Spark Shell源码分析

Spark Shell是提供给用户即时交互的一个命令窗口，可以在里面编写Spark代码，然后根据命令进行运算，其被称为REPL(Read-Eval-Print Loop)，交互式开发环境。

其启动过程如下：spark-shell —> spark-submit —> spark-class，依次调用者三个shell脚本文件，最终在spark-class脚本里加载主类并执行：

1. spark-shell脚本的主要工作是收集spark-submit相关参数，最终调用spark-submint，源码如下：

*function main() {*

*……*

*else*

*export SPARK\_SUBMIT\_OPTS*

*"${SPARK\_HOME}"/bin/spark-submit --class org.apache.spark.repl.Main --name "Spark shell" "$@" //例如$@ => --master yarn*

*fi*

*}*

1. 在spark-shell中执行spark-submit脚本，在该脚本中执行spark-class，增加了参数SparkSubmit，该类是真正的入口类。

*exec "${SPARK\_HOME}"/bin/spark-class org.apache.spark.deploy.SparkSubmit "$@"*

3）spark-class脚本，其代码清单：

*if [ -n "${JAVA\_HOME}" ]; then*

*RUNNER="${JAVA\_HOME}/bin/java"*

*else*

*......*

*LAUNCH\_CLASSPATH="$SPARK\_JARS\_DIR/\*"*

*build\_command() {*

*"$RUNNER" -Xmx128m -cp "$LAUNCH\_CLASSPATH" org.apache.spark.launcher.Main "$@"*

*printf "%d\0" $?*

*}*

*exec "${CMD[@]}"*

启动后，Spark启动了以SparkSubmit为主类的jvm进程：

*/usr/jdk64/jdk1.8.0\_77/bin/java -cp /usr/lib/spark/conf/:/usr/lib/spark/jars/\*:/usr/lib/hadoop/etc/hadoop/ -Dscala.usejavacp=true -Xmx1g org.apache.spark.deploy.SparkSubmit --master yarn --class org.apache.spark.repl.Main --name Spark shell spark-shell*

Spark应用程序可以以Client和Cluster模式启动，区别在于Client模式下的Driver是在执行spark-submit命令节点上启动的，而Cluster模式是CM随机选择一台Worker通过DriverWrapper来启动Driver。大概流程如下：



1. 通过spark-submit提交提交会调用SparkSubmit类，在SparkSubmit类中通过反射调用Client，Client与CM通信来Submit Driver，当应用成功返回后退出JVM。在master为yarn为配置下，使用类：

*org.apache.spark.deploy.yarn.Client*

1. yarn.Client通过YarnClient与RM通信，提交应用创建请求，其中ContainerLaunchContext与ApplicationSubmissionContext的初始化，如下所示：

*private def createContainerLaunchContext(newAppResponse: GetNewApplicationResponse)*

*: ContainerLaunchContext = {*

*val launchEnv <= setupLaunchEnv*

*[SPARK\_YARN\_MODE, true]*

*[SPARK\_YARN\_STATE\_DIR]*

*[SPARK\_USER].....*

*val localResources <=*

*val javaOpts <=*

*[-Xms, amMemory]*

*[-Djava.io.tmpDir]......*

*if(isClusterMode) {*

*librayPaths*

*}*

*val userJar*

*val amClass: ClusterMode? ApplicationMaster :ExecutorLauncher*

*val commands:*

*val securityManager*

*}*

ApplicationSubmissionContext的初始化如下：

*def createApplicationSubmissionContext (newApp: YarnClientApplication,*

*containerContext: ContainerLauncherContext){*

*val appContext:*

*spark.app.name*

*QUEUE\_NAME*

*containerContext*

*ApplicationType*

*maxAppAttempts*

*val capability:*

*amMemory + amMemoryOverHead*

*amCores*

*val amRequest:*

*ResourceName*

*Priority*

*Capability*

*numContainers*

*NodeLabel*

*V al logAggregationContext*

*}*

3) 启动SparkShell应用的ApplicationMaster，在ClientMode下启动使用ExecutorLauncher

*val amClass =*

*if (isClusterMode) {*

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

*} else {*

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

*}*

Client与RM通信会启动ExecutorLauncher，其作为AM，与RM通信获取资源，启动Executor，具体的工作由YarnAllocator完成，Executor的进程名为CoarseGrainedExecutorBackend

*def allocateResources(): Unit = synchronized {*

*updateResourceRequests()*

*val progressIndicator = 0.1f*

*val allocateResponse = amClient.allocate(progressIndicator)*

*val allocatedContainers = allocateResponse.getAllocatedContainers()*

*if (allocatedContainers.size > 0) {*

*handleAllocatedContainers(allocatedContainers.asScala)*

*}*

*val completedContainers = allocateResponse.getCompletedContainersStatuses()*

*......*

*}*

默认的Executor数目为：YarnSparkHadoopUtil.DEFAULT\_NUMBER\_EXECUTOR，值为2。

YarnAllocator在启动Executor的时候，同时会将Executor向Driver注册：

其启动线程：ExecutorRunnable#startContainer，源码如下：

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

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

*.asInstanceOf[ContainerLaunchContext]*

*val env = prepareEnvironment().asJava*

*ctx.setLocalResources(localResources.asJava)*

*ctx.setEnvironment(env)*

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

*val dob = new DataOutputBuffer()*

*credentials.writeTokenStorageToStream(dob)*

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

*//container启动命令*

*val commands = prepareCommand()*

*ctx.setCommands(commands.asJava)*

*ctx.setApplicationACLs(*

*YarnSparkHadoopUtil.getApplicationAclsForYarn(securityMgr).asJava)*

*//安全配置*

*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))*

*}*

*//通过NMClient，启动CoNTAINER*

*try {*

*nmClient.startContainer(container.get, ctx)*

*} catch {*

*case ex: Exception =>*

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

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

*}*

Commands如下：

*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")*

4）SparkShell中cluster-mode方式为Client，Driver启动在SparkSubmit中，在其JVM中，启动Driver程序：*org.apache.spark.repl.Main*，执行方法为SparkSubmit.runMain：

*private def runMain(*

*childArgs: Seq[String],*

*childClasspath: Seq[String],*

*sysProps: Map[String, String],*

*childMainClass: String,*

*verbose: Boolean): Unit = {*

*mainClass = Utils.classForName(childMainClass)*

*if (classOf[scala.App].isAssignableFrom(mainClass)) {*

*printWarning("Subclasses of scala.App may not work correctly. Use a main() method instead.")*

*}*

*val mainMethod = mainClass.getMethod("main", new Array[String](0).getClass)*

*}*

*org.apache.spark.repl.Main#main*

*private[repl] def doMain(args: Array[String], \_interp: SparkILoop): Unit = {*

*interp = \_interp*

*val jars = Utils.getLocalUserJarsForShell(conf)*

*// Remove file:///, file:// or file:/ scheme if exists for each jar*

*.map { x => if (x.startsWith("file:")) new File(new URI(x)).getPath else x }*

*.mkString(File.pathSeparator)*

*val interpArguments = List(*

*"-Yrepl-class-based",*

*"-Yrepl-outdir", s"${outputDir.getAbsolutePath}",*

*"-classpath", jars*

*) ++ args.toList*

*val settings = new GenericRunnerSettings(scalaOptionError)*

*settings.processArguments(interpArguments, true)*

*if (!hasErrors) {*

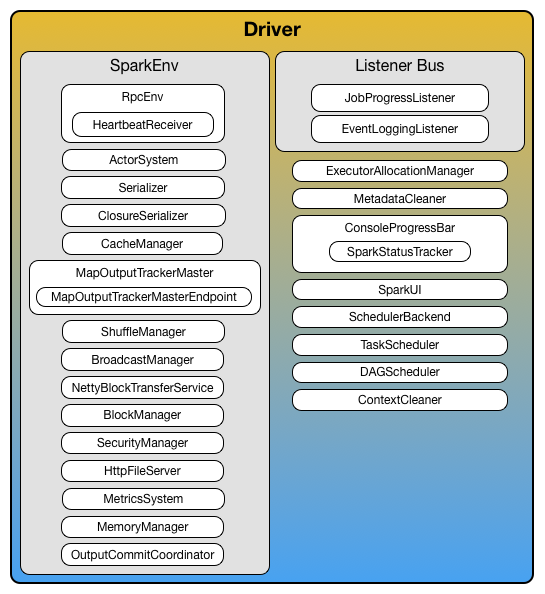
*interp.process(settings) // 启动REPL，在Shell中输出命令并执行*

*Option(sparkContext).foreach(\_.stop)*

*}*

*}*

Spark-Shell的核心类为SparkILoop。Driver程序主要是SparkContext（SparkSession），主要模块为DAGScheduler和TaskScheduler，其他如下图所示：



5）AM启动后，会向Driver注册，代码如下：

*def register(*

*driverUrl: String,*

*driverRef: RpcEndpointRef,*

*conf: YarnConfiguration,*

*sparkConf: SparkConf,*

*uiAddress: Option[String],*

*uiHistoryAddress: String,*

*securityMgr: SecurityManager,*

*localResources: Map[String, LocalResource]*

*): YarnAllocator = {*

*amClient = AMRMClient.createAMRMClient()*

*amClient.init(conf)*

*amClient.start()*

*this.uiHistoryAddress = uiHistoryAddress*

*val trackingUrl = uiAddress.getOrElse {*

*if (sparkConf.get(ALLOW\_HISTORY\_SERVER\_TRACKING\_URL)) uiHistoryAddress else ""*

*}*

*logInfo("Registering the ApplicationMaster")*

*synchronized {*

*amClient.registerApplicationMaster(Utils.localHostName(), 0, trackingUrl)*

*registered = true*

*}*

*new YarnAllocator(driverUrl, driverRef, conf, sparkConf, amClient, getAttemptId(), securityMgr,*

*localResources, new SparkRackResolver())*

*}*