## 1 切片源码分析

|  |
| --- |
| def defaultMinPartitions: Int = math.min(defaultParallelism, 2)  override def defaultParallelism(): Int =  scheduler.conf.getInt("spark.default.parallelism", totalCores)  // defaultParallelism没有在配置文件中配置，会选择默认的核心数，  defaultMinPartitions在totalCores与2中选择一个较小值！  sc.textFile调用hadoopFile方法在其中 new HadoopRDD  在HadoopRDD这个类中，重写了getPartitions方法！  override def getPartitions: Array[Partition] = {  val jobConf = getJobConf()  // add the credentials here as this can be called before SparkContext initialized  SparkHadoopUtil.get.addCredentials(jobConf)  val inputFormat = getInputFormat(jobConf)  val inputSplits = inputFormat.getSplits(jobConf, minPartitions)  val array = new Array[Partition](inputSplits.size)  for (i <- 0 until inputSplits.size) {  array(i) = new HadoopPartition(id, i, inputSplits(i))  }  array  }  在getPartitions中调用inputFormat.getSplits方法获取切片情况！  FileInputFormat中getSplits方法说明！  public InputSplit[] getSplits(JobConf job, int numSplits)  throws IOException {  Stopwatch sw = new Stopwatch().start();    FileStatus[] files = listStatus(job); // 所有的文件    // Save the number of input files for metrics/loadgen  job.setLong(NUM\_INPUT\_FILES, files.length);  long totalSize = 0; // compute total size  for (FileStatus file: files) { // check we have valid files  if (file.isDirectory()) {  throw new IOException("Not a file: "+ file.getPath());  }  totalSize += file.getLen(); |

|  |
| --- |
| }  // 记录所有文件的总大小！比如上课时是100+100+100+400=700KB    // numSplits传进来的值是2，所以此处goalSize的大小是700/2=350  long goalSize = totalSize / (numSplits == 0 ? 1 : numSplits);    // private long minSplitSize = 1;这个值默认是1，因此minSize的值为1  long minSize = Math.max(job.getLong(org.apache.hadoop.mapreduce.lib.input.  FileInputFormat.SPLIT\_MINSIZE, 1), minSplitSize);    // generate splits  // 这个数组保存所有的切片，最后返回  ArrayList<FileSplit> splits = new ArrayList<FileSplit>(numSplits);  NetworkTopology clusterMap = new NetworkTopology();    for (FileStatus file: files) { // 遍历所有的文件  Path path = file.getPath();  long length = file.getLen();  if (length != 0) {  FileSystem fs = path.getFileSystem(job);  BlockLocation[] blkLocations;  if (file instanceof LocatedFileStatus) {  blkLocations = ((LocatedFileStatus) file).getBlockLocations();  } else {  blkLocations = fs.getFileBlockLocations(file, 0, length);  }    // 判断文件是否可以切割  if (isSplitable(fs, path)) {  long blockSize = file.getBlockSize(); // 块大小local: 32M 集群: 128M    // 本例中splitSize=350，computeSplitSize方法在goalSize和blockSize中取最小值，然后在minSize取一个最大值  long splitSize = computeSplitSize(goalSize, minSize, blockSize);  long bytesRemaining = length; // 用来记录文件切后剩余大小    // 如果文件大小大于了总大小一半的1.1倍, 就会进行切  while (((double) bytesRemaining)/splitSize > SPLIT\_SLOP) {  String[][] splitHosts = getSplitHostsAndCachedHosts(blkLocations,  length-bytesRemaining, splitSize, clusterMap);  splits.add(makeSplit(path, length-bytesRemaining, splitSize,  splitHosts[0], splitHosts[1])); |
| bytesRemaining -= splitSize;  }  if (bytesRemaining != 0) {  String[][] splitHosts = getSplitHostsAndCachedHosts(blkLocations, length  - bytesRemaining, bytesRemaining, clusterMap);  splits.add(makeSplit(path, length - bytesRemaining, bytesRemaining,  splitHosts[0], splitHosts[1]));  }  } else {  // 如果文件不可切割，直接作为一片  String[][] splitHosts =  getSplitHostsAndCachedHosts(blkLocations,0,length,clusterMap);  splits.add(makeSplit(path, 0, length, splitHosts[0], splitHosts[1]));  }  } else {  //Create empty hosts array for zero length files  splits.add(makeSplit(path, 0, length, new String[0]));  }  }  sw.stop();  if (LOG.isDebugEnabled()) {  LOG.debug("Total # of splits generated by getSplits: " + splits.size()  + ", TimeTaken: " + sw.elapsedMillis());  }  return splits.toArray(new FileSplit[splits.size()]);  }  protected long computeSplitSize(long goalSize, long minSize,long blockSize) {  return Math.max(minSize, Math.min(goalSize, blockSize));  } |

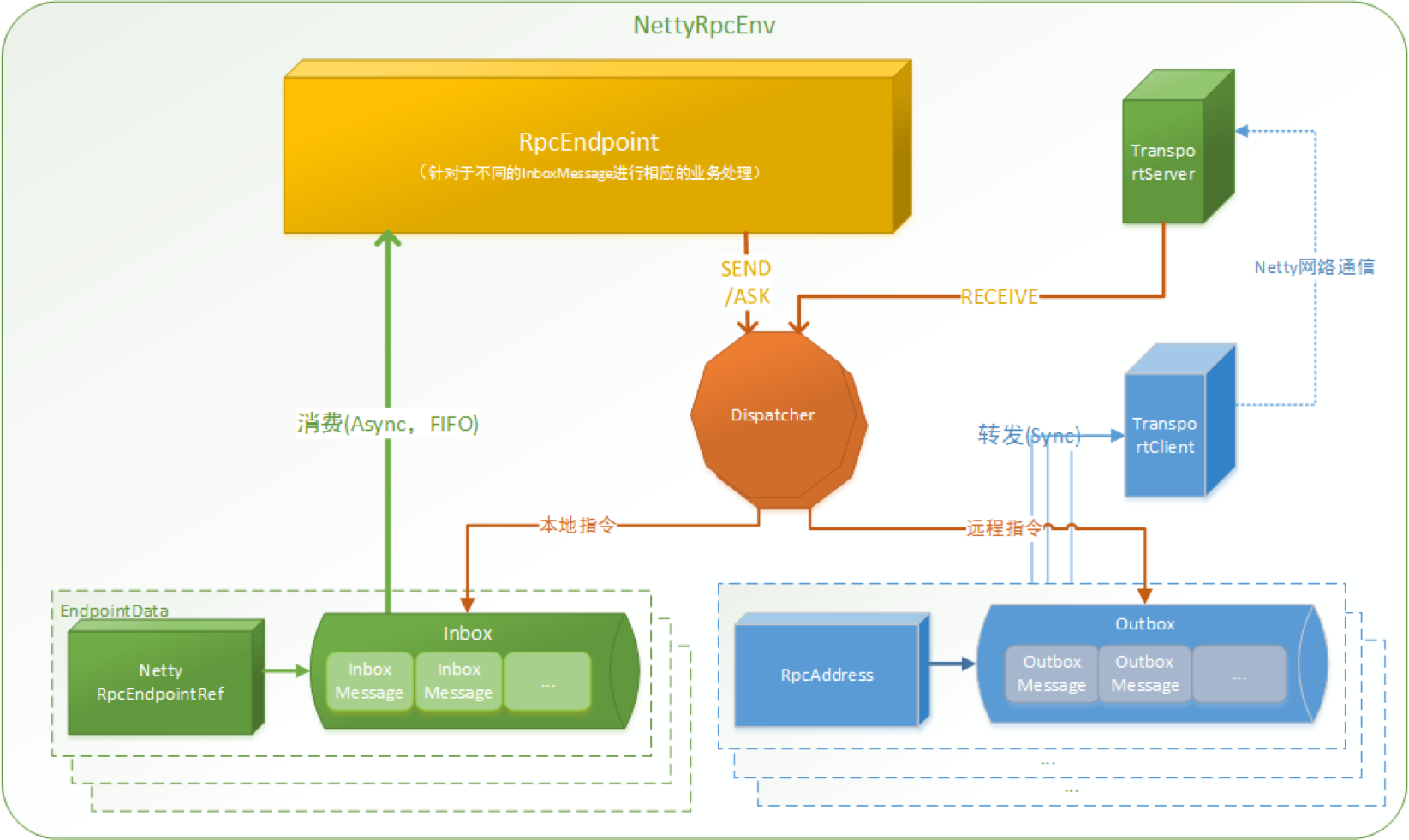
|  |
| --- |
| 总结如下：  (1) sc.textFile调用hadoopFile方法在其中 new HadoopRDD在HadoopRDD这个类中，重写了getPartitions方法！  (2) 在getPartitions中调用inputFormat.getSplits方法获取切片情况！FileInputFormat中getSplits方法说明！  读取目录所有文件；  totalSize记录所有文件大小；goalSize计算目标文件大小；computeSplitSize计算切片大小；  splits数组保存所有的切片信息；  遍历所有文件，判断是否可切，如果文件大小大于了总大小一半的1.1倍, 就会进行切，如果文件不可切割，直接作为一片！ |

## 2 Spark Netty 通信架构解析

|  |
| --- |
| RpcEndpoint类的说明：   1. An end point for the RPC that defines what functions to trigger given a message   接收消息时，创建RpcEndpoint等操作时，定义哪些函数自动触发；   1. constructor -> onStart -> receive\* -> onStop   It is guaranteed that `onStart`, `receive` and `onStop` will be called in sequence.  保证三个方法顺序调用，receive负责接收消息，end point收到一个信息回调一次；  RpcEndpoint用来接收消息 |

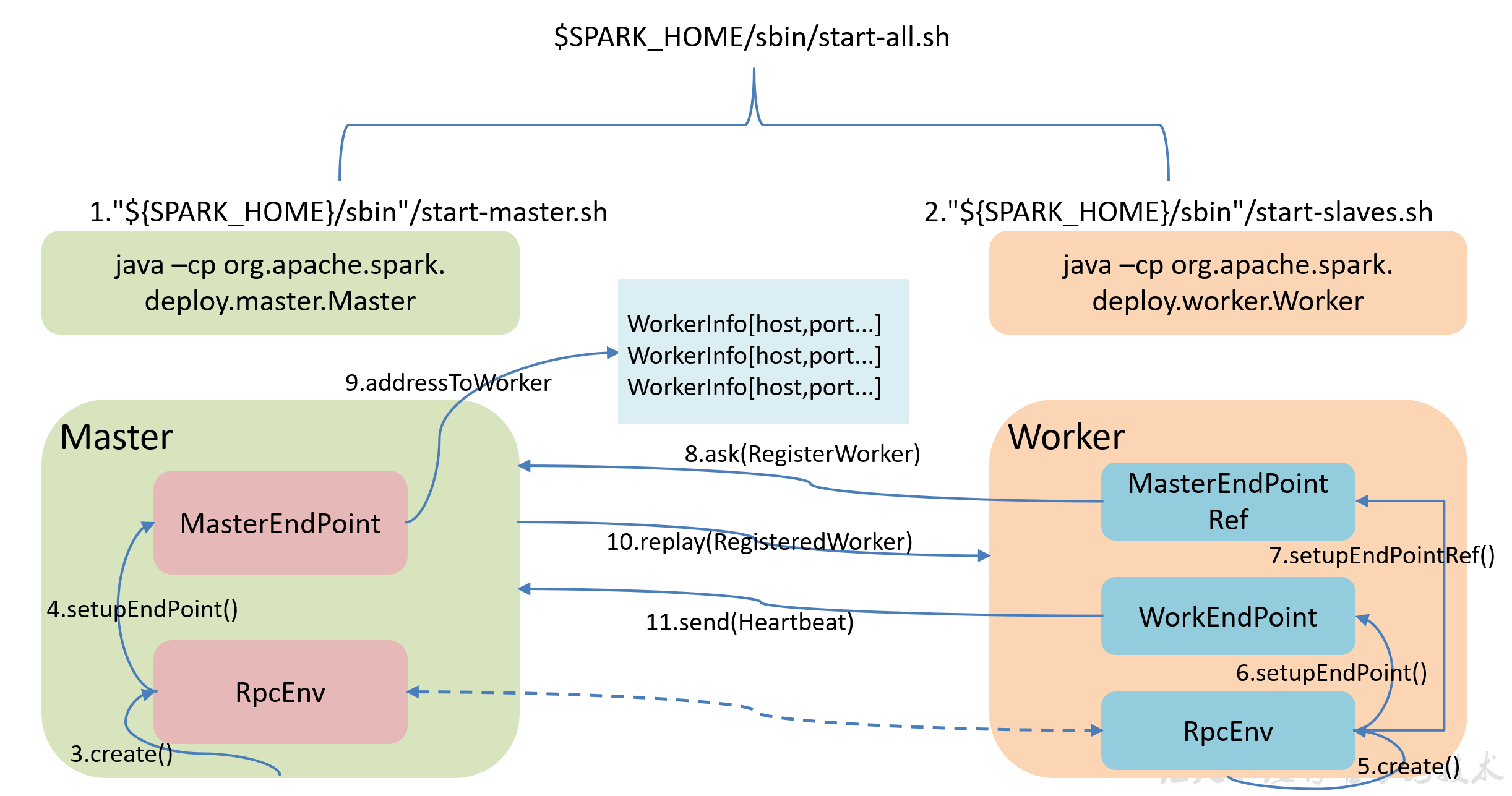
|  |
| --- |
| RpcEndpointRef用来发送消息  def send(message: Any): Unit // 发送单向异步消息，不要求回复  // 发送消息给对应RpcEndpoint的receiveAndReply方法，只发一次不重试  def ask[T: ClassTag](message: Any, timeout: RpcTimeout): Future[T]  // 发送同步请求，并在指定时间内等待返回类型为T的处理结果，  // 当抛出异常的时候, 会进行请求重试, 直到超出重试次数  // 这个方法会有阻塞行为, 所以不要循环调用它  def askWithRetry[T: ClassTag](message: Any): T  RpcEndpointRef：RpcEndpointRef 是对远程 RpcEndpoint 的一个引用；  当我们需要向一个具体的 RpcEndpoint 发送消息时，一般我们需要获取到该RpcEndpoint 的引用，然后通过该引用发送消息。 |

|  |
| --- |
| 消息传递涉及的类：  class NettyRpcEnv  abstract class RpcEndpointRef  class Dispatcher(nettyEnv: NettyRpcEnv)  private class EndpointData(val name: String,val endpoint: RpcEndpoint,  val ref: NettyRpcEndpointRef) {  // 每个Endpoint的收件箱  val inbox: Inbox = new Inbox(ref, endpoint)  }  class Outbox(nettyEnv: NettyRpcEnv, val address: RpcAddress)  class TransportClient  class TransportServer  class Inbox(val endpointRef: NettyRpcEndpointRef,val endpoint: RpcEndpoint) |



## 3 standalone模式启动分析

Master与Worker的启动流程分析：



|  |
| --- |
| 1. start-all.sh脚本，实际是执行java -cp Master和java -cp Worker；  2. Master启动时首先创建一个RpcEnv对象，负责管理所有通信逻辑；  3. Master通过RpcEnv对象创建一个Endpoint，Master就是一个Endpoint，Worker可以与其进行通信；  4. Worker启动时也是创建一个RpcEnv对象；  5. Worker通过RpcEnv对象创建一个Endpoint；  6. Worker通过RpcEnv对象建立到Master的连接，获取到一个RpcEndpointRef对象，通过该对象可以与Master通信；  7. Worker向Master注册，注册内容包括主机名、端口、CPU Core数量、内存数量；  8. Master接收到Worker的注册，将注册信息维护在内存中的集合中，其中还包含了一个到Worker的RpcEndpointRef对象引用；  9. Master回复Worker已经接收到注册：  Master调用RpcCallContext 类的context.reply(RegisteredWorker(self, masterWebUiUrl))方法告知Worker已经注册成功；  10. Worker端收到成功注册响应后，开始周期性【默认15s】向Master发送心跳。 |

|  |
| --- |
| 执行start-all.sh：  依次执行：spark-config.sh，start-master.sh，start-slaves.sh；  start-master.sh：CLASS="org.apache.spark.deploy.master.Master"  执行：start-master.sh，调用spark-daemon.sh启动主类！  "${SPARK\_HOME}/sbin"/spark-daemon.sh start $CLASS 1 \  --host $SPARK\_MASTER\_HOST  --port $SPARK\_MASTER\_PORT  --webui-port $SPARK\_MASTER\_WEBUI\_PORT \ $ORIGINAL\_ARGS  spark-daemon.sh中调用bin/spark-class脚本，bin/spark-class主要配置Java运行相关的参数，最后执行：exec "${CMD[@]}"  在logs文件夹下查看master.out文件，执行的命令如下：  /opt/module/jdk1.8.0\_121/bin**/java -cp**  /opt/module/spark-standalone/conf/:/opt/module/spark-standalone/jars/\*  -Dspark.deploy.recoveryMode=ZOOKEEPER  -Dspark.deploy.zookeeper.url=hadoop102:2181,hadoop102:2181,hadoop103:2181  -Dspark.deploy.zookeeper.dir=/spark  -Xmx1g  org.apache.spark.deploy.master.Master  --host hadoop103 --port 7077 --webui-port 8080 |

看这个类org.apache.spark.deploy.master.Master伴生对象的main方法：

|  |
| --- |
| // 启动 Master 的入口函数  def main(argStrings: Array[String]) {  Utils.initDaemon(log)  val conf = new SparkConf  // 构建用于参数解析的实例 --host hadoop103 --port 7077 --webui-port 8080  val args: MasterArguments = new **MasterArguments**(argStrings, conf)  // 启动 RPC 通信环境和 MasterEndPoint(通信终端)  val (rpcEnv, \_, \_): (RpcEnv, Int, Option[Int]) =  startRpcEnvAndEndpoint(args.host, args.port, args.webUiPort, conf)  // 防止master进程退出 master 需要一直在后台进行  rpcEnv.awaitTermination()  } |

|  |
| --- |
| // **MasterArguments这个类中**一堆参数解析的方法！采用递归的方式，值得借鉴！  @tailrec  private def parse(args: List[String]): Unit = args match {  case ("--ip" | "-i") :: value :: tail => // --ip 或者 -i开头都可以匹配到  Utils.checkHost(value, "ip no longer supported, please use hostname " + value)  host = value  parse(tail)    case ("--host" | "-h") :: value :: tail =>  Utils.checkHost(value, "Please use hostname " + value)  host = value  parse(tail)    case ("--port" | "-p") :: IntParam(value) :: tail =>  port = value  parse(tail)    case "--webui-port" :: IntParam(value) :: tail =>  webUiPort = value  parse(tail)    case ("--properties-file") :: value :: tail =>  propertiesFile = value  parse(tail)    case ("--help") :: tail =>  printUsageAndExit(0)    case Nil => // No-op    case \_ =>  printUsageAndExit(1)  } |

**1 MasterArguments这个类解析参数！**

2 startRpcEnvAndEndpoint方法创建并启动 RPC 通信环境！

startRpcEnvAndEndpoint中调用RpcEnv.create返回**NettyRpcEnv**

RpcEnv.create使用工厂创建RpcEnv：new NettyRpcEnvFactory().create(config)

调用nettyEnv.startServer启动NettyRpcEnv；

【两点：创建一个TransportServer；注册一个RpcEndpoint】

|  |
| --- |
| 在startServer方法中注册RpcEndpoint：dispatcher.registerRpcEndpoint  注册RpcEndpoint做了哪些事？  endpoints: ConcurrentMap[String, EndpointData]【端点实例名称与端点数据的映射！】  endpointRefs: ConcurrentMap[RpcEndpoint, RpcEndpointRef]  【端点实例与端点ref之间的映射】  receivers = new LinkedBlockingQueue[EndpointData]【存储EndpointData的阻塞队列】  将RpcEndpoint 放入到 endpoints集合中！  获取刚刚放入endpoints集合的RpcEndpoint并放在endpointRefs集合中！  将RpcEndpoint放在阻塞队列中！ |

3 startRpcEnvAndEndpoint方法创建Master对象，返回RpcEndpointRef，

val masterEndpoint: RpcEndpointRef = rpcEnv.setupEndpoint(ENDPOINT\_NAME,

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

setupEndpoint方法：

|  |
| --- |
| override def setupEndpoint(name: String, endpoint: RpcEndpoint): RpcEndpointRef = {  dispatcher.registerRpcEndpoint(name, endpoint) // 上面已经看过  } |

Inbox类中的def process(dispatcher: Dispatcher) // 调用RpcEndpoint的onStart方法

case OnStart => endpoint.onStart()

Master中def onStart()方法说明：

|  |
| --- |
| // 先创建web UI；  // 按照固定的频率去启动线程来检查 Worker 是否超时，  // 其实就是给自己发信息：CheckForWorkerTimeOut  // 默认是每分钟检查一次，使用固定的频率  checkForWorkerTimeOutTask =  forwardMessageThread.scheduleAtFixedRate(new Runnable {  override def run(): Unit = Utils.tryLogNonFatalError {  // self：RpcEndpointRef // 检查worker是否超时，样例对象 receive方法接收  self.send(CheckForWorkerTimeOut) // 自己给自己发送信息统一编程风格  }  // 0 表示立即执行，先执行一次，以后每隔60秒执行一次  }, 0, WORKER\_TIMEOUT\_MS, TimeUnit.MILLISECONDS) |

Master中receive方法接收发送的信息：override def receive: PartialFunction[Any, Unit]返回偏函数，根据信息匹配：

case CheckForWorkerTimeOut => timeOutDeadWorkers() // 移除超时的workers

4 startRpcEnvAndEndpoint方法中，向Master的通信终端发法请求，获取 BoundPortsResponse 对象！

至此，master启动成功！

worker的启动：

/opt/module/jdk1.8.0\_121/bin/java

-cp /opt/module/spark-standalone/conf/:/opt/module/spark-standalone/jars/\*

-Dspark.deploy.recoveryMode=ZOOKEEPER

-Dspark.deploy.zookeeper.url=hadoop102:2181,hadoop102:2181,hadoop103:2181

-Dspark.deploy.zookeeper.dir=/spark

-Xmx1g

**org.apache.spark.deploy.worker.Worker** --webui-port 8081 spark://hadoop103:7077

main方法：

WorkerArguments类构建解析参数实例；startRpcEnvAndEndpoint方法启动RpcEnv!

Worker的onStart方法说明：

创建工作目录，启动shuffle服务，

调用registerWithMaster()向 Master 注册 Worker；

|  |
| --- |
| 先调用tryRegisterAllMasters()向所有的master注册！如果注册失败，再以固定的频率向master注册！  private def registerWithMaster(masterEndpoint: RpcEndpointRef): Unit = {  // 向 Master 对应的 receiveAndReply 方法发送信息  // 信息的类型是 RegisterWorker, 包括 Worker 的一些信息: id, 主机地址, 端口号, 内存, webUi  // ask: 发送信息的时候, 要求对方有回应  // send: 发送信息, 不强制要求对方有回应  masterEndpoint.ask[RegisterWorkerResponse](RegisterWorker(  workerId, host, port, self, cores, memory, workerWebUiUrl))  .onComplete {  // This is a very fast action so we can use "ThreadUtils.sameThread"  case Success(msg) =>  Utils.tryLogNonFatalError {  **handleRegisterResponse**(msg)  }  case Failure(e) =>  logError(s"Cannot register with master: ${masterEndpoint.address}", e)  System.exit(1)  }(ThreadUtils.sameThread)  } |

|  |
| --- |
| Master中的registerWorker方法保存注册的worker：  private def removeWorker(worker: WorkerInfo)  注册或者移除worker就是更新下面三个集合！  val workers = new HashSet[WorkerInfo] // 新的worker放在hashset中  private val idToWorker = new HashMap[String, WorkerInfo] // id映射worker  private val addressToWorker = new HashMap[RpcAddress, WorkerInfo] // 地址映射worker  private[spark] class WorkerInfo(val id: String,  val host: String,val port: Int,  val cores: Int, val memory: Int,  val endpoint: RpcEndpointRef,  val webUiAddress: String) |

|  |
| --- |
| Dispatcher说明：在NettyRpcEnv中创建  // 创建消息分发器, 会创建一个收件箱. 可以提升异步处理消息的能力  private val dispatcher: Dispatcher = new Dispatcher(this) |

## 4 YARN Cluster 模式

提交应用到yarn集群：

|  |
| --- |
| bin/spark-submit \  --master yarn \  --deploy-mode cluster \  --class org.apache.spark.examples.SparkPi \  ./examples/jars/spark-examples\_2.11-2.1.1.jar 100 |

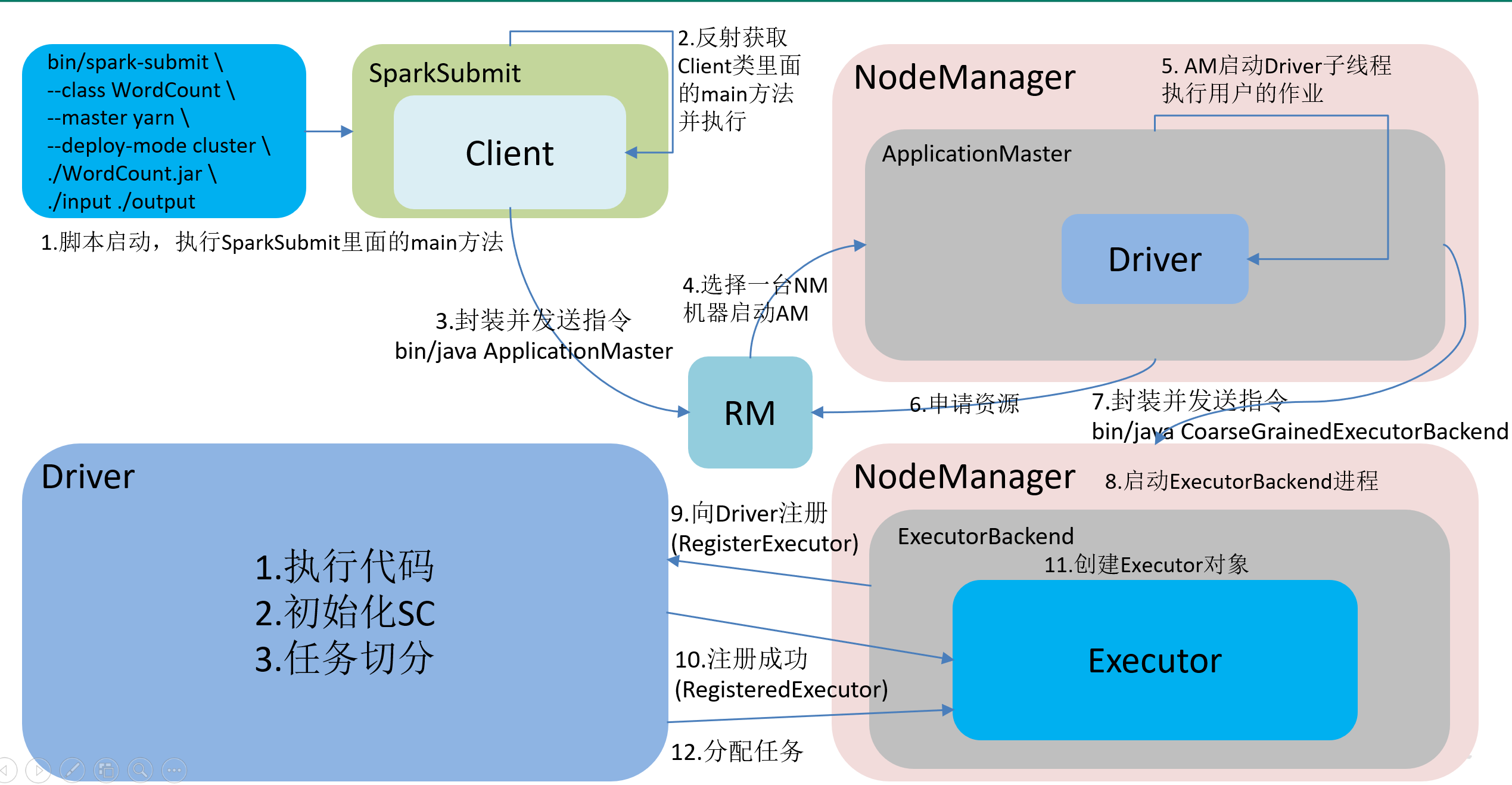
依次启动三个进程：

SparkSubmit

ApplicationMaster

CoarseGrainedExecutorBackend【一个或多个】

调用bin/spark-class org.apache.spark.deploy.SparkSubmit



org.apache.spark.deploy.SparkSubmit的main方法：

main方法做了两件事：封装传递过来的参数；执行submit方法！

|  |
| --- |
| 一、封装传递过来的参数！  val appArgs: SparkSubmitArguments = new SparkSubmitArguments(args)  创建对象调用loadEnvironmentArguments()方法，  这个方法中action = Option(action).getOrElse(SUBMIT)，所以下面执行的时候默认参数是SUMMIT！ |

|  |
| --- |
| **二、**执行submit方法：case SparkSubmitAction.SUBMIT => submit(appArgs)  submit方法：  // 准备提交环境，返回四个值  // 当是 yarn-cluster 模式: childMainClass = org.apache.spark.deploy.yarn.Client  // 当是 yarn-client 模式: childMainClass = args.mainClass // 用户主类 spark.Pi等  **1** val (childArgs, childClasspath, sysProps, childMainClass) = prepareSubmitEnvironment(args)  这个方法确定childMainClass，457行代码！  2 然后执行doRunMain()方法，  调用runMain(childArgs, childClasspath, sysProps, childMainClass, args.verbose)  在runMain方法中，  使用反射的方式加载 childMainClass = "org.apache.spark.deploy.yarn.Client"  反射出来的方法名是mainMethod，必须是静态方法！  执行这个main方法：mainMethod.invoke【Client.main(childArgs.toArray)】  **3** Client.Main说明：  new Client(args, sparkConf).run() // 直接调用run方法！  **4** Client类的run方法：  this.appId = submitApplication() // 提交应用, 返回应用的id  **5** def submitApplication(): ApplicationId方法说明：  首先初始化并启动yarnClient客户端！  // 然后创建一个新的应用newApp!  val newApp = yarnClient.createApplication()  val containerContext = createContainerLaunchContext(newAppResponse)  // 这个方法确定 ApplicationMaster类： // 核心代码  // 设置正确的上下文对象来启动 ApplicationMaster // 设置JVM参数！  如果是cluster模式：org.apache.spark.deploy.yarn.ApplicationMaster  如果是client模式：org.apache.spark.deploy.yarn.ExecutorLauncher  创建appContext后提交应用，启动ApplicationMaster进程：// 在client模式启动的是EL！  yarnClient.submitApplication(appContext)  // 走到这SparkSubmit进程结束！作用启动一个进程 ApplicationMaster  // 下面提交应用给 ResourceManager 启动 ApplicationMaster |

|  |
| --- |
| ApplicationMaster类的main方法：  // 构建 ApplicationMasterArguments对象，对传来的参数做封装  **1** val amArgs: ApplicationMasterArguments = new ApplicationMasterArguments(args)  SparkHadoopUtil.get.runAsSparkUser { () =>  2 // 构建 ApplicationMaster实例，ApplicationMaster 需要与 RM 通讯  master = new ApplicationMaster(amArgs, new YarnRMClient)  3 // 运行ApplicationMaster的 run方法，run 方法结束之后，结束 ApplicationMaster进程  System.exit(master.run())  }  4 // ApplicationMaster的run方法核心代码  if (isClusterMode) { // cluster 模式  runDriver(securityMgr)  } else { // client 模式  runExecutorLauncher(securityMgr)  }  5 ApplicationMaster的runDriver方法说明：  private def runDriver(securityMgr: SecurityManager): Unit  userClassThread = startUserApplication() // 在runDriver中调用startUserApplication方法  在startUserApplication方法中首先反射得到用户类的main方法，然后新起一个线程，名叫 "Driver"，这个线程在ApplicationMaster进程中！在这个线程中，反射调用用户类的main方法！  userThread.setName("Driver")  userThread.start() // 运行用户main方法的子线程  userThread // 返回Driver线程  // ApplicationMaster类的main线程会礼让userClassThread.join()，直到Driver运行结束！  6 // 等待SparkContext初始化完成！  SparkContext类：  创建SparkContext对象所做的事情！try代码块！  6.1 设置executorMemory，默认1G；  6.2 创建并注册心跳接收器：heartbeatReceiver = env.rpcEnv.setupEndpoint  6.3 创建DAGScheduler;  6.4 发送心跳；  6.5 启动TaskScheduler：taskScheduler.start()；  调用的是这个类的TaskSchedulerImpl，start方法！  最终调用CoarseGrainedSchedulerBackend的start方法，  最终创建了一个driverEndpoint: RpcEndpointRef！  7 runDriver方法中调用registerAM方法说明！  7.1 // 首先注册ApplicationMaster，其实就是请求资源！  // 向RM注册, 得到 YarnAllocator  allocator = client.register  7.2 // 请求分配资源  allocator.allocateResources()  def allocateResources(): Unit = synchronized方法说明：  // 获取RM给这个AM分配的所有容器（yarn给这个应用分配的资源）  val allocatedContainers = allocateResponse.getAllocatedContainers()  // 处理分配到的容器  handleAllocatedContainers(allocatedContainers.asScala)  在handleAllocatedContainers方法中runAllocatedContainers(containersToUse)这个方法运行分配的容器，封装远程指令，启动JVM进程！  runAllocatedContainers方法分别在每个容器内启动一个JVM进程，在每个进程中创建线程调用ExecutorRunnable的run方法！  ExecutorRunnable的run方法，首先创建nmClient，然后初始化nmClient，  最后执行startContainer()方法！  在startContainer()方法中首先准备要执行的命令：val commands = prepareCommand()，  准备JVM环境，启动这个类org.apache.spark.executor.CoarseGrainedExecutorBackend！  然后启动容器：nmClient.startContainer(container.get, ctx)  到此，AM启动完毕，做了两件事，创建Driver线程并运行；注册AM，向yarn集群申请容器，启动CoarseGrainedExecutorBackend进程！ |

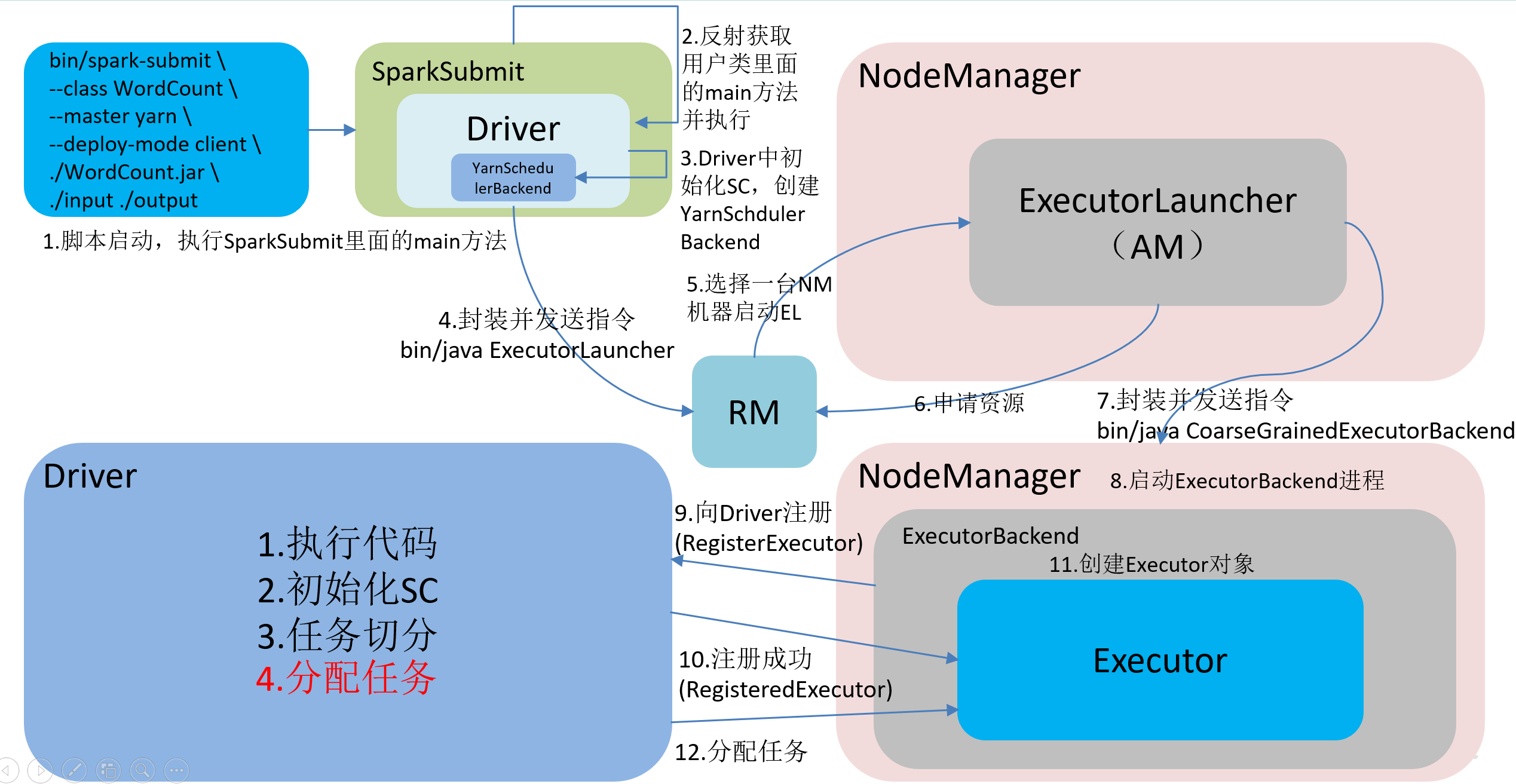
|  |
| --- |
| 启动org.apache.spark.executor.CoarseGrainedExecutorBackend这个类，main方法！  Main方法中启动CoarseGrainedExecutorBackend  run(driverUrl, executorId, hostname, cores, appId, workerUrl, userClassPath)  CoarseGrainedExecutorBackend伴生对象的run方法说明：  1 // 创建 SparkEnv对象是在rpcenv的基础上进一步封装  val env: SparkEnv = SparkEnv.createExecutorEnv  createExecutorEnv中调用create方法，  // 生成系统名称：sparkDriver 或 sparkExecutor  val systemName = if (isDriver) driverSystemName else executorSystemName  // 创建RpcEnv  val rpcEnv = RpcEnv.create  // 初始化序列化器  // 创建广播管理器  2 // 设置Executor端的RpcEndpoint: CoarseGrainedExecutorBackend  // 会调用生命周期的方法  env.rpcEnv.setupEndpoint("Executor", new CoarseGrainedExecutorBackend(  env.rpcEnv, driverUrl, executorId, hostname, cores, userClassPath, env))  3 override def onStart()  // CoarseGrainedSchedulerBackend进程向driver反向注册executor  ref.ask[Boolean](RegisterExecutor(executorId, self, hostname, cores, extractLogUrls))  发送RegisterExecutor到class DriverEndpoint这个类的receiveAndReply方法，注册成功，  receiveAndReply方法：  更新集合：val addressToExecutorId = new HashMap[RpcAddress, String]  addressToExecutorId(executorAddress) = executorId  totalCoreCount.addAndGet(cores)  totalRegisteredExecutors.addAndGet(1)  发送executorRef.send(RegisteredExecutor)消息！  发送RegisteredExecutor，然后看CoarseGrainedExecutorBackend这个类的receive方法，  注册成功，创建Executor对象：  executor = new Executor(executorId, hostname, env, userClassPath, isLocal = false)  到此，任务提交完成！  下面Driver开始分配任务交给Executor执行！ |

## 5 YARN Client 模式

SparkSubmit

ExecutorLauncher

CoarseGrainedExecutorBacken



## 6 任务调度

|  |
| --- |
| 第一步：执行行动算子，多次调用SparkContext的runJob方法；  执行行动算子：collect  打开RDD类，org.apache.spark.rdd.RDD  从第一个collect方法开始：  def collect(): Array[T]  这个方法中调用sc.runJob  sc.runJob中继续调用SparkContext的runJob重载方法  继续调用其重载方法，在重载方法中首先处理闭包，然后继续调用其runJob方法  继续调用其runJob方法，  在这个方法中调用DAGScheduler类的runJob方法，其中DAGScheduler是在初始化SparkContext的时候创建的：  dagScheduler.runJob(rdd, cleanedFunc, partitions, callSite, resultHandler, localProperties.get)  第二步，SparkContext的runJob方法，调用DAGScheduler类的runJob方法，  DAGScheduler类的runJob方法中，调用submitJob方法提交任务：  val waiter: JobWaiter[U] = submitJob(rdd, func, partitions, callSite, resultHandler, properties)  第三步，eventProcessLoop将任务事件提交到阻塞队列；  在DAGScheduler类中的submitJob方法，首先处理分区，然后创建JobWaiter对象，  最后调用eventProcessLoop.post方法：  // 向内部事件循环器发送消息JobSubmitted  eventProcessLoop.post(JobSubmitted(jobId, rdd, func2, partitions.toArray, callSite, waiter,  SerializationUtils.clone(properties)))  // eventProcessLoop: DAGSchedulerEventProcessLoop  任务会提交到一个阻塞队列中，eventQueue.put(event)，其中eventQueue是一个阻塞队列，LinkedBlockingDeque；  下面分析哪个方法会去这个阻塞队列中获取事件？  第四步，创建DAGScheduler对象的时候会调用，eventProcessLoop.start()，在EventLoop的run方法中会将事件逐一取出，调用onReceive(event)，执行doOnReceive(event)方法；  DAGScheduler类的最后有下面这行代码，在创建DAGScheduler对象的时候会调用：  eventProcessLoop.start()，这个方法会调用eventProcessLoop对象所在类的run方法；  eventProcessLoop的类型是DAGSchedulerEventProcessLoop，  需要查看这个类：DAGSchedulerEventProcessLoop，  这个类中没有run，在父类EventLoop的run方法；  在EventLoop的run方法中会将事件逐一取出：  val event = eventQueue.take() // 如果没有事件线程会阻塞在这儿！  然后调用：onReceive(event)，会调用DAGSchedulerEventProcessLoop类的onReceive方法，在这个方法中执行doOnReceive(event)方法：  在def doOnReceive(event: DAGSchedulerEvent)方法中：  case JobSubmitted(jobId, rdd, func, partitions, callSite, listener, properties) => // 处理提交 Job  dagScheduler.handleJobSubmitted(jobId, rdd, func, partitions, callSite, listener, properties)  第五步，DAGScheduler处理提交的任务，handleJobSubmitted方法，创建最后一个阶段并提交！  handleJobSubmitted方法的说明：  在这个方法中主要做了两件事：创建最后一个阶段，提交最后一个阶段！  // Stage的划分是从后向前推断的，所以先创建最后的阶段  finalStage = createResultStage(finalRDD, func, partitions, jobId, callSite)  // 提交finaStage，在内部会递归的提交那些没有提交的父阶段！  submitStage(finalStage)  createResultStage方法说明：  private def createResultStage(  rdd: RDD[\_],func: (TaskContext, Iterator[\_]) => \_, partitions: Array[Int], jobId: Int,  callSite: CallSite): ResultStage = {  // 1. 获取当前 RDD 所有父 Stage 的列表  val parents: List[Stage] = getOrCreateParentStages(rdd, jobId)  // 2. 给resultStage生成一个id  val id = nextStageId.getAndIncrement()  // 3. 创建 ResultStage  val stage: ResultStage = new ResultStage(id, rdd, func, partitions, parents, jobId, callSite)  // 4. stageId 和 ResultStage 做映射  stageIdToStage(id) = stage // 更新map  // 5. 更新ResultStage及其所有祖先的映射关系  updateJobIdStageIdMaps(jobId, stage)  // 6. 返回 ResultStage 对象  stage  }  private def getOrCreateParentStages(rdd: RDD[\_], firstJobId: Int): List[Stage] = {  1. // 得到所有的Shuffle宽依赖  getShuffleDependencies(rdd).map { shuffleDep: ShuffleDependency[\_, \_, \_] =>  2. // 对每个Shuffle宽依赖，获取或者创建新的Stage  getOrCreateShuffleMapStage(shuffleDep, firstJobId)  }.toList  }  // 找宽依赖  private[scheduler] def getShuffleDependencies(rdd: RDD[\_]):  HashSet[ShuffleDependency[\_, \_, \_]] = {  // 存储所有的宽依赖  val parents = new HashSet[ShuffleDependency[\_, \_, \_]]  val visited = new HashSet[RDD[\_]] // 记录以及查找过的  val waitingForVisit = new Stack[RDD[\_]] // 记录待查找的  waitingForVisit.push(rdd)  while (waitingForVisit.nonEmpty) { // 待找的非空  val toVisit: RDD[\_] = waitingForVisit.pop()  if (!visited(toVisit)) { // 没找过  visited += toVisit  toVisit.dependencies.foreach {  case shuffleDep: ShuffleDependency[\_, \_, \_] =>  parents += shuffleDep // shuffle加进hashset  case dependency =>  waitingForVisit.push(dependency.rdd) // 放到待找的里面  }  }  }  parents  }  getOrCreateShuffleMapStage方法创建阶段  // Finally, create a stage for the given shuffle dependency.  createShuffleMapStage(shuffleDep, firstJobId)  // 上面是创建最后一个阶段，下面是提交最后一个阶段  submitStage(finalStage)方法说明：// 深度优先搜索算法  // 提交stage，但是会递归的提交没有添加的父stage  // 先去找没有提交父阶段的，没有直接提交，有递归提交  private def submitStage(stage: Stage) {  val jobId: Option[Int] = activeJobForStage(stage)  if (jobId.isDefined) {  logDebug("submitStage(" + stage + ")")  if (!waitingStages(stage) && !runningStages(stage) && !failedStages(stage)) {  // 1. 找到没有提交的父stage，先创建的id 小  val missing: List[Stage] = getMissingParentStages(stage).sortBy(\_.id)  logDebug("missing: " + missing)  if (missing.isEmpty) { // 2. 如果为空, 则从当前stage直接提交  logInfo("Submitting " + stage + " (" + stage.rdd + "), which has no missing parents")  submitMissingTasks(stage, jobId.get) // 提交没有提交的任务  } else { // 3. 如果不为空, 则递归的向上查找  for (parent <- missing) {  submitStage(parent)  }  // 4. 当前 stage加入到等待提交的 stage 对列中  waitingStages += stage  }  }  } else {  abortStage(stage, "No active job for stage " + stage.id, None)  }  }  最终提交任务的方法：private def submitMissingTasks(stage: Stage, jobId: Int)  // 计算所有分区的 id  val partitionsToCompute: Seq[Int] = stage.findMissingPartitions()  // 根据stage的不同，ShuffleMapStage还是ResultStage，划分task将任务封装成TaskSet，交给TaskScheduler调度：  taskScheduler.submitTasks(new TaskSet(  tasks.toArray, stage.id, stage.latestInfo.attemptId, jobId, properties))  // DAG级别的调度，主要做了两件事：划分stage;  提交stage：将每个stage封装成TaskSet，交给TaskScheduler去调度！ |

|  |
| --- |
| TaskSchedulerImpl类中的override def submitTasks(taskSet: TaskSet)方法说明：  1 // 创建TaskSetManger对象，用来追踪每个任务，每个stage对应一个TaskSetManager  val manager: TaskSetManager = createTaskSetManager(taskSet, maxTaskFailures)  // 最大允许失败次数  2 // 任务调度构建器把任务交给任务调度池来调度  schedulableBuilder.addTaskSetManager(manager, manager.taskSet.properties)  3 // SchedulerBackend发送信息进行通信  backend.reviveOffers()  2 说明  // 把任务放在调度池中  def addTaskSetManager(manager: Schedulable, properties: Properties): Unit  SchedulableBuilder对应两个实现类：  FIFOSchedulableBuilder(val rootPool: Pool) // 根调度池在初始化sc时创建  FairSchedulableBuilder(val rootPool: Pool, conf: SparkConf)  // 分别对应两个调度算法  FIFOSchedulingAlgorithm  FairSchedulingAlgorithm  3 说明： // 通信  backend.reviveOffers()查看SchedulerBackend的子类CoarseGrainedSchedulerBackend  override def reviveOffers() {  driverEndpoint.send(ReviveOffers) // RpcEndpointRef类型  }  发送ReviveOffers 在DriverEndpoint 类中接收，看receive方法！  在receive方法中模式匹配接收：  case ReviveOffers => makeOffers()  查看makeOffers()方法：  做了三件事：过滤出Active的Executor，封装资源，启动任务！  launchTasks(scheduler.resourceOffers(workOffers))  launchTasks方法：  做了两件事：序列化任务，发送任务到Executor  executorData.executorEndpoint.send(LaunchTask(new SerializableBuffer(serializedTask))) |

|  |
| --- |
| 下面查看CoarseGrainedExecutorBackend的receive方法：  case LaunchTask(data) =>  // 反序列化任务以后，启动任务  executor.launchTask  Executor类的launchTask方法说明：  创建Runnable接口对象：val tr = new TaskRunner  在线程池中执行：threadPool.execute(tr)  下面查看TaskRunner的run方法：  更新task的状态，反序列化任务相关的数据，调用task的run方法！  Task类的run方法：  创建context，运行task：runTask(context)  Task有两个子类，ResultTask和ShuffleMapTask  ResultTask中runTask方法：  // 匿名函数 func在每个executor上执行 // excutor走到这，开始执行啦  func(context, rdd.iterator(partition, context))  ShuffleMapTask的runTask方法：  // 写出RDD中的数据，然后reduce阶段的task去读  writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[\_ <: Product2[Any, Any]]])  至此，调度结束！ |

## 7 shuffle过程

略