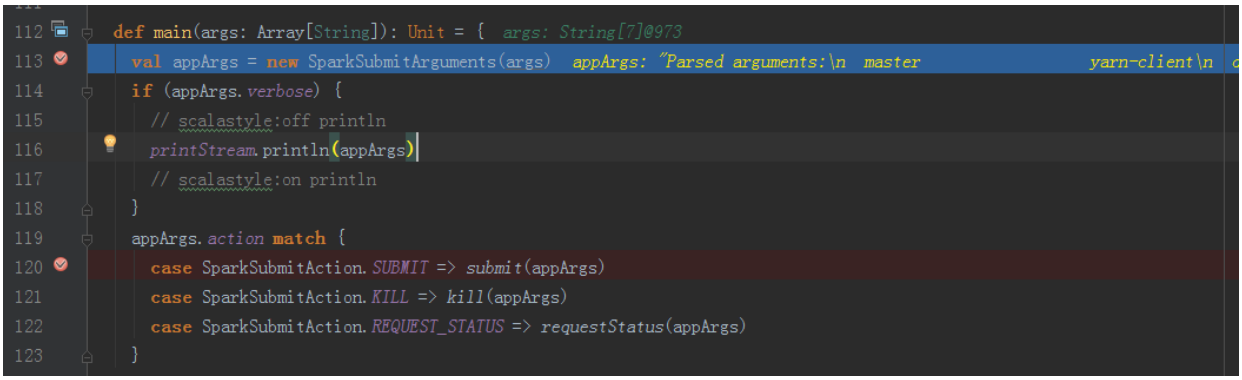
Spark On Yarn 任务提交流程分析

yarn-client mode

当我们在客户端运行 `spark-submit --master yarn-client ***.jar` 命令，使用yarn-client模式提交一个spark任务时，SparkSubmit首先作为主进程启动：



SparkSubmit将任务提交命令中包含的如 `--conf` 、 `--master` 和 `***.jar` 参数全部交给SparkSubmitArguments(args)，由它为后面submit(appArgs)动作合并参数。
SparkSubmitArguments()中的方法：

- mergeDefaultSparkProperties()：合并获取的参数。配置文件 `--properties-file` 优先级高于 `spark-defaults.conf` ,从任务提交命令中获取的 `--conf` 参数优先级高于从配置文件中获取的参数。最后合并的参数存储在HashMap— `sparkProperties` 里面。
- ignoreNonSparkProperties()：过滤掉所有不是以 `spark.` 开头的配置参数。
- loadEnvironmentArguments()：加载获取参数到提交环境。
- validateArguments()：对于提交动作来说，参数校验判断参数是否为空，如果是 `spark on yarn` 提交，需要验证 `HADOOP_CONF_DIR` 或者 `YARN_CONF_DIR` 是否存在，不存在时抛出异常。

参数准备工作就绪，接下来执行submit(appArgs)动作，所有经过合并校验的参数都可以通过 `appArgs` 获取。
我们来看看submit(appArgs)干了什么活：

- prepareSubmitEnvironment(args): 返回一个四元组，关键参数 `(childArgs, childClasspath, sysProps, childMainClass)` ，参数含义可以理解为（应用类参数，应用类引用的外包包路径和本身路径，系统配置参数包括从配置文件中和命令行获取的参数、应用主类名称）
- doRunMain(): 根据上面获取的参数，启动 `childMainClass` 。在yarn-client模式下，就是启动用户应用程序。

用户应用程序初始化 `SparkConf` ，把系统参数传递给SparkConf。由SparkConf创建 `SparkContext` ，实例化任务调度器和调度器后台程序：

```scala
case "yarn-client" =>
  val scheduler = try {
    val clazz = Utils.classForName("org.apache.spark.scheduler.cluster.YarnScheduler")
    val cons = clazz.getConstructor(classOf[SparkContext])
    cons.newInstance(sc).asInstanceOf[TaskSchedulerImpl]

  } catch {
    case e: Exception => {
      throw new SparkException("YARN mode not available ?", e)
    }
  }

  val backend = try {
    val clazz =
      Utils.classForName("org.apache.spark.scheduler.cluster.YarnClientSchedulerBackend")
    val cons = clazz.getConstructor(classOf[TaskSchedulerImpl], classOf[SparkContext])
    cons.newInstance(scheduler, sc).asInstanceOf[CoarseGrainedSchedulerBackend]
  } catch {
    case e: Exception => {
      throw new SparkException("YARN mode not available ?", e)
    }
  }
```

在 `yarn-client` 模式下， `YarnClusterScheduler` 调度器拉起后台进程 `YarnClientSchedulerBackend` 。

YarnClientSchedulerBackend.start()通过 `Client` 向 `Resource Manager` 提交应用（AM）：

```scala
override def start() {
  val driverHost = conf.get("spark.driver.host")
  val driverPort = conf.get("spark.driver.port")
  val hostport = driverHost + ":" + driverPort
  sc.ui.foreach { ui => conf.set("spark.driver.appUIAddress", ui.appUIAddress) }

  val argsArrayBuf = new ArrayBuffer[String]()
  argsArrayBuf += ("--arg", hostport)
  argsArrayBuf ++= getExtraClientArguments

  logDebug("ClientArguments called with: " + argsArrayBuf.mkString(" "))
  val args = new ClientArguments(argsArrayBuf.toArray, conf)
  totalExpectedExecutors = args.numExecutors
  client = new Client(args, conf)
  appId = client.submitApplication()

  // SPARK-8687: Ensure all necessary properties have already been set before
  // we initialize our driver scheduler backend, which serves these properties
  // to the executors
  super.start()
```

在 `Client` 中真正执行 `submitApplication` 的是 `yarnClient` ， `yarnClient` 经过**初始化->启动->创建App**流程，从RM端将AM申请下来，RM返回申请结果 `newAppResponse` ，这里面携带了 `AppID` 等信息。

```
    yarnClient.init(yarnConf)
    yarnClient.start()

    logInfo("Requesting a new application from cluster with %d NodeManagers"
      .format(yarnClient.getYarnClusterMetrics.getNumNodeManagers))

    // Get a new application from our RM
    val newApp = yarnClient.createApplication()
    val newAppResponse = newApp.getNewApplicationResponse()
    appId = newAppResponse.getApplicationId()

    // Verify whether the cluster has enough resources for our AM
    verifyClusterResources(newAppResponse)

    // Set up the appropriate contexts to launch our AM
    val containerContext = createContainerLaunchContext(newAppResponse)
    val appContext = createApplicationSubmissionContext(newApp, containerContext)

    // Finally, submit and monitor the application
    logInfo(s"Submitting application ${appId.getId} to ResourceManager")
    yarnClient.submitApplication(appContext)
```

通过RM的返回 `Client` 进行了资源可用性校验，AM或者Executor所用内存（包括 `MemoryOverhead` ）不能大于YarnContainer的最大可用内存，否则抛出异常，代码如下：

```
private def verifyClusterResources(newAppResponse: GetNewApplicationResponse): Unit = {
  val maxMem = newAppResponse.getMaximumResourceCapability().getMemory()
  logInfo("Verifying our application has not requested more than the maximum " +
    s"memory capability of the cluster ($maxMem MB per container)")
  val executorMem = args.executorMemory + executorMemoryOverhead
  if (executorMem > maxMem) {
    throw new IllegalArgumentException(s"Required executor memory (${args.executorMemory}" +
      s"+${executorMemoryOverhead} MB) is above the max threshold ($maxMem MB) of this cluster! " +
      "Please increase the value of 'yarn.scheduler.maximum-allocation-mb'.")
  }
  val amMem = args.amMemory + amMemoryOverhead
  if (amMem > maxMem) {
    throw new IllegalArgumentException(s"Required AM memory (${args.amMemory}" +
      s"+${amMemoryOverhead} MB) is above the max threshold ($maxMem MB) of this cluster! " +
      "Please increase the value of 'yarn.scheduler.maximum-allocation-mb'.")
  }
  logInfo("Will allocate AM container, with %d MB memory including %d MB overhead".format(
    amMem,
    amMemoryOverhead))
```

`containerContext` 内容包括hive配置文件路径， `spark.yarn.jars` 配置jar包的HDFS路径和 `spark-conf` 的路径

因为是yarn-client模式，ApplicationMaster启动后会运行 `runExecutorLauncher` ，启动AMContanier进程:

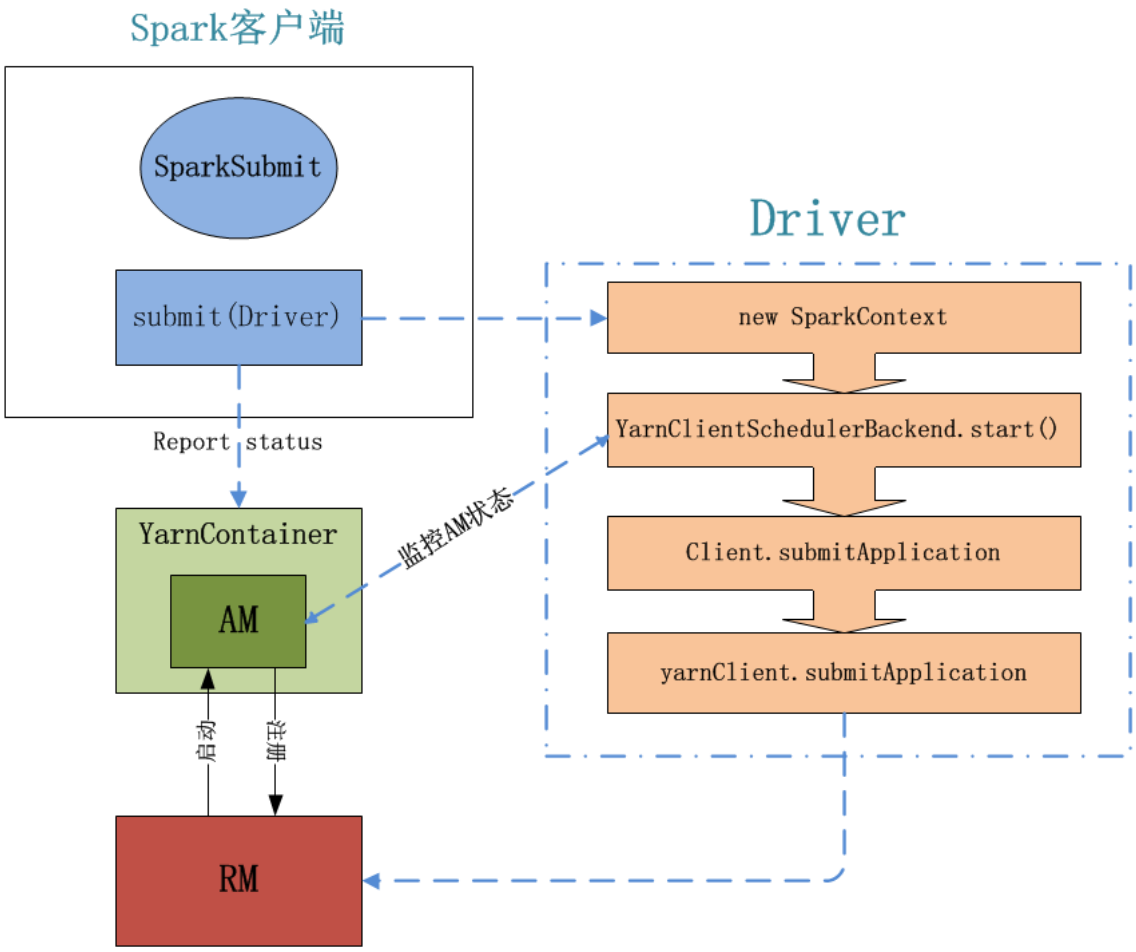
```
    if (isClusterMode) {
      runDriver(securityMgr)
    } else {
      runExecutorLauncher(securityMgr)
    }
```

```scala
private def runExecutorLauncher(securityMgr: SecurityManager): Unit = {
  val port = sparkConf.getInt("spark.yarn.am.port", 0)
  rpcEnv = RpcEnv.create("sparkYarnAM", Utils.localHostName, port, sparkConf, securityMgr)
  val driverRef = waitForSparkDriver()
  addAmIpFilter()
  registerAM(rpcEnv, driverRef, sparkConf.get("spark.driver.appUIAddress", ""), securityMgr)

  // In client mode the actor will stop the reporter thread.
  reporterThread.join()
}
```

`runExecutorLauncher` 等待Driver可用后，向RM注册该AM，注册结果的返回后， `YarnAllocator` 介入，承担更新资源需求，申请资源和加载Container的工作。

Yarn-Client任务提交整体流程图



yarn-cluster

对于yarn-cluster模式来说，不同之处在于spark客户端SparkSubmit启动的的子进程是Client，通过Client.submitApplication()向RM申请启动AM。在yarn-client模式下，这个工作是由调度器后台(`YarnClientSchedulerBackend`)来做的。

AM被授权启动后，通过判断是否集群模式决定是启动Driver还是AMContanier线程：

```
    if (isClusterMode) {
      runDriver(securityMgr)
    } else {
      runExecutorLauncher(securityMgr)
    }
  } catch {
```

runDriver的工作是启动用户应用程序线程，向RM注册AM，并等待用户线程结束。

Yarn-Cluster任务提交整体流程图