# 深入理解Spark 2.1 Core（五）：Standalone模式运行的原理与源码分析



## 概述

前几篇博文都在介绍Spark的调度，这篇博文我们从更加宏观的调度看Spark，讲讲Spark的部署模式。Spark部署模式分以下几种：

- local 模式
- local-cluster 模式
- Standalone 模式
- YARN 模式
- Mesos 模式

我们先来简单介绍下YARN模式，然后深入讲解Standalone模式。

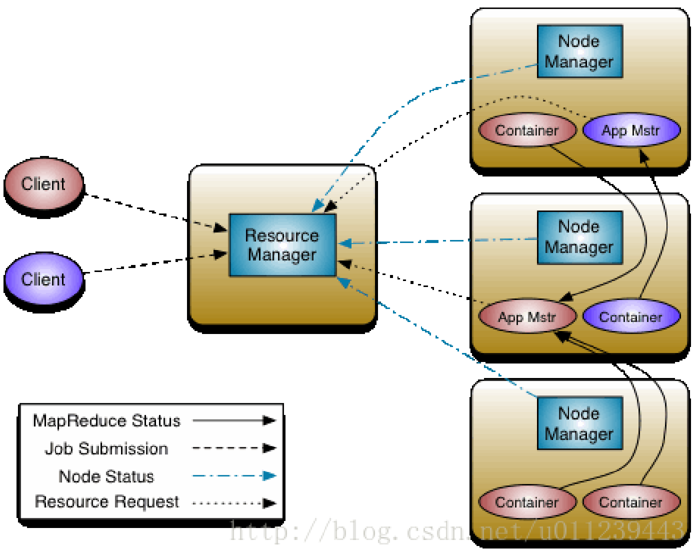
## YARN 模式介绍

### YARN介绍

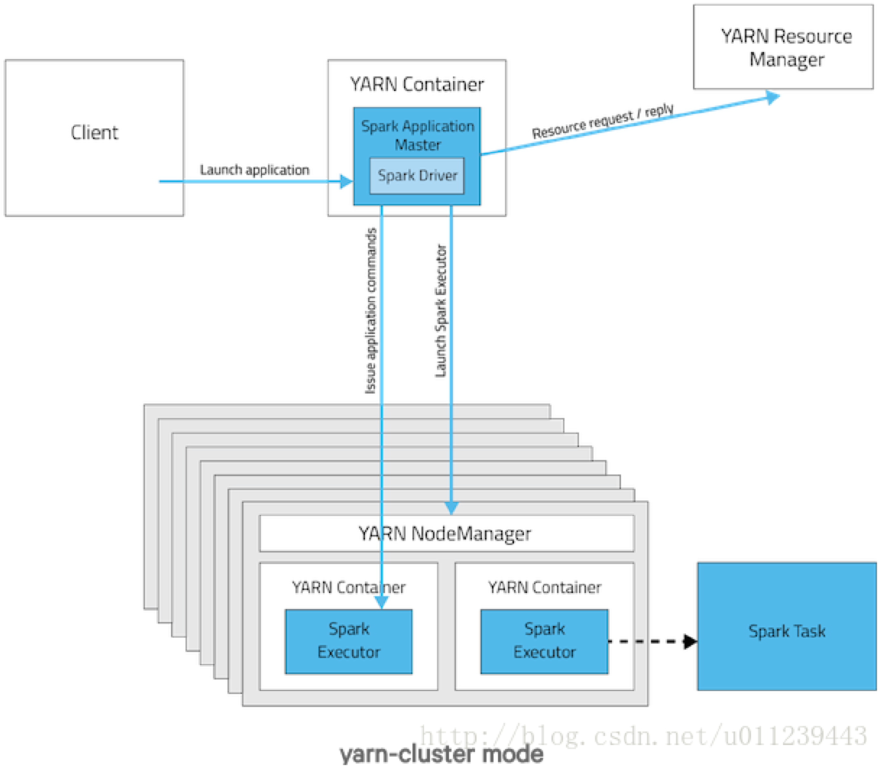
YARN 是一个资源管理、任务调度的框架，主要包含三大模块：ResourceManager（RM）、NodeManager（NM）、ApplicationMaster（AM）。

其中，ResourceManager负责所有资源的监控、分配和管理；ApplicationMaster负责每一个具体应用程序的调度和协调；NodeManager负责每一个节点的维护。

对于所有的applications，RM拥有绝对的控制权和对资源的分配权。而每个AM则会和RM协商资源，同时和NodeManager通信来执行和监控task。几个模块之间的关系如图所示。



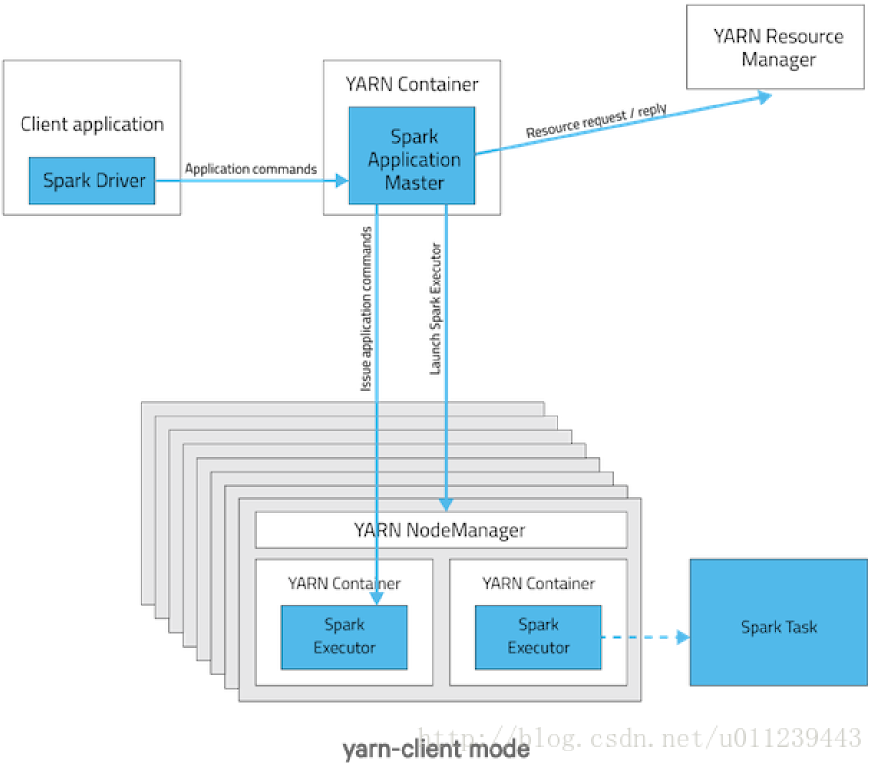

### Yarn Cluster 模式




Spark的Yarn Cluster 模式流程如下：

- 本地用YARN Client 提交App 到 Yarn Resource Manager
- Yarn Resource Manager 选个 YARN Node Manager，用它来
  - 创建个ApplicationMaster，SparkContext相当于是这个ApplicationMaster管的APP，生成YarnClusterScheduler与YarnClusterSchedulerBackend
  - 选择集群中的容器启动CoarseCrainedExecutorBackend，用来启动spark.executor。
- ApplicationMaster与CoarseCrainedExecutorBackend会有远程调用。

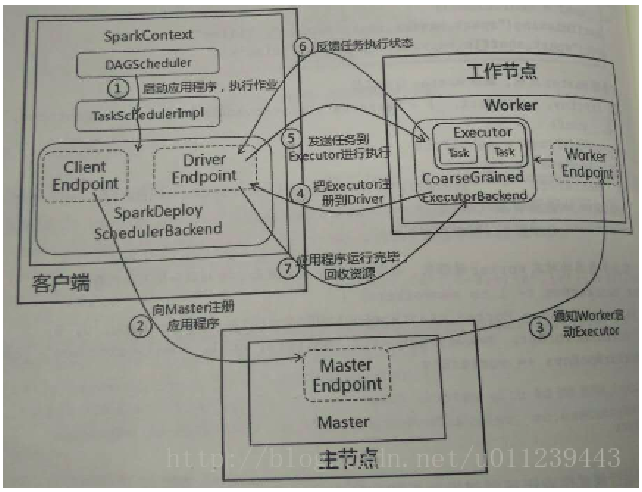
## Yarn Client 模式




Spark的Yarn Client 模式流程如下：

- 本地启动SparkContext，生成YarnClientClusterScheduler 和 YarnClientClusterSchedulerBackend
- YarnClientClusterSchedulerBackend启动yarn.Client，用它提交App 到 Yarn Resource Manager
- Yarn Resource Manager 选个 YARN Node Manager，用它来选择集群中的容器启动CoarseCrainedExecutorBackend，用来启动spark.executor
- YarnClientClusterSchedulerBackend与CoarseCrainedExecutorBackend会有远程调用。

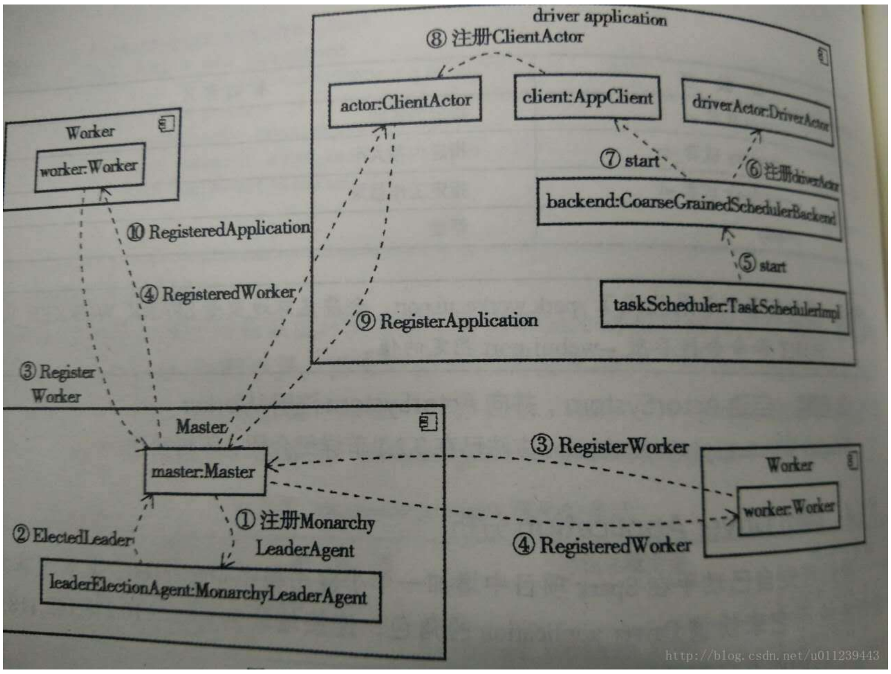
## Standalone 模式介绍




1. 启动app，在SparkContxt启动过程中，先初始化DAGScheduler 和 TaskScheduler，并初始化 SparkDeploySchedulerBackend，并在其内部启动DriverEndpoint和ClientEndpoint。

2. ClientEndpoint想Master注册app，Master收到注册信息后把该app加入到等待运行app列表中，等待由Master分配给该app worker。
3. app获取到worker后，Master通知Worker的WorkerEndpont创建CoarseGrainedExecutorBackend进程，在该进程中创建执行容器executor
4. executor创建完毕后发送信息给Master和DriverEndpoint，告知Executor创建完毕，在SparkContext注册，后等待DriverEndpoint发送执行任务的消息。
5. SparkContext分配TaskSet给CoarseGrainedExecutorBackend，按一定调度策略在executor执行。详见：《深入理解Spark 2.1 Core （二）：DAG调度器的实现与源码分析 》与《深入理解Spark 2.1 Core （三）：任务调度器的实现与源码分析 》
6. CoarseGrainedExecutorBackend在Task处理的过程中，把处理Task的状态发送给DriverEndpoint，Spark根据不同的执行结果来处理。若处理完毕，则继续发送其他TaskSet。详见：《深入理解Spark 2.1 Core （四）：运算结果处理和容错的实现与源码分析 》
7. app运行完成后，SparkContext会进行资源回收，销毁Worker的CoarseGrainedExecutorBackend进程，然后注销自己。

# Standalone 启动集群




## 启动Master

### master.Master

我们先来看下Master对象的main函数做了什么：

```
private[deploy] object Master extends Logging {
  val SYSTEM_NAME = "sparkMaster"
  val ENDPOINT_NAME = "Master"

  def main(argStrings: Array[String]) {
    Utils.initDaemon(log)
    //创建SparkConf
    val conf = new SparkConf
    //解析SparkConf参数
    val args = new MasterArguments(argStrings, conf)
```

```scala
    val (rpcEnv, _, _) = startRpcEnvAndEndpoint(args.host, args.port, args.webUiPort, conf)
    rpcEnv.awaitTermination()
  }

  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)
    //创建Master
    val masterEndpoint = rpcEnv.setupEndpoint(ENDPOINT_NAME,
      new Master(rpcEnv, rpcEnv.address, webUiPort, securityMgr, conf))
    val portsResponse = masterEndpoint.askWithRetry[BoundPortsResponse](BoundPortsRequest)
    //返回 Master RpcEnv,
    //web UI 端口,
    //其他服务的端口
    (rpcEnv, portsResponse.webUIPort, portsResponse.restPort)
  }
}
```

## master.MasterArguments

接下来我们看看master是如何解析参数的：

```scala
private[master] class MasterArguments(args: Array[String], conf: SparkConf) extends Logging {
  //默认配置
  var host = Utils.localHostName()
  var port = 7077
  var webUiPort = 8080
  //Spark属性文件
  //默认为 spark-default.conf
  var propertiesFile: String = null

  // 检查环境变量
  if (System.getenv("SPARK_MASTER_IP") != null) {
    logWarning("SPARK_MASTER_IP is deprecated, please use SPARK_MASTER_HOST")
    host = System.getenv("SPARK_MASTER_IP")
  }

  if (System.getenv("SPARK_MASTER_HOST") != null) {
    host = System.getenv("SPARK_MASTER_HOST")
  }
  if (System.getenv("SPARK_MASTER_PORT") != null) {
    port = System.getenv("SPARK_MASTER_PORT").toInt
  }
  if (System.getenv("SPARK_MASTER_WEBUI_PORT") != null) {
    webUiPort = System.getenv("SPARK_MASTER_WEBUI_PORT").toInt
  }

  parse(args.toList)

  // 转变SparkConf
  propertiesFile = Utils.loadDefaultSparkProperties(conf, propertiesFile)
  //环境变量的SPARK_MASTER_WEBUI_PORT
  //会被Spark属性spark.master.ui.port所覆盖
  if (conf.contains("spark.master.ui.port")) {
    webUiPort = conf.get("spark.master.ui.port").toInt
  }

  //解析命令行参数
  //命令行参数会把环境变量和Spark属性都覆盖
  @tailrec
```

```
  private def parse(args: List[String]): Unit = args match {
    case ("--ip" | "-i") :: value :: tail =>
      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 =>

    case _ =>
      printUsageAndExit(1)
  }


  private def printUsageAndExit(exitCode: Int) {
    System.err.println(
      "Usage: Master [options]\n" +
      "\n" +
      "Options:\n" +
      "  -i HOST, --ip HOST     Hostname to listen on (deprecated, please use --host or -h) \n" +
      "  -h HOST, --host HOST   Hostname to listen on\n" +
      "  -p PORT, --port PORT   Port to listen on (default: 7077)\n" +
      "  --webui-port PORT      Port for web UI (default: 8080)\n" +
      "  --properties-file FILE Path to a custom Spark properties file.\n" +
      "                         Default is conf/spark-defaults.conf.")
    System.exit(exitCode)
  }
}
```

我们可以看到上述参数设置的优先级别为：

系统环境变量 < spark-default.conf中的属性 < 命令行参数 < 应用级代码中的参数设置

## 启动Worker

### worker.Worker

我们先来看下Worker对象的main函数做了什么：

```
private[deploy] object Worker extends Logging {
  val SYSTEM_NAME = "sparkWorker"
  val ENDPOINT_NAME = "Worker"

  def main(argStrings: Array[String]) {
    Utils.initDaemon(log)
```

```
    //创建SparkConf
    val conf = new SparkConf
    //解析SparkConf参数
    val args = new WorkerArguments(argStrings, conf)
    val rpcEnv = startRpcEnvAndEndpoint(args.host, args.port, args.webUiPort, args.cores,
      args.memory, args.masters, args.workDir, conf = conf)
    rpcEnv.awaitTermination()
  }

  def startRpcEnvAndEndpoint(
      host: String,
      port: Int,
      webUiPort: Int,
      cores: Int,
      memory: Int,
      masterUrls: Array[String],
      workDir: String,
      workerNumber: Option[Int] = None,
      conf: SparkConf = new SparkConf): RpcEnv = {


    val systemName = SYSTEM_NAME + workerNumber.map(_.toString).getOrElse("")
    val securityMgr = new SecurityManager(conf)
    val rpcEnv = RpcEnv.create(systemName, host, port, conf, securityMgr)
    val masterAddresses = masterUrls.map(RpcAddress.fromSparkURL(_))
    //创建Worker
    rpcEnv.setupEndpoint(ENDPOINT_NAME, new Worker(rpcEnv, webUiPort, cores, memory,
      masterAddresses, ENDPOINT_NAME, workDir, conf, securityMgr))
    rpcEnv
  }

  ***
```

## worker.WorkerArguments

worker.WorkerArguments与master.MasterArguments类似：

```
private[worker] class WorkerArguments(args: Array[String], conf: SparkConf) {
  var host = Utils.localHostName()
  var port = 0
  var webUiPort = 8081
  var cores = inferDefaultCores()
  var memory = inferDefaultMemory()
  var masters: Array[String] = null
  var workDir: String = null
  var propertiesFile: String = null

 // 检查环境变量
  if (System.getenv("SPARK_WORKER_PORT") != null) {
    port = System.getenv("SPARK_WORKER_PORT").toInt
  }
  if (System.getenv("SPARK_WORKER_CORES") != null) {
    cores = System.getenv("SPARK_WORKER_CORES").toInt
  }
  if (conf.getenv("SPARK_WORKER_MEMORY") != null) {
    memory = Utils.memoryStringToMb(conf.getenv("SPARK_WORKER_MEMORY"))
  }
  if (System.getenv("SPARK_WORKER_WEBUI_PORT") != null) {
    webUiPort = System.getenv("SPARK_WORKER_WEBUI_PORT").toInt
  }
  if (System.getenv("SPARK_WORKER_DIR") != null) {
    workDir = System.getenv("SPARK_WORKER_DIR")
  }

```

```
parse(args.toList)

// 转变SparkConf
propertiesFile = Utils.loadDefaultSparkProperties(conf, propertiesFile)

if (conf.contains("spark.worker.ui.port")) {
  webUiPort = conf.get("spark.worker.ui.port").toInt
}

checkWorkerMemory()

@tailrec
private def parse(args: List[String]): Unit = args match {
  case ("--ip" | "-i") :: value :: tail =>
    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 ("--cores" | "-c") :: IntParam(value) :: tail =>
    cores = value
    parse(tail)

  case ("--memory" | "-m") :: MemoryParam(value) :: tail =>
    memory = value
    parse(tail)

  //工作目录
  case ("--work-dir" | "-d") :: value :: tail =>
    workDir = 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 value :: tail =>
    if (masters != null) {  // Two positional arguments were given
      printUsageAndExit(1)
    }
    masters = Utils.parseStandaloneMasterUrls(value)
    parse(tail)

  case Nil =>
    if (masters == null) {  // No positional argument was given
      printUsageAndExit(1)
    }

  case _ =>
```

```
      printUsageAndExit(1)
    }

***
```

# 资源回收

我们在概述中提到了“ app运行完成后，SparkContext会进行资源回收，销毁Worker的CoarseGrainedExecutorBackend进程，然后注销自己。”接下来我们就来讲解下Master和Executor是如何感知到Application的退出的。
调用栈如下：

- SparkContext.stop
  - DAGScheduler.stop
    - TaskSchedulerImpl.stop
      - CoarseGrainedSchedulerBackend.stop
        - CoarseGrainedSchedulerBackend.stopExecutors
          - CoarseGrainedSchedulerBackend.DriverEndpoint.receiveAndReply
            - CoarseGrainedExecutorBackend.receive
              - Executor.stop
          - CoarseGrainedSchedulerBackend.DriverEndpoint.receiveAndReply

## SparkContext.stop

SparkContext.stop会调用DAGScheduler.stop

```
***
    if (_dagScheduler != null) {
      Utils.tryLogNonFatalError {
        _dagScheduler.stop()
      }
      _dagScheduler = null
    }

***
```

## DAGScheduler.stop

DAGScheduler.stop会调用TaskSchedulerImpl.stop

```
  def stop() {
  //停止消息调度
    messageScheduler.shutdownNow()
  //停止事件处理循环
    eventProcessLoop.stop()
  //调用TaskSchedulerImpl.stop
    taskScheduler.stop()
  }
```

## TaskSchedulerImpl.stop

TaskSchedulerImpl.stop会调用CoarseGrainedSchedulerBackend.stop

```
override def stop() {
  //停止推断
  speculationScheduler.shutdown()
  //调用CoarseGrainedSchedulerBackend.stop
  if (backend != null) {
    backend.stop()
  }
  //停止结果获取
  if (taskResultGetter != null) {
    taskResultGetter.stop()
  }
  starvationTimer.cancel()
}
```

## CoarseGrainedSchedulerBackend.stop

```
override def stop() {
  //调用stopExecutors()
  stopExecutors()
  try {
    if (driverEndpoint != null) {
    //发送StopDriver信号
      driverEndpoint.askWithRetry[Boolean](StopDriver)
    }
  } catch {
    case e: Exception =>
      throw new SparkException("Error stopping standalone scheduler's driver endpoint", e)
  }
}
```

## CoarseGrainedSchedulerBackend.stopExecutors

我们先来看下CoarseGrainedSchedulerBackend.stopExecutors

```
def stopExecutors() {
  try {
    if (driverEndpoint != null) {
      logInfo("Shutting down all executors")
      //发送StopExecutors信号
      driverEndpoint.askWithRetry[Boolean](StopExecutors)
    }
  } catch {
    case e: Exception =>
      throw new SparkException("Error asking standalone scheduler to shut down executors", e)
  }
}
```

## CoarseGrainedSchedulerBackend.DriverEndpoint.receiveAndReply

DriverEndpoint接收并回应该信号：

```
    case StopExecutors =>
      logInfo("Asking each executor to shut down")
      for ((_, executorData) <- executorDataMap) {
        //给CoarseGrainedExecutorBackend发送StopExecutor信号
        executorData.executorEndpoint.send(StopExecutor)
      }
      context.reply(true)
```

## CoarseGrainedExecutorBackend.receive

CoarseGrainedExecutorBackend接收该信号：

```
    case StopExecutor =>
      stopping.set(true)
      logInfo("Driver commanded a shutdown")
      //这里并没有直接关闭Executor，
      //因为Executor必须先返回确认帧给CoarseGrainedSchedulerBackend
      //所以，这的策略是给自己再发一个Shutdown信号，然后处理
      self.send(Shutdown)

    case Shutdown =>
      stopping.set(true)
      new Thread("CoarseGrainedExecutorBackend-stop-executor") {
        override def run(): Unit = {
          // executor.stop() 会调用 `SparkEnv.stop()`
          // 直到 RpcEnv 彻底结束
          // 但是，如果 `executor.stop()` 运行在和RpcEnv相同的线程里面，
          // RpcEnv 会等到`executor.stop()`结束后才能结束，
          // 这就产生了死锁
          // 因此，我们需要新建一个线程
          executor.stop()
        }
```

## Executor.stop

```
  def stop(): Unit = {
    env.metricsSystem.report()
    //关闭心跳
    heartbeater.shutdown()
    heartbeater.awaitTermination(10, TimeUnit.SECONDS)
    //关闭线程池
    threadPool.shutdown()
    if (!isLocal) {
    //停止SparkEnv
      env.stop()
    }
  }
```

## CoarseGrainedSchedulerBackend.DriverEndpoint.receiveAndReply

我们回过头来看CoarseGrainedSchedulerBackend.stop，调用stopExecutors()结束后，会给 driverEndpoint发送StopDriver信号。CoarseGrainedSchedulerBackend.DriverEndpoint.接收信号并回复：

```
      case StopDriver =>
        context.reply(true)
        //停止driverEndpoint
        stop()
```