配呢？如何通知worker去执行这些任务呢？接下来，我们会一一讲解。



根据以上两个阶段，我们会来详细介绍两个Scheduler，一个是DAGScheduler，另外一个是TaskScheduler。

我们先来看一来在SparkContext中是如何创建它们的：

val (sched, ts) = SparkContext.createTaskScheduler(this, master, deployMode)

\_schedulerBackend = sched

\_taskScheduler = ts

\_dagScheduler = new DAGScheduler(this)

可以看到，我们是先用函数createTaskScheduler创建了taskScheduler，再new了一个DAGScheduler。这个顺序可以改变吗？答案是否定的，我们看下DAGScheduler类就知道了：

class DAGScheduler(

private[scheduler] val sc: SparkContext,

private[scheduler] val taskScheduler: TaskScheduler,

listenerBus: LiveListenerBus,

mapOutputTracker: MapOutputTrackerMaster,

blockManagerMaster: BlockManagerMaster,

env: SparkEnv,

clock: Clock = new SystemClock())

extends Logging {

def this(sc: SparkContext, taskScheduler: TaskScheduler) = {

this(

sc,

taskScheduler,

sc.listenerBus,

sc.env.mapOutputTracker.asInstanceOf[MapOutputTrackerMaster],

sc.env.blockManager.master,

sc.env)

}

def this(sc: SparkContext) = this(sc, sc.taskScheduler)

\*\*\*

}

SparkContext中创建的TaskScheduler，会传入DAGScheduler赋值给它的成员变量，再DAG阶段结束后，使用它进行下一步对任务调度等的操作。

**提交Job**

调用栈如下：

* rdd.count
  + SparkContext.runJob
    - DAGScheduler.runJob
      * DAGScheduler.submitJob
        + DAGSchedulerEventProcessLoop.doOnReceive

DAGScheduler.handleJobSubmitted

接下来，我们来逐个深入：

**rdd.count**

RDD的一些action操作都会触发SparkContext的runJob函数，如count()

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

**SparkContext.runJob**

SparkContext的runJob会触发 DAGScheduler的runJob：

def runJob[T, U: ClassTag](

rdd: RDD[T],

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

partitions: Seq[Int],

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

if (stopped.get()) {

throw new IllegalStateException("SparkContext has been shutdown")

}

val callSite = getCallSite

val cleanedFunc = clean(func)

logInfo("Starting job: " + callSite.shortForm)

if (conf.getBoolean("spark.logLineage", false)) {

logInfo("RDD's recursive dependencies:\n" + rdd.toDebugString)

}

dagScheduler.runJob(rdd, cleanedFunc, partitions, callSite, resultHandler, localProperties.get)

progressBar.foreach(\_.finishAll())

rdd.doCheckpoint()

}

这里的rdd.doCheckpoint()并不是对自己Checkpoint，而是递归的回溯parent rdd 检查checkpointData是否被定义了，若定义了就将该rdd Checkpoint：

private[spark] def doCheckpoint(): Unit = {

RDDOperationScope.withScope(sc, "checkpoint", allowNesting = false, ignoreParent = true) {

if (!doCheckpointCalled) {

doCheckpointCalled = true

if (checkpointData.isDefined) {

if (checkpointAllMarkedAncestors) {

//若想要把checkpointData定义过的RDD的parents也进行checkpoint的话，

//那么我们需要先对parents checkpoint。

//这是因为，如果RDD把自己checkpoint了，

//那么它就将lineage中它的parents给切除了。

dependencies.foreach(\_.rdd.doCheckpoint())

}

checkpointData.get.checkpoint()

} else {

dependencies.foreach(\_.rdd.doCheckpoint())

}

}

}

}

具体的checkpoint实现可见上一篇博文。

**DAGScheduler.runJob**

DAGScheduler的runJob会触发DAGScheduler的submitJob：

/\*\*

\* 参数介绍：

\* @param rdd： 执行任务的目标TDD

\* @param func： 在RDD的分区上所执行的函数

\* @param partitions： 需要执行的分区集合;有些job并不会对RDD的所有分区都进行计算的，比如说first()

\* @param callSite：用户程序的调用点

\* @param resultHandler：回调结果

\* @param properties：关于这个job的调度器特征，比如说公平调度的pool名字，这个会在后续讲到

\*/

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 start = System.nanoTime

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

\*\*\*

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))

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

exception.setStackTrace(exception.getStackTrace ++ callerStackTrace)

throw exception

}

}

**DAGScheduler.submitJob**

我们接下来看看submitJob里面做了什么：

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] = {

// 确认没在不存在的partition上执行任务

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)

}

//递增得到jobId

val jobId = nextJobId.getAndIncrement()

if (partitions.size == 0) {

//若Job没对任何一个partition执行任务，

//则立即返回

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

}

**DAGSchedulerEventProcessLoop.doOnReceive**

eventProcessLoop是一个DAGSchedulerEventProcessLoop类对象，即一个DAG调度事件处理的监听。eventProcessLoop中调用doOnReceive来进行监听

private def doOnReceive(event: DAGSchedulerEvent): Unit = event match {

//当事件为JobSubmitted时，

//会调用DAGScheduler.handleJobSubmitted

case JobSubmitted(jobId, rdd, func, partitions, callSite, listener, properties) =>

dagScheduler.handleJobSubmitted(jobId, rdd, func, partitions, callSite, listener, properties)

\*\*\*

}

**DAGScheduler.handleJobSubmitted**

自此Job的提交就完成了：

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 = newResultStage(finalRDD, func, partitions, jobId, callSite)

} catch {

case e: Exception =>

logWarning("Creating new stage failed due to exception - job: " + jobId, e)

listener.jobFailed(e)

return

}

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

clearCacheLocs()

logInfo("Got job %s (%s) with %d output partitions".format(

job.jobId, callSite.shortForm, partitions.length))

logInfo("Final stage: " + finalStage + " (" + finalStage.name + ")")

logInfo("Parents of final stage: " + finalStage.parents)

logInfo("Missing parents: " + getMissingParentStages(finalStage))

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))

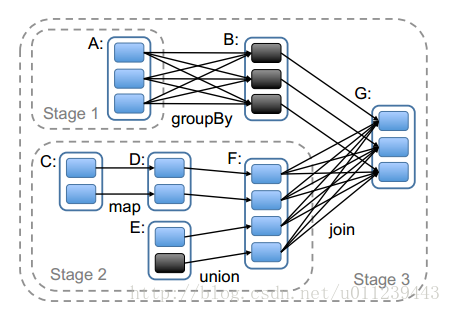
submitStage(finalStage)

submitWaitingStages()

}

接下来我们来看看handleJobSubmitted中的newResultStage，一个非常有趣的划分Stage过程。

**划分Stage**



如我们之前提到的：Spark将宽依赖为划分界限，将Job换分为多个Stage。调用栈为：

* DAGScheduler.newResultStage
  + DAGScheduler.getParentStagesAndId
    - DAGScheduler.getParentStages
      * DAGScheduler.getShuffleMapStage
        + DAGScheduler.getAncestorShuffleDependencies
        + DAGScheduler.newOrUsedShuffleStage

DAGScheduler.newShuffleMapStage

接下来，我们来逐个深入：

**DAGScheduler.newResultStage**

Spark的Stage调用是从最后一个RDD所在的Stage，ResultStage开始划分的，这里即为G所在的Stage。但是在生成这个Stage之前会生成它的parent Stage，就这样递归的把parent Stage都先生成了。

private def newResultStage(

rdd: RDD[\_],

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

partitions: Array[Int],

jobId: Int,

callSite: CallSite): ResultStage = {

val (parentStages: List[Stage], id: Int) = getParentStagesAndId(rdd, jobId)

val stage = new ResultStage(id, rdd, func, partitions, parentStages, jobId, callSite)

stageIdToStage(id) = stage

updateJobIdStageIdMaps(jobId, stage)

stage

}

**DAGScheduler.getParentStagesAndId**

getParentStagesAndId中得到了ParentStages以及其StageId：

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

val parentStages = getParentStages(rdd, firstJobId)

val id = nextStageId.getAndIncrement()

(parentStages, id)

}

**DAGScheduler.getParentStages**

我们再来深入看看getParentStages做了什么：

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

//将存储ParentStages

val parents = new HashSet[Stage]

//存储已将访问过了的RDD

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

// 存储需要被处理的RDD

val waitingForVisit = new Stack[RDD[\_]]

def visit(r: RDD[\_]) {

if (!visited(r)) {

//加入访问集合

visited += r

//遍历该RDD所有的依赖

for (dep <- r.dependencies) {

dep match {

//若是宽依赖则生成新的Stage

case shufDep: ShuffleDependency[\_, \_, \_] =>

parents += getShuffleMapStage(shufDep, firstJobId)

//若是窄依赖则加入Stack，等待处理

case \_ =>

waitingForVisit.push(dep.rdd)

}

}

}

}

//在Stack中加入最后一个RDD

waitingForVisit.push(rdd)

//广度优先遍历

while (waitingForVisit.nonEmpty) {

visit(waitingForVisit.pop())

}

//返回ParentStages List

parents.toList

}

其实getParentStages使用的就是广度优先遍历的算法，若知道这点也容易理解了。虽然现在Stage并没有生成，但是我们可以看到划分策略是：广度遍历方式的划分parent RDD 的Stage。

若parent RDD 和 child RDD 为窄依赖，则将parent RDD 纳入 child RDD 所在的Stage中。如图，B被纳入了Stage3中。

若parent RDD 和 child RDD 为宽依赖，则parent RDD将纳入一新的Stage中。如图，F被纳入了Stage2中。

**DAGScheduler.getShuffleMapStage**

下面我们来看下getShuffleMapStage是如何生成新的Stage的。   
首先shuffleToMapStage中保存了关于Stage的HashMap

private[scheduler] val shuffleToMapStage = new HashMap[Int, ShuffleMapStage]

* 1

getShuffleMapStage会先去根据shuffleId去查找shuffleToMapStage

private def getShuffleMapStage(

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

firstJobId: Int): ShuffleMapStage = {

shuffleToMapStage.get(shuffleDep.shuffleId) match {

//若找到则直接返回

case Some(stage) => stage

case None =>

// 检查这个Stage的Parent Stage是否生成

// 若没有，则生成它们

getAncestorShuffleDependencies(shuffleDep.rdd).foreach { dep =>

if (!shuffleToMapStage.contains(dep.shuffleId)) {

shuffleToMapStage(dep.shuffleId) = newOrUsedShuffleStage(dep, firstJobId)

}

}

// 生成新的Stage

val stage = newOrUsedShuffleStage(shuffleDep, firstJobId)

//将新的Stage 加入到 HashMap

shuffleToMapStage(shuffleDep.shuffleId) = stage

//返回新的Stage

stage

}

}

可以发现这部分的代码和上述的newResultStage部分很像，所以可以看成一种递归的方法。

**DAGScheduler.getAncestorShuffleDependencies**

我们再来看下getAncestorShuffleDependencies，可想而知，它应该会和newResultStage中的getParentStages会非常类似：

private def getAncestorShuffleDependencies(rdd: RDD[\_]): Stack[ShuffleDependency[\_, \_, \_]] = {

val parents = new Stack[ShuffleDependency[\_, \_, \_]]

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

val waitingForVisit = new Stack[RDD[\_]]

def visit(r: RDD[\_]) {

if (!visited(r)) {

visited += r

for (dep <- r.dependencies) {

dep match {

case shufDep: ShuffleDependency[\_, \_, \_] =>

if (!shuffleToMapStage.contains(shufDep.shuffleId)) {

parents.push(shufDep)

}

case \_ =>

}

waitingForVisit.push(dep.rdd)

}

}

}

waitingForVisit.push(rdd)

while (waitingForVisit.nonEmpty) {

visit(waitingForVisit.pop())

}

parents

}

可以看到的确和newResultStage中的getParentStages会非常类似，不同的是这里会先判断shuffleToMapStage是否存在这个Stage，不存在的话会push到parents这个Stack，最会返回给上述的getShuffleMapStage，调用newOrUsedShuffleStage生成新的Stage。

**DAGScheduler.newOrUsedShuffleStage**

那现在就来看newOrUsedShuffleStage是如何生成新的Stage的。   
首先ShuffleMapTask的计算结果（其实是计算结果数据所在的位置、大小等元数据信息）都会传给Driver的mapOutputTracker。所以需要先判断Stage是否已经被计算过：

private def newOrUsedShuffleStage(

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

firstJobId: Int): ShuffleMapStage = {

val rdd = shuffleDep.rdd

val numTasks = rdd.partitions.length

//生成新的Stage

val stage = newShuffleMapStage(rdd, numTasks, shuffleDep, firstJobId, rdd.creationSite)

//判断Stage是否已经被计算过

//若计算过，则把结果复制到新的stage

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

val serLocs = mapOutputTracker.getSerializedMapOutputStatuses(shuffleDep.shuffleId)

val locs = MapOutputTracker.deserializeMapStatuses(serLocs)

(0 until locs.length).foreach { i =>

if (locs(i) ne null) {

stage.addOutputLoc(i, locs(i))

}

}

} else {

logInfo("Registering RDD " + rdd.id + " (" + rdd.getCreationSite + ")")

//如果没计算过，就在注册mapOutputTracker Stage

//为存储元数据占位

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

}

stage

}

**DAGScheduler.newShuffleMapStage**

递归就发生在newShuffleMapStage，它的实现和最一开始的newResultStage类似，也是先getParentStagesAndId，然后生成一个ShuffleMapStage：

private def newShuffleMapStage(

rdd: RDD[\_],

numTasks: Int,

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

firstJobId: Int,

callSite: CallSite): ShuffleMapStage = {

val (parentStages: List[Stage], id: Int) = getParentStagesAndId(rdd, firstJobId)

val stage: ShuffleMapStage = new ShuffleMapStage(id, rdd, numTasks, parentStages,

firstJobId, callSite, shuffleDep)

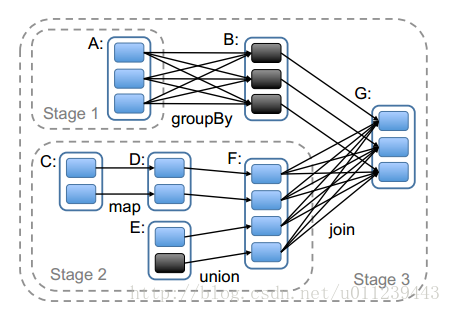
stageIdToStage(id) = stage

updateJobIdStageIdMaps(firstJobId, stage)

stage

}

**回顾**

到此，Stage划分过程就结束了。我们在根据一开始的图，举例回顾下：   


* 首先，我们想 newResultStage RDD\_G所在的Stage3
* 但在new Stage之前会调用getParentStagesAndId
* getParentStagesAndId中又会调用getParentStages，来广度优先的遍历RDD\_G所依赖的RDD。如果是窄依赖，就纳入G所在的Stage3，如RDD\_B就纳入了Stage3
* 若过是宽依赖，我们这里以RDD\_F为例（与RDD\_A处理过程相同）。我们就会调用getShuffleMapStage，来判断RDD\_F所在的Stage2是否已经生成了，如果生成了就直接返回。
* 若还没生成，我们先调用getAncestorShuffleDependencies。getAncestorShuffleDependencies类似于getParentStages，也是用广度优先的遍历RDD\_F所依赖的RDD。如果是窄依赖，如RDD\_C、RDD\_D和RDD\_E，都被纳入了F所在的Stage2。但是假设RDD\_E有个parent RDD ``RDD\_H，RDD\_H和RDD\_E之间是宽依赖，那么该怎么办呢？我们会先判断RDD\_H所在的Stage是否已经生成。若还没生成，我们把它put到一个parents Stack 中，最后返回。
* 对于那些返回的还没生成的Stage我们会调用newOrUsedShuffleStage
* newOrUsedShuffleStage会调用newShuffleMapStage，来生成新的Stage。而newShuffleMapStage的实现类似于newResultStage。这样我们就可以递归下去，使得每个Stage所依赖的Stage都已经生成了，再来生成这个的Stage。如这里，会将RDD\_H所在的Stage生成了，然后在再生成Stage2。
* newOrUsedShuffleStage生成新的Stage后，会判断Stage是否被计算过。若已经被计算过，就从mapOutPutTracker中复制计算结果。若没计算过，则向mapOutPutTracker注册占位。
* 最后，回到newResultStage中，new ResultStage，这里即生成了Stage3。至此，Stage划分过程就结束了。

**生成任务**

调用栈如下：

* DAGScheduler.handleJobSubmitted
  + DAGScheduler.submitStage
    - DAGScheduler.getMissingParentStages
    - DAGScheduler.submitMissingTasks

**DAGScheduler.handleJobSubmitted**

我们再回过头来看***“提交Job”***的最后一步handleJobSubmitted：

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 = newResultStage(finalRDD, func, partitions, jobId, callSite)

} catch {

case e: Exception =>

logWarning("Creating new stage failed due to exception - job: " + jobId, e)

listener.jobFailed(e)

return

}

\*\*\*

}

在***“划分Stage”***中我们已经深入的讲解了finalStage的生成：

finalStage = newResultStage(finalRDD, func, partitions, jobId, callSite)

接下来，我们继续往下看handleJobSubmitted的代码：

//生成新的job

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

clearCacheLocs()

logInfo("Got job %s (%s) with %d output partitions".format(

job.jobId, callSite.shortForm, partitions.length))

logInfo("Final stage: " + finalStage + " (" + finalStage.name + ")")

logInfo("Parents of final stage: " + finalStage.parents)

logInfo("Missing parents: " + getMissingParentStages(finalStage))

//得到job提交的时间

val jobSubmissionTime = clock.getTimeMillis()

//得到job id

jobIdToActiveJob(jobId) = job

//添加到activeJobs HashSet

activeJobs += job

//将finalStage甚至ActiveJob为该job

finalStage.setActiveJob(job)

//得到stage 的id 信息

val stageIds = jobIdToStageIds(jobId).toArray

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

//监听

listenerBus.post(

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

//提交

submitStage(finalStage)

//等待

submitWaitingStages()

**DAGScheduler.submitStage**

接下来我们来看Stage是如何提交的。我们需要找到哪些parent Stage缺失，然后我们先运行生成这些Stage。这是一个深度优先遍历的过程：

private def submitStage(stage: Stage) {

val jobId = activeJobForStage(stage)

if (jobId.isDefined) {

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

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

//得到缺失的Parent Stage

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

logDebug("missing: " + missing)

if (missing.isEmpty) {

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

//如果没有缺失的Parent Stage，

//那么代表着该Stage可以运行了

//submitMissingTasks会完成DAGScheduler最后的工作，

//向TaskScheduler 提交 Task

submitMissingTasks(stage, jobId.get)

} else {

//深度优先遍历

for (parent <- missing) {

submitStage(parent)

}

waitingStages += stage

}

}

} else {

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

}

}

**DAGScheduler.getMissingParentStages**

getMissingParentStages类似于getParentStages，也是使用广度优先遍历：

private def getMissingParentStages(stage: Stage): List[Stage] = {

val missing = new HashSet[Stage]

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

val waitingForVisit = new Stack[RDD[\_]]

def visit(rdd: RDD[\_]) {

if (!visited(rdd)) {

visited += rdd

val rddHasUncachedPartitions = getCacheLocs(rdd).contains(Nil)

if (rddHasUncachedPartitions) {

for (dep <- rdd.dependencies) {

dep match {

//若是宽依赖 并且 不可用 ，

//则加入 missing HashSet

case shufDep: ShuffleDependency[\_, \_, \_] =>

val mapStage = getShuffleMapStage(shufDep, stage.firstJobId)

if (!mapStage.isAvailable) {

missing += mapStage

}

//若是窄依赖

//则加入等待访问的 HashSet

case narrowDep: NarrowDependency[\_] =>

waitingForVisit.push(narrowDep.rdd)

}

}

}

}

}

waitingForVisit.push(stage.rdd)

while (waitingForVisit.nonEmpty) {

visit(waitingForVisit.pop())

}

missing.toList

}

**DAGScheduler.submitMissingTasks**

最后，我们来看下DAGScheduler最后的工作，提交Task：

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

logDebug("submitMissingTasks(" + stage + ")")

// pendingPartitions 是 HashSet[Int]

//存储待处理的Task

stage.pendingPartitions.clear()

// 找出还未就算的Partition

val partitionsToCompute: Seq[Int] = stage.findMissingPartitions()

//从一个ActiveJob中得到关于这个Stage的

//调度池，job组描述等信息

val properties = jobIdToActiveJob(jobId).properties

// runningStages 是 HashSet[Stage]

//将当前Stage加入到运行中Stage集合

runningStages += stage

stage match {

case s: ShuffleMapStage =>

outputCommitCoordinator.stageStart(stage = s.id, maxPartitionId = s.numPartitions - 1)

case s: ResultStage =>

outputCommitCoordinator.stageStart(

stage = s.id, maxPartitionId = s.rdd.partitions.length - 1)

}

val taskIdToLocations: Map[Int, Seq[TaskLocation]] = try {

stage match {

case s: ShuffleMapStage =>

partitionsToCompute.map { id => (id, getPreferredLocs(stage.rdd, id))}.toMap

case s: ResultStage =>

partitionsToCompute.map { id =>

val p = s.partitions(id)

(id, getPreferredLocs(stage.rdd, p))

}.toMap

}

} catch {

case NonFatal(e) =>

stage.makeNewStageAttempt(partitionsToCompute.size)

listenerBus.post(SparkListenerStageSubmitted(stage.latestInfo, properties))

abortStage(stage, s"Task creation failed: $e\n${Utils.exceptionString(e)}", Some(e))

runningStages -= stage

return

}

stage.makeNewStageAttempt(partitionsToCompute.size, taskIdToLocations.values.toSeq)

//向listenerBus发送SparkListenerStageSubmitted事件

listenerBus.post(SparkListenerStageSubmitted(stage.latestInfo, properties))

var taskBinary: Broadcast[Array[Byte]] = null

try {

//对于最后一个Stage的Task，

//序列化并广播(rdd, func)。

//若是其他的Stage的Task，

//序列化并广播(rdd, shuffleDep)

val taskBinaryBytes: Array[Byte] = stage match {

case stage: ShuffleMapStage =>

JavaUtils.bufferToArray(

closureSerializer.serialize((stage.rdd, stage.shuffleDep): AnyRef))

case stage: ResultStage =>

JavaUtils.bufferToArray(closureSerializer.serialize((stage.rdd, stage.func): AnyRef))

}

taskBinary = sc.broadcast(taskBinaryBytes)

} catch {

//若序列化失败，停止这个stage

case e: NotSerializableException =>

abortStage(stage, "Task not serializable: " + e.toString, Some(e))

runningStages -= stage

// 停止执行

return

case NonFatal(e) =>

abortStage(stage, s"Task serialization failed: $e\n${Utils.exceptionString(e)}", Some(e))

runningStages -= stage

return

}

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

//对于最后一个Stage的Task，

//则创建ResultTask。

//若是其他的Stage的Task，

//则创建ShuffleMapTask。

stage match {

case stage: ShuffleMapStage =>

partitionsToCompute.map { id =>

val locs = taskIdToLocations(id)

val part = stage.rdd.partitions(id)

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

taskBinary, part, locs, stage.latestInfo.taskMetrics, properties, 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, stage.latestInfo.taskMetrics,

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

}

}

} catch {

case NonFatal(e) =>

abortStage(stage, s"Task creation failed: $e\n${Utils.exceptionString(e)}", Some(e))

runningStages -= stage

return

}

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)

//创建TaskSet并提交

taskScheduler.submitTasks(new TaskSet(

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

stage.latestInfo.submissionTime = Some(clock.getTimeMillis())

} else {

markStageAsFinished(stage, None)

val debugString = stage match {

case stage: ShuffleMapStage =>

s"Stage ${stage} is actually done; " +

s"(available: ${stage.isAvailable}," +

s"available outputs: ${stage.numAvailableOutputs}," +

s"partitions: ${stage.numPartitions})"

case stage : ResultStage =>

s"Stage ${stage} is actually done; (partitions: ${stage.numPartitions})"

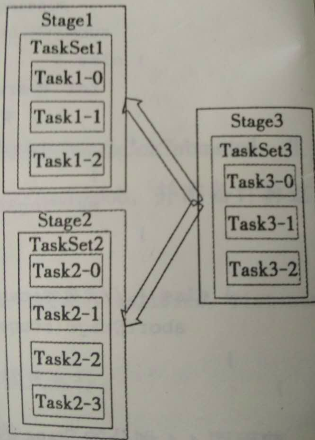
}

logDebug(debugString)

submitWaitingChildStages(stage)

}

}



TaskSet保存了Stage包含的一组完全相同的Task，每个Task的处理逻辑完全相同，不同的是处理的数据，每个Task负责一个Partition。

至此，DAGScheduler就完成了它的任务了。接下来一篇博文，我们会从上述代码中的：

taskScheduler.submitTasks(new TaskSet(

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

开始讲起，深入理解TaskScheduler的工作过程。