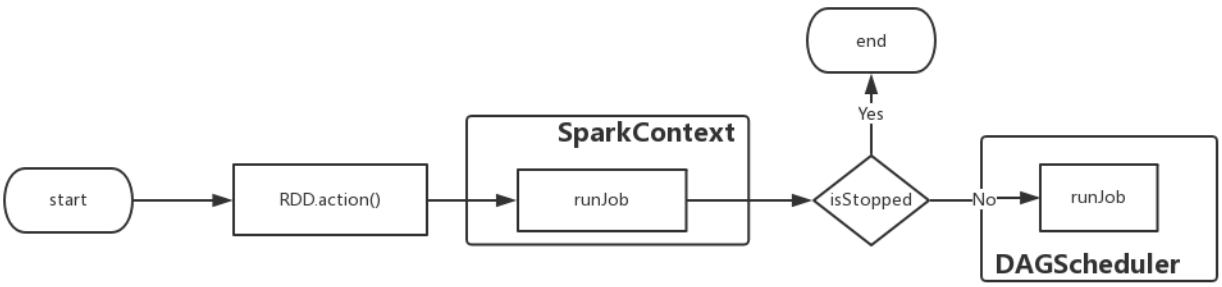
架构

调度

首先，当触发了rdd的action操作之后。将隐式的调用SparkContext中的runJob方法，这样便开始了整个调度过程

```
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)
  // ... ignore some codes
  dagScheduler.runJob(rdd, cleanedFunc, partitions, callSite, resultHandler,
localProperties.get)
  progressBar.foreach(_.finishAll())
  rdd.doCheckpoint()
}
```

从上述代码可以看到，先判断当前SparkContext是否已经停下来了，之后做一些初始化的工作之后开始调用dagScheduler的runJob方法



```
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)
  val awaitPermission = null.asInstanceOf[scala.concurrent.CanAwait]
  waiter.completionFuture.ready(Duration.Inf)(awaitPermission)
  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
  }
}
```

runJob方法中继续调用submitJob方法将任务进行提交，并且创建JobWaiter对象，这里会发生阻塞，直到submitJob方法完成，返回作业完成或者失败的结果

```
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] = {
  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

}
```

submitJob方法中首先根据输入的rdd得到最大分区的数目，然后做一个正确性判断，之后通过nextJobId得到当前任务的编号。接下来，如果输入的分区数目为0，则表明没有要计算的数据，可以直接返回；否则创建一个新的JobWaiter对象，主要将resultHandler传入，这样，在一个任务完成之后，可以调用taskSucceeded方法对结果进行处理。

```
private[spark] class JobWaiter[T](
    dagScheduler: DAGScheduler,
    val jobId: Int,
    totalTasks: Int,
    resultHandler: (Int, T) => Unit)
  extends JobListener with Logging {
  /*
  ... ignore some codes
  */
  override def taskSucceeded(index: Int, result: Any): Unit = {
    synchronized {
      resultHandler(index, result.asInstanceOf[T])
    }
    if (finishedTasks.incrementAndGet() == totalTasks) {
      jobPromise.success(())
    }
  }

  override def jobFailed(exception: Exception): Unit = {
    if (!jobPromise.tryFailure(exception)) {
      logWarning("Ignore failure", exception)
    }
  }
}
```

对于创建好的JobWaiter，根据DAGSchedulerEvent中定义的事件类型创建“提交任务事件”，也就是代码里面看到的

```
JobSubmitted(jobId, rdd, func2, partitions.toArray, callSite,
   waiter, SerializationUtils.clone(properties))
```

之后通过DAGScheduler的内部类DAGSchedulerEventProcessLoop进行消息传递，DAGSchedulerEventProcessLoop继承了EventLoop，设计了针对接收到的不同事件的处理方法

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

  private[this] val timer = dagScheduler.metricsSource.messageProcessingTimer

  override def onReceive(event: DAGSchedulerEvent): Unit = {
    val timerContext = timer.time()
    try {
      doOnReceive(event)
    } finally {
      timerContext.stop()
    }
  }

  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)
    // ... ignore some codes
    case ResubmitFailedStages =>
      dagScheduler.resubmitFailedStages()
  }
}
```

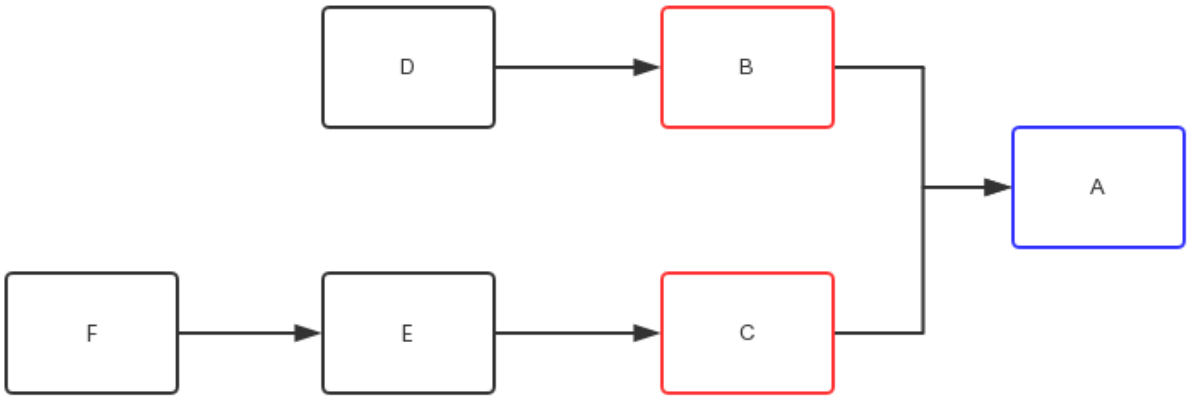
可以发现，尽管接收到了处理的消息，但是具体处理的方法仍然在DAGScheduler中完成，对于JobSubmitted事件，调用handleJobSubmitted方法

```
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)
  } catch {
    case e: Exception =>
      logWarning("Creating new stage failed due to exception - job: " + jobId, e)
      listener.jobFailed(e)
      return
  }
  // ... ignore some codes
}
```

handleJobSubmitted方法里面首先根据输入的rdd创建好Result Stage，这部分过程相对复杂，下面将结合代码和流程图分析。

```
private def createResultStage(
    rdd: RDD[_],
    func: (TaskContext, Iterator[_]) => _,
    partitions: Array[Int],
    jobId: Int,
    callSite: CallSite): ResultStage = {
  val parents = getOrCreateParentStages(rdd, jobId)
  val id = nextStageId.getAndIncrement()
  val stage = new ResultStage(id, rdd, func, partitions, parents, jobId, callSite)
  stageIdToStage(id) = stage
  updateJobIdStageIdMaps(jobId, stage)
  stage
}

private def getOrCreateParentStages(rdd: RDD[_], firstJobId: Int): List[Stage] = {
  getShuffleDependencies(rdd).map { shuffleDep =>
    getOrCreateShuffleMapStage(shuffleDep, firstJobId)
  }.toList
}

private[scheduler] def getShuffleDependencies(
    rdd: RDD[_]): HashSet[ShuffleDependency[_, _, _]] = {
  val parents = new HashSet[ShuffleDependency[_, _, _]]
  val visited = new HashSet[RDD[_]]
  val waitingForVisit = new Stack[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
}
```

createResultStage方法里面首先需要针对输入的rdd寻找它的依赖，getOrCreateParentStages方法里面先按照getShuffleDependencies，根据输入的RDD按照广度优先的方法，找到第一层的宽依赖，如下图



假设图中两两RDD之间均为宽依赖，那么如果RDD A作为getShuffleDependencies函数的输入，那么得到的结果就是包含RDD B和RDD C的集合

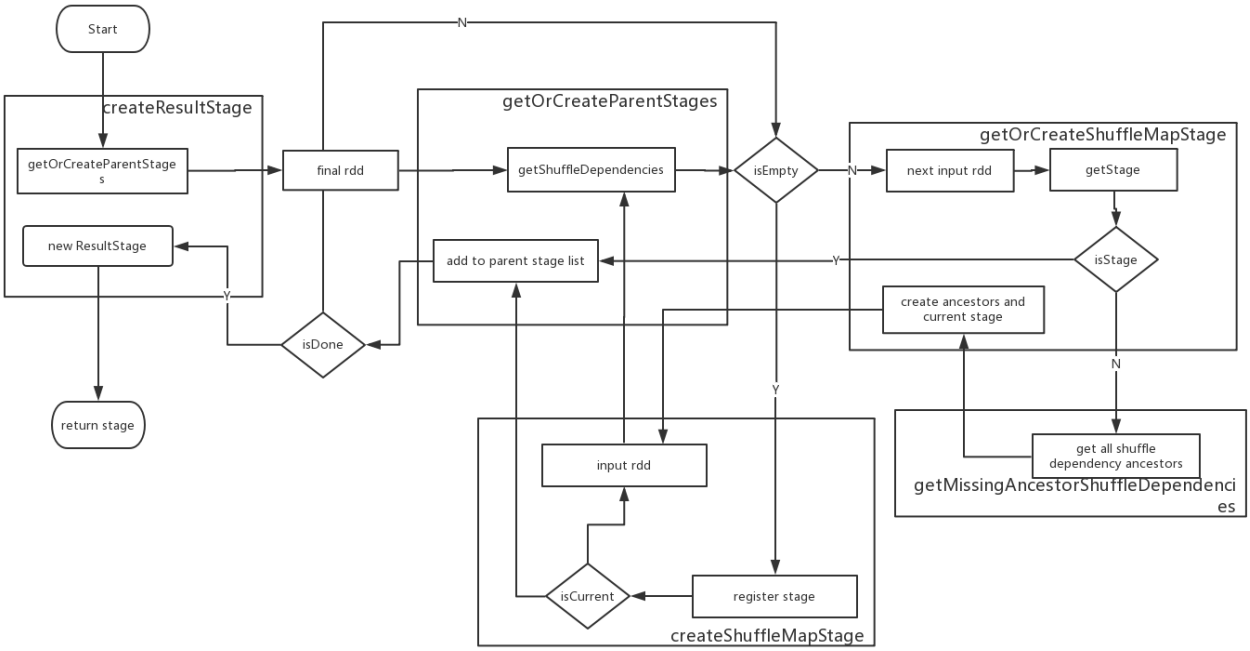
```
private def getOrCreateShuffleMapStage(
    shuffleDep: ShuffleDependency[_, _, _],
    firstJobId: Int): ShuffleMapStage = {
  shuffleIdToMapStage.get(shuffleDep.shuffleId) match {
    case Some(stage) =>
      stage

    case None =>
      getMissingAncestorShuffleDependencies(shuffleDep.rdd).foreach { dep =>
        if (!shuffleIdToMapStage.contains(dep.shuffleId)) {
          // 这里仅仅是创建了shuffleMapStage并注册，并不作为返回，只有下面一个真正返回
          createShuffleMapStage(dep, firstJobId)
        }
      }
      createShuffleMapStage(shuffleDep, firstJobId)
  }
}

// 获取所有祖先中的宽依赖
private def getMissingAncestorShuffleDependencies(
    rdd: RDD[_]): Stack[ShuffleDependency[_, _, _]] = {
// ... ignore some codes
}

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)

  stageIdToStage(id) = stage
  shuffleIdToMapStage(shuffleDep.shuffleId) = stage
  updateJobIdStageIdMaps(jobId, stage)

  // ... ignore some codes
  stage
}
```

getOrCreateShuffleMapStage方法是将输入的宽依赖，判断所对应的ShuffleMapStage是否已经创建了，如果没有的话，首先将他的所有祖先创建ShuffleMapStage，然后在创建自己的ShuffleMapStage并返回

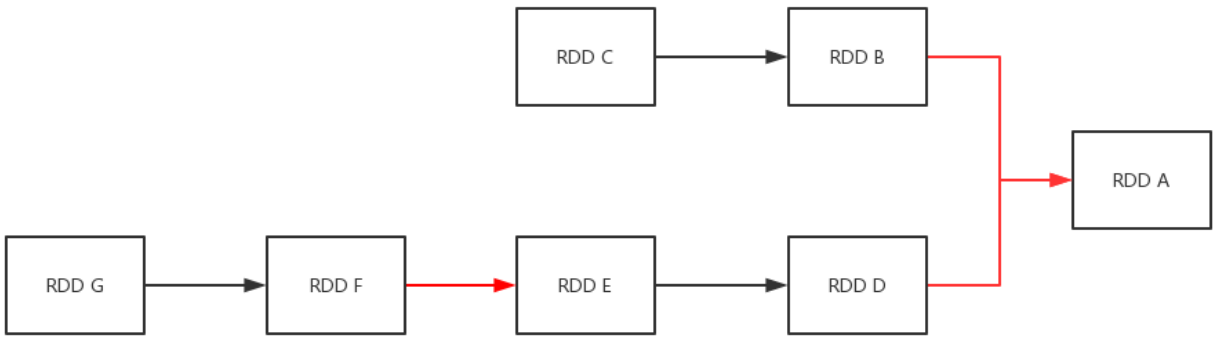
getMissingAncestorShuffleDependencies方法负责找到输入rdd的所有祖先里的宽依赖rdd

createShuffleMapStage负责对于输入的宽依赖，建立ShuffleMapStage，这里还会递归调用getOrCreateParentStages找到并建立祖先的ShuffleMapStage

上面的几个函数是DAGScheduler根据宽依赖切分Stage的核心过程，该过程采用递归调用的方法，比较绕。流程示意图如下



举例分析这个过程，下图中红色的箭头表示宽依赖，黑色的箭头表示窄依赖



首先从RDD A出发，调用createResultStage方法，该方法中调用getOrCreateParentStages方法，首选获取到所有的宽依赖，这里是RDD B和RDD D。

先看RDD B

1. 将其作为参数输入到getOrCreateShuffleMapStage方法
   1.1. 判断这个宽依赖是否已经注册，显然没有
   1.2. 这里调用getMissingAncestorShuffleDependencies找到他的祖先宽依赖，也没有
   1.3. 那么现在调用getOrCreateShuffleMapStage方法
      1.3.1 getOrCreateShuffleMapStage 方法里面会调用 getOrCreateShuffleMapStage 找到所有的祖先ShuffleMapStage，这里也没有
   1.3.2 最后直接创建一个ShuffleMapStage，这里编号记为0，包含RDD C和RDD D。
2. 将ShuffleMapStage 0返回个给最上层getOrCreateParentStages方法里面数组里面的一个值

再看RDD D，这个相对就比较麻烦了，涉及了递归调用。

1. 一开始将它作为参数传入到getOrCreateShuffleMapStage中，

1.1 首先判断这个宽依赖是否已经注册，显然没有，

1.2 这里调用getMissingAncestorShuffleDependencies找到他的祖先宽依赖，只有RDD F

1.2.1 针对RDD F作为输入调用createShuffleMapStage方法

> 1.2.1.1 该方法里面首先找到他的祖先依赖，这里没有

> 1.2.1.2 那么就可以构建一个包含RDD G和RDD F的ShuffleMapStage，编号记为1。

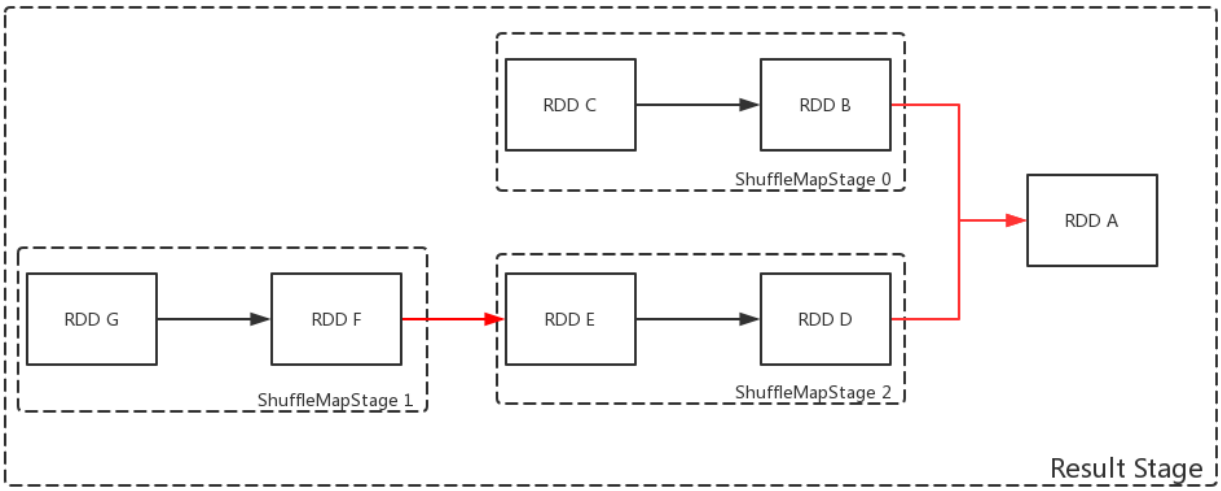
1.3 然后回到之间的getOrCreateShuffleMapStage的方法，刚才是调用了getMissingAncestorShuffleDependencies方法，现在针对RDD D代用createShuffleMapStage

1.3.1 针对RDD D作为输入调用createShuffleMapStage方法

> 1.3.1.1 该方法里面首先找到他的祖先依赖，这里是刚才构建的包含RDD G个RDD F的ShuffleMapStage 1

> 1.3.1.2 之后就可以构建一个包含RDD G和RDD F的ShuffleMapStage，编号记为2。它的祖先是ShuffleMapStage 1

到这里过程算是结束了，生成了一个ResultStage和三个ShuffleMapStage，如下图。



上述流程结束之后，在回到最开始的handleJobSubmitted方法，第二步是创建ActiveJob，将它交给监听总线，最后调用submitStage方法

```
private[scheduler] def handleJobSubmitted(jobId: Int,
    finalRDD: RDD[_],
    func: (TaskContext, Iterator[_]) => _,
    partitions: Array[Int],
    callSite: CallSite,
    listener: JobListener,
    properties: Properties) {
  // ... ignore some codes

  val job = new ActiveJob(jobId, finalStage, callSite, listener, properties)
  clearCacheLocs()
  // ... ignore some codes
  listenerBus.post(
    SparkListenerJobStart(job.jobId, jobSubmissionTime, stageInfos, properties))
  submitStage(finalStage)
}
```

submitStage方法主要将输入的Stage按照先后顺序一个个submit，先提交祖先Stage，再提交当前Stage。

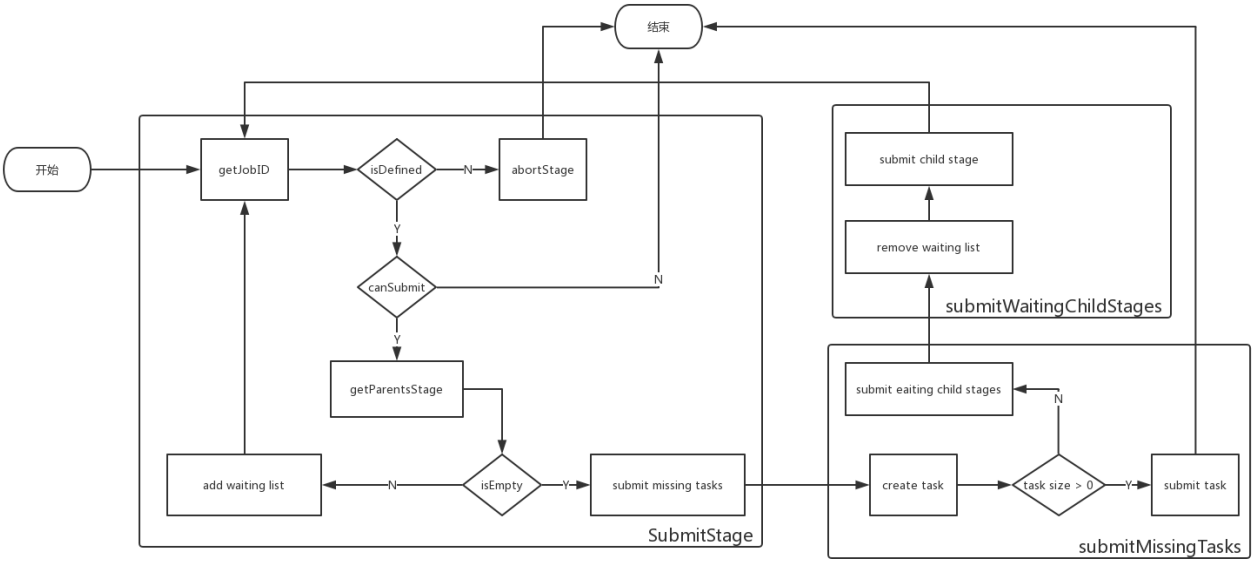
```
/** Submits stage, but first recursively submits any missing parents. */
private def submitStage(stage: Stage) {
  val jobId = activeJobForStage(stage)
  if (jobId.isDefined) {
    logDebug("submitStage(" + stage + ")")
    if (!waitingStages(stage) && !runningStages(stage) && !failedStages(stage)) {
      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)
  }
}

// 获取上一层祖先的Stage
private def getMissingParentStages(stage: Stage): List[Stage] = {
// ... ignore some codes
}
```

代码中首先判断一个Job是否合法

- 不合法的话需要终止当前Stage
- 合法的话再判断是否可以提交该Stage
  - 这一步通过之后，调用getMissingParentStages获取上一层的所有Stage
    - 如果上一层的所有Stage为空，则调用submitMissingTasks提交当前Stage作为任务

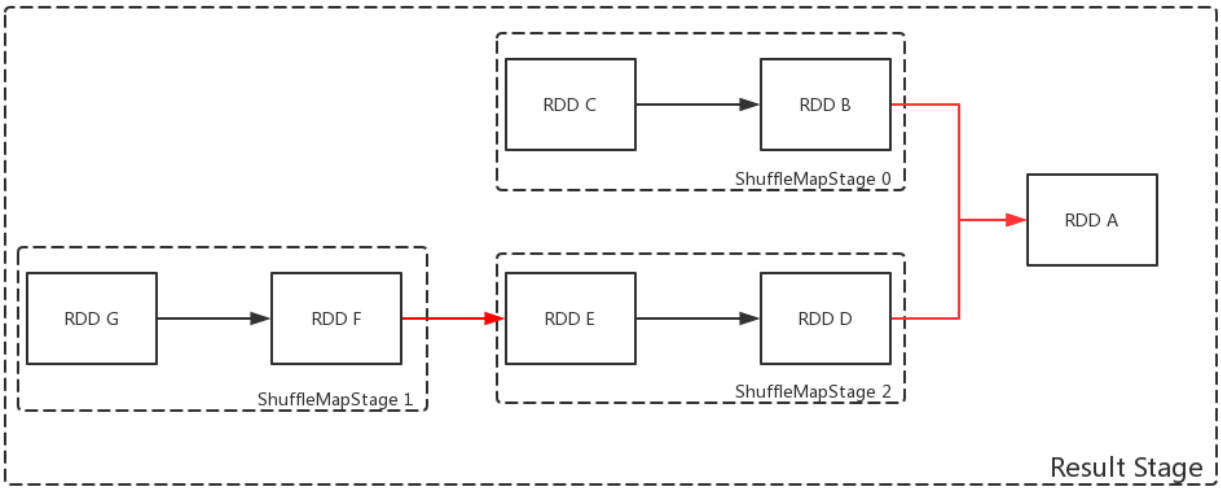
- 上一层Stage不为空，先提交祖先Stage，把自身加入到等待队列中

这里看上去等待队列没有再被你访问过，貌似是提交了上一层的MapStage，其实在submitMissingTasks方法的最后还有一个submitWaitingChildStages方法，把当前任务的孩子任务提交，所以并没有遗漏。流程示意图如下

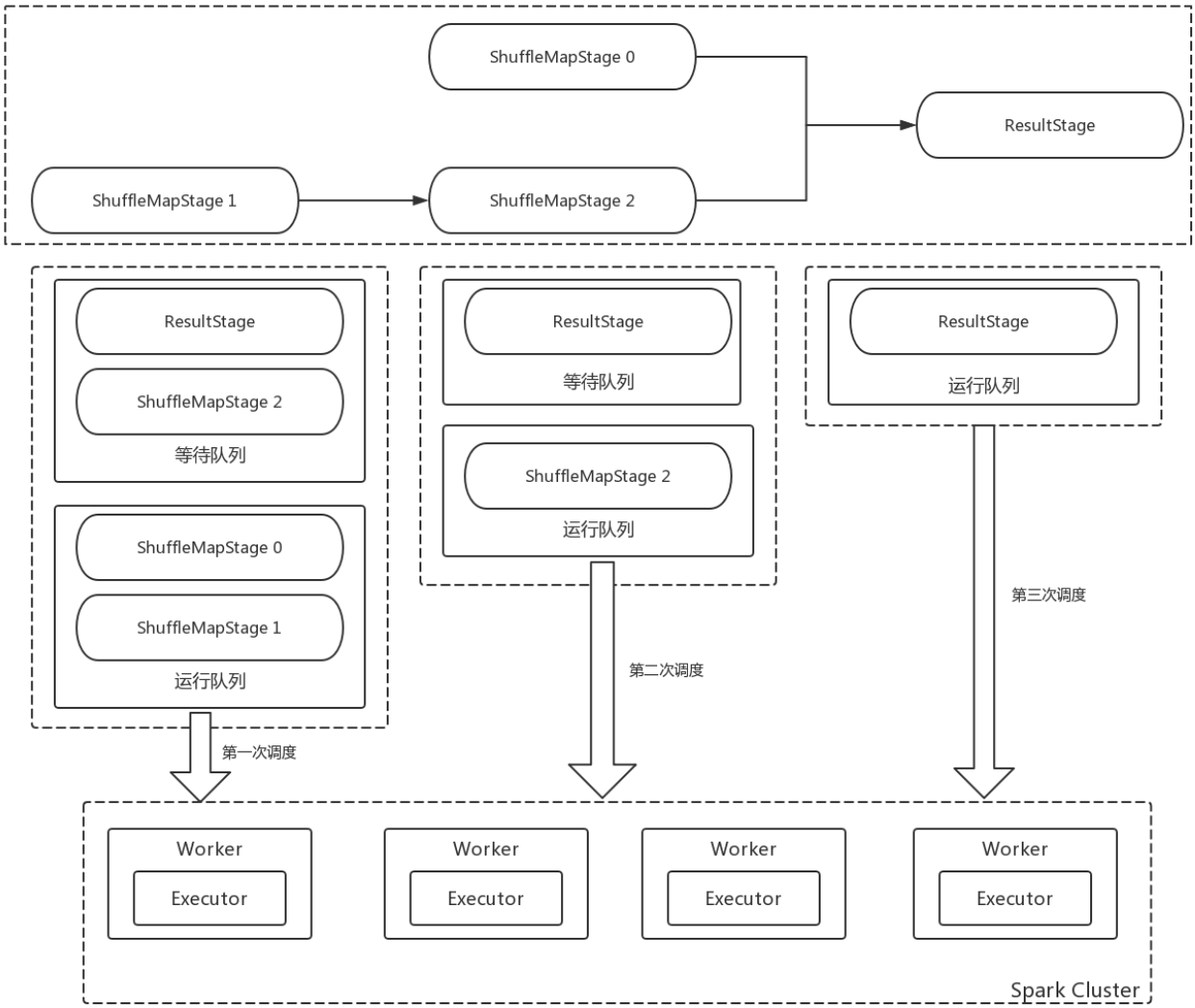


```
private def submitMissingTasks(stage: Stage, jobId: Int) {
  // ... ignore some codes

  if (tasks.size > 0) {
    stage.pendingPartitions ++= tasks.map(_.partitionId)
    taskScheduler.submitTasks(new TaskSet(
      tasks.toArray, stage.id, stage.latestInfo.attemptId, jobId, properties))
    stage.latestInfo.submissionTime = Some(clock.getTimeMillis())
  } else {
    markStageAsFinished(stage, None)
      // ... ignore some codes
    submitWaitingChildStages(stage)
  }
}

private def submitWaitingChildStages(parent: Stage) {
  // ... ignore some codes
  val childStages = waitingStages.filter(_.parents.contains(parent)).toArray
  waitingStages --= childStages
  for (stage <- childStages.sortBy(_.firstJobId)) {
    submitStage(stage)
  }
}

private[scheduler] def handleTaskCompletion(event: CompletionEvent) {
  // ... ignore some codes
  val stage = stageIdToStage(task.stageId)
  event.reason match {
    case Success =>
      stage.pendingPartitions -= task.partitionId
      task match {
        case smt: ShuffleMapTask =>
            // ... ignore some codes
            clearCacheLocs()
            if (!shuffleStage.isAvailable) {
              submitStage(shuffleStage)
            } else {
              if (shuffleStage.mapStageJobs.nonEmpty) {
                val stats = mapOutputTracker.getStatistics(shuffleStage.shuffleDep)
                for (job <- shuffleStage.mapStageJobs) {
                  markMapStageJobAsFinished(job, stats)
                }
              }
              submitWaitingChildStages(shuffleStage)
            }
          }
      }

  }
}
```

用上面的例子分析一下每个Stage提交的过程



- handleJobSubmitted里面得到result stage，生成ActiveJob并调用submitStage方法提交该调度阶段
- 首先调用getMissingParentStages方法看它的祖先是否已经提交了，ShuffleMapStage 0 和ShuffleMapStage 2都没有提交
- 递归调用submitStage方法
  - ShuffleMapStage 0 没有祖先依赖，调用submitMissingTasks方法，生成任务并执行
    - 这里尽管ShuffleMapStage 0所对应的Task执行完会调用submitWaitingChildStages方法，但是因为对于result stage来说，他的另一个依赖ShuffleMapStage 1还没有做，所以这里result stage还是会等待
  - ShuffleMapStage 2 有祖先依赖ShuffleMapStage 1，递归调用submitStage方法
    - ShuffleMapStage 1 没有祖先依赖，调用submitMissingTasks方法，生成任务并执行
    - 执行完成之后调用submitWaitingChildStages方法，提交ShuffleMapStage
  - ShuffleMapStage 2的祖先依赖已经完成，那么submitMissingTasks方法里面将提交自己生成任务并执行
- 执行完成之后调用submitWaitingChildStages方法，提交result stage，result stage已经没有没有提交的祖先依赖，那么这个时候就可以提交给task scheduler执行


ShuffleMapStage 0
ResultStage
ShuffleMapStage 1
ShuffleMapStage 2
ResultStage
ShuffleMapStage 2
等待队列
ShuffleMapStage 0
ShuffleMapStage 1
运行队列
ResultStage
等待队列
ShuffleMapStage 2
运行队列
ResultStage
运行队列
第一次调度
第二次调度
第三次调度
Worker
Executor
Worker
Executor
Worker
Executor
Worker
Executor
Spark Cluster