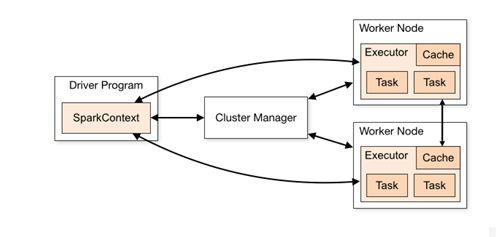
Spark消息通信

Spark以Spark Core为核心，能够读取传统文件（如文本文件）、HDFS、Amazon S3、Alluxio和NoSQL等数据源，利用Standalone、YARN和Mesos等资源调度管理，完成应用程序的分析和处理。这些应用程序来自Spark的不同组件，如Spark Shell或Spark Submit交互式批处理方式、Spark Streaming的实时流处理系统、Spark SQL的即席查询、采用近似查询引擎BlinkDB的权衡查询、MLbase/Mllib的机器学习、GraphX的图处理和SparkR的数学计算等，如下图所示：



Spark通过客户端将程序，提交给集群，作业运行在分布式节点上，基本流程图如下：



其中Cluster Manager指的是在集群中获取资源的外部服务，目前有三种类型：

1. Standalone，Spark原生的集群资源管理，由Master负责资源的分配
2. Apache Mesos，Spark从Mesos中获取资源
3. Hadoop YARN，RM负责将集群的资源分配给各个应用使用，而资源分配和调度的基本单位是Container

在上图中，基本概念如下:

1. Application，用户编写的Spark应用程序，包括Driver功能代码和分布在集群的多个节点上运行的Executor代码
2. Driver，Spark中的Driver即运行上述Application的main函数，并创建SparkContext，其准备Spark应用程序的运行话，并且负责与CM通信，进行资源申请、任务分配和监控等，当Executor部分运行完毕后，Driver同时负责将SparkContext关闭。
3. Executor，某个Application运行在Worker节点的一个进程，该进程负责运行某些Task，并且负责将数据存到内存或磁盘上，每个Application都有各自独立的Executor，在Spark On Yarn模式下，其进程名称为CoarseGrainedExecutor Backend，其有且仅有一个Executor对象，负责将Task包装成TaskRunner，并从线程池中抽取一个空闲线程运行Task。
4. Worker，集群中任务可以运行Application代码的节点，在Standalone模式中指的是通过slave文件配置的worker节点，在Spark on Yarn模式下就是NodeManager节点
5. Master，总控进程，Spark Standalone运行模式下的主节点，负责管理和分配集群资源来运行Spark Application

# 1.Spark消息通信原理

在Spark Core中定义了通信框架接口，调用Netty的具体方法（在2.0之前使用Akka）。在框架中以RpcEndPoint和RpcEndpointRef实现了Actor和ActorRef相关动作，其中RpcEndpointRef是RpcEndpoint的引用，在消息通信中消息发送方持有引用RpcEndpointRef，相关的关系如下图所示：



通信框架使用了工厂设计模式实现，这种设计方法对Netty解耦，能够根据需要引入其他的消息通信工具。在各模块使用中，如Master,Worker等，会先使用RpcEnv的静态方法创建RpcEnv实例，实现实例化Master，由于Master继承于ThreadSafeRpcEndpoint，创建的Master实例时一个线程Safe的Endpoint，接着调用RpcEnv启动Endpoint方法，把Master的Endpoint和其对应的引用注册到RpcEnv中。在消息通信中，其他对象只要获取Master终端点的引用，就能够发送消息给Master进行通信。

下面是Master.scala类的startRpcEnvAndEndpoint方法中，启动消息通信框架的代码：

*def startRpcEnvAndEndpoint(*

*host: String,*

*port: Int,*

*webUiPort: Int,*

*conf: SparkConf): (RpcEnv, Int, Option[Int]) = {*

*val securityMgr = new SecurityManager(conf)*

*val rpcEnv = RpcEnv.create(SYSTEM\_NAME, host, port, conf, securityMgr)*

*val masterEndpoint = rpcEnv.setupEndpoint(ENDPOINT\_NAME,*

*new Master(rpcEnv, rpcEnv.address, webUiPort, securityMgr, conf))*

*val portsResponse = masterEndpoint.askSync[BoundPortsResponse](BoundPortsRequest)*

*(rpcEnv, portsResponse.webUIPort, portsResponse.restPort)*

*}*

Spark运行过程中Master,Driver,Worker以及Executor等模块之间由事件驱动消息的发送。下面分析Spark On Yarn运行架构下，Spark启动过程和应用程序运行过程是如何进行通信。

# 2.Spark消息通信

## 2.1 启动消息通信

Spark在启动过程中主要进行Master和Worker之间的通信，其消息发送关系如下图所示：



Worker启动后向Master发送注册消息，然后Master处理完毕后，返回注册成功消息或失败消息，如果成功注册，则Worker定时发送心跳消息给Master。具体过程如下：

1. 当Master启动后，随着启动Worker，Worker启动时会创建RpcEnv和WorkerEndpoint，并向Master发送注册Worker的消息RegisterWorker。调用的方法为：

*private def tryRegisterAllMasters(): Array[JFuture[\_]] = {*

*masterRpcAddresses.map { masterAddress =>*

*registerMasterThreadPool.submit(new Runnable {*

*override def run(): Unit = {*

*try {*

*val masterEndpoint = rpcEnv.setupEndpointRef(masterAddress, Master.ENDPOINT\_NAME)*

*sendRegisterMessageToMaster(masterEndpoint)*

*} catch {*

*case ie: InterruptedException => // Cancelled*

*case NonFatal(e) => logWarning(s"Failed to connect to master $masterAddress", e)*

*}*

*}})}}*

1. Master收到消息后，需要对Worker发送的信息进行验证、记录。如果注册成功，则发送RegisteredWorker消息给对应的Worker，告诉Worker已经完成注册，随着进行步骤3，即Worker定期发送心跳给Master。在Master中，接收到Worker注册消息后，先判断Master当前状态是否处于STANDBY，如果是则忽略该消息。判断完毕后使用registerWorker方法把该Worker加入到列表中，用于集群进行处理任务时进行调度。Master.receiveAndReply方法中注册Worker的代码如下所示：

*private def registerWorker(worker: WorkerInfo): Boolean = {*

*workers.filter { w =>*

*(w.host == worker.host && w.port == worker.port) && (w.state == WorkerState.DEAD)*

*}.foreach { w =>*

*workers -= w*

*}*

*val workerAddress = worker.endpoint.address*

*if (addressToWorker.contains(workerAddress)) {*

*val oldWorker = addressToWorker(workerAddress)*

*if (oldWorker.state == WorkerState.UNKNOWN) {*

*// A worker registering from UNKNOWN implies that the worker was restarted during recovery.*

*// The old worker must thus be dead, so we will remove it and accept the new worker.*

*removeWorker(oldWorker, "Worker replaced by a new worker with same address")*

*} else {*

*logInfo("Attempted to re-register worker at same address: " + workerAddress)*

*return false*

*}*

*}*

*workers += worker*

*idToWorker(worker.id) = worker*

*addressToWorker(workerAddress) = worker*

*true*

*}*

1. Worker收到注册成功后，会定时发送心跳信息Heartbeat给Master，以便Master了解Worker的实时状态。间隔时间在spark.worker.timeout中设置。当Worker获取注册成功消息后，先记录日志并更新Master信息，然后启动定时调度进程发送心跳信息。

*case RegisteredWorker(masterRef, masterWebUiUrl, masterAddress) =>*

*if (preferConfiguredMasterAddress) {*

*logInfo("Successfully registered with master " + masterAddress.toSparkURL)*

*} else {*

*logInfo("Successfully registered with master " + masterRef.address.toSparkURL)*

*}*

*registered = true*

*changeMaster(masterRef, masterWebUiUrl, masterAddress)*

*forwordMessageScheduler.scheduleAtFixedRate(new Runnable {*

*override def run(): Unit = Utils.tryLogNonFatalError {*

*self.send(SendHeartbeat)*

*}*

*}, 0, HEARTBEAT\_MILLIS, TimeUnit.MILLISECONDS)*

*if (CLEANUP\_ENABLED) {*

*logInfo(*

*s"Worker cleanup enabled; old application directories will be deleted in: $workDir")*

*forwordMessageScheduler.scheduleAtFixedRate(new Runnable {*

*override def run(): Unit = Utils.tryLogNonFatalError {*

*self.send(WorkDirCleanup)*

*}*

*}, CLEANUP\_INTERVAL\_MILLIS, CLEANUP\_INTERVAL\_MILLIS, TimeUnit.MILLISECONDS)*

*}*

*val execs = executors.values.map { e =>*

*new ExecutorDescription(e.appId, e.execId, e.cores, e.state)*

*}*

*masterRef.send(WorkerLatestState(workerId, execs.toList, drivers.keys.toSeq))*

*case RegisterWorkerFailed(message) =>*

*if (!registered) {*

*logError("Worker registration failed: " + message)*

*System.exit(1)*

*}*

*case MasterInStandby =>*

*// Ignore. Master not yet ready.*

*}*

## 2.2 Spark运行时消息通信

用户提交应用程序时，应用程序的SparkContext会向Master发送应用注册信息，并由Master给该应用分配Executor，Executor启动后，会向SparkContext发送注册成功消息。当SparkContext的RDD触发行动操作后，将创建RDD的DAG，通过DAGScheduler进行划分Stage，并将Stage转换为TaskSet；接着由TaskScheduler向注册的Executor发送执行消息，Executor接收到任务消息后启动并运行；最后当所有任务运行时，由Driver处理结果并回收资源，下图是Spark运行消息通信的交互过程：



1. 执行应用程序需启动SparkContext，在SparkContext启动过程中会先实例SchedulerBack-

end对象，在独立运行（Standalone）模式中实际创建的是SparkDeploySchedulerBackend对象，在该对象的启动中会继承父类DriverEndpoint和创建AppClient的ClientEndpoint的两个终端点。

在ClientEndpoint的tryRegisterAllMasters方法中创建注册线程池registerMasterThreadPool，在该线程池中启动线程池并向Master发送RegisterApplication注册应用的消息，代码如下：

*private def tryRegisterAllMasters(): Array[JFuture[\_]] = {*

*for (masterAddress <- masterRpcAddresses) yield {*

*registerMasterThreadPool.submit(new Runnable {*

*override def run(): Unit = try {*

*if (registered.get) {*

*return*

*}*

*logInfo("Connecting to master " + masterAddress.toSparkURL + "...")*

*val masterRef = rpcEnv.setupEndpointRef(masterAddress, Master.ENDPOINT\_NAME)*

*masterRef.send(RegisterApplication(appDescription, self))*

*} catch {*

*case ie: InterruptedException => // Cancelled*

*case NonFatal(e) => logWarning(s"Failed to connect to master $masterAddress", e)*

*}})}}*

当Master接收到注册应用的消息时，在registerApplication方法中记录应用信息并把该应用加入到等待运行应用列表中，注册完毕后发送成功消息RegisteredApplication给ClientEndpoint，同时调用startExecutorsOnWorkers方法运行应用。在执行前需要获取应用的Worker。其中Master.startExecutorsOnWorkers方法代码如下：

*private def startExecutorsOnWorkers(): Unit = {*

*// Right now this is a very simple FIFO scheduler. We keep trying to fit in the first app*

*// in the queue, then the second app, etc.*

*for (app <- waitingApps) {*

*val coresPerExecutor = app.desc.coresPerExecutor.getOrElse(1)*

*// If the cores left is less than the coresPerExecutor,the cores left will not be allocated*

*if (app.coresLeft >= coresPerExecutor) {*

*// Filter out workers that don't have enough resources to launch an executor*

*val usableWorkers = workers.toArray.filter(\_.state == WorkerState.ALIVE)*

*.filter(worker => worker.memoryFree >= app.desc.memoryPerExecutorMB &&*

*worker.coresFree >= coresPerExecutor)*

*.sortBy(\_.coresFree).reverse*

*val assignedCores = scheduleExecutorsOnWorkers(app, usableWorkers, spreadOutApps)*

*// Now that we've decided how many cores to allocate on each worker, let's allocate them*

*for (pos <- 0 until usableWorkers.length if assignedCores(pos) > 0) {*

*allocateWorkerResourceToExecutors(*

*app, assignedCores(pos), app.desc.coresPerExecutor, usableWorkers(pos))*

*}*

*}*

*}*

*}*

1. AppClient.ClientEndpoint接收到Master发送的RegisteredApplication消息，需要将注册标识registered置为true，Master注册线程获取状态变换后，完成注册Application进程，代码如下：

*case RegisteredApplication(appId\_, masterRef) =>*

*appId.set(appId\_)*

*registered.set(true)*

*master = Some(masterRef)*

*listener.connected(appId.get)*

1. 在Master类的startExecutorsOnWorkers方法中分配资源应用程序，调用allocateWorkerResourceToExecutors方法实现Worker中启动Executor。当Worker收到Master发送过来的LaunchExecutor消息，先实例化ExecutorRunner对象，在ExecutorRunner启动中会创建进程生成器ProcessBuilder，然后由该生成器使用command创建CoarseGrainedExecutorBackend对象，该对象是Executor运行的容器，最后Worker发送ExecutorStateChanged消息给Master，通知Executor容器已经创建完毕

*case LaunchExecutor(masterUrl, appId, execId, appDesc, cores\_, memory\_) =>*

*if (masterUrl != activeMasterUrl) {*

*logWarning("Invalid Master (" + masterUrl + ") attempted to launch executor.")*

*} else {*

*try {*

*logInfo("Asked to launch executor %s/%d for %s".format(appId, execId, appDesc.name))*

*// Create the executor's working directory*

*val executorDir = new File(workDir, appId + "/" + execId)*

*if (!executorDir.mkdirs()) {*

*throw new IOException("Failed to create directory " + executorDir)*

*}*

*// Create local dirs for the executor. These are passed to the executor via the*

*// SPARK\_EXECUTOR\_DIRS environment variable, and deleted by the Worker when the*

*// application finishes.*

*val appLocalDirs = appDirectories.getOrElse(appId, {*

*val localRootDirs = Utils.getOrCreateLocalRootDirs(conf)*

*val dirs = localRootDirs.flatMap { dir =>*

*try {*

*val appDir = Utils.createDirectory(dir, namePrefix = "executor")*

*Utils.chmod700(appDir)*

*Some(appDir.getAbsolutePath())*

*} catch {*

*case e: IOException =>*

*logWarning(s"${e.getMessage}. Ignoring this directory.")*

*None*

*}*

*}.toSeq*

*if (dirs.isEmpty) {*

*throw new IOException("No subfolder can be created in " +*

*s"${localRootDirs.mkString(",")}.")*

*}*

*dirs*

*})*

*appDirectories(appId) = appLocalDirs*

*val manager = new ExecutorRunner(*

*appId,*

*execId,*

*appDesc.copy(command = Worker.maybeUpdateSSLSettings(appDesc.command, conf)),*

*cores\_,*

*memory\_,*

*self,*

*workerId,*

*host,*

*webUi.boundPort,*

*publicAddress,*

*sparkHome,*

*executorDir,*

*workerUri,*

*conf,*

*appLocalDirs, ExecutorState.RUNNING)*

*executors(appId + "/" + execId) = manager*

*manager.start()*

*coresUsed += cores\_*

*memoryUsed += memory\_*

*sendToMaster(ExecutorStateChanged(appId, execId, manager.state, None, None))*

*}*

在ExecutorRunner创建中调用了fetchAndRunExecutor方法进行实现，在该方法中构建command，指定构造Executor运行容器CoarseGrainedExecutorBackend，其中创建过程代码如下：

*private def fetchAndRunExecutor() {*

*try {*

*// Launch the process*

*val builder = CommandUtils.buildProcessBuilder(appDesc.command, new SecurityManager(conf),*

*memory, sparkHome.getAbsolutePath, substituteVariables)*

*val command = builder.command()*

*val formattedCommand = command.asScala.mkString("\"", "\" \"", "\"")*

*logInfo(s"Launch command: $formattedCommand")*

*builder.directory(executorDir)*

*builder.environment.put("SPARK\_EXECUTOR\_DIRS", appLocalDirs.mkString(File.pathSeparator))*

*// In case we are running this from within the Spark Shell, avoid creating a "scala"*

*// parent process for the executor command*

*builder.environment.put("SPARK\_LAUNCH\_WITH\_SCALA", "0")*

*// Add webUI log urls*

*val baseUrl =*

*if (conf.getBoolean("spark.ui.reverseProxy", false)) {*

*s"/proxy/$workerId/logPage/?appId=$appId&executorId=$execId&logType="*

*} else {*

*s"http://$publicAddress:$webUiPort/logPage/?appId=$appId&executorId=$execId&logType="*

*}*

*builder.environment.put("SPARK\_LOG\_URL\_STDERR", s"${baseUrl}stderr")*

*builder.environment.put("SPARK\_LOG\_URL\_STDOUT", s"${baseUrl}stdout")*

*process = builder.start()*

*val header = "Spark Executor Command: %s\n%s\n\n".format(*

*formattedCommand, "=" \* 40)*

*// Redirect its stdout and stderr to files*

*val stdout = new File(executorDir, "stdout")*

*stdoutAppender = FileAppender(process.getInputStream, stdout, conf)*

*val stderr = new File(executorDir, "stderr")*

*Files.write(header, stderr, StandardCharsets.UTF\_8)*

*stderrAppender = FileAppender(process.getErrorStream, stderr, conf)*

*// Wait for it to exit; executor may exit with code 0 (when driver instructs it to shutdown)*

*// or with nonzero exit code*

*val exitCode = process.waitFor()*

*state = ExecutorState.EXITED*

*val message = "Command exited with code " + exitCode*

*worker.send(ExecutorStateChanged(appId, execId, state, Some(message), Some(exitCode)))*

*} catch {*

*case interrupted: InterruptedException =>*

*logInfo("Runner thread for executor " + fullId + " interrupted")*

*state = ExecutorState.KILLED*

*killProcess(None)*

*case e: Exception =>*

*logError("Error running executor", e)*

*state = ExecutorState.FAILED*

*killProcess(Some(e.toString))*

*}*

*}*

1. Master接收到Worker发送的ExecutorStateChanged消息，根据ExecuteState

*case StatusUpdate(executorId, taskId, state, data) =>*

*scheduler.statusUpdate(taskId, state, data.value)*

*if (TaskState.isFinished(state)) {*

*executorDataMap.get(executorId) match {*

*case Some(executorInfo) =>*

*executorInfo.freeCores += scheduler.CPUS\_PER\_TASK*

*makeOffers(executorId)*

*case None =>*

*// Ignoring the update since we don't know about the executor.*

*logWarning(s"Ignored task status update ($taskId state $state) " +*

*s"from unknown executor with ID $executorId")*

*}*

*}*

1. 在步骤3中的CoarseGrainedExecutorBackend启动方法onStart中，会发送Executor消息RegisterExecutor给DriverEndpoint，先判断Executor是否已经注册，如果已经存在发送注册失败RegisterExecutorFailed消息，否则Driver终端点会记录该Executor信息，发送注册成功RegisteredExecutor消息，在makeOffers方法中分配运行任务的资源，最后发送LaunchTask消息执行任务。其中在DriverEndpoint进行注册Executor的过程如下：

*case RegisterExecutor(executorId, executorRef, hostname, cores, logUrls) =>*

*if (executorDataMap.contains(executorId)) {*

*executorRef.send(RegisterExecutorFailed("Duplicate executor ID: " + executorId))*

*context.reply(true)*

*} else if (scheduler.nodeBlacklist != null &&*

*scheduler.nodeBlacklist.contains(hostname)) {*

*// If the cluster manager gives us an executor on a blacklisted node (because it*

*// already started allocating those resources before we informed it of our blacklist,*

*// or if it ignored our blacklist), then we reject that executor immediately.*

*logInfo(s"Rejecting $executorId as it has been blacklisted.")*

*executorRef.send(RegisterExecutorFailed(s"Executor is blacklisted: $executorId"))*

*context.reply(true)*

*} else {*

*// If the executor's rpc env is not listening for incoming connections, `hostPort`*

*// will be null, and the client connection should be used to contact the executor.*

*val executorAddress = if (executorRef.address != null) {*

*executorRef.address*

*} else {*

*context.senderAddress*

*}*

*logInfo(s"Registered executor $executorRef ($executorAddress) with ID $executorId")*

*addressToExecutorId(executorAddress) = executorId*

*totalCoreCount.addAndGet(cores)*

*totalRegisteredExecutors.addAndGet(1)*

*val data = new ExecutorData(executorRef, executorRef.address, hostname,*

*cores, cores, logUrls)*

*// This must be synchronized because variables mutated*

*// in this block are read when requesting executors*

*CoarseGrainedSchedulerBackend.this.synchronized {*

*executorDataMap.put(executorId, data)*

*if (currentExecutorIdCounter < executorId.toInt) {*

*currentExecutorIdCounter = executorId.toInt*

*}*

*if (numPendingExecutors > 0) {*

*numPendingExecutors -= 1*

*logDebug(s"Decremented number of pending executors ($numPendingExecutors left)")*

*}*

*}*

*executorRef.send(RegisteredExecutor)*

*// Note: some tests expect the reply to come after we put the executor in the map*

*context.reply(true)*

*listenerBus.post(*

*SparkListenerExecutorAdded(System.currentTimeMillis(), executorId, data))*

*makeOffers()*

*}*

6） 当CoarsedGrainedExecutorBackend接收到Executor注册成功RegisteredExecutor消息时，在CoarsedGrainedExecutorBackend容器中实例化Executor对象。启动完毕后，会定时向Driver发送心跳信息，等待接收从DriverEndpoint终端点发送执行任务的消息，CoarsedGrainedExeuctorBackend处理注册成功代码如下：

*case RegisteredExecutor =>*

*logInfo("Successfully registered with driver")*

*try {*

*executor = new Executor(executorId, hostname, env, userClassPath, isLocal = false)*

*} catch {*

*case NonFatal(e) =>*

*exitExecutor(1, "Unable to create executor due to " + e.getMessage, e)*

*}*

该Exeuctor会定时向Driver发送心跳信息，等待Driver下发任务：

*val intervalMs = conf.getTimeAsMs("spark.executor.heartbeatInterval", "10s")*

*// Wait a random interval so the heartbeats don't end up in sync*

*val initialDelay = intervalMs + (math.random \* intervalMs).asInstanceOf[Int]*

*val heartbeatTask = new Runnable() {*

*override def run(): Unit = Utils.logUncaughtExceptions(reportHeartBeat())*

*}*

*heartbeater.scheduleAtFixedRate(heartbeatTask, initialDelay, intervalMs, TimeUnit.MILLISECONDS)*

*}*

7）CoarsedGrainedExecutorBackEnd的Executor启动后，接收从DriverEndpoint终端点发送LaunchTask执行任务消息，任务执行时在Executor的launchTask方法实现的。在执行时会创建TaskRunner进程，由该进程进行任务的处理，处理完毕后发送StatusUpdate消息返回给CoarseGrainedExecutorBackend。

*private def launchTasks(tasks: Seq[Seq[TaskDescription]]) {*

*for (task <- tasks.flatten) {*

*val serializedTask = TaskDescription.encode(task)*

*if (serializedTask.limit >= maxRpcMessageSize) {*

*scheduler.taskIdToTaskSetManager.get(task.taskId).foreach { taskSetMgr =>*

*try {*

*var msg = "Serialized task %s:%d was %d bytes, which exceeds max allowed: " +*

*"spark.rpc.message.maxSize (%d bytes). Consider increasing " +*

*"spark.rpc.message.maxSize or using broadcast variables for large values."*

*msg = msg.format(task.taskId, task.index, serializedTask.limit, maxRpcMessageSize)*

*taskSetMgr.abort(msg)*

*} catch {*

*case e: Exception => logError("Exception in error callback", e)*

*}*

*}*

*}*

*else {*

*val executorData = executorDataMap(task.executorId)*

*executorData.freeCores -= scheduler.CPUS\_PER\_TASK*

*logDebug(s"Launching task ${task.taskId} on executor id: ${task.executorId} hostname: " +*

*s"${executorData.executorHost}.")*

*executorData.executorEndpoint.send(LaunchTask(new SerializableBuffer(serializedTask)))*

*}*

*}*

*}*

调用Executor的launchTask方法，在该方法中创建TaskRunner进程，然后把该进程加入到执行池threadPool中，由Executor统一调度：

*def launchTask(context: ExecutorBackend, taskDescription: TaskDescription): Unit = {*

*val tr = new TaskRunner(context, taskDescription)*

*runningTasks.put(taskDescription.taskId, tr)*

*threadPool.execute(tr)*

*}*

8）在TaskRunner执行任务完成后，会由DriverEndpoint终端点发送状态变更StatusUpdate消息，当DriverEndpoint接收到该消息时，调用TaskSchedulerImpl的statusUpdate方法，根据任务执行不同的结果进行处理，处理完毕后再给该Executor分配执行任务。其中在DriverEndpoint终端点处理状态变更代码如下所示：

*case StatusUpdate(executorId, taskId, state, data) =>*

*scheduler.statusUpdate(taskId, state, data.value)*

*if (TaskState.isFinished(state)) {*

*executorDataMap.get(executorId) match {*

*case Some(executorInfo) =>*

*executorInfo.freeCores += scheduler.CPUS\_PER\_TASK*

*makeOffers(executorId)*

*case None =>*

*// Ignoring the update since we don't know about the executor.*

*logWarning(s"Ignored task status update ($taskId state $state) " +*

*s"from unknown executor with ID $executorId")*

*}*

*}*

## 2.3. Spark On Yarn消息通信

Spark On yarn的配置比较简单，只需要将HADOOP\_CONF\_DIR配置到spark-env.sh中，让Spark获取YARN的配置信息即可,程序提交到YARN的命令格式如下：

*bin/spark-submit --class org.apache.spark.examples.SparkPi --master yarn --deploy-mode cluster lib/spark-examples-1.6.2-bc1.3.5-hadoop2.6.0-bc1.3.5.jar 10*

命令执行后，会通过YARN Client启动ApplicationMaster，SparkPi作为ApplicationMaster的子线程，Yarn Client周期性的从AM中获取状态并在控制界面上显示。Spark AM实例化SparkContext，并且向RM申请资源，在NM中的Container中启动CoarsGrainedExecutor等，消息通信图如下所示：



1. 由Client向RM提交应用请求，创建YarnClient，上传app 及spark-assembly 等jar

构建ApplicationSubmissionContext，并提交应用

1. RM接收到应用创建后，向NodeManager申请资源，并启动Spark ApplicationMaster。在Spark ApplicationMaster中启动SparkContext，代码如下：

*private def runDriver(securityMgr: SecurityManager): Unit = {*

*try {*

*val sc = ThreadUtils.awaitResult(sparkContextPromise.future,*

*Duration(totalWaitTime, TimeUnit.MILLISECONDS))*

*if (sc != null) {*

*rpcEnv = sc.env.rpcEnv*

*val driverRef = createSchedulerRef(*

*sc.getConf.get("spark.driver.host"),*

*sc.getConf.get("spark.driver.port"))*

*registerAM(sc.getConf, rpcEnv, driverRef, sc.ui.map(\_.webUrl), securityMgr)*

*registered = true*

*} else {*

*// Sanity check; should never happen in normal operation, since sc should only be null*

*// if the user app did not create a SparkContext.*

*throw new IllegalStateException("User did not initialize spark context!")*

*}*

*userClassThread.join()*

*}*

*}*

在SparkContext中由YarnSchedulerBackend完成Task的分发。在Backend中，DriverEndpont与Executor之间进行交互。

1. Spark App Master启动后，向ResourceManager ASM注册，并启动DAGScheduler和YARN Cluster Scheduler
2. AM向RM申请资源，分配Container后，启动CoarseGrainedExecutorBackend
3. AM中的YarnSchedulerBackend与CoarseGrainedExecutorBackend，通过Endpoint进行交互，两端的Endpoint分别是YarnDriverEndpoint和Executor Endpoint。AMEndpoint接收到CoarseGrainedExecutorBackend的接收到事件后：

*override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {*

*case r: RequestExecutors =>*

*amEndpoint match {*

*case Some(am) =>*

*am.ask[Boolean](r).andThen {*

*case Success(b) => context.reply(b)*

*case Failure(NonFatal(e)) =>*

*logError(s"Sending $r to AM was unsuccessful", e)*

*context.sendFailure(e)*

*}(ThreadUtils.sameThread)*

*case None =>*

*logWarning("Attempted to request executors before the AM has registered!")*

*context.reply(false)*

*}*

*case k: KillExecutors =>*

*amEndpoint match {*

*case Some(am) =>*

*am.ask[Boolean](k).andThen {*

*case Success(b) => context.reply(b)*

*case Failure(NonFatal(e)) =>*

*logError(s"Sending $k to AM was unsuccessful", e)*

*context.sendFailure(e)*

*}(ThreadUtils.sameThread)*

*case None =>*

*logWarning("Attempted to kill executors before the AM has registered!")*

*context.reply(false)*

*}*

*case RetrieveLastAllocatedExecutorId =>*

*context.reply(currentExecutorIdCounter)*

*}*

ApplicationMaster通过YarnAllocator进行Container启动，allocator的启动在registerAM中完成：

*allocator = client.register(driverUrl,*

*driverRef,*

*yarnConf,*

*\_sparkConf,*

*uiAddress,*

*historyAddress,*

*securityMgr,*

*localResources)*

*rpcEnv.setupEndpoint("YarnAM", new AMEndpoint(rpcEnv, driverRef))*

*allocator.allocateResources()*

allocator.allocateResources中分配资源：

*allocateResponse =amClient.allocate(progressIndicator) amClient => AMRMClient*

*allocatedContainers = allocateResponse.getAllocatedContainers()*

Containers的启动调用方法runAllocatedContainers

*private def runAllocatedContainers(containersToUse: ArrayBuffer[Container]): Unit = {*

*for (container <- containersToUse) {*

*executorIdCounter += 1*

*val executorHostname = container.getNodeId.getHost*

*val containerId = container.getId*

*val executorId = executorIdCounter.toString*

*def updateInternalState(): Unit = synchronized {*

*numExecutorsRunning.incrementAndGet()*

*numExecutorsStarting.decrementAndGet()*

*executorIdToContainer(executorId) = container*

*containerIdToExecutorId(container.getId) = executorId*

*val containerSet = allocatedHostToContainersMap.getOrElseUpdate(executorHostname,*

*new HashSet[ContainerId])*

*containerSet += containerId*

*allocatedContainerToHostMap.put(containerId, executorHostname)*

*}*

*if (numExecutorsRunning.get < targetNumExecutors) {*

*numExecutorsStarting.incrementAndGet()*

*if (launchContainers) {*

*launcherPool.execute(new Runnable {*

*override def run(): Unit = {*

*try {*

*new ExecutorRunnable(*

*Some(container),*

*conf,*

*sparkConf,*

*driverUrl,*

*executorId,*

*executorHostname,*

*executorMemory,*

*executorCores,*

*appAttemptId.getApplicationId.toString,*

*securityMgr,*

*localResources*

*).run()*

*updateInternalState()*

*}*

*}*

ExecutorRunnable中startContainer，在NM中启动Container，

*def startContainer(): java.util.Map[String, ByteBuffer] = {*

*val ctx = Records.newRecord(classOf[ContainerLaunchContext])*

*.asInstanceOf[ContainerLaunchContext]*

*val env = prepareEnvironment().asJava //构建环境参数*

*ctx.setLocalResources(localResources.asJava) //资源*

*ctx.setEnvironment(env)*

*val credentials = UserGroupInformation.getCurrentUser().getCredentials()*

*val dob = new DataOutputBuffer()*

*credentials.writeTokenStorageToStream(dob)*

*ctx.setTokens(ByteBuffer.wrap(dob.getData()))*

*val commands = prepareCommand() //构建Container启动命令*

*ctx.setCommands(commands.asJava)*

*ctx.setApplicationACLs(*

*YarnSparkHadoopUtil.getApplicationAclsForYarn(securityMgr).asJava)*

*if (sparkConf.get(SHUFFLE\_SERVICE\_ENABLED)) {*

*val secretString = securityMgr.getSecretKey()*

*val secretBytes =*

*if (secretString != null) {*

*// This conversion must match how the YarnShuffleService decodes our secret*

*JavaUtils.stringToBytes(secretString)*

*} else {*

*// Authentication is not enabled, so just provide dummy metadata*

*ByteBuffer.allocate(0)*

*}*

*ctx.setServiceData(Collections.singletonMap("spark\_shuffle", secretBytes))*

*}*

*// Send the start request to the ContainerManager*

*try {*

*nmClient.startContainer(container.get, ctx) nmClient =>NMClient*

*} catch {*

*case ex: Exception =>*

*throw new SparkException(s"Exception while starting container ${container.get.getId}" +*

*s" on host $hostname", ex)*

*}*

*}*

命令如下：

*private def prepareCommand(): List[Strin g] = {*

*// Extra options for the JVM*

*val javaOpts = ListBuffer[String]()*

*var prefixEnv: Option[String] = None*

*// Set the JVM memory*

*val executorMemoryString = executorMemory + "m"*

*javaOpts += "-Xmx" + executorMemoryString*

*// Set extra Java options for the executor, if defined*

*sparkConf.get(EXECUTOR\_JAVA\_OPTIONS).foreach { opts =>*

*javaOpts ++= Utils.splitCommandString(opts).map(YarnSparkHadoopUtil.escapeForShell)*

*}*

*sparkConf.get(EXECUTOR\_LIBRARY\_PATH).foreach { p =>*

*prefixEnv = Some(Client.getClusterPath(sparkConf, Utils.libraryPathEnvPrefix(Seq(p))))*

*}*

*javaOpts += "-Djava.io.tmpdir=" +*

*new Path(Environment.PWD.$$(), YarnConfiguration.DEFAULT\_CONTAINER\_TEMP\_DIR)*

*sparkConf.getAll*

*.filter { case (k, v) => SparkConf.isExecutorStartupConf(k) }*

*.foreach { case (k, v) => javaOpts += YarnSparkHadoopUtil.escapeForShell(s"-D$k=$v") }*

*javaOpts += ("-Dspark.yarn.app.container.log.dir=" + ApplicationConstants.LOG\_DIR\_EXPANSION\_VAR)*

*val userClassPath = Client.getUserClasspath(sparkConf).flatMap { uri =>*

*val absPath =*

*if (new File(uri.getPath()).isAbsolute()) {*

*Client.getClusterPath(sparkConf, uri.getPath())*

*} else {*

*Client.buildPath(Environment.PWD.$(), uri.getPath())*

*}*

*Seq("--user-class-path", "file:" + absPath)*

*}.toSeq*

*YarnSparkHadoopUtil.addOutOfMemoryErrorArgument(javaOpts)*

*val commands = prefixEnv ++*

*Seq(Environment.JAVA\_HOME.$$() + "/bin/java", "-server") ++*

*javaOpts ++*

*Seq("org.apache.spark.executor.CoarseGrainedExecutorBackend",*

*"--driver-url", masterAddress,*

*"--executor-id", executorId,*

*"--hostname", hostname,*

*"--cores", executorCores.toString,*

*"--app-id", appId) ++*

*userClassPath ++*

*Seq(*

*s"1>${ApplicationConstants.LOG\_DIR\_EXPANSION\_VAR}/stdout",*

*s"2>${ApplicationConstants.LOG\_DIR\_EXPANSION\_VAR}/stderr")*

*// TODO: it would be nicer to just make sure there are no null commands here*

*commands.map(s => if (s == null) "null" else s).toList*

*}*

launcher.sh中的启动命令如下：

*/bin/bash -c /usr/jdk64/jdk1.7.0\_67/bin/java -server*

*-XX:OnOutOfMemoryError='kill %p' -Xms1024m -Xmx1024m*

*-Djava.io.tmpdir=../container\_e08\_1506347582099\_0012\_01\_000004/tmp*

*'-Dspark.history.ui.port=18080' '-Dspark.driver.port=43526'*

*-Dspark.yarn.app.container.log.dir=../container\_e08\_1506347582099\_0012\_01\_000004*

*-XX:MaxPermSize=256m org.apache.spark.executor.CoarseGrainedExecutorBackend*

*--driver-url spark://CoarseGrainedScheduler@10.139.9.123:43526*

*--executor-id 2 --hostname fys1.cmss.com*

*--cores 1*

*--app-id application\_1506347582099\_0012*

*--user-class-path file:.. /container\_e08\_1506347582099\_0012\_01\_000004/\_\_app\_\_.jar 1>*

*../container\_e08\_1506347582099\_0012\_01\_000004/stdout*

*2> ../container\_e08\_1506347582099\_0012\_01\_000004/stderr*