# 深入理解Spark 2.1 Core （四）：运算结果处理和容错的原理与源码分析

在上一篇博文《深入理解Spark 2.1 Core （三）：任务调度器的实现与源码分析 》TaskScheduler在发送任务给executor前的工作就全部完成了。这篇博文，我们来看看当executor计算完任务后，Spark是如何处理获取的计算结果与容错的。

调用栈如下：

- TaskSchedulerImpl.statusUpdate
    - TaskResultGetter.enqueueSuccessfulTask
        - TaskSchedulerImpl.handleSuccessfulTask
            - TaskSetManager.handleSuccessfulTask
                - DAGScheduler.taskEnded
                    - DAGSchedulerEventProcessLoop.doOnReceive
                        - DAGScheduler.handleTaskCompletion
    - TaskResultGetter.enqueueFailedTask
        - TaskSchedulerImpl.handleFailedTask
            - TaskSetManager.handleFailedTask
                - DAGScheduler.taskEnded
                    - DAGSchedulerEventProcessLoop.doOnReceive
                        - DAGScheduler.handleTaskCompletion

# TaskSchedulerImpl.statusUpdate

TaskRunner将任务的执行结果发送给DriverEndPoint，DriverEndPoint会转给TaskSchedulerImpl的statusUpdate：

```scala
def statusUpdate(tid: Long, state: TaskState, serializedData: ByteBuffer) {
    var failedExecutor: Option[String] = None
    var reason: Option[ExecutorLossReason] = None
    synchronized {
      try {
        taskIdToTaskSetManager.get(tid) match {
          case Some(taskSet) =>
          //这只针对Mesos调度模式
            if (state == TaskState.LOST) {
              val execId = taskIdToExecutorId.getOrElse(tid, throw new IllegalStateException(
                "taskIdToTaskSetManager.contains(tid) <=> taskIdToExecutorId.contains(tid)"))
              if (executorIdToRunningTaskIds.contains(execId)) {
                reason = Some(
                  SlaveLost(s"Task $tid was lost, so marking the executor as lost as well."))
                removeExecutor(execId, reason.get)
                failedExecutor = Some(execId)
              }
            }
            //FINISHED KILLED LOST 都属于 isFinished
            if (TaskState.isFinished(state)) {
              cleanupTaskState(tid)
      taskSet.removeRunningTask(tid)
      //若FINISHED调用taskResultGetter.enqueueSuccessfulTask,
      //否则调用taskResultGetter.enqueueFailedTask(taskSet, tid, state, serializedData)
      if (state == TaskState.FINISHED) {
       taskResultGetter.enqueueSuccessfulTask(taskSet, tid, serializedData)
      } else if (Set(TaskState.FAILED, TaskState.KILLED, TaskState.LOST).contains(state)) {
       taskResultGetter.enqueueFailedTask(taskSet, tid, state, serializedData)
      }
     }
    case None =>
     logError(
      ("Ignoring update with state %s for TID %s because its task set is gone (this is " +
       "likely the result of receiving duplicate task finished status updates) or its " +
       "executor has been marked as failed.")
       .format(state, tid))
   }
  } catch {
   case e: Exception => logError("Exception in statusUpdate", e)
  }
 }
 if (failedExecutor.isDefined) {
  assert(reason.isDefined)
  dagScheduler.executorLost(failedExecutor.get, reason.get)
  backend.reviveOffers()
 }
}
```

## 处理执行成功的结果

我们先来看下处理执行成功的结果的运行机制：

### TaskResultGetter.enqueueSuccessfulTask

```
def enqueueSuccessfulTask(
  taskSetManager: TaskSetManager,
  tid: Long,
  serializedData: ByteBuffer): Unit = {
  //通过线程池来获取结果
 getTaskResultExecutor.execute(new Runnable {
  override def run(): Unit = Utils.logUncaughtExceptions {
   try {
    val (result, size) = serializer.get().deserialize[TaskResult[_]](serializedData) match {
    //可以直接获取到的结果
     case directResult: DirectTaskResult[_] =>
     //判断大小是否符合要求
      if (!taskSetManager.canFetchMoreResults(serializedData.limit())) {
       return
      }

   directResult.value(taskResultSerializer.get())
      (directResult, serializedData.limit())
      //若不能直接获取到结果
     case IndirectTaskResult(blockId, size) =>
      if (!taskSetManager.canFetchMoreResults(size)) {
       // 判断大小是否符合要求,
       //若不符合则远程的删除计算结果
       sparkEnv.blockManager.master.removeBlock(blockId)
       return
      }
      logDebug("Fetching indirect task result for TID %s".format(tid))
      scheduler.handleTaskGettingResult(taskSetManager, tid)
      val serializedTaskResult = sparkEnv.blockManager.getRemoteBytes(blockId)
      //从远程获取计算结果
      if (!serializedTaskResult.isDefined) {
      //若在任务执行结束后与我们去获取结果之间机器出现故障了
      //或者block manager 不得不刷新结果了
      //那么我们将不能够获取到结果
        scheduler.handleFailedTask(
          taskSetManager, tid, TaskState.FINISHED, TaskResultLost)
        return
      }
      val deserializedResult = serializer.get().deserialize[DirectTaskResult[_]](
        serializedTaskResult.get.toByteBuffer)
      // 反序列化
      deserializedResult.value(taskResultSerializer.get())
      sparkEnv.blockManager.master.removeBlock(blockId)
      (deserializedResult, size)
    }


    result.accumUpdates = result.accumUpdates.map { a =>
      if (a.name == Some(InternalAccumulator.RESULT_SIZE)) {
        val acc = a.asInstanceOf[LongAccumulator]
        assert(acc.sum == 0L, "task result size should not have been set on the executors")
        acc.setValue(size.toLong)
        acc
      } else {
        a
      }
    }

    //处理获取到的计算结果
    scheduler.handleSuccessfulTask(taskSetManager, tid, result)
   } catch {
    case cnf: ClassNotFoundException =>
     val loader = Thread.currentThread.getContextClassLoader
     taskSetManager.abort("ClassNotFound with classloader: " + loader)
    case NonFatal(ex) =>
     logError("Exception while getting task result", ex)
     taskSetManager.abort("Exception while getting task result: %s".format(ex))
   }
  }
 })
}
```

## TaskSchedulerImpl.handleSuccessfulTask

调用taskSetManager.handleSuccessfulTask

```
def handleSuccessfulTask(
   taskSetManager: TaskSetManager,
   tid: Long,
   taskResult: DirectTaskResult[_]): Unit = synchronized {
  taskSetManager.handleSuccessfulTask(tid, taskResult)
 }
```

## TaskSetManager.handleSuccessfulTask

```
 def handleSuccessfulTask(tid: Long, result: DirectTaskResult[_]): Unit = {
  val info = taskInfos(tid)
  val index = info.index
  info.markFinished(TaskState.FINISHED)
  //从RunningTask中移除该task
  removeRunningTask(tid)
 //通知dagScheduler该task完成
  sched.dagScheduler.taskEnded(tasks(index), Success, result.value(), result.accumUpdates, info)
  //杀死所有其他与之相同的task的尝试
  for (attemptInfo <- taskAttempts(index) if attemptInfo.running) {
    logInfo(s"Killing attempt ${attemptInfo.attemptNumber} for task ${attemptInfo.id} " +
      s"in stage ${taskSet.id} (TID ${attemptInfo.taskId}) on ${attemptInfo.host} " +
      s"as the attempt ${info.attemptNumber} succeeded on ${info.host}")
    sched.backend.killTask(attemptInfo.taskId, attemptInfo.executorId, true)
  }
  if (!successful(index)) {
  //计数
    tasksSuccessful += 1
    logInfo(s"Finished task ${info.id} in stage ${taskSet.id} (TID ${info.taskId}) in" +
      s" ${info.duration} ms on ${info.host} (executor ${info.executorId})" +
      s" ($tasksSuccessful/$numTasks)")
    //若果有所task成功了，
    //那么标记successful，并且停止
    successful(index) = true
    if (tasksSuccessful == numTasks) {
      isZombie = true
    }
  } else {
    logInfo("Ignoring task-finished event for " + info.id + " in stage " + taskSet.id +
      " because task " + index + " has already completed successfully")
  }
  maybeFinishTaskSet()
 }
```

## DAGScheduler.taskEnded

我们再深入看下是如何通知dagScheduler该task完成的：

```
def taskEnded(
    task: Task[_],
    reason: TaskEndReason,
    result: Any,
    accumUpdates: Seq[AccumulatorV2[_, _]],
    taskInfo: TaskInfo): Unit = {
    //发送CompletionEvent信号
  eventProcessLoop.post(
    CompletionEvent(task, reason, result, accumUpdates, taskInfo))
}
```

## DAGSchedulerEventProcessLoop.doOnReceive

上一篇博文讲过，DAGSchedulerEventProcessLoop的doOnReceive会对信号进行监听：

```
 case completion: CompletionEvent =>
   dagScheduler.handleTaskCompletion(completion)
```

## DAGScheduler.handleTaskCompletion

我们来看下DAGScheduler.handleTaskCompletion部分核心代码：

```
***
  //根据stageId 得到stage
  val stage = stageIdToStage(task.stageId)
  //这里的event就是completion
  event.reason match {
  //这里只看成功的流程
    case Success =>
    //将这个task 从stage等待处理分区中删去
      stage.pendingPartitions -= task.partitionId
      task match {
```

```
//若是最后一个Stage的task
  case rt: ResultTask[_, _] =>
  //将stage 转为 ResultStage
    val resultStage = stage.asInstanceOf[ResultStage]
    resultStage.activeJob match {
    //获取这Stage的job
      case Some(job) =>
        if (!job.finished(rt.outputId)) {
          updateAccumulators(event)
          //标记状态
          job.finished(rt.outputId) = true
          //计数
          job.numFinished += 1
          // 若Job的所有partition都完成了，
          //移除这个Job
          if (job.numFinished == job.numPartitions) {
            markStageAsFinished(resultStage)
            cleanupStateForJobAndIndependentStages(job)
            listenerBus.post(
              SparkListenerJobEnd(job.jobId, clock.getTimeMillis(), JobSucceeded))
          }
          //通知 JobWaiter 有任务成功
          //但 taskSucceeded 会运行用户自定义的代码
          //因此可能抛出异常
          try {
            job.listener.taskSucceeded(rt.outputId, event.result)
          } catch {
            case e: Exception =>
              // 标记为失败
              job.listener.jobFailed(new SparkDriverExecutionException(e))
          }
        }
      case None =>
        logInfo("Ignoring result from " + rt + " because its job has finished")
    }
  //若不是最后一个Stage的Task
  case smt: ShuffleMapTask =>
    val shuffleStage = stage.asInstanceOf[ShuffleMapStage]
    updateAccumulators(event)
    val status = event.result.asInstanceOf[MapStatus]
    val execId = status.location.executorId
    logDebug("ShuffleMapTask finished on " + execId)
    if (failedEpoch.contains(execId) && smt.epoch <= failedEpoch(execId)) {
      logInfo(s"Ignoring possibly bogus $smt completion from executor $execId")
    } else {
    //将Task的partitionId和status
    //追加到OutputLoc
      shuffleStage.addOutputLoc(smt.partitionId, status)
    }

    if (runningStages.contains(shuffleStage) && shuffleStage.pendingPartitions.isEmpty) {
      markStageAsFinished(shuffleStage)
      logInfo("looking for newly runnable stages")
      logInfo("running: " + runningStages)
      logInfo("waiting: " + waitingStages)
      logInfo("failed: " + failedStages)

    //将outputLoc信息注册到mapOutputTracker
    //上篇博文中有提到:
    //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mapOutputTr
    // 所以 DAGScheduler.newOrUsedShuffleStage需要先判断Stage是否已经被计算过
    ///若计算过，DAGScheduler.newOrUsedShuffleStage则把结果复制到新创建的stage
    //如果没计算过，DAGScheduler.newOrUsedShuffleStage就向注册mapOutputTracker Stage，为存储元数据占位
      mapOutputTracker.registerMapOutputs(
              shuffleStage.shuffleDep.shuffleId,
              shuffleStage.outputLocInMapOutputTrackerFormat(),
              changeEpoch = true)

            clearCacheLocs()

            if (!shuffleStage.isAvailable) {
              //若Stage不可用（一些任务失败），则从新提交Stage
              logInfo("Resubmitting " + shuffleStage + " (" + shuffleStage.name +
                ") because some of its tasks had failed: " +
                shuffleStage.findMissingPartitions().mkString(", "))
              submitStage(shuffleStage)
            } else {
              // 若该Stage的所有分区都完成了
              if (shuffleStage.mapStageJobs.nonEmpty) {
                val stats = mapOutputTracker.getStatistics(shuffleStage.shuffleDep)
                //将各个Task的标记为Finished
                for (job <- shuffleStage.mapStageJobs) {
                  markMapStageJobAsFinished(job, stats)
                }
              }
              //提交该Stage的正在等在的Child Stages
              submitWaitingChildStages(shuffleStage)
            }
          }
      }
***
```

# 处理执行失败的结果

## TaskResultGetter.enqueueFailedTask

下面，我们回归头来看如何处理失败的结果。

```
def enqueueFailedTask(taskSetManager: TaskSetManager, tid: Long, taskState: TaskState,
  serializedData: ByteBuffer) {
  var reason : TaskFailedReason = UnknownReason
  try {
  //通过线程池来处理结果
    getTaskResultExecutor.execute(new Runnable {
      override def run(): Unit = Utils.logUncaughtExceptions {
        val loader = Utils.getContextOrSparkClassLoader
        try {
        //若序列化数据，即TaskFailedReason，存在且长度大于0
        //则反序列化获取它
          if (serializedData != null && serializedData.limit() > 0) {
            reason = serializer.get().deserialize[TaskFailedReason](
              serializedData, loader)
          }
        } catch {
        //若是ClassNotFoundException，
        //打印log
          case cnd: ClassNotFoundException =>
            logError(
              "Could not deserialize TaskEndReason: ClassNotFound with classloader " + loader)
              //若其他异常，
              //不进行操作
          case ex: Exception =>
        }
        //处理失败的任务
        scheduler.handleFailedTask(taskSetManager, tid, taskState, reason)
      }
    })
  } catch {
    case e: RejectedExecutionException if sparkEnv.isStopped =>
  }
}
```

## TaskSchedulerImpl.handleFailedTask

```
  def handleFailedTask(
      taskSetManager: TaskSetManager,
      tid: Long,
      taskState: TaskState,
      reason: TaskFailedReason): Unit = synchronized {
      //处理失败任务
    taskSetManager.handleFailedTask(tid, taskState, reason)
    if (!taskSetManager.isZombie && taskState != TaskState.KILLED) {
    //handleFailedTask会将失败任务放入待运行的队列等待下一次调度
    //所以这里开始新的一轮调度
      backend.reviveOffers()
    }
  }
```

## TaskSetManager.handleFailedTask

我们来看下handleFailedTask核心代码：

```
***
    //调用dagScheduler处理失败任务
    sched.dagScheduler.taskEnded(tasks(index), reason, null, accumUpdates, info)

    if (successful(index)) {
      logInfo(
        s"Task ${info.id} in stage ${taskSet.id} (TID $tid) failed, " +
        "but another instance of the task has already succeeded, " +
        "so not re-queuing the task to be re-executed.")
    } else {
    //将这个任务重新加入到等待队列中
      addPendingTask(index)
    }

    if (!isZombie && reason.countTowardsTaskFailures) {
      taskSetBlacklistHelperOpt.foreach(_.updateBlacklistForFailedTask(
        info.host, info.executorId, index))
      assert (null != failureReason)
      //计数 这个任务的重试次数
      numFailures(index) += 1
      //若大于等于最大重试次数，默认为4，
      //则取消这个任务
      if (numFailures(index) >= maxTaskFailures) {
        logError("Task %d in stage %s failed %d times; aborting job".format(
          index, taskSet.id, maxTaskFailures))
        abort("Task %d in stage %s failed %d times, most recent failure: %s\nDriver stacktrace:"
          .format(index, taskSet.id, maxTaskFailures, failureReason), failureException)
        return
      }
    }
    maybeFinishTaskSet()
  }
```

## DAGScheduler.handleTaskCompletion

与处理成功结果的过程相同，接下来也会调用DAGScheduler.taskEnded。DAGSchedulerEventProcessLoop的doOnReceive接收CompletionEvent信号，调用dagScheduler.handleTaskCompletion(completion)

我们来看下DAGScheduler.handleTaskCompletion 处理失败任务部分的核心代码：

```
 //重新提交任务
case Resubmitted =>
  logInfo("Resubmitted " + task + ", so marking it as still running")
  //把任务加入的等待队列
  stage.pendingPartitions += task.partitionId

//获取结果失败
case FetchFailed(bmAddress, shuffleId, mapId, reduceId, failureMessage) =>
  val failedStage = stageIdToStage(task.stageId)
  val mapStage = shuffleIdToMapStage(shuffleId)
  //若失败的尝试ID 不是 stage尝试ID，
  //则忽略这个失败
  if (failedStage.latestInfo.attemptId != task.stageAttemptId) {
    logInfo(s"Ignoring fetch failure from $task as it's from $failedStage attempt" +
      s" ${task.stageAttemptId} and there is a more recent attempt for that stage " +
      s"(attempt ID ${failedStage.latestInfo.attemptId}) running")
  } else {
    //若失败的Stage还在运行队列，
    //标记这个Stage完成
    if (runningStages.contains(failedStage)) {
      logInfo(s"Marking $failedStage (${failedStage.name}) as failed " +
        s"due to a fetch failure from $mapStage (${mapStage.name})")
      markStageAsFinished(failedStage, Some(failureMessage))
    } else {
      logDebug(s"Received fetch failure from $task, but its from $failedStage which is no " +
        s"longer running")
    }
    //若不允许重试，
    //则停止这个Stage
    if (disallowStageRetryForTest) {
      abortStage(failedStage, "Fetch failure will not retry stage due to testing config",
        None)
    }
   //若达到最大重试次数，
   //则停止这个Stage
    else if (failedStage.failedOnFetchAndShouldAbort(task.stageAttemptId)) {
      abortStage(failedStage, s"$failedStage (${failedStage.name}) " +
        s"has failed the maximum allowable number of " +
        s"times: ${Stage.MAX_CONSECUTIVE_FETCH_FAILURES}. " +
        s"Most recent failure reason: ${failureMessage}", None)
    } else {
      if (failedStages.isEmpty) {
      //若失败的Stage中，没有个task完成了，
      //则重新提交Stage。
      //若果有完成的task的话，我们不能重新提交Stage，
      //因为有些task已经被调度过了。
      //task级别的重新提交是在TaskSetManager.handleFailedTask进行的
        logInfo(s"Resubmitting $mapStage (${mapStage.name}) and " +
          s"$failedStage (${failedStage.name}) due to fetch failure")
        messageScheduler.schedule(new Runnable {
          override def run(): Unit = eventProcessLoop.post(ResubmitFailedStages)
        }, DAGScheduler.RESUBMIT_TIMEOUT, TimeUnit.MILLISECONDS)
      }
      failedStages += failedStage
      failedStages += mapStage
    }
    // 移除OutputLoc中的数据
    // 取消注册mapOutputTracker
    if (mapId != -1) {
      mapStage.removeOutputLoc(mapId, bmAddress)
      mapOutputTracker.unregisterMapOutput(shuffleId, mapId, bmAddress)
    }
```

```

     //当有executor上发生多次获取结果失败,
     //则标记这个executor丢失
     if (bmAddress != null) {
       handleExecutorLost(bmAddress.executorId, filesLost = true, Some(task.epoch))
     }
   }

 //拒绝处理
 case commitDenied: TaskCommitDenied =>
   // 不做任何事,
   //让 TaskScheduler 来决定如何处理

 //异常
 case exceptionFailure: ExceptionFailure =>
   // 更新accumulator
   updateAccumulators(event)

 //task结果丢失
 case TaskResultLost =>
 // 不做任何事,
 // 让 TaskScheduler 处理这些错误和重新提交任务

// executor 丢失
// 任务被杀死
// 未知错误
 case _: ExecutorLostFailure | TaskKilled | UnknownReason =>
   // 不做任何事,
   // 若这task不断的错误,
   // TaskScheduler 会停止 job
```