Spark on Yarn

YARN运行模式根据Driver在集群中的位置分为两种模式：YARN-Client模式和YARN-Cluster模式。其启动过程如下所示：

工作流程分为以下步骤，流程如下图所示：



1. 启动应用程序后，在SparkContext启动过程中，初始化DAGScheduler调度器，使用反射方法初始化YarnScheduler和YarnClientSchedulerBackend在内部启动终端点DriverEndpoint和Client，然后Client向Yarn集群的RM申请并启动Application Master
2. RM收到请求后，在集群中选择一个NM，为该程序启动分配一个Container，并启动AM。与YARN Cluster区别是在Application Master不运行Spark Context，只与SparkContext进行联系进行资源分派。
3. 在客户端SparkContext启动完毕后，与AM建立通信，AM向RM注册，根据任务信息向RM申请资源
4. 一旦AM申请到资源，并与NM通信，在Container中启动CoarseGrainedExecutorBackEnd，其启动后向客户端中的SparkContext注册并申请任务集
5. SparkContext分配任务集给CoarseGrainedExecutorBackend执行，在执行过程中向DriverEndpoint汇报运行的状态和进度，让客户端随时掌握各个任务的运行状态，从而可以在任务失败时重启启动任务
6. 应用程序运行完成后，客户端SparkContext向RM申请注销并关闭自身。

# YARN-Client运行模式

前面对YARN-Client运行模式的运行流程进行了分析，下面将根据运行流程的步骤分析其代码实现原理，具体如下：

1. 在SparkContext启动时，初始化DAGScheduler调度器，然后创建createTaskScheduler方法中匹配为YARN-Client运行模式时，通过反射方法初始化YarnScheduler和YarnClientSchedulerBackend两个对象，其中YarnClientSchedulerBackend类是CoarseGrainedSchedulerBackend类的子类，YarnScheduler是TaskSchedulerImpl的子类，仅仅重写了TaskSchedulerImpl的getRackForHost方法，在SparkContext中createTaskScheduler方法中，匹配模式代码及初始化如下：

*master match {*

*// Local模式，调用TaskSchedulerImpl*

*case "local" =>*

*case LOCAL\_N\_REGEX(threads) =>*

*case LOCAL\_N\_FAILURES\_REGEX(threads, maxFailures) =>*

*val threadCount = if (threads == "\*") localCpuCount else threads.toInt*

*val scheduler = new TaskSchedulerImpl(sc, maxFailures.toInt, isLocal = true)*

*val backend = new LocalSchedulerBackend(sc.getConf, scheduler, threadCount)*

*//StandAlone模式，TaskSchedulerImpl*

*case SPARK\_REGEX(sparkUrl) =>*

*val scheduler = new TaskSchedulerImpl(sc)*

*val masterUrls = sparkUrl.split(",").map("spark://" + \_)*

*val backend = new StandaloneSchedulerBackend(scheduler, sc, masterUrls)*

*scheduler.initialize(backend)*

*(backend, scheduler)*

*//根据masterUrl，判断ClusterMaster是YARN还是Mesos*

*case masterUrl =>*

*val cm = getClusterManager(masterUrl) match {*

*case Some(clusterMgr) => clusterMgr*

*case None => throw new SparkException("Could not parse Master URL: '" + master + "'")*

*}*

*try {*

*//使用ClusterMaster创建TaskScheduler*

*val scheduler = cm.createTaskScheduler(sc, masterUrl)*

*val backend = cm.createSchedulerBackend(sc, masterUrl, scheduler)*

*cm.initialize(scheduler, backend)*

*(backend, scheduler)*

*} catch {}*

*}*

其中YarnClusterManager中createTaskScheduler，源码如下：

*override def createTaskScheduler(sc: SparkContext, masterURL: String): TaskScheduler = {*

*sc.deployMode match {*

*case "cluster" => new YarnClusterScheduler(sc)*

*case "client" => new YarnScheduler(sc)*

*case \_ => throw new SparkException(s"Unknown deploy mode '${sc.deployMode}' for Yarn")*

*}*

*}*

其中createSchedulerBackend的，源码如下：

*override def createSchedulerBackend(sc: SparkContext,*

*masterURL: String,*

*scheduler: TaskScheduler): SchedulerBackend = {*

*sc.deployMode match {*

*case "cluster" =>*

*new YarnClusterSchedulerBackend(scheduler.asInstanceOf[TaskSchedulerImpl], sc)*

*case "client" =>*

*new YarnClientSchedulerBackend(scheduler.asInstanceOf[TaskSchedulerImpl], sc)*

*case \_ =>*

*throw new SparkException(s"Unknown deploy mode '${sc.deployMode}' for Yarn")*

*}*

*}*

在YarnClientSchedulerBackend.start方法中先在内部启动Client，然后调用父类start方法启动DriverEndpoint终端点，并通过在submitApplication方法中申请启动ApplicationMaster。



YARN-Client运行模式下应用程序启动类图

其中Client.submitApplication方法中，先向RM确认是否有足够的资源，如果有则构造用于启动ApplicationMaster环境并提交应用程序到YARN集群中。代码如下：

*def submitApplication(): ApplicationId = {var appId: ApplicationId = null*

*try {*

*launcherBackend.connect()*

*setupCredentials()*

*//创建YarnClient，用于和YARN集群进行交互*

*yarnClient.init(yarnConf)*

*yarnClient.start()*

*//向RM申请应用程序编号*

*val newApp = yarnClient.createApplication()*

*val newAppResponse = newApp.getNewApplicationResponse()*

*appId = newAppResponse.getApplicationId()*

*//确认在YARN集群中是否有足够的资源启动AM*

*verifyClusterResources(newAppResponse)*

*// 构造适当的环境用于启动AM*

*val containerContext = createContainerLaunchContext(newAppResponse)*

*val appContext = createApplicationSubmissionContext(newApp, containerContext)*

*//向RM提交并监控应用程序*

*yarnClient.submitApplication(appContext)*

*launcherBackend.setAppId(appId.toString)*

*reportLauncherState(SparkAppHandle.State.SUBMITTED)*

*appId*

*} catch {*

*......*

*}*

*}*

1. 当RM收到请求后，在集群中选择一个NM并启动ExecutorLauncher，在ExecutorLauncher初始化中启动AM，启动ExecutorLauncher是在Client.createContainerContext方法中指定，代码如下：

*val amClass =*

*if (isClusterMode) {*

*Utils.classForName("org.apache.spark.deploy.yarn.ApplicationMaster").getName*

*} else {*

*Utils.classForName("org.apache.spark.deploy.yarn.ExecutorLauncher").getName*

*}*

*}*

1. AM启动完毕后，在registerAM方法中由RM向终端点DriverEndpoint发送消息通知AM已经启动完毕。通过YarnAllocator中的allocateResources方法向RM申请资源，其中AM.registerAM方法如下：

*private def registerAM(\_sparkConf: SparkConf,\_rpcEnv: RpcEnv, driverRef: RpcEndpointRef,*

*uiAddress: Option[String],securityMgr: SecurityManager) = {*

*//获取应用程序和Attempt编号*

*val appId = client.getAttemptId().getApplicationId().toString()*

*val attemptId = client.getAttemptId().getAttemptId().toString()*

*val historyAddress =*

*\_sparkConf.get(HISTORY\_SERVER\_ADDRESS)*

*.map { text => SparkHadoopUtil.get.substituteHadoopVariables(text, yarnConf) }*

*.map { address => s"${address}${HistoryServer.UI\_PATH\_PREFIX}/${appId}/${attemptId}" }*

*.getOrElse("")*

*//获取DriverEndpoint终端点引用地址*

*val driverUrl = RpcEndpointAddress(*

*\_sparkConf.get("spark.driver.host"),*

*\_sparkConf.get("spark.driver.port").toInt,*

*CoarseGrainedSchedulerBackend.ENDPOINT\_NAME).toString*

*// 在RM发送消息通知DriverEndpoint，通知其AM已启动*

*allocator = client.register(driverUrl,*

*driverRef,*

*yarnConf,*

*\_sparkConf,*

*uiAddress,*

*historyAddress,*

*securityMgr,*

*localResources)*

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

*//申请运行Executor资源*

*allocator.allocateResources()*

*reporterThread = launchReporterThread()*

*}*

1. 在YarnAllocator.allocateResources方法中获取可用的Container，然后调用YarnAllocator中runAllocatorContainers方法，在该方法中调用ExecutorRunnable的run方法在Container启动CoarseGrainedExecutorBackend，启动后会向客户端中的SparkContext注册并申请任务集

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

*for (container <- containersToUse) {*

*//更新计数器*

*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*

*//在机器与Container对应列表中加入当前Container信息*

*val containerSet = allocatedHostToContainersMap.getOrElseUpdate(executorHostname,*

*new HashSet[ContainerId])*

*containerSet += containerId*

*allocatedContainerToHostMap.put(containerId, executorHostname)*

*}*

*//在container中实例化ExecutorRunnable，同时启动CoarseGrainedExecutorBackend*

*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()*

*} catch {})} }}*

1. SparkContext中分配任务集给CoarsGrainedExecutorBackend执行并跟踪运行状态，该过程和独立运行模式类似，其作业运行调用关系如图：



1. 应用程序运行完成后，SparkContext向RM申请注销并关闭

# YARN-Cluster运行模式

YARN-Cluster运行模式的流程与YARN-Client的流程基本相同，不同点是SparkContext运行在AM中。

1. 在SparkContext启动时,初始化DAGScheduler调度器，然后在createTaskScheduler方法中匹配为YARN-Cluster运行模式时，通过反射方法初始化YarnClusterScheduler和YarnClusterSchedulerBackend两个对象，其中YarnClusterSchedulerBackend类是CoarseGrainedSchedulerBackend类的子类，YarnClusterScheduler是TaskSchedulerImpl的子类。在SparkContext的createTaskScheduler方法中，匹配模式代码及初始化如下：

其中YarnClusterManager中createTaskScheduler，源码如下：

*override def createTaskScheduler(sc: SparkContext, masterURL: String): TaskScheduler = {*

*sc.deployMode match {*

*case "cluster" => new YarnClusterScheduler(sc)*

*case "client" => new YarnScheduler(sc)*

*case \_ => throw new SparkException(s"Unknown deploy mode '${sc.deployMode}' for Yarn")*

*}*

*}*

其中createSchedulerBackend的，源码如下：

*override def createSchedulerBackend(sc: SparkContext,*

*masterURL: String,*

*scheduler: TaskScheduler): SchedulerBackend = {*

*sc.deployMode match {*

*case "cluster" =>*

*new YarnClusterSchedulerBackend(scheduler.asInstanceOf[TaskSchedulerImpl], sc)*

*case "client" =>*

*new YarnClientSchedulerBackend(scheduler.asInstanceOf[TaskSchedulerImpl], sc)*

*case \_ =>*

*throw new SparkException(s"Unknown deploy mode '${sc.deployMode}' for Yarn")*

*}*

*}*

1. Spark-Submit脚本提交程序时，会在SparkSubmit.submit方法中根据YARN-Cluster运行模式匹配Client类，匹配后在SparkSubmit.runMain方法中通过反射初始化Client实例，Client在初始化后向YARN集群提交应用程序，提交后会通过AM.startUserApplication方法中启动用户提交的应用程序，启动方法是通过反射对传入的类进行初始化并加载SparkContext。在AM.run方法中，当YARN-Cluster运行模式后，会运行runDriver方法：



在runDriver方法会在AM中运行SparkContext，该处是与YARN-Client运行模式的最大区别，等待SparkContext启动完毕后，由RM向终端点DriverEndpoint发送消息通知AM已经启动完毕：

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

*addAmIpFilter(None)*

*//启动用户应用程序，返回该应用程序进程*

*userClassThread = startUserApplication()*

*val totalWaitTime = sparkConf.get(AM\_MAX\_WAIT\_TIME)*

*try {*

*//等待SparkContext启动，默认情况下等待100s*

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

*Duration(totalWaitTime, TimeUnit.MILLISECONDS))*

*//判断SparkContext是否成功启动，如果没有则返回失败*

*if (sc != null) {*

*rpcEnv = sc.env.rpcEnv*

*val driverRef = createSchedulerRef(*

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

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

*//registerAM方法中，由RM向终端点DriverEndpoint发送消息通知AM已经启动完毕*

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

*registered = true*

*} else {}}*

1. AM启动完毕后在在registerAM方法中，由RM向终端点DriverEndpoint发送消息给AM已经启动完毕，通过YarnAllocator中的allocateResources方法向RM申请资源，该处理与YARN-Client步骤3相同
2. 在YarnAllocator.allocateResources方法中获取可用的Container，然后调用YarnAllocator中runAllocatedContainers方法，在该方法中调用ExecutorRunnable的run方法在Container启动CoarseGrainedExecutorBackend，启动后向客户端中的SparkContainer注册并申请任务集，该处理与YARN-Client步骤4相同。
3. AM中的SparkContext分配任务集给CoarseGrainedExecutorBackend执行，其运行任务并向AM汇报运行状态和进度，以让AM随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务，其作业运行调用关系如下图：



# YARN-Client和YARN-Cluster对比

在YARN中，每个Application实例都有一个AM进程，它是Application启动的第一个Container。它负责和RM打交道并请求资源，获取资源后告诉NM为其启动Container，从深层次的含义将YARN-Cluster和YARN-Client模式的区别是AM进程的区别：

1. YARN-Client模式下，AM仅向YARN请求Executor，Client会和请求的Container通信来调度他们工作，也就是Client不能离开
2. YARN-Cluster模式下，Driver运行在AM中，负责向YARN申请资源，并监督作业的运行状况。当用户提交提交作业之后，就可以关闭Client，作业会继续YARN上运行，因此YARN-Cluster模式不适合交互类型的作业。