深入理解Spark 2.1 Core（七）：任务执行的原理与源码分析

版权声明：本文为博主原创文章，转载请附上原文地址。 http://blog.csdn.net/u011239443/article/details/54143839

上篇博文《深入理解Spark 2.1 Core （六）：资源调度的实现与源码分析》中我们讲解了，AppClient和Executor是如何启动，如何为逻辑上与物理上的资源调度，以及分析了在Spark1.4之前逻辑上资源调度算法的bug。

这篇博文，我们就来讲讲Executor启动后，是如何在Executor上执行Task的，以及其后续处理。

执行Task

我们在《深入理解Spark 2.1 Core （三）：任务调度器的原理与源码分析》中提到了，任务调度完成后，CoarseGrainedSchedulerBackend.DriverEndpoint会调用launchTasks向CoarseGrainedExecutorBackend发送带着serializedTask的LaunchTask信号。接下来，我们就来讲讲CoarseGrainedExecutorBackend接收到LaunchTask信号后，是如何执行Task的。

调用栈如下：

- CoarseGrainedExecutorBackend.receive
  - Executor.launchTask
    - Executor.TaskRunner.run
      - Executor.updateDependencies
      - Task.run
        - ShuffleMapTask.runTask
        - ResultTask.runTask

CoarseGrainedExecutorBackend.receive

```
    case LaunchTask(data) =>
      if (executor == null) {
        exitExecutor(1, "Received LaunchTask command but executor was null")
      } else {
        // 反序列话task描述
        val taskDesc = ser.deserialize[TaskDescription](data.value)
        logInfo("Got assigned task " + taskDesc.taskId)
        // 调用executor.launchTask
        executor.launchTask(this, taskId = taskDesc.taskId, attemptNumber = taskDesc.attemptNumber,
          taskDesc.name, taskDesc.serializedTask)
      }
```

Executor.launchTask

```
  def launchTask(
      context: ExecutorBackend,
      taskId: Long,
      attemptNumber: Int,
      taskName: String,
      serializedTask: ByteBuffer): Unit = {
      // 创建TaskRunner
    val tr = new TaskRunner(context, taskId = taskId, attemptNumber = attemptNumber, taskName,
      serializedTask)
      // 把taskID 以及 对应的 TaskRunner,
      // 加入到 ConcurrentHashMap[Long, TaskRunner]
    runningTasks.put(taskId, tr)
```

```
    // 线程池 执行 TaskRunner
    threadPool.execute(tr)
  }
```

Executor.TaskRunner.run

```
    override def run(): Unit = {
      val threadMXBean = ManagementFactory.getThreadMXBean
      val taskMemoryManager = new TaskMemoryManager(env.memoryManager, taskId)
      // 记录开始反序列化的时间
      val deserializeStartTime = System.currentTimeMillis()
      // 记录开始反序列化的时的Cpu时间
      val deserializeStartCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {
        threadMXBean.getCurrentThreadCpuTime
      } else 0L
      Thread.currentThread.setContextClassLoader(replClassLoader)
      val ser = env.closureSerializer.newInstance()
      logInfo(s"Running $taskName (TID $taskId)")
      execBackend.statusUpdate(taskId, TaskState.RUNNING, EMPTY_BYTE_BUFFER)
      var taskStart: Long = 0
      var taskStartCpu: Long = 0
      // 开始GC的时间
      startGCTime = computeTotalGcTime()

      try {
       //反序列化任务信息
        val (taskFiles, taskJars, taskProps, taskBytes) =
          Task.deserializeWithDependencies(serializedTask)

       // 根据taskProps设置executor属性
        Executor.taskDeserializationProps.set(taskProps)
       // 根据taskFiles和taskJars,
       // 下载任务所需的File 和 加载所需的Jar包
        updateDependencies(taskFiles, taskJars)
       // 根据taskBytes生成task
        task = ser.deserialize[Task[Any]](taskBytes, Thread.currentThread.getContextClassLoader)
        //设置task属性
        task.localProperties = taskProps
        //设置task内存管理
        task.setTaskMemoryManager(taskMemoryManager)

        // 若在反序列话之前Task就被kill了,
        // 抛出异常
        if (killed) {
          throw new TaskKilledException
        }

        logDebug("Task " + taskId + "'s epoch is " + task.epoch)
        //更新mapOutputTracker Epoch 为task epoch
        env.mapOutputTracker.updateEpoch(task.epoch)

        // 记录任务开始时间
        taskStart = System.currentTimeMillis()
        // 记录任务开始时的cpu时间
        taskStartCpu = if (threadMXBean.isCurrentThreadCpuTimeSupported) {
          threadMXBean.getCurrentThreadCpuTime
        } else 0L
        var threwException = true
        val value = try {
        // 运行Task
          val res = task.run(
            taskAttemptId = taskId,
            attemptNumber = attemptNumber,
            metricsSystem = env.metricsSystem)
```

```
          threwException = false
          res
        } finally {
          val releasedLocks = env.blockManager.releaseAllLocksForTask(taskId)
          val freedMemory = taskMemoryManager.cleanUpAllAllocatedMemory()

          if (freedMemory > 0 && !threwException) {
            val errMsg = s"Managed memory leak detected; size = $freedMemory bytes, TID = $taskId"
            if (conf.getBoolean("spark.unsafe.exceptionOnMemoryLeak", false)) {
              throw new SparkException(errMsg)
            } else {
              logWarning(errMsg)
            }
          }

          if (releasedLocks.nonEmpty && !threwException) {
            val errMsg =
              s"${releasedLocks.size} block locks were not released by TID = $taskId:\n" +
                releasedLocks.mkString("[", ", ", "]")
            if (conf.getBoolean("spark.storage.exceptionOnPinLeak", false)) {
              throw new SparkException(errMsg)
            } else {
              logWarning(errMsg)
            }
          }
        }
        // 记录任务结束时间
        val taskFinish = System.currentTimeMillis()
        // 记录任务结束时的cpu时间
        val taskFinishCpu = if (threadMXBean.isCurrentThreadCpuTimeSupported) {
          threadMXBean.getCurrentThreadCpuTime
        } else 0L

        // 若task在运行中被kill了
        // 则抛出异常
        if (task.killed) {
          throw new TaskKilledException
        }


        val resultSer = env.serializer.newInstance()
        // 结果记录序列化开始的系统时间
        val beforeSerialization = System.currentTimeMillis()
        // 序列化结果
        val valueBytes = resultSer.serialize(value)
        // 结果记录序列化完成的系统时间
        val afterSerialization = System.currentTimeMillis()

        // 反序列话发生在两个地方：
        // 1. 在该函数下反序列化Task信息以及Task实例。
        // 2. 在任务启动后，Task.run 反序列化 RDD 和 函数

        // 计算task的反序列化费时
        task.metrics.setExecutorDeserializeTime(
          (taskStart - deserializeStartTime) + task.executorDeserializeTime)
        // 计算task的反序列化cpu费时
        task.metrics.setExecutorDeserializeCpuTime(
          (taskStartCpu - deserializeStartCpuTime) + task.executorDeserializeCpuTime)
        // 计算task运行费时
        task.metrics.setExecutorRunTime((taskFinish - taskStart) - task.executorDeserializeTime)
        // 计算task运行cpu费时
        task.metrics.setExecutorCpuTime(
          (taskFinishCpu - taskStartCpu) - task.executorDeserializeCpuTime)
         // 计算GC时间
```

```
        task.metrics.setJvmGCTime(computeTotalGcTime() - startGCTime)

         //计算结果序列化时间
        task.metrics.setResultSerializationTime(afterSerialization - beforeSerialization)


        val accumUpdates = task.collectAccumulatorUpdates()
        // 这里代码存在缺陷：
        // value相当于被序列化了两次
        val directResult = new DirectTaskResult(valueBytes, accumUpdates)
        val serializedDirectResult = ser.serialize(directResult)
        // 得到结果的大小
        val resultSize = serializedDirectResult.limit

       // 对于计算结果，会根据结果的大小有不同的策略：
       // 1.生成结果在(正无穷,1GB)：
       // 超过1GB的部分结果直接丢弃，
       // 可以通过spark.driver.maxResultSize实现
       // 默认为1G
       // 2.生成结果大小在$[1GB,128MB - 200KB]
       // 会把该结果以taskId为编号存入BlockManager中，
       // 然后把该编号通过Netty发送给Driver，
       // 该阈值是Netty框架传输的最大值
       // spark.akka.frameSize（默认为128MB）和Netty的预留空间reservedSizeBytes（200KB）的差值
       // 3.生成结果大小在（128MB - 200KB,0）：
       // 直接通过Netty发送到Driver
        val serializedResult: ByteBuffer = {
          if (maxResultSize > 0 && resultSize > maxResultSize) {
            logWarning(s"Finished $taskName (TID $taskId). Result is larger than maxResultSize " +
              s"(${Utils.bytesToString(resultSize)} > ${Utils.bytesToString(maxResultSize)}), " +
              s"dropping it.")
            ser.serialize(new IndirectTaskResult[Any](TaskResultBlockId(taskId), resultSize))
          } else if (resultSize > maxDirectResultSize) {
            val blockId = TaskResultBlockId(taskId)
            env.blockManager.putBytes(
              blockId,
              new ChunkedByteBuffer(serializedDirectResult.duplicate()),
              StorageLevel.MEMORY_AND_DISK_SER)
            logInfo(
              s"Finished $taskName (TID $taskId). $resultSize bytes result sent via BlockManager)")
            ser.serialize(new IndirectTaskResult[Any](blockId, resultSize))
          } else {
            logInfo(s"Finished $taskName (TID $taskId). $resultSize bytes result sent to driver")
            serializedDirectResult
          }
        }
        // 更新execBackend 状态
        execBackend.statusUpdate(taskId, TaskState.FINISHED, serializedResult)

      } catch {
        case ffe: FetchFailedException =>
          val reason = ffe.toTaskFailedReason
          setTaskFinishedAndClearInterruptStatus()
          execBackend.statusUpdate(taskId, TaskState.FAILED, ser.serialize(reason))

        case _: TaskKilledException =>
          logInfo(s"Executor killed $taskName (TID $taskId)")
          setTaskFinishedAndClearInterruptStatus()
          execBackend.statusUpdate(taskId, TaskState.KILLED, ser.serialize(TaskKilled))

        case _: InterruptedException if task.killed =>
          logInfo(s"Executor interrupted and killed $taskName (TID $taskId)")
          setTaskFinishedAndClearInterruptStatus()
          execBackend.statusUpdate(taskId, TaskState.KILLED, ser.serialize(TaskKilled))
```

```
187
188         case CausedBy(cDE: CommitDeniedException) =>
189           val reason = cDE.toTaskFailedReason
190           setTaskFinishedAndClearInterruptStatus()
191           execBackend.statusUpdate(taskId, TaskState.FAILED, ser.serialize(reason))
192
193         case t: Throwable =>
194           logError(s"Exception in $taskName (TID $taskId)", t)
195
196           val accums: Seq[AccumulatorV2[_, _]] =
197             if (task != null) {
198               task.metrics.setExecutorRunTime(System.currentTimeMillis() - taskStart)
199               task.metrics.setJvmGCTime(computeTotalGcTime() - startGCTime)
200               task.collectAccumulatorUpdates(taskFailed = true)
201             } else {
202               Seq.empty
203             }
204
205           val accUpdates = accums.map(acc => acc.toInfo(Some(acc.value), None))
206
207           val serializedTaskEndReason = {
208             try {
209               ser.serialize(new ExceptionFailure(t, accUpdates).withAccums(accums))
210             } catch {
211               case _: NotSerializableException =>
212                 ser.serialize(new ExceptionFailure(t, accUpdates, false).withAccums(accums))
213             }
214           }
215           setTaskFinishedAndClearInterruptStatus()
216           execBackend.statusUpdate(taskId, TaskState.FAILED, serializedTaskEndReason)
217
218           if (Utils.isFatalError(t)) {
219             SparkUncaughtExceptionHandler.uncaughtException(t)
220           }
221
222       } finally {
223       // 任务结束后移除
224         runningTasks.remove(taskId)
225       }
226     }
```

Executor.updateDependencies

接下来，我们来看看更新executor的依赖，即下载任务所需的File和加载所需的Jar包：

```
1   private def updateDependencies(newFiles: HashMap[String, Long], newJars: HashMap[String, Long]) {
2     lazy val hadoopConf = SparkHadoopUtil.get.newConfiguration(conf)
3     synchronized {
4       // 下载任务所需的File
5       for ((name, timestamp) <- newFiles if currentFiles.getOrElse(name, -1L) < timestamp) {
6         logInfo("Fetching " + name + " with timestamp " + timestamp)
7         Utils.fetchFile(name, new File(SparkFiles.getRootDirectory()), conf,
8           env.securityManager, hadoopConf, timestamp, useCache = !isLocal)
9         currentFiles(name) = timestamp
10      }
11      // 加载所需的Jar包
12      for ((name, timestamp) <- newJars) {
13        val localName = name.split("/").last
14        val currentTimeStamp = currentJars.get(name)
15          .orElse(currentJars.get(localName))
16          .getOrElse(-1L)
17        if (currentTimeStamp < timestamp) {
18          logInfo("Fetching " + name + " with timestamp " + timestamp)
19          Utils.fetchFile(name, new File(SparkFiles.getRootDirectory()), conf,
```

```
            env.securityManager, hadoopConf, timestamp, useCache = !isLocal)
          currentJars(name) = timestamp
          // 把它加入到 class loader
          val url = new File(SparkFiles.getRootDirectory(), localName).toURI.toURL
          if (!urlClassLoader.getURLs().contains(url)) {
            logInfo("Adding " + url + " to class loader")
            urlClassLoader.addURL(url)
          }
        }
      }
    }
```

Task.run

接下来，我们来看看这篇博文最核心的部分——task运行：

```
  final def run(
      taskAttemptId: Long,
      attemptNumber: Int,
      metricsSystem: MetricsSystem): T = {
    SparkEnv.get.blockManager.registerTask(taskAttemptId)
    //创建TaskContextImpl
    context = new TaskContextImpl(
      stageId,
      partitionId,
      taskAttemptId,
      attemptNumber,
      taskMemoryManager,
      localProperties,
      metricsSystem,
      metrics)
      //在TaskContext中设置TaskContextImpl
    TaskContext.setTaskContext(context)
    taskThread = Thread.currentThread()

    if (_killed) {
      kill(interruptThread = false)
    }

    new CallerContext("TASK", appId, appAttemptId, jobId, Option(stageId), Option(stageAttemptId),
      Option(taskAttemptId), Option(attemptNumber)).setCurrentContext()

    try {
    // 调用runTask
      runTask(context)
    } catch {
      case e: Throwable =>
        try {
          context.markTaskFailed(e)
        } catch {
          case t: Throwable =>
            e.addSuppressed(t)
        }
        throw e
    } finally {
      // 标记Task完成
      context.markTaskCompleted()
      try {
        Utils.tryLogNonFatalError {
          // 释放内存
          SparkEnv.get.blockManager.memoryStore.releaseUnrollMemoryForThisTask(MemoryMode.ON_HEAP)
          SparkEnv.get.blockManager.memoryStore.releaseUnrollMemoryForThisTask(MemoryMode.OFF_HEAP)
          val memoryManager = SparkEnv.get.memoryManager
```

```
          memoryManager.synchronized { memoryManager.notifyAll() }
        }
      } finally {
      //取消TaskContext设置
        TaskContext.unset()
      }
    }
  }
```

Task有两个子类，一个是非最后的Stage的Task，ShuffleMapTask；一个是最后的Stage的Task，ResultTask。它们都覆盖了Task的runTask方法，接下来我们就分别来讲下它们的runTask方法。

ShuffleMapTask.runTask

根据每个Stage的partition数量来生成ShuffleMapTask，ShuffleMapTask会根据下游的Partition数量和Shuffle的策略来生成一系列文件。

```
override def runTask(context: TaskContext): MapStatus = {

  val threadMXBean = ManagementFactory.getThreadMXBean
  // 记录反序列化开始时间
  val deserializeStartTime = System.currentTimeMillis()
  // 记录反序列化开始时的Cpu时间
  val deserializeStartCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {
    threadMXBean.getCurrentThreadCpuTime
  } else 0L
  val ser = SparkEnv.get.closureSerializer.newInstance()
  // 反序列化rdd 及其 依赖
  val (rdd, dep) = ser.deserialize[(RDD[_], ShuffleDependency[_, _, _])](
    ByteBuffer.wrap(taskBinary.value), Thread.currentThread.getContextClassLoader)
    // 计算 反序列化费时
  _executorDeserializeTime = System.currentTimeMillis() - deserializeStartTime
  // 计算 反序列化Cpu费时
  _executorDeserializeCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {
    threadMXBean.getCurrentThreadCpuTime - deserializeStartCpuTime
  } else 0L

  var writer: ShuffleWriter[Any, Any] = null
  try {
  //获取shuffleManager
    val manager = SparkEnv.get.shuffleManager
  // writer
    writer = manager.getWriter[Any, Any](dep.shuffleHandle, partitionId, context)
   // 调用writer.write 开始计算RDD,
   // 这部分 我们会在后续博文讲解
    writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[_ <: Product2[Any, Any]]])
    // 停止计算，并返回结果
    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
  }
}
```

ResultTask.runTask

```
override def runTask(context: TaskContext): U = {
  val threadMXBean = ManagementFactory.getThreadMXBean
  // 记录反序列化开始时间
  val deserializeStartTime = System.currentTimeMillis()
  // 记录反序列化开始时的Cpu时间
  val deserializeStartCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {
    threadMXBean.getCurrentThreadCpuTime
  } else 0L
  val ser = SparkEnv.get.closureSerializer.newInstance()
  // 反序列化rdd 及其 作用于RDD的结果函数
  val (rdd, func) = ser.deserialize[(RDD[T], (TaskContext, Iterator[T]) => U)](
    ByteBuffer.wrap(taskBinary.value), Thread.currentThread.getContextClassLoader)
    // 计算 反序列化费时
  _executorDeserializeTime = System.currentTimeMillis() - deserializeStartTime
  // 计算 反序列化Cpu费时
  _executorDeserializeCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {
    threadMXBean.getCurrentThreadCpuTime - deserializeStartCpuTime
  } else 0L
  // 这部分 我们会在后续博文讲解
  func(context, rdd.iterator(partition, context))
}
```

后续处理

计量统计

对各个费时的统计，上章已经讲解。

回收内存

这在上章Task.run也已经讲解。

处理执行结果

Executor.TaskRunner.run的execBackend.statusUpdate，在《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 》中我们已经讲解过。