Task执行原理流程图

1. 当Driver端中的CoarseGrainedSchedulerBackend给CoarseGrainedExecutorBackend发送LaunchTask之后，CoarseGrainedExecutorBackend在收到LaunchTask消息之后，首先会反序列化TaskDescription:，下面就是CoarseGrainedExecutorBackend接收到Driver的CoarseGrainedSchedulerBackend发过来的luanchTask之后所作的处理：

**case** *LaunchTask*(data) =>  
 **if** (executor == **null**) {  
 logError("Received LaunchTask command but executor was null")  
 System.*exit*(1)  
 } **else** {  
 **val** taskDesc = *ser*.deserialize[TaskDescription](data.value)  
 logInfo("Got assigned task " + taskDesc.taskId)  
 executor.launchTask(**this**, taskId = taskDesc.taskId, attemptNumber = taskDesc.attemptNumber, // **attemptNumber重试次数**  
 taskDesc.name, taskDesc.serializedTask)  
 }

1. Executor会通过launchTask来执行Task；

*executor*.launchTask(**this**, taskId = taskDesc.taskId, attemptNumber = taskDesc.attemptNumber,  
 taskDesc.name, taskDesc.serializedTask)

**进入executor.launchTask具体看executor是如何处理的。**

1. Executor会通过TaskRunner在ThreadPool来运行具体的Task。创建TaskRunner实例，然后将实例传入到treadPool的execute中具体运行Task。

**val** tr = **new** TaskRunner(context, taskId = taskId, attemptNumber = attemptNumber, taskName,  
 serializedTask)  
*runningTasks*.put(taskId, tr)  
*threadPool*.execute(tr)

**那么TaskRunner有什么用呢？为啥不直接反序列化之后直接传入呢？**

TaskRunner本身是一个Runnable接口。

**class** TaskRunner(  
 execBackend: ExecutorBackend,  
 **val** taskId: Long,  
 **val** attemptNumber: Int,  
 taskName: String,  
 serializedTask: ByteBuffer)  
 **extends** Runnable {

**TaskRunner在运行的时候会调用run方法**

**override def** run(): Unit = {  
 **val** taskMemoryManager = **new** TaskMemoryManager(env.memoryManager, taskId)  
 **val** deserializeStartTime = System.*currentTimeMillis*()//反序列化时间  
 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  
 *startGCTime* = computeTotalGcTime()

**对run方法进行具体分析：**

1. 在TaskRunner的run方法中首先会调用statusUpdate给Driver发信息汇报自己的状态。向Driver报告自己是Running状态。

execBackend.statusUpdate(taskId, TaskState.*RUNNING*, *EMPTY\_BYTE\_BUFFER*)

statusUpdate方法是ExecutorBackend里面定义的方法。

*/\*\*  
 \* A pluggable interface used by the Executor to send updates to the cluster scheduler.  
 \*/***private**[spark] **trait** ExecutorBackend {  
 **def** statusUpdate(taskId: Long, state: TaskState, data: ByteBuffer)  
}

1. TaskRunner内部的run中会做一些准备工作；例如反序列化Task的依赖：

Task.*deserializeWithDependencies*(serializedTask)

反序列化的目的？因为Task在Driver端发过来的时候封装了Task的依赖，这些依赖包括依赖的Jar，文件等，得到这些信息后，就要通过**updateDependencies把这些信息下载下来。**

updateDependencies(taskFiles, taskJars)

**而上面工作都是准备了Task的运行时环境。下面Task就会执行了。**

1. 然后反序列Task本身。

*task* = ser.deserialize[Task[Any]](taskBytes, Thread.*currentThread*.getContextClassLoader)

调用反序列化后的Task.run方法来执行任务并获得执行结果。Value是run之后的结果。

// Run the actual task and measure its runtime.  
taskStart = System.*currentTimeMillis*()//开始执行Task，并记录执行的耗时  
**var** threwException = **true  
val** (value, accumUpdates) = **try** {   
 **val** res = *task*.run(//run方法执行Task，并将结果保存到value中  
 taskAttemptId = taskId,  
 attemptNumber = attemptNumber,  
 metricsSystem = env.metricsSystem)  
 threwException = **false** res

其中Task的run方法调用的时候会导致Task的抽象方法runTask的调用，在Task的runTask内部会调用RDD的iterator()方法，该方法就是我们针对当前Task所对应的Paritition进行计算的关键之所在，在具体的处理内部会迭代Partition的元素交给我们自定义的function进行处理！

其中Task的run方法调用的时候会导致Task的中run方法执行，而run方法会调用内部的runTask方法执行。

**try** {  
 (runTask(context), context.collectAccumulators())

**而runTask的实现分为两种一个是在ShuffleMapTask,另一个是ResultTask。**

**对runTask的处理分两种：**

**ShuffleMapTask:**

runTask在ShuffleMapTask中的实现核心代码如下：

对于ShuffleMapTask，首先要对RDD以及其依赖关系反序列化。

**val** (rdd, dep) = ser.deserialize[(RDD[\_], ShuffleDependency[\_, \_, \_])](

从SparkEnv中获得Shuffle Manager。

**val** manager = SparkEnv.*get*.shuffleManager

**//从Shuffle Manager中取得Writer**  
writer = manager.getWriter[Any, Any](dep.*shuffleHandle*, partitionId, context)

**//调用rdd，并且将计算结果通过writer写入文件系统**  
writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[\_ <: Product2[Any, Any]]])  
writer.stop(success = **true**).get

**ResultTask：**

**val** ser = SparkEnv.*get*.closureSerializer.newInstance()

**//获得RDD和作用RDD结果的函数，func返回是iterator**  
**val** (rdd, func) = ser.deserialize[(RDD[T], (TaskContext, Iterator[T]) => U)](  
 ByteBuffer.*wrap*(taskBinary.value), Thread.*currentThread*.getContextClassLoader)  
*\_executorDeserializeTime* = System.*currentTimeMillis*() - deserializeStartTime  
  
*metrics* = *Some*(context.taskMetrics)

**//调用rdd.iterator方法来计算RDD**  
func(context, rdd.iterator(partition, context))

最终计算会调用RDD的compute方法

compute(split, context)

1. 具体计算的时候有具体的RDD，例如MapParittionsRDD的compute。

其中，f()就是我们在当前的Stage中计算具体Partition的业务逻辑代码；里面是迭代计算。

**override def** compute(split: Partition, context: TaskContext): Iterator[U] =  
 f(context, split.index, firstParent[T].iterator(split, context))

**至此，结果已经计算完了。**

**下面就是对结果的处理。**

1. SerializedDirectResult就是在executor中运行Task的最终结果。

**val** directResult = **new** DirectTaskResult(valueBytes, accumUpdates, *task*.*metrics*.orNull)  
**val** serializedDirectResult = ser.serialize(directResult)  
**val** resultSize = serializedDirectResult.limit

但是这个结果并不是直接传给Driver而是，要先判断，且需往下看？

1. 首先会对结果大小进行判断

// directSend = sending directly back to the driver  
**val** serializedResult: ByteBuffer = {

//如果结果大于1G，那么直接丢弃  
 **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 >= *akkaFrameSize* - AkkaUtils.*reservedSizeBytes*) {

//后续课程继续完善  
 **val** blockId = *TaskResultBlockId*(taskId)  
 env.blockManager.putBytes(  
 blockId, serializedDirectResult, 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")

//直接返回结果  
 serializedDirectResult  
 }  
}  
  
execBackend.statusUpdate(taskId, TaskState.*FINISHED*, serializedResult)

1. 把执行结果序列。

**val** valueBytes = resultSer.serialize(value)

并根据大小判断不同的结果传回给Driver的方式。

Driver端对接受到Task的处理

1. CoarseGrainedExecutorBackend给DriverEndpoint发送StatusUpdate来传输执行结果。

**override def** statusUpdate(taskId: Long, state: TaskState, data: ByteBuffer) {  
 **val** msg = *StatusUpdate*(executorId, taskId, state, data)  
 *driver* **match** {  
 **case** *Some*(driverRef) => driverRef.send(msg)  
 **case** None => logWarning(s"Drop **$**msg because has not yet connected to driver")  
 }  
 }  
}

1. DriverEndpoint会把执行结果传递给TaskSchedulerImpl处理。

如果TaskState为LOST，则executor会被move掉。

**def** statusUpdate(tid: Long, state: TaskState, serializedData: ByteBuffer) {  
 **var** failedExecutor: Option[String] = None  
 synchronized {  
 **try** {  
 **if** (state == TaskState.*LOST* && *taskIdToExecutorId*.contains(tid)) {  
 // We lost this entire executor, so remember that it's gone  
 **val** execId = *taskIdToExecutorId*(tid)  
  
 **if** (*executorIdToTaskCount*.contains(execId)) {  
 removeExecutor(execId,  
 *SlaveLost*(s"Task **$**tid was lost, so marking the executor as lost as well."))  
 failedExecutor = *Some*(execId)  
 }  
 }  
 *taskIdToTaskSetManager*.get(tid) **match** {  
 **case** *Some*(taskSet) =>  
 **if** (TaskState.*isFinished*(state)) {  
 *taskIdToTaskSetManager*.remove(tid)  
 *taskIdToExecutorId*.remove(tid).foreach { execId =>  
 **if** (*executorIdToTaskCount*.contains(execId)) {  
 *executorIdToTaskCount*(execId) -= 1

如果TaskState为FAILED，则会调用enqueueSuccessfulTask方法处理

**if** (state == TaskState.*FINISHED*) {  
 taskSet.removeRunningTask(tid)  
 *taskResultGetter*.enqueueSuccessfulTask(taskSet, tid, serializedData)

如果TaskSet是FAILD,KILLD,LOST中的一个则会调用enqueueFailedTask

**else if** (*Set*(TaskState.*FAILED*, TaskState.*KILLED*, TaskState.*LOST*).contains(state)) {  
 taskSet.removeRunningTask(tid)  
 *taskResultGetter*.enqueueFailedTask(taskSet, tid, state, serializedData)  
}

1. 然后交给TaskResultGetter去分别处理Task执行成功和失败时候的不同情况。在TaskSetManager.handleSuccessfulTask中，然后告诉DAGScheduler去处理结果。

sched.*dagScheduler*.taskEnded(

补充：

1. 在执行具体Task的业务逻辑前会进行三次反序列化：
2. TaskDecription的反序列化(在CoarseGrainedExecutorBackend接收到CoarseGrainedSchedulerBackend(Driver)端发来的luanchTask的时候)

**val** taskDesc = *ser*.deserialize[TaskDescription](data.value)

1. Task反序列化
2. 反序列化Task的依赖
3. RDD反序列化
4. updateDependencies方法是在executor的run方法中被调用的。

updateDependencies(taskFiles, taskJars)

**为啥需要synchronized？**

updateDependencies方法是在线程里面执行的，同一个Stage内部的任务是共享资源的，这就涉及到了ExecutorBackend内部的全局共享资源，多条线程执行的时候，是需要同步块的，

在同一个Stage里面，executorBackend会有很多并发线程，这里面的线程所依赖的Jar，文件，完全都是一样的。updateDependencies运行在run方法里面，会被很多线程去调用，多线程调用的时候就需要加锁，而这个方法是在全局中的，所以访问要加锁。

*/\*\*  
 \* Download any missing dependencies if we receive a new set of files and JARs from the  
 \* SparkContext. Also adds any new JARs we fetched to the class loader.  
 \*/***private def** updateDependencies(newFiles: HashMap[String, Long], newJars: HashMap[String, Long]) {  
 **lazy val** hadoopConf = SparkHadoopUtil.*get*.newConfiguration(*conf*)  
 synchronized {

1. Task.run方法执行的时候会反过来调用

**val** res = *task*.run(

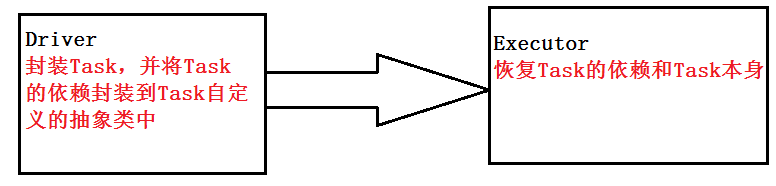
run方法具体在运行运行的时候会调用

**try** {  
 (runTask(*context*), *context*.collectAccumulators())

1. 在Spark1.6中AKKFrameSize是128MB，所以可以广播非常大的任务，任务的执行结果最大可以达到1GB。

在Spark1.6中AKKFrameSize是128MB：Driver给Executor发信息，Task序列化之后可以多大；

Task执行完之后，最大结果为1G



Driver端在封装Task时，会将Task依赖的文件封装到Task中。

**val** out = **new** ByteArrayOutputStream(4096)  
**val** dataOut = **new** DataOutputStream(out)  
  
// Write currentFiles  
dataOut.writeInt(currentFiles.size)  
**for** ((name, timestamp) <- currentFiles) {  
 dataOut.writeUTF(name)  
 dataOut.writeLong(timestamp)  
}  
  
// Write currentJars  
dataOut.writeInt(currentJars.size)  
**for** ((name, timestamp) <- currentJars) {  
 dataOut.writeUTF(name)  
 dataOut.writeLong(timestamp)  
}  
  
// Write the task itself and finish  
dataOut.flush()  
**val** taskBytes = serializer.serialize(task).array()  
out.write(taskBytes)  
ByteBuffer.*wrap*(out.toByteArray)

Executor端，恢复Task本身和Task的依赖。