

一：Stage划分算法解密

1. Spark Application中可以因为不同的Action触发众多的Job,也就是说一个Application中可以有很多的Job，每个Job是由一个或者多个Stage构成的，后面的Stage依赖于前面的Stage，也就是说只有前面依赖的Stage计算完毕后，后面的Stage才会运行。
2. Stage划分的依据就是宽依赖，什么时候产生宽依赖呢？例如reduceByKey，groupByKey等等。
3. 为啥是case class？因为一个application中可能会有很多job，而不同的Job的JobSubmitted的实例不一样的，而case object是全局唯一的，所以此时的话，JobSubmitted的实例就一样了。

*/\*\* A result-yielding job was submitted on a target RDD \*/***private**[scheduler] **case class** JobSubmitted(

1. 由Action(例如：collect)导致了SparkContext.runjob的执行，最终导致了DAGScheduler中的submitjob的执行，其核心是通过发送一个case class JobSubmitted对象给eventProcessLoop(消息循环器)，其中JobSubmitted源码如下： 包含了当前作业的信息。

*/\*\* A result-yielding job was submitted on a target RDD \*/***private**[scheduler] **case class** JobSubmitted(  
 jobId: Int,  
 finalRDD: RDD[\_],  
 func: (TaskContext, Iterator[\_]) => \_,  
 partitions: Array[Int],  
 callSite: CallSite,  
 listener: JobListener,  
 properties: Properties = **null**)  
 **extends** DAGSchedulerEvent

eventProcessLoop是DAGSchedulerEventProcessLoop的具体实例.

**private**[scheduler] **val** *eventProcessLoop* = **new** DAGSchedulerEventProcessLoop(**this**)

DAGSchedulerEventProcessLoop是子类。

**private**[scheduler] **class** DAGSchedulerEventProcessLoop(dagScheduler: DAGScheduler)  
 **extends** EventLoop[DAGSchedulerEvent]("dag-scheduler-event-loop") **with** Logging {

**具体看一下EventLoop类：**

setDaemon(true)是后台进程，为啥是后台进程呢？作为后台线程，在后台不断的循环，如果是前台线程的话，对垃圾的回收是有影响的。

**private**[spark] **abstract class** EventLoop[E](name: String) **extends** Logging {  
  
**private val** *eventQueue*: BlockingQueue[E] = **new** LinkedBlockingDeque[E]()//往里面发信息  
  
 **private val** *stopped* = **new** AtomicBoolean(**false**)  
  
 **private val** *eventThread* = **new** Thread(name) {  
 setDaemon(**true**)  
  
 **override def** run(): Unit = {  
 **try** {  
 **while** (!*stopped*.get) {//不断的循环队列  
 **val** event = *eventQueue*.take()//从*eventQueue中获得消息队列*  
 **try** {  
 onReceive(event) //接收消息。在这里并没有直接实现OnReceive方法

//具体方法实现是在DAGScheduler#onReceive

} **catch** {  
 **case** *NonFatal*(e) => {  
 **try** {  
 onError(e)  
 } **catch** {  
 **case** *NonFatal*(e) => logError("Unexpected error in " + name, e)  
 }  
 }  
 }  
 }

总结：

eventProcessLoop是DAGSchedulerEventProcessLoop的具体实例，而DAGSchedulerEventProcessLoop是EventLoop的子类，具体实现EventLoop的onReceive方法，onReceive方法转过来回调doOnReceive

**private**[scheduler] **class** DAGSchedulerEventProcessLoop(dagScheduler: DAGScheduler)  
 **extends** EventLoop[DAGSchedulerEvent]("dag-scheduler-event-loop") **with** Logging {  
  
 **private**[**this**] **val** *timer* = dagScheduler.*metricsSource*.*messageProcessingTimer  
  
 /\*\*  
 \* The main event loop of the DAG scheduler.  
 \*/* **override def** onReceive(event: DAGSchedulerEvent): Unit = { //这里面实现了  
 **val** timerContext = *timer*.time()  
 **try** {  
 doOnReceive(event) // 调用doOnReceive  
 } **finally** {  
 timerContext.stop()  
 }  
 }

**//模式匹配，调用*JobSubmitted，是用过post的方式，把信息post给他的。***  
 **private def** doOnReceive(event: DAGSchedulerEvent): Unit = event **match** {  
 **case** *JobSubmitted*(jobId, rdd, func, partitions, callSite, listener, properties) =>  
 dagScheduler.handleJobSubmitted(jobId, rdd, func, partitions, callSite, listener, properties)

**EventLoop里面开辟了一个线程，这个线程不断的循环队列，post的时候其实就是将消息放入到这个队列里面，由于线程不断循环，因此放到队列里面可以拿到，拿到后就会回调DAGSchedulerEventProcessLoop里面的onReceive，处理的时候onReceive调用doOnReceive。**

问题：

DAGScheduler中发消息为啥不是直接掉doOnReceive，而是用消息循环器呢？

1. 异步处理多Job，把Job放到队列里面一个一个的处理。
2. 无论是自己发消息，还是别人发消息的话，都采用循环线程去处理的话，这个时候大家处理的方式就是统一的，逻辑和思路都是一致的。扩展性就会非常好。

**下面就开始Stage划分了。**

1. 在handleJobSubmitted中首先创建finalStage，newResultStage也就是最开始创建的一个Stage,对应上面图的Stage3.

// New stage creation may throw an exception if, for example, jobs are run on a  
// HadoopRDD whose underlying HDFS files have been deleted.  
finalStage = newResultStage(finalRDD, func, partitions, jobId, callSite)

在创建的时候可能会出现异常：HDFS文件被修改，或者被删除了。

**case** e: Exception =>  
 logWarning("Creating new stage failed due to exception - job: " + jobId, e)  
 listener.jobFailed(e)  
 **return**

**private def** getParentStages(rdd: RDD[\_], firstJobId: Int): List[Stage] = {  
 **val** parents = **new** HashSet[Stage] // parents RDD; HashSet为了防止里面元素重复   
 **val** visited = **new** HashSet[RDD[\_]]//存储已经被访问的RDD，构建的时候是从后往前回溯的一个过程，回溯过之后就会被保存起来。  
 // We are manually maintaining a stack here to prevent StackOverflowError  
 // caused by recursively visiting  
 **val** waitingForVisit = **new** Stack[RDD[\_]] //存储需要被处理的RDD  
 **def** visit(r: RDD[\_]) {  
 **if** (!visited(r)) {  
 visited += r //如果没有被回溯过，那么就将此RDD加入HashSet中  
 // Kind of ugly: need to register RDDs with the cache here since  
 // we can't do it in its constructor because # of partitions is unknown  
 **for** (dep <- r.dependencies) { //逐个处理parent RDD  
 dep **match** {

//shuffle类型的Dependency  
 **case** shufDep: ShuffleDependency[\_, \_, \_] =>  
 parents += getShuffleMapStage(shufDep, firstJobId) //增加Stage

//其他就是窄依赖,和自己在同一个Stage,就将此加入  
 **case** \_ =>  
 waitingForVisit.push(dep.rdd)  
 }  
 }  
 }  
 }  
 waitingForVisit.push(rdd)  
 **while** (waitingForVisit.nonEmpty) {  
 visit(waitingForVisit.pop())  
 }  
 parents.toList  
}

**下面我们以上面的图为例，来详细叙述Stage的划分**

**第一次循环：**

1. RDD G传进来，将RDD G压栈.

waitingForVisit.push(rdd)

1. 此时的栈不空，将栈里面的RDD G弹出，作为参数传入visit函数内。

**while** (waitingForVisit.nonEmpty) {  
 visit(waitingForVisit.pop())  
}

1. RDD G没有被访问过，所以执行if中的代码

**val** waitingForVisit = **new** Stack[RDD[\_]]  
**def** visit(r: RDD[\_]) {  
 **if** (!visited(r)) {

1. 对RDD G进行处理，加入visited中。

visited += r

1. 以此处理RDD G依赖的parent RDD

// Kind of ugly: need to register RDDs with the cache here since  
 // we can't do it in its constructor because # of partitions is unknown  
 **for** (dep <- r.dependencies) {  
 dep **match** {

//若是 RDD F 创建新的Stage,并将新创建的Stage存储到parents中。  
 **case** shufDep: ShuffleDependency[\_, \_, \_] =>  
 parents += getShuffleMapStage(shufDep, firstJobId)  
 **case** \_ =>

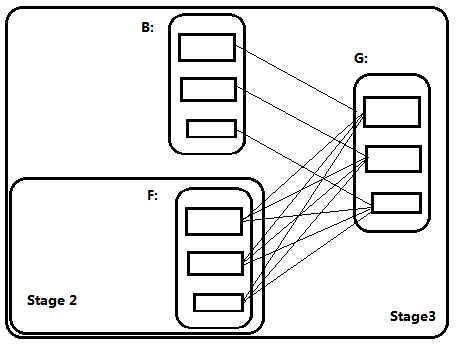
//若是RDD F 则为和RDD G是一个Stage,然后就将RDD F压栈  
 waitingForVisit.push(dep.rdd)  
 }  
 }

上述过程就执行了一次循环了。

RDD G依赖两个RDD：

和RDD B的依赖关系是窄依赖，因此，并不会产生新的RDD，**只是将RDD B 压入Stack栈中。**

和RDD F的依赖关系是宽依赖，因此，RDD G 和RDD F会被划分成两个Stage,Shuffle依赖的关系信息保存在parents中，并且，RDD F所在的Stage 2是RDD G所在的Stage 3的parent Stage。



**第二次循环：**

与上述一样，RDD B所依赖的parent RDD是RDD A 之间是宽依赖关系，因此要创建一个新的Stage为Stage 1.而RDD F的parent RDD都是窄依赖，所以不产生新的Stage,均为Stage 2.

综上：

parents来记录Stage的创建，而不产生新的Stage的处理就是压入到Stack栈中。

**详解getShuffleMapStage**

**getShuffleMapStage获得ShuffleDependency所依赖的Stage,如果没有，则创建新的Stage.**

*/\*\*  
 \* Get or create a shuffle map stage for the given shuffle dependency's map side.  
 \*/***private def** getShuffleMapStage(  
 shuffleDep: ShuffleDependency[\_, \_, \_],  
 firstJobId: Int): ShuffleMapStage = {  
 *shuffleToMapStage*.get(shuffleDep.*shuffleId*) **match** {  
 **case** *Some*(stage) => stage //如果已创建，则直接返回。  
 **case** None =>

//如果有Stage则直接使用，否则，根据Stage创建新的Parent Stage  
 // We are going to register ancestor shuffle dependencies  
 getAncestorShuffleDependencies(shuffleDep.rdd).foreach { dep =>  
 *shuffleToMapStage*(dep.*shuffleId*) = newOrUsedShuffleStage(dep, firstJobId)  
 }  
 // Then register current shuffleDep  
 **val** stage = newOrUsedShuffleStage(shuffleDep, firstJobId)  
 *shuffleToMapStage*(shuffleDep.*shuffleId*) = stage  
 stage  
 }  
}

newOrUsedStage也是生成Stage的，不过如果Stage已经存在，则直接使用。

*/\*\*  
 \* Create a shuffle map Stage for the given RDD. The stage will also be associated with the  
 \* provided firstJobId. If a stage for the shuffleId existed previously so that the shuffleId is  
 \* present in the MapOutputTracker, then the number and location of available outputs are  
 \* recovered from the MapOutputTracker  
 \*/***private def** newOrUsedShuffleStage(  
 shuffleDep: ShuffleDependency[\_, \_, \_],  
 firstJobId: Int): ShuffleMapStage = {  
 **val** rdd = shuffleDep.rdd  
 **val** numTasks = rdd.partitions.length  
 **val** stage = newShuffleMapStage(rdd, numTasks, shuffleDep, firstJobId, rdd.*creationSite*)  
 **if** (mapOutputTracker.containsShuffle(shuffleDep.*shuffleId*)) {

//Stage已经被计算过，从MapOutputTracker中获取计算结果  
 **val** serLocs = mapOutputTracker.getSerializedMapOutputStatuses(shuffleDep.*shuffleId*)  
 **val** locs = MapOutputTracker.*deserializeMapStatuses*(serLocs)  
 (0 until locs.length).foreach { i =>  
 **if** (locs(i) ne **null**) {  
 // locs(i) will be null if missing  
 stage.addOutputLoc(i, locs(i))  
 }  
 }  
 } **else** {  
 // Kind of ugly: need to register RDDs with the cache and map output tracker here  
 // since we can't do it in the RDD constructor because # of partitions is unknown  
 logInfo("Registering RDD " + rdd.*id* + " (" + rdd.getCreationSite + ")")  
 mapOutputTracker.registerShuffle(shuffleDep.*shuffleId*, rdd.partitions.length)  
 }  
 stage  
}

**前面Stage被划分好了，现在就要提交了**

1. HandleJobSubmitted 生成finalStage后，就会为该Job生成一个ActiveJob,来当前Job的一些信息。

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

1. HandleJobSubmitted调用submitStage来提交Stage.

*/\*\* Submits stage, but first recursively submits any missing parents. \*/***private def** submitStage(stage: Stage) {  
 **val** jobId = activeJobForStage(stage)  
 **if** (jobId.isDefined) {  
 logDebug("submitStage(" + stage + ")")

//如果当前Stage不再等待其parent stage的返回，不是正在运行，且没有提示失败，那//么就尝试提交它。  
 **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就是没有父Stage了。  
 submitMissingTasks(stage, jobId.get)  
 } **else** {

//如果parent stage未完成，则递归提交它。  
 **for** (parent <- missing) {  
 submitStage(parent) //从后往前回溯  
 }  
 *waitingStages* += stage  
 }  
 }  
 } **else** {  
 abortStage(stage, "No active job for stage " + stage.id, None)  
 }  
}

**补充说明**：所谓的missing就是说要进行当前的计算了。

**二：Task任务本地性算法实现：**

1. 在submitMissingTasks中会通过调用以下代码来获取任务的本地性。

**val** taskIdToLocations: Map[Int, Seq[TaskLocation]] = **try** {  
 stage **match** {  
 **case** s: ShuffleMapStage =>  
 partitionsToCompute.map { id => (id, getPreferredLocs(stage.rdd, id))}.toMap  
 **case** s: ResultStage =>  
 **val** job = s.activeJob.get  
 partitionsToCompute.map { id =>  
 **val** p = s.partitions(id)  
 (id, getPreferredLocs(stage.rdd, p))  
 }.toMap  
 }

1. 具体一个Partition中的数据本地性的算法实现为下述代码中：

**private**[spark]  
**def** getPreferredLocs(rdd: RDD[\_], partition: Int): Seq[TaskLocation] = {  
 getPreferredLocsInternal(rdd, partition, **new** HashSet)  
}

在具体算法实现的时候，首先查询DAGScheduler的内存数据结构中是否存在当前Paritition的数据本地性的信息，如果有的话直接返回，如果没有首先会调用rdd.getPreferedLocations

例如想让Spark运行在HBase上或一种现在还没有直接支持的数据库上面，此时开发者需要自定义RDD，为了保证Task计算的数据本地性，最为关键的方式就是必须实现RDD的getPreferedLocations，**数据本地性是在底层运行之前就完成了。**

3． DAGScheduler计算数据本地性的时候巧妙的借助了RDD自身的getPreferedLocations中的数据，最大化的优化的效率，因为getPreferedLocations中表明了每个Partition的数据本地性，虽然当前Partition可能被persist或者checkpoint，但是persist或者checkpoint默认情况下肯定是和getPreferedLocations中的Partition的数据本地性是一致的，所以这就极大的简化Task数据本地性算法的实现和效率的优化；

**数据本地性是指**：确定数据在哪个节点上，就到哪个节点上的Executor上去运行。