Spark基础以及Shuffle实现分析

版本：1.1.0

不管是hadoop中map/reduce还是spark中各种算子，shuffle过程都是其中核心过程，shuffle的设计是否高效，基本确定了整个计算过程是否高效。 设计难点在于shuffle过程涉及到大数据的IO操作（包括本地临时文件IO和网络IO），以及可能存在的cpu密集型排序计算操作。

刚刚发布的spark1.1版本，spark针对大型数据引入一个新的shuffle实现，即“sort-based shuffle”

> This release introduces a new shuffle implementation optimized for very large scale shuffles. This “sort-based shuffle” will be become the default in the next release, and is now available to users. For jobs with large numbers of reducers, we recommend turning this on.

本文针对shuffle相关的代码逻辑做一次串读，其中包括shuffle的原理，以及shuffle代码级别的实现。

Job，Stage，Task, Dependency

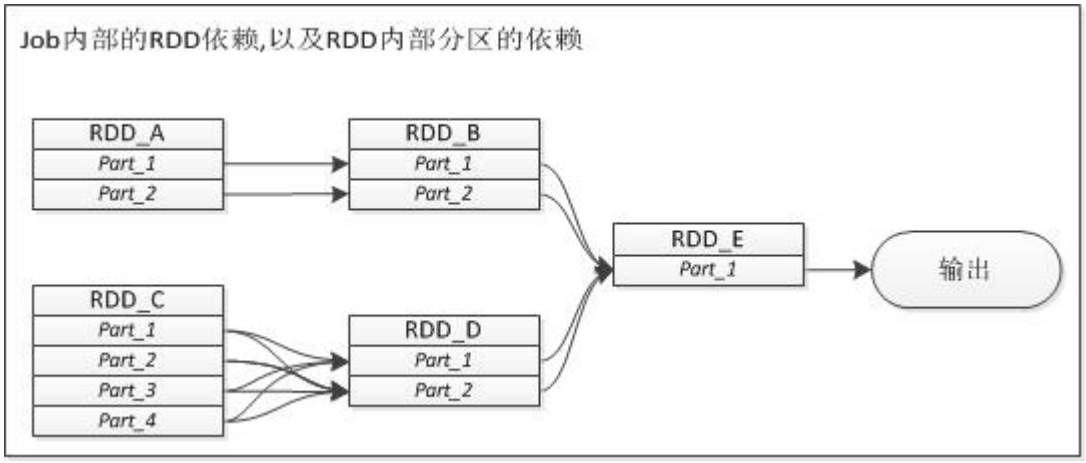
在Spark中，RDD是操作对象的单位，其中操作可以分为转换(transformation)和动作(actions),只有动作操作才会触发一个spark计算操作。
以rdd.map操作和rdd.count操作做比较

```
def map[U: ClassTag](f: T => U): RDD[U] = new MappedRDD(this, sc.clean(f))
def count(): Long = sc.runJob(this, Utils.getIteratorSize _).sum
```

map是一个转换操作，它只是在当前的rdd的基础上创建一个MappedRDD对象，而count是一个动作操作，它会调用sc.runJob向spark提交一个Job

Job是一组rdd的转换以及最后动作的操作集合，它是Spark里面计算最大最虚的概念，甚至在spark的任务页面中都无法看到job这个单位。 但是不管怎么样，在spark用户的角度，job是我们计算目标的单位，每次在一个rdd上做一个动作操作时，都会触发一个job，完成计算并返回我们想要的数据。

Job是由一组RDD上转换和动作组成，这组RDD之间的转换关系表现为一个有向无环图(DAG)，每个RDD的生成依赖于前面1个或多个RDD。
在Spark中，两个RDD之间的依赖关系是Spark的核心。站在RDD的角度，两者依赖表现为点对点依赖，但是在Spark中，RDD存在分区（partition）的概念，两个RDD之间的转换会被细化为两个RDD分区之间的转换。



如上图所示，站在job角度，RDD_B由RDD_A转换而成，RDD_D由RDD_C转换而成，最后RDD_E由RDD_B和RDD_D转换，最后输出RDD_E上做了一个动作，将结果输出。 但是细化到RDD内分区之间依赖，RDD_B对RDD_A的依赖，RDD_D对RDD_C的依赖是不一样，他们的区别用专业词汇来描述即为窄依赖和宽依赖。

所谓的窄依赖是说子RDD中的每一个数据分区只依赖于父RDD中的对应的有限个固定的数据分区，而宽依赖是指子RDD中的每个数据分区依赖于父RDD中的所有数据分区。

宽依赖很好理解，但是对于窄依赖比较绕口，特别是定义中有限与固定两个要求，宽依赖也满足有限和固定这两个要求？难道他们俩个之间区别也仅仅在于“有限”这个数字的大小？ 其实就是这样的理解，“有限”就表现为所依赖的分区数目相比完整分区数相差很大，而且spark靠窄依赖来实现的RDD基本上都大部分都是一对一的依赖，所以就不需要纠结这个有限的关键字。

这里还有一个问题，count操作是依赖父RDD的所有分区进行计算而得到，那么它是宽依赖吗？这么疑问，答案肯定就是否定的，首先这里依赖是父RDD和子RDD之间的关系描述，count操作只有输出， 没有子rdd的概念，就不要把依赖的关系硬套上给你带来麻烦。看上面的实现，count只是把sc.runJob计算返回的Array[U]做一次sum操作而已。

窄依赖和宽依赖的分类是Spark中很重要的特性，不同依赖在实现，任务调度机制，容错恢复上都有不同的机制。

- 实现上：对于窄依赖，rdd之间的转换可以直接pipe化，而宽依赖需要采用shuffle过程来实现。
- 任务调度上：窄依赖意味着可以在某一个计算节点上直接通过父RDD的某几块数据（通常是一块）计算得到子RDD某一块的数据； 而相对的，宽依赖意味着子RDD某一块数据的计算必须等到它的父RDD所有数据都计算完成之后才可以进行，而且需要对父RDD的计算结果需要经过shuffle才能被下一个rdd所操作。
- 容错恢复上：窄依赖的错误恢复会比宽依赖的错误恢复要快很多，因为对于窄依赖来说，只有丢失的那一块数据需要被重新计算， 而宽依赖意味着所有的祖先RDD中所有的数据块都需要被重新计算一遍，这也是我们建议在长“血统”链条特别是有宽依赖的时候， 需要在适当的时机设置一个数据检查点以避免过长的容错恢复。

理清楚了Job层面RDD之间的关系，RDD层面分区之间的关系，那么下面讲述一下Stage概念。

Stage的划分是对一个Job里面一系列RDD转换和动作进行划分。

- 首先job是因动作而产生，因此每个job肯定都有一个ResultStage，否则job就不会启动。

- 其次，如果Job内部RDD之间存在宽依赖，Spark会针对它产生一个中间Stage，即为ShuffleStage，严格来说应该是ShuffleMapStage，这个stage是针对父RDD而产生的， 相当于在父RDD上做一个父rdd.map().collect()的操作。ShuffleMapStage生成的map输入，对于子RDD，如果检测到所自己所"宽依赖"的stage完成计算，就可以启动一个shuffleFectch， 从而将父RDD输出的数据拉取过程，进行后续的计算。

因此一个Job由一个ResultStage和多个ShuffleMapStage组成。

无Shuffle Job的执行过程

对一个无Shuffle的job执行过程的剖析可以知晓我们执行一个"动作"时,spark的处理流程. 下面我们就以一个简单例子进行讲解:

```
sc.textFile("filepath").count
//def count(): Long = sc.runJob(this, Utils.getIteratorSize _).sum
```

这个例子很简单就是统计这个文件的行数;上面一行代码,对应了下面三个过程中:

- sc.textFile("filepath")会返回一个rdd,
- 然后在这个rdd上做count动作,触发了一次Job的提交sc.runJob(this, Utils.getIteratorSize _)
- 对runJob返回的Array结构进行sum操作;

核心过程就是第二步,下面我们以代码片段的方式来描述这个过程,这个过程肯定是线性的,就用step来标示每一步,以及相关的代码类:

```
//step1:SparkContext
def runJob[T, U: ClassTag](
      rdd: RDD[T],
      func: (TaskContext, Iterator[T]) => U,
      partitions: Seq[Int],
      allowLocal: Boolean
      ): Array[U] = {
    val results = new Array[U](partitions.size)
    runJob[T, U](rdd, func, partitions, allowLocal, (index, res) => results(index) = res)
    results
}
```

sc.runJob(this, Utils.getIteratorSize _)的过程会经过一组runJob的重载函数,进入上述step1中的runJob函数,相比原始的runJob,到达这边做的工作不多,比如设置partitions个数, Utils.getIteratorSize _到func转化等,以后像这样简单的过程就不再描述.

Step1做的一个很重要的工作是构造一个Array,并构造一个函数对象"(index, res) => results(index) = res"继续传递给runJob函数,然后等待runJob函数运行结束,将results返回; 对这里的解释相当在runJob添加一个回调函数,将runJob的运行结果保存到Array到, 回调函数,index表示mapindex, res为单个map的运行结果,对于我们这里例子.res就为每个分片的 文件行数.

```
  //step2:SparkContext
  def runJob[T, U: ClassTag](
        rdd: RDD[T],
        func: (TaskContext, Iterator[T]) => U,
        partitions: Seq[Int],
        allowLocal: Boolean,
        resultHandler: (Int, U) => Unit) {
      if (dagScheduler == null) {
        throw new SparkException("SparkContext has been shutdown")
      }
      val callSite = getCallSite
      val cleanedFunc = clean(func)
      dagScheduler.runJob(rdd, cleanedFunc, partitions, callSite, allowLocal,
        resultHandler, localProperties.get)
      rdd.doCheckpoint()
    }
```

Step2中runJob就有一个resultHandler参数,这就是Step1构造的回调函数,dagScheduler是Spark里面最外层调度器,通过调用它的runJob函数,将相关参见传入到Spark调度器中. 只有Step1中的runJob函数的返回值有返回值,这里的runJob,包括dagScheduler.runJob都是灭有返回值的;返回是通过Step1的回调函数进行设置的.

> 为什么我要一再强调返回值是通过Step1的回调函数来设置的?这个很重要,否则你都不知道spark调度的job的运行结果是怎么样被我们自己的逻辑代码所获取的!!

还有一点很重要,Step2是Step1以后的直接步骤,所以Step2中的dagScheduler.runJob是堵塞的操作,即直到Spark完成Job的运行之前,rdd.doCheckpoint()是不会执行的;

```
  //Step3:DAGScheduler
  def runJob[T, U: ClassTag](rdd: RDD[T],func: (TaskContext, Iterator[T]) => U,
        partitions: Seq[Int],callSite: CallSite,
        allowLocal: Boolean,resultHandler: (Int, U) => Unit,properties: Properties = null)
    {
      val start = System.nanoTime
      val waiter = submitJob(rdd, func, partitions, callSite, allowLocal, resultHandler,
  properties)
      waiter.awaitResult() match {
        case JobSucceeded => {
          logInfo("Job %d finished: %s, took %f s".format
            (waiter.jobId, callSite.shortForm, (System.nanoTime - start) / 1e9))
        }
        case JobFailed(exception: Exception) =>
          logInfo("Job %d failed: %s, took %f s".format
            (waiter.jobId, callSite.shortForm, (System.nanoTime - start) / 1e9))
          throw exception
      }
    }
```

Step2中说了dagScheduler.runJob是堵塞的,堵塞就堵塞在Step3的waiter.awaitResult()操作,即submitJob会返回一个waiter对象,而我们的awaitResult()就堵塞了;

到目前为止,我们终于从runJob这个多处出现的函数名称跳到submitJob这个函数名称;继续下一步

```
//Step4:DAGScheduler
def submitJob[T, U](){//省略了函数的参数
    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)
    eventProcessActor ! JobSubmitted(
          jobId, rdd, func2, partitions.toArray, allowLocal, callSite, waiter, properties)
    waiter
}
```

在Step4的submitJob中,我们给这次job分配了一个jobID, 通过创建了一个JobWaiter对象,返回给Step3;最重要的步骤就是调用eventProcessActor ! JobSubmitted向DAG调度器 发送一个JobSubmitted的消息;

到目前为止我们都没有关系函数的参数,这里我们要分析一下发送的JobSubmitted的消息包:

- jobId,rdd,func2,partitions.toArray这几个都比较好理解,就不阐述了
- allowLocal:是否运行直接在本地完成job的运行,毕竟有些操作是没有必要通过task提交给spark运行,当然这里不是重点
- callSite/properties:个人不是很感兴趣,姑且理解为不重要的
- waiter就是上面创建的JobWaiter对象,这个很重要,因为这个对象封装了几个重要的参数:
  - jobId:Job编号
  - partitions.size:分区编号
  - resultHandler:我们Step1设置的回调函数

> 为什么JobWaiter重要,这个对象包含了我们分区的个数.我们知道分区的个数和task个数是相同的,因此JobWaiter成功返回的前提是: 它接受到partitions.size个归属于jobid的task成功运行的结果,并通过resultHandler来将这些task运行结果回调给Step2的Array

这句话应该不难理解,其实这句话也包含了我们后面job调度的整体过程, 下面我们就一步一步来分析从job到Stage,到task以及直到task运行成功,调用我们的resultHandler回调的过程.

```
  //Step5:DAGScheduler
  private[scheduler] def handleJobSubmitted() {
      var finalStage: Stage = null
      try {
        finalStage = newStage(finalRDD, partitions.size, None, jobId, callSite)
        submitStage(finalStage)
      }
    }
```

Step4发送的消息最后被Step5中的handleJobSubmitted函数进行处理,我这里删除了handleJobSubmitted中很多我们不关心的代码,Step5的核心代码就是创建一个finalStage, 并调用submitStage将stage提交给Dag进行调度;这里我们从Job单位层面进入Stage层;

这个Stage命名很好:finalStage,它是整个DAG上的最后一个stage,它不是一个集合,而是单一的stage,这说明一个道理,runJob肯定只对应一个finalStage,即最终的输出肯定只有一个, 中间的stage就是我们传说中的shuffleStage,shuffleStage的生成就是在生成finalStage过程中生成的,即newStage.

那么我们就进入newStage这个函数,等一下我们还会回到submitStage,来分析怎么将Stage解析为Task提交给Spark进行运行;

```
  //Step5.1:DAGScheduler
  private def newStage(): Stage =
    {
      val parentStages = getParentStages(rdd, jobId)
      val id = nextStageId.getAndIncrement()
      val stage = new Stage(id, rdd, numTasks, shuffleDep, parentStages, jobId, callSite)
      stageIdToStage(id) = stage
      updateJobIdStageIdMaps(jobId, stage)
      stage
    }
```

Step5.1中首先是在当前的rdd上调用getParentStages来生成父Stage,父Stages是一个列表;我们这里分析的cache是没有Shuffle的,那么肯定就没有父Stage这个过程;我们就不深入 去分析这个过程;

然后就创建一个Stage对象,并更新Stage和job之间的关系.

下面我们要从维度5.1跳转到一个和执行流程无关的代码,即Stage类的实现,毕竟是Spark的核心对象,对它的理解还是很重要的;

```
  private[spark] class Stage(
      val id: Int,
      val rdd: RDD[_],
      val numTasks: Int,
      val shuffleDep: Option[ShuffleDependency[_, _, _]],  // Output shuffle if stage is a map
  stage
      val parents: List[Stage],
      val jobId: Int,
```

```
      val callSite: CallSite){
  }
```

首先我们看Stage几个字段,其中shuffleDep和parents最为重要,首先如果一个Stage的shuffleDep不为空,那么当前的Stage是因为shuffleMap输出而生成的Stage;

> 怎么解释呢?shuffleDep就是该Stage的生成原因;因为下游rdd对当前的rdd有这个依赖而生成在当前rdd上生成一个Stage. 因此FinalStage,shuffleDep值为none

parents参数就是父Stage列表,当前rdd被调度的前提是所有的父Stage都调度完成;对于我们当前研究这个case来说,shuffleDep和parents都为none;

Stage这个类还有两个比较重要的函数:

```
  //Stage.class
  val isShuffleMap = shuffleDep.isDefined
  def isAvailable: Boolean = {
      if (!isShuffleMap) {
        true
      } else {
        numAvailableOutputs == numPartitions
      }
    }

    def addOutputLoc(partition: Int, status: MapStatus) {
      val prevList = outputLocs(partition)
      outputLocs(partition) = status :: prevList
      if (prevList == Nil) {
        numAvailableOutputs += 1
      }
    }
```

isAvailable这个函数,判读该Stage是否已经运行完成;首先看这个函数,它判读isShuffleMap即当前Stage是否是ShuffleStage.如果不是Shuffle, 这个Stage永远是OK; 换句话说:这个函数对非ShuffleStage没有意义;非ShuffleStage就是上面说的FinalStage, FinalStage永远只有一个,我们不会去判读一个Job的FinalStage是否Ok;

那么为什么要判读ShuffleStage是否OK呢?因为ShuffleStage肯定是中间Stage,只有这个中间Stage完成了才可以提交对该Stage有依赖的下游Stage去计算;

如果当前Stage是ShuffleStage, 那么该Stage Available的前提是该ShuffleStage的所有的MapOutput已经生成成功,即该Stage的所有shuffle Map task已经运行成功;
numAvailableOutputs这个变量就是在addOutputLoc这个函数中进行加一操作;所有我们可以大胆的假设,每个Shuffle Map Task完成Task的输出以后,就会调用该函数 设置当前Task所对应的分片的MapStatus;一旦所有的分片的MapStatus都设置了,那么就代表该Stage所有task都已经运行成功

简单解析一下MapStatus这个类.

```
  private[spark] sealed trait MapStatus {
    def location: BlockManagerId

    def getSizeForBlock(reduceId: Int): Long
  }
```

这个类很简单,首先BlockManagerId代表BlockManager的标示符,里面包含了Host之类的性能,换句话通过BlockManagerId我们知道一个Task的Map输出在哪台Executor机器 上;

对于一个ShuffleStage,我们知道当前ShuffleID, 知道每个Map的index,知道Reduce个数和index,那么通过这里的MapStatus.BlockManagerId就可以读取每个map针对每个reduce的输出Block; 这里getSizeForBlock就是针对每个map上针对reduceId输出的Block大小做一个估算

关于BlockManager相关的知识参阅我的另外一篇文章;简单一句话:MapStatus就是ShuffleMap的输出位置;

上面说了isAvailable不能用于判读FinalStage是否完成,那么我们有没有办法来判读一个FinalStage是否完成呢?有的;我们知道Job肯定只有一个FinalStage,一个FinalStage是否 运行完成,其实就是这个Job是否完成,那么怎么判读一个Job是否完成呢?

```
  //Stage.class
   var resultOfJob: Option[ActiveJob] = None
  //ActiveJob
  private[spark] class ActiveJob(
     val jobId: Int,
     val finalStage: Stage,
     val func: (TaskContext, Iterator[_]) => _,
     val partitions: Array[Int],
     val callSite: CallSite,
     val listener: JobListener,
     val properties: Properties) {

   val numPartitions = partitions.length
   val finished = Array.fill[Boolean](numPartitions)(false)
   var numFinished = 0
  }
```

Stage里面有resultOfJob对这个变量,表示我们当前Stage所对应的Job,它里面有一个finished数组存储这当前Stage/Job所有已经完成Task,换句话说,如果finished里面全部是true, 这个Job运行完成了,这个Job对应的FinalStage也运行完成了,FinalStage依赖的ShuffleStage,以及ShuffleStage依赖的ShuffleStage肯定都运行完成了;

这里说到ActiveJob这个类就多说一句,它的val listener: JobListener变量,其实它就是我们Step4中的waiter,JobWaiter是JobListener类的子类,后面我们要讲到

对Stage这个类做一个总结:Stage可以分为ShuffleStage和FinalStage, 对于ShuffleStage,提供了查询入口来判读Stage是否运行完成,也存储了每个Shuffle Map Task output的BlockManager信息; 对于FinalStage,它和Job是一一绑定,通过Job可以确定Job是否运行完成;

下面继续回到Step 5.1;

在Step 5.1中完成对newStage函数的调用,创建了一个Stage,该Stage的shuffleDep和parents都为none;即该Stage为一个FinalStage,没有任何parent Stage的依赖,那么Spark调度器就 可以把我们的Stage拆分为Task提交给Spark进行调度,即Step 5.2:submitStage;好,我们继续;

```
//5.2:DAGScheduler
private def submitStage(stage: Stage) {
       val jobId = activeJobForStage(stage)
      if (!waitingStages(stage) && !runningStages(stage) && !failedStages(stage)) {
        val missing = getMissingParentStages(stage).sortBy(_.id)
        if (missing == Nil) {
          submitMissingTasks(stage, jobId.get)
        } else {
          for (parent <- missing) {
            submitStage(parent)
          }
          waitingStages += stage
        }
      }
  }
```

Step里面首先判读当前Stage是否处于等待状态,是否处于运行状态,是否处于失败状态. 运行和失败都很好理解,对于等待状态,这里做一个简单的解释:所谓的等待就是 当前Stage依赖的ParentStages还没有运行完成;就是getMissingParentStages这个函数,这个函数的功能肯定对我们Stage的parentStage进行遍历,判读是否isAvailable; 如果为Nil,那么我们就可以调用submitMissingTasks将我们当前的Stage转化为Task进行提交,否则将当前的Stage添加到waitingStages中,即设置当前Stage为等待状态;

其实Stage5.2的逻辑很简单,但是它是Stage层面的最为重要的调度逻辑,即DAG序列化, DAG调度不就是将我们的DAG图转化为有先后次序的序列图吗?!所以简单但是还是要理解;

对于我们的case,流程就进入了submitMissingTasks, 即真正的将Stage转化为Task的功能, 继续;

```
//Step6:DAGScheduler
  private def submitMissingTasks(stage: Stage, jobId: Int) {
    //Step6.1
    stage.pendingTasks.clear()
    val partitionsToCompute: Seq[Int] = {
      if (stage.isShuffleMap) {
        (0 until stage.numPartitions).filter(id => stage.outputLocs(id) == Nil)
      } else {
        val job = stage.resultOfJob.get
        (0 until job.numPartitions).filter(id => !job.finished(id))
      }
```

```
      }
      runningStages += stage

      //Step6.2
      var taskBinary: Broadcast[Array[Byte]] = null
      val taskBinaryBytes: Array[Byte] =
      if (stage.isShuffleMap) {
        closureSerializer.serialize((stage.rdd, stage.shuffleDep.get) : AnyRef).array()
      } else {
        closureSerializer.serialize((stage.rdd, stage.resultOfJob.get.func) : AnyRef).array()
      }
     taskBinary = sc.broadcast(taskBinaryBytes)

      //Step6.3
      val tasks: Seq[Task[_]] = if (stage.isShuffleMap) {
        partitionsToCompute.map { id =>
          val locs = getPreferredLocs(stage.rdd, id)
          val part = stage.rdd.partitions(id)
          new ShuffleMapTask(stage.id, taskBinary, part, locs)
        }
      } else {
        val job = stage.resultOfJob.get
        partitionsToCompute.map { id =>
          val p: Int = job.partitions(id)
          val part = stage.rdd.partitions(p)
          val locs = getPreferredLocs(stage.rdd, p)
          new ResultTask(stage.id, taskBinary, part, locs, id)
        }
      }
      //Step6.4
      if (tasks.size > 0) {
        closureSerializer.serialize(tasks.head)
        stage.pendingTasks ++= tasks
        taskScheduler.submitTasks(
          new TaskSet(tasks.toArray, stage.id, stage.newAttemptId(), stage.jobId, properties))
      } else {
        runningStages -= stage
      }
    }
```

submitMissingTasks的可以删除的代码逻辑不多,都很重要,剩下上面的Step6.1~Step 6.4.下面我们一一进行分析;我们还是先综述一下Step6的工作:

Step6是对Stage到Task的拆分,首先利于上面说到的Stage知识获取所需要进行计算的task的分片;因为该Stage有些分片可能已经计算完成了;然后将Task运行依赖的RDD,Func,shuffleDep 进行序列化,通过broadcast发布出去; 然后创建Task对象,提交给taskScheduler调度器进行运行;

- Step6.1:就是对Stage进行遍历所有需要运行的Task分片;这个不是很好理解,难道每次运行不是对所有分片都进行运行吗?没错,正常的逻辑是对所有的分片进行运行,但是 存在部分task失败之类的情况,或者task运行结果所在的BlockManager被删除了,就需要针对特定分片进行重新计算;即所谓的恢复和重算机制;不是我们这里重点就不进行深入分析;

- Step6.2:对Stage的运行依赖进行序列化并broadcast出去,我对broadcast不是很了解,但是这里我们发现针对ShuffleStage和FinalStage所序列化的内容有所不同;

    - 对于ShuffleStage序列化的是RDD和shuffleDep;而对FinalStage序列化的是RDD和Func

    - 怎么解释呢?对于FinalStage我们知道,每个Task运行过程中,需要知道RDD和运行的函数,比如我们这里讨论的Count实现的Func;而对于ShuffleStage,没有所有Func, 它的task运行过程肯定是按照ShuffleDep的要求,将Map output到相同的物理位置;所以它需要将ShuffleDep序列化出去

```
class ShuffleDependency[K, V, C](
    @transient _rdd: RDD[_ <: Product2[K, V]],
    val partitioner: Partitioner,
    val serializer: Option[Serializer] = None,
    val keyOrdering: Option[Ordering[K]] = None,
    val aggregator: Option[Aggregator[K, V, C]] = None,
    val mapSideCombine: Boolean = false)
  extends Dependency[Product2[K, V]] {

  override def rdd = _rdd.asInstanceOf[RDD[Product2[K, V]]]

  val shuffleId: Int = _rdd.context.newShuffleId()

  val shuffleHandle: ShuffleHandle =
_rdd.context.env.shuffleManager.registerShuffle(
    shuffleId, _rdd.partitions.size, this)

  _rdd.sparkContext.cleaner.foreach(_.registerShuffleForCleanup(this))
}
```

上面就是ShuffleDependency,我们看到ShuffleDep包含了partitioner告诉我们要按照什么分区函数将Map分Bucket进行输出, 有serializer告诉我们怎么对Map的输出进行 序列化, 有keyOrdering和aggregator告诉我们怎么按照Key进行分Bucket,已经怎么进行合并,以及mapSideCombine告诉我们是否需要进行Map端reduce;
还有最为重要的和Shuffle完全相关的shuffleId和ShuffleHandle,这两个东西我们后面具体研究Shuffle再去分析;

    - 因此对于ShuffleStage,我们需要把ShuffleDependency序列化下去

- Step6.3;针对每个需要计算的分片构造一个Task对象,和Step6.2, finalStage和ShuffleStage对应了不同类型的Task,分别为ShuffleMapTask和ResultTask; 她们都接受我们Step6.2broadcast的Stage序列化内容;这样我们就很清楚每个Task的工作,对于ResultTask就是在分片上调用我们的Func,而ShuffleMapTask按照ShuffleDep进行 MapOut,

- Step6.4就是调用taskScheduler将task提交给Spark进行调度

这一节我们不会把分析到Task里面,因此我们就不去深扣ShuffleMapTask和ResultTask两种具体的实现,只需要知道上面谈到的功能就可以;同时我们不会去去分析task调度器的工作原理; 下一篇问题我们会详细分析task调度器;相比DAG调度仅仅维护Stage之间的关系,Task调度器需要将具体Task发送到Executor上执行,涉及内容较多,下一篇吧;

到目前为止,我们已经将我们count job按照ResultTask的提交给Spark进行运行. 好,下面进入最后一个步骤就是我们task运行结果怎么传递给我们的上面Step1回调函数和Step4的waiter对象

```
//Step7:DAGScheduler
private[scheduler] def handleTaskCompletion(event: CompletionEvent) {
    val task = event.task
    val stageId = task.stageId
    val taskType = Utils.getFormattedClassName(task)
    event.reason match {
      //event.reason表示TasK运行结果
      case Success =>
        stage.pendingTasks -= task
        task match {
          //Task是ResultTask
          case rt: ResultTask[_, _] =>
            stage.resultOfJob match {
              case Some(job) =>
                if (!job.finished(rt.outputId)) {
                  job.finished(rt.outputId) = true
                  job.numFinished += 1
                  if (job.numFinished == job.numPartitions) {
                    markStageAsFinished(stage)
                    cleanupStateForJobAndIndependentStages(job)
                    listenerBus.post(SparkListenerJobEnd(job.jobId, JobSucceeded))
                  }
                  job.listener.taskSucceeded(rt.outputId, event.result)
                }
              case None =>
                logInfo("Ignoring result from " + rt + " because its job has finished")
            }
          //Task是ShuffleTask
          case smt: ShuffleMapTask =>
            val status = event.result.asInstanceOf[MapStatus]
            val execId = status.location.executorId
            if (failedEpoch.contains(execId) && smt.epoch <= failedEpoch(execId)) {
              logInfo("Ignoring possibly bogus ShuffleMapTask completion from " + execId)
            } else {
              stage.addOutputLoc(smt.partitionId, status)
            }
            if (runningStages.contains(stage) && stage.pendingTasks.isEmpty) {
              markStageAsFinished(stage)
              if (stage.shuffleDep.isDefined) {
                mapOutputTracker.registerMapOutputs(
```

```
                    stage.shuffleDep.get.shuffleId,
                    stage.outputLocs.map(list => if (list.isEmpty) null else
  list.head).toArray,
                    changeEpoch = true)
                }
                clearCacheLocs()
                if (stage.outputLocs.exists(_ == Nil)) {
                  submitStage(stage)
                } else {
                  val newlyRunnable = new ArrayBuffer[Stage]
                  for (stage <- waitingStages if getMissingParentStages(stage) == Nil) {
                    newlyRunnable += stage
                  }
                  waitingStages --= newlyRunnable
                  runningStages ++= newlyRunnable
                  for {
                    stage <- newlyRunnable.sortBy(_.id)
                    jobId <- activeJobForStage(stage)
                  } {
                    submitMissingTasks(stage, jobId)
                  }
                }
              }
            }
      }
    }
```

上述代码来自DAGScheduler的handleTaskCompletion函数,这个函数逻辑较多,这里只扣出event.reason=SUCCESS部分逻辑;该函数处理来自每个Task 运行成功以后,由TaskScheduler向DAGScheduler发送的消息,函数的参数为CompletionEvent:

```
  private[scheduler] case class CompletionEvent(
      task: Task[_],
      reason: TaskEndReason,
      result: Any,
      accumUpdates: Map[Long, Any],
      taskInfo: TaskInfo,
      taskMetrics: TaskMetrics)
    extends DAGSchedulerEvent
```

有两个重要参数reason和result,分别表示Task结束的原因以及Task的运行结果,我们这里只截取了reason=Success的逻辑,表示Task运行成功;而result就为Task运行结果;

上面我们说了Task有两种类型ResultTask和ShuffleMapTask,我们这里的case为ResultTask;它的业务逻辑主要包括以下几个步骤:

- job.numFinished += 1和job.finished(rt.outputId) = true设置job的一个分片运行结果为true

- job.listener.taskSucceeded(rt.outputId, event.result)这个过程中很重要,我们下面分析以下:job.listener是什么

在上面的ActiveJob我们提到job的listener字段,其实就是我们step4设置的waiter对象;如下所示,我们可以看到JobWaiter是JobListener的子类

```
private[spark] class JobWaiter[T](
    dagScheduler: DAGScheduler,
    val jobId: Int,
    totalTasks: Int,
    resultHandler: (Int, T) => Unit)
  extends JobListener {
  override def taskSucceeded(index: Int, result: Any): Unit = synchronized {
    resultHandler(index, result.asInstanceOf[T])
    finishedTasks += 1
    if (finishedTasks == totalTasks) {
      _jobFinished = true
      jobResult = JobSucceeded
      this.notifyAll()
    }
  }
  def awaitResult(): JobResult = synchronized {
      while (!_jobFinished) {
        this.wait()
      }
      return jobResult
    }
```

上面是JobWaiter类以及被Step7调用的taskSucceeded函数,我们发现就在taskSucceeded这个函数里面,调用了我们在Step2设置的resultHandler,就是这里,将我们Task运行的结果 通过event.result传递给我们Step2中Array;

并且在随后判读finishedTasks == totalTasks所有的Task是否都运行完成, 如果是,那么就this.notifyAll(),唤醒了Step3的awaitResult函数;

到目前为止,我们以及走通了一个非ShuffleStage的运行过程;虽然中间我啰嗦了几句关于Shuffle的东西,如果理解不够没有关系,下一步我们就会来分析非Shuffle的执行过程;

Shuffle Job的执行过程

我们知道Shuffle包含两个过程中,即Shuffle Map过程以及Shuffle Reduce;这里详细解释一下;

> 上面我们谈到了Shuffle Stage,其实是Shuffle Map的过程,即Shuffle Stage的ShuffleTask按照一定的规则将数据写到相应的文件中,并把写的文件"位置信息" 以MapOutput返回给DAGScheduler,MapOutput将它更新到特定位置就完成了整个Shuffle Map过程.
> 在Spark中,Shuffle reduce过程抽象化为ShuffledRDD,即这个RDD的compute方法计算每一个分片即每一个reduce的数据是通过拉取ShuffleMap输出的文件并返回Iterator来实现的

上面两句话基本上表述清楚了Spark的Shuffle的过程,下面我们会针对ShuffleMap和ShuffledRDD的实现分别进行阐述

Shuffle Map过程

对于ShuffleMap的过程的认识,首先需要解释其中一个组件的功能:MapOutputTracker;

MapOutputTracker

写这部分的代码的人肯定写过Hadoop的代码,Tracker的命名方式在Hadoop很常见,但是在Spark中就好像仅此一处;Hadoop中Track对我的影响就是提供一个对象的访问入口,详细可以 参见其他几篇对NodeManager的分析;在这里的MapOutputTracker,也是为MapOutput提供一个访问入口;

首先MapOutput是什么?MapStatus; 每个Shuffle都对应一个ShuffleID,该ShuffleID下面对应多个MapID,每个MapID都会输出一个MapStatus,通过该MapStatus,可以定位每个 MapID所对应的ShuffleMapTask运行过程中所对应的机器;

MapOutputTracker也是提供了这样的接口,可以把每个Map输出的MapStatus注册到Tracker,同时Tracker也提供了访问接口,可以从该Tracker中读取指定每个ShuffleID所对应的map输出的位置;

同时MapOutputTracker也是主从结构,其中Master提供了将Map输出注册到Tracker的入口, slave运行在每个Executor上,提供读取入口, 但是这个读取过程需要和Master进行交互,将指定的 ShuffleID所对应的MapStatus信息从Master中fetch过来;

好了,下面我们具体来分析实现;

```
private[spark] class MapOutputTrackerMaster(conf: SparkConf){
  protected val mapStatuses = new TimeStampedHashMap[Int, Array[MapStatus]]()

  def registerShuffle(shuffleId: Int, numMaps: Int) {
    if (mapStatuses.put(shuffleId, new Array[MapStatus](numMaps)).isDefined) {
      throw new IllegalArgumentException("Shuffle ID " + shuffleId + " registered twice")
    }
  }
  def registerMapOutput(shuffleId: Int, mapId: Int, status: MapStatus) {
    val array = mapStatuses(shuffleId)
    array.synchronized {
      array(mapId) = status
    }
  }
  def getSerializedMapOutputStatuses(shuffleId: Int): Array[Byte] = {
      var statuses: Array[MapStatus] = null
      statuses = mapStatuses.getOrElse(shuffleId, Array[MapStatus]())
      val bytes = MapOutputTracker.serializeMapStatuses(statuses)
      bytes
  }
```

上面是运行在Driver中MapOutputTrackerMaster的实现,它其中包含了一个mapStatuses的TimeStampedHashMap,通过shuffleID进行索引,存储了所有注册到tracker的Shuffle, 通过registerShuffle可以进行注册Shuffle, 通过registerMapOutput可以在每次ShuffleMapTask结束以后,将Map的输出注册到Track中; 同时提供了getSerializedMapOutputStatuses接口 将一个Shuffle所有的MapStatus进行序列化并进行返回;

现在你肯定会为问:什么时候会进行registerShuffle和registerMapOutput的注册?这里简单回答一下:在创建Stage过程中,如果遇到了ShuffleStage,那么就会进行registerShuffle的注册; 在上面谈到的handleTaskCompletion时候, 如果这里的Task是ShuffleMapTask, 就会调用registerMapOutput将结果进行注册;(具体的实现其实有点差别,为了保存简单,就不去阐述这个差别)

上面我说了,MapOutputTrackerMaster是一个主从结构的Master,但是我们这里没有看到通信的实现, 在Master中,通信的实现是使用MapOutputTrackerMasterActor来实现的;

```
private[spark] class MapOutputTrackerMasterActor(tracker: MapOutputTrackerMaster, conf:
SparkConf)
  extends Actor with ActorLogReceive with Logging {
  override def receiveWithLogging = {
    case GetMapOutputStatuses(shuffleId: Int) =>
      val hostPort = sender.path.address.hostPort
      val mapOutputStatuses = tracker.getSerializedMapOutputStatuses(shuffleId)
      sender ! mapOutputStatuses
  }
}
```

它对外提供了GetMapOutputStatuses的入口,slave通过该消息从Master中读取指定Shuffle的MapStatus;

```
private[spark] class MapOutputTrackerWorker(conf: SparkConf) extends MapOutputTracker(conf)
{
  protected val mapStatuses: Map[Int, Array[MapStatus]] =
    new ConcurrentHashMap[Int, Array[MapStatus]]

  private val fetching = new HashSet[Int]

  protected def sendTracker(message: Any) {
      val response = askTracker(message)
      if (response != true) {
        throw new SparkException(
          "Error reply received from MapOutputTracker. Expecting true, got " +
response.toString)
      }
   }
  def getServerStatuses(shuffleId: Int, reduceId: Int): Array[(BlockManagerId, Long)] = {
      val statuses = mapStatuses.get(shuffleId).orNull
      if (statuses == null) {
        var fetchedStatuses: Array[MapStatus] = null
        fetching.synchronized {
          if (fetching.contains(shuffleId)) {
```

```
            while (fetching.contains(shuffleId)) {
              try {
                fetching.wait()
              } catch {
                case e: InterruptedException =>
              }
            }
            fetchedStatuses = mapStatuses.get(shuffleId).orNull
            if (fetchedStatuses == null) {
              fetching += shuffleId
            }
          }

          if (fetchedStatuses == null) {
            try {
              val fetchedBytes =
                askTracker(GetMapOutputStatuses(shuffleId)).asInstanceOf[Array[Byte]]
              fetchedStatuses = MapOutputTracker.deserializeMapStatuses(fetchedBytes)
              mapStatuses.put(shuffleId, fetchedStatuses)
            } finally {
              fetching.synchronized {
                fetching -= shuffleId
                fetching.notifyAll()
              }
            }
          }
          if (fetchedStatuses != null) {
            fetchedStatuses.synchronized {
              return MapOutputTracker.convertMapStatuses(shuffleId, reduceId, fetchedStatuses)
            }
          } else {
            throw new MetadataFetchFailedException(
              shuffleId, reduceId, "Missing all output locations for shuffle " + shuffleId)
          }
        } else {
          statuses.synchronized {
            return MapOutputTracker.convertMapStatuses(shuffleId, reduceId, statuses)
          }
        }
      }
    }
```

MapOutputTrackerWorker为上面的slave, 它的实现很简单,核心功能就是getServerStatuses, 它获取指定Shuffle的每个reduce所对应的MapStatus信息; 想想看, mapTask按照Map 为单位进行输出,而reduceTask肯定是按照reduce为单独进行读取,这也是为什么getServerStatuses有一个reduceId的参数; 具体的我们在ShuffledRDD的实现中来分析;

上面基本分析完了MapOutputTracker,从逻辑上还是比较单独, 但是在整个Shuffle过程中很重要,它存储了Shuffle Map所有的输出;

Map按照什么规则进行output?--ShuffleManager的实现

上面我们说每个shuffleMapStage由多个map组成,每个map将该map中属于每个reduce的数据按照一定规则输出到"文件"中,并返回MapStatus给Driver;这里还有几个问题?

- 每个mapTask按照什么规则进行write?
- 每个reduceTask按照什么规则进行reduce?因为每个reduceTask通过shuffleID和Reduce,只能获取一组表示map输出的mapStatus,reduce怎么从这组mapStatus读取指定 reduce的数据?

这一切都是由ShuffleManager来实现的. 各个公司都说自己针对Shuffle做了什么优化来提供Spark的性能, 本质上就是对ShuffleManager进行优化和提供新的实现; 在1.1以后版本的Spark中ShuffleManager实现为可插拔的接口, 用户可以实现自己的ShuffleManager, 同时提供了两个默认的ShuffleManager的实现;

```
val shortShuffleMgrNames = Map(
      "hash" -> "org.apache.spark.shuffle.hash.HashShuffleManager",
      "sort" -> "org.apache.spark.shuffle.sort.SortShuffleManager")
val shuffleMgrName = conf.get("spark.shuffle.manager", "sort")
val shuffleMgrClass = shortShuffleMgrNames.getOrElse(shuffleMgrName.toLowerCase,
shuffleMgrName)
val shuffleManager = instantiateClass[ShuffleManager](shuffleMgrClass)
```

即老版本的HashShuffleManager和1.1新发布的SortShuffleManager, 可以通过"spark.shuffle.manager""进行配置,默认为SortShuffleManager.

在看具体的ShuffleManager的实现之前,我们先看看ShuffleManager接口提供了哪些功能:

```
private[spark] class BaseShuffleHandle[K, V, C](
    shuffleId: Int,
    val numMaps: Int,
    val dependency: ShuffleDependency[K, V, C])
  extends ShuffleHandle(shuffleId)

private[spark] trait ShuffleManager {
  def registerShuffle(shuffleId: Int, numMaps: Int,dependency: ShuffleDependency):
ShuffleHandle

  def getWriter(handle: ShuffleHandle, mapId: Int, context: TaskContext): ShuffleWriter

  def getReader(handle: ShuffleHandle,startPartition: Int,endPartition: Int,context:
TaskContext): ShuffleReader

  def unregisterShuffle(shuffleId: Int): Boolean

  def shuffleBlockManager: ShuffleBlockManager

  def stop(): Unit
}
```

首先看上面的ShuffleHandle的实现, 它只是一个shuffleId, numMaps和ShuffleDep的封装; 再看ShuffleManager提供的接口;

- registerShuffle/unregisterShuffle:提供了Shuffle的注册和注销的功能, 和上面谈到的MapOutputTracker一直,特别是注册返回的ShuffleHandle来对shuffle的一个封装
- getWriter:针对一个mapTask返回一组Writer,为什么是一组?因为需要针对mapTask上每个可能的reduce提供一个Writer, 所以是一组
- getReader:提供Start分区编号和end分区编号;当然一般情况如果每个reduce单独运行,那么start-end区间也只对应一个reduce, HashShuffleManager也支持一个reduce

最下面有一个ShuffleBlockManager, 和Spark内部的BlockManager相似, 只是把Shuffle的write/reduce都抽象为block的操作, 并由ShuffleBlockManager进行Block管理; 关于这点可以参考Spark内部的BlockManager的getBlockData

```
  override def getBlockData(blockId: String): Option[ManagedBuffer] = {
      val bid = BlockId(blockId)
      if (bid.isShuffle) {

  Some(shuffleManager.shuffleBlockManager.getBlockData(bid.asInstanceOf[ShuffleBlockId]))
      } else {
        val blockBytesOpt = doGetLocal(bid, asBlockResult =
  false).asInstanceOf[Option[ByteBuffer]]
        if (blockBytesOpt.isDefined) {
          val buffer = blockBytesOpt.get
          Some(new NioByteBufferManagedBuffer(buffer))
        } else {
          None
        }
      }
    }
```

我们看到, Spark的BlockManager还是所有的Block的访问入口,如果访问的Block是ShuffleBlock, 会把读取的入口转移到"shuffleManager.shuffleBlockManager"进行读取;

但是站在ShuffleManager的角度来看, 基本上不会对ShuffleBlock按照BlockID进行直接访问,至少很少, 都是通过getWriter和getReader进行访问;

我们先不对具体的ShuffleManager的实现进行研究,我们先看看ShuffleManager是怎么被实现的;

上面我们谈到,在ShuffleMapTask中, mapTask按照一定规则进行write操作,那么我们就来看看ShuffleMapTask的实现;

```
  private[spark] class ShuffleMapTask(
      stageId: Int,
      taskBinary: Broadcast[Array[Byte]],
      partition: Partition,
      @transient private var locs: Seq[TaskLocation])
    extends Task[MapStatus](stageId, partition.index) with Logging {
```

```
    override def runTask(context: TaskContext): MapStatus = {
      val ser = SparkEnv.get.closureSerializer.newInstance()
      val (rdd, dep) = ser.deserialize[(RDD[_], ShuffleDependency[_, _, _])](
        ByteBuffer.wrap(taskBinary.value), Thread.currentThread.getContextClassLoader)

      metrics = Some(context.taskMetrics)
      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]]])
        return writer.stop(success = true).get
      } catch {
        case e: Exception =>
          try {
            if (writer != null) {
              writer.stop(success = false)
            }
          } catch {
            case e: Exception =>log.debug("Could not stop writer", e)
          }
          throw e
      }
    }
  }
```

核心就是ShuffleMapTask.runTask的实现, 每个运行在Executor上的Task, 通过SparkEnv获取shuffleManager对象, 然后调用getWriter来当前MapID=partitionId的一组Writer. 然后将rdd的迭代器传递给writer.write函数, 由每个Writer的实现去实现具体的write操作;

这里就很详细解释了shuffleManager怎么解决"每个mapTask按照什么规则进行write?"这个问题; 关于ShuffleMap还有几个问题没有没有进行解释:

- Job中的ShuffleStage是怎么划分出来的呢?即Step5.1中getParentStages(rdd, jobId)的实现;
- 具体shuffleManager和shuffleBlockManager的实现的分析;

这两个问题需要开一章来分析,这里我就不进行阐述了;这节的目前已经解释清楚了Shuffle的Map过程的实现原理;基本上Shuffle已经走通了一半了;

ShuffledRDD的实现过程

上面讲到了,Shuffle过程是包括Map和Reduce两个过程;其中Shuffle Map以ShuffleMapStage的形式存在, Shuffle Reduce被抽象为一个RDD,该RDD的compute函数有点特殊而已, 如下所示:

```
  override def compute(split: Partition, context: TaskContext): Iterator[(K, C)] = {
      val dep = dependencies.head.asInstanceOf[ShuffleDependency[K, V, C]]
```

```
      SparkEnv.get.shuffleManager.getReader(dep.shuffleHandle, split.index, split.index + 1,
context)
        .read()
        .asInstanceOf[Iterator[(K, C)]]
    }
```

即它通过shuffleManager来拉取特定reduceid的数据:"split.index, split.index + 1"; 下面拿个例子说明

```
scala> val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "spider", "eagle"), 2)
a: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[3] at parallelize at
<console>:12

scala> val b = a.keyBy(_.length)
b: org.apache.spark.rdd.RDD[(Int, String)] = MappedRDD[4] at keyBy at <console>:14

scala> b.groupByKey
res2: org.apache.spark.rdd.RDD[(Int, Iterable[String])] = ShuffledRDD[5] at groupByKey at
<console>:17
```

第三步就会生成一个ShuffledRDD, 而第二步生成的 MappedRDD到第三步的ShuffledRDD需要经过一次ShuffleMap的过程;

对了,还有一个东西:就是上面谈到MapOutputTracker,我们知道MapStatus是在Task运行结束的时候被添加到Tracker,但是我们还没有说过什么时候会通过MapOutputTrackerWorker 来读取Mapstatus, 具体的细节不进一步阐述,它就发生在我们上面阐述的"SparkEnv.get.shuffleManager.getReader.read"中间, 因为每个Reduce需要去拉取该reduce的所有输出的地理位置, 这些都需要去问MapOutputTracker.

OK, 到目前我们已经解释清楚了Spark Shuffle过程, 虽然还有两个问题没有进一步阐述,但是已经足够可以理解Shuffle的实现过程;至于那两个问题,后面有时间再开文写吧

======= end!