深入理解Spark 2.1 Core（六）：资源调度的原理与源码分析

http://blog.csdn.net/u011239443/article/details/54098376

在上篇博文《深入理解Spark 2.1 Core （五）：Standalone模式运行的实现与源码分析》 中，我们讲到了如何启动Master和Worker，还讲到了如何回收资源。但是，我们没有将AppClient是如何启动的，其实它们的启动也涉及到了资源是如何调度的。这篇博文，我们就来讲一下AppClient的启动和逻辑与物理上的资源调度。

启动AppClient

调用栈如下：

- StandaloneSchedulerBackend.start
    - StandaloneAppClient.start
        - StandaloneAppClient.ClientEndpoint.onStart
            - StandaloneAppClient.registerWithMaster
                - StandaloneAppClient.tryRegisterAllMasters
- Master.receive
    - Master.createApplication
    - Master.registerApplication
- StandaloneAppClient.ClientEndpoint.receive

StandaloneSchedulerBackend.start

在Standalone模式下，SparkContext中的backend是StandaloneSchedulerBackend。在StandaloneSchedulerBackend.start中可以看到：

```
***
    val appUIAddress = sc.ui.map(_.appUIAddress).getOrElse("")
    val coresPerExecutor = conf.getOption("spark.executor.cores").map(_.toInt)

    val initialExecutorLimit =
      if (Utils.isDynamicAllocationEnabled(conf)) {
        Some(0)
      } else {
        None
      }
    val appDesc = new ApplicationDescription(sc.appName, maxCores, sc.executorMemory, command,
      appUIAddress, sc.eventLogDir, sc.eventLogCodec, coresPerExecutor, initialExecutorLimit)
      //创建AppClient
    client = new StandaloneAppClient(sc.env.rpcEnv, masters, appDesc, this, conf)
    //启动AppClient
    client.start()
 ***
```

StandaloneAppClient.start

```
def start() {
  //生成了ClientEndpoint，于是调用其onStart
  endpoint.set(rpcEnv.setupEndpoint("AppClient", new ClientEndpoint(rpcEnv)))
}
```

StandaloneAppClient.ClientEndpoint.onStart

调用registerWithMaster

```
  override def onStart(): Unit = {
    try {
      registerWithMaster(1)
    } catch {
      case e: Exception =>
        logWarning("Failed to connect to master", e)
        markDisconnected()
        stop()
    }
  }
```

StandaloneAppClient.registerWithMaster

```

  private def registerWithMaster(nthRetry: Int) {
          //向所有的Master注册当前App
         //一旦成功连接的一个master，其他将被取消
         registerMasterFutures.set(tryRegisterAllMasters())
    registrationRetryTimer.set(registrationRetryThread.schedule(new Runnable {
      override def run(): Unit = {
        if (registered.get) {

         registerMasterFutures.get.foreach(_.cancel(true))
          registerMasterThreadPool.shutdownNow()
        }
        //若达到最大尝试次数，则标志死亡，默认为3
        else if (nthRetry >= REGISTRATION_RETRIES) {
          markDead("All masters are unresponsive! Giving up.")
        } else {
          registerMasterFutures.get.foreach(_.cancel(true))
          registerWithMaster(nthRetry + 1)
        }
      }
    }, REGISTRATION_TIMEOUT_SECONDS, TimeUnit.SECONDS))
  }
```

StandaloneAppClient.tryRegisterAllMasters

给Master发送RegisterApplication信号：

```
  private def tryRegisterAllMasters(): Array[JFuture[_]] = {
    for (masterAddress <- masterRpcAddresses) yield {
      registerMasterThreadPool.submit(new Runnable {
        override def run(): Unit = try {
          if (registered.get) {
            return
          }
          logInfo("Connecting to master " + masterAddress.toSparkURL + "...")
          val masterRef = rpcEnv.setupEndpointRef(masterAddress, Master.ENDPOINT_NAME)
          masterRef.send(RegisterApplication(appDescription, self))
        } catch {
          case ie: InterruptedException => // Cancelled
          case NonFatal(e) => logWarning(s"Failed to connect to master $masterAddress", e)
```

```
          }
        })
      }
    }
```

Master.receive

Master.receive接收并处理RegisterApplication信号

```
    case RegisterApplication(description, driver) =>
      // 若之前注册过
      if (state == RecoveryState.STANDBY) {
        // 忽略
      } else {
        logInfo("Registering app " + description.name)
        //创建app
        val app = createApplication(description, driver)
        //注册app
        registerApplication(app)
        logInfo("Registered app " + description.name + " with ID " + app.id)
        //持久化
        persistenceEngine.addApplication(app)
        //回复RegisteredApplication信号
        driver.send(RegisteredApplication(app.id, self))
        //资源调度
        schedule()
      }
```

让我们深入来看下Master是如何注册app的。

Master.createApplication

先创建app：

```
  private def createApplication(desc: ApplicationDescription, driver: RpcEndpointRef):
      ApplicationInfo = {
    val now = System.currentTimeMillis()
    val date = new Date(now)
    //根据日期生成appId
    val appId = newApplicationId(date)
    //传入 时间，appId，描述信息，日期，driver，默认核数，
    //生成app信息
    new ApplicationInfo(now, appId, desc, date, driver, defaultCores)
  }
```

Master.registerApplication

再注册app：

```
  private def registerApplication(app: ApplicationInfo): Unit = {
     //若已有这个app地址，
     //则返回
    val appAddress = app.driver.address
    if (addressToApp.contains(appAddress)) {
      logInfo("Attempted to re-register application at same address: " + appAddress)
      return
    }

   //向 applicationMetricsSystem 注册appSource
   applicationMetricsSystem.registerSource(app.appSource)
   //将app加入到 集合
   //HashSet[ApplicationInfo]
```

```
    apps += app
    //更新 id到App
    //HashMap[String, ApplicationInfo]
    idToApp(app.id) = app
    //更新 endpoint到App
    // HashMap[RpcEndpointRef, ApplicationInfo]
    endpointToApp(app.driver) = app
    //更新 address到App
    // HashMap[RpcAddress, ApplicationInfo]
    addressToApp(appAddress) = app
    // 加入到等待数组中
    //ArrayBuffer[ApplicationInfo]
    waitingApps += app
    if (reverseProxy) {
      webUi.addProxyTargets(app.id, app.desc.appUiUrl)
    }
  }
```

StandaloneAppClient.ClientEndpoint.receive

```
      case RegisteredApplication(appId_, masterRef) =>
      //这里的代码有两个缺陷：
      //1. 一个Master可能接收到多个注册请求，
      // 并且回复多个RegisteredApplication信号，
      //这会导致网络不稳定。
      //2.若master正在变化，
      //则会接收到多个RegisteredApplication信号
        //设置appId
        appId.set(appId_)
        //编辑已经注册
        registered.set(true)
        //创建master信息
        master = Some(masterRef)
        //绑定监听
        listener.connected(appId.get)
```

逻辑资源调度

我们可以看到在上一章，Master.receive接收并处理RegisterApplication信号时的最后一行代码：

```
        //资源调度
        schedule()
```

下面，我们就来讲讲资源调度。

调用栈如下：

- Master.schedule
  - Master.startExecutorsOnWorkers
    - Master.scheduleExecutorsOnWorkers
    - Master.allocateWorkerResourceToExecutors

Master.schedule

该方法主要来在等待的app之间调度资源。每次有新的app加入或者可用资源改变的时候，这个方法都会被调用：

```
  private def schedule(): Unit = {
    if (state != RecoveryState.ALIVE) {
      return
```

```
    }
    // 得到活的Worker，
    // 并打乱它们
    val shuffledAliveWorkers = Random.shuffle(workers.toSeq.filter(_.state == WorkerState.ALIVE))
    // worker数量
    val numWorkersAlive = shuffledAliveWorkers.size
    var curPos = 0
    //为driver分配资源，
    //该调度策略为FIFO的策略，
    //先来的driver会先满足其资源所需的条件
    for (driver <- waitingDrivers.toList) {
      var launched = false
      var numWorkersVisited = 0
      while (numWorkersVisited < numWorkersAlive && !launched) {
        val worker = shuffledAliveWorkers(curPos)
        numWorkersVisited += 1
        if (worker.memoryFree >= driver.desc.mem && worker.coresFree >= driver.desc.cores) {
          launchDriver(worker, driver)
          waitingDrivers -= driver
          launched = true
        }
        curPos = (curPos + 1) % numWorkersAlive
      }
    }
    //启动worker上的executor
    startExecutorsOnWorkers()
  }
```

Master.startExecutorsOnWorkers

接下来我们来看下executor的启动：

```
  private def startExecutorsOnWorkers(): Unit = {
    // 这里还是使用的FIFO的调度方式
    for (app <- waitingApps if app.coresLeft > 0) {
      val coresPerExecutor: Option[Int] = app.desc.coresPerExecutor
      // 过滤掉资源不够启动executor的worker
      // 并按资源逆序排序
      val usableWorkers = workers.toArray.filter(_.state == WorkerState.ALIVE)
        .filter(worker => worker.memoryFree >= app.desc.memoryPerExecutorMB &&
          worker.coresFree >= coresPerExecutor.getOrElse(1))
        .sortBy(_.coresFree).reverse
        //调度worker上的executor，
        //确定在每个worker上给这个app分配多少核
      val assignedCores = scheduleExecutorsOnWorkers(app, usableWorkers, spreadOutApps)

      //分配
      for (pos <- 0 until usableWorkers.length if assignedCores(pos) > 0) {
        allocateWorkerResourceToExecutors(
          app, assignedCores(pos), coresPerExecutor, usableWorkers(pos))
      }
    }
  }
```

Master.scheduleExecutorsOnWorkers

接下来我们就来讲讲核心的worker上的executor资源调度。在将现在的Spark代码之前，我们看看在Spark1.4之前，这部分逻辑是如何实现的：

```
***
        val numUsable = usableWorkers.length
        // 用来记录每个worker已经分配的核数
        val assigned = new Array[Int](numUsable)
```

```
        var toAssign = math.min(app.coresLeft, usableWorkers.map(_.coresFree).sum)
        var pos = 0
        while (toAssign > 0) {
        //遍历worker，
        //若当前worker还存在资源，
        //则分配掉1个核。
        //直到workers的资源全都被分配掉，
        //或者是app所需要的资源被满足。
          if (usableWorkers(pos).coresFree - assigned(pos) > 0) {
            toAssign -= 1
            assigned(pos) += 1
                      }
          pos = (pos + 1) % numUsable
        }
***
```

在Spark1.4的时候，这段代码被修改了。我们来想一下，以上代码有什么问题?

问题就在于，core是一个一个的被分配的。设想，一个集群中有4 worker，每个worker有16个core。用户想启动3个executor，且每个executor拥有16个core。于是，他会这样配置参数：

```
spark.cores.max = 48
spark.executor.cores = 16
```

显然，我们集群的资源是能满足用户的需求的。但如果一次只能分配一个core，那最终的结果是每个worker上都分配了12个core。由于12 < 16, 所有没有一个executor能够启动。

下面，我们回过头来看现在的源码中是如何实现这部分逻辑的：

```
  private def scheduleExecutorsOnWorkers(
      app: ApplicationInfo,
      usableWorkers: Array[WorkerInfo],
      spreadOutApps: Boolean): Array[Int] = {
    val coresPerExecutor = app.desc.coresPerExecutor
    val minCoresPerExecutor = coresPerExecutor.getOrElse(1)
    val oneExecutorPerWorker = coresPerExecutor.isEmpty
    val memoryPerExecutor = app.desc.memoryPerExecutorMB
    val numUsable = usableWorkers.length
    // 用来记录每个worker已经分配的核数
    val assignedCores = new Array[Int](numUsable)
    // 用来记录每个worker已经分配的executor数
    val assignedExecutors = new Array[Int](numUsable)
    // 剩余总共资源
    var coresToAssign = math.min(app.coresLeft, usableWorkers.map(_.coresFree).sum)

    //判断是否能启动Executor
    def canLaunchExecutor(pos: Int): Boolean = {
      //先省略
    }


    //过滤去能启动executor的Worker
    var freeWorkers = (0 until numUsable).filter(canLaunchExecutor)
    //调度资源，
    //直到worker上的executor被分配完
    while (freeWorkers.nonEmpty) {
      freeWorkers.foreach { pos =>
        var keepScheduling = true
        while (keepScheduling && canLaunchExecutor(pos)) {
           // minCoresPerExecutor 是用户设置的 spark.executor.cores
          coresToAssign -= minCoresPerExecutor
          assignedCores(pos) += minCoresPerExecutor

```

```
        // 若用户没有设置 spark.executor.cores
        // 则oneExecutorPerWorker就为True
        // 也就是说，assignedCores中的core都被一个executor使用
        // 若用户设置了spark.executor.cores，
        // 则该Worker的assignedExecutors会加1
        if (oneExecutorPerWorker) {
          assignedExecutors(pos) = 1
        } else {
          assignedExecutors(pos) += 1
        }


        //资源分配算法有两种：
        // 1. 尽量打散，将一个app尽可能的分配到不同的节点上，
        // 这有利于充分利用集群的资源，
        // 在这种情况下，spreadOutApps设为True，
        // 于是，该worker分配好了一个executor之后就退出循环
        // 轮询到下一个worker
        // 2. 尽量集中，将一个app尽可能的分配到同一个的节点上，
        // 这适合cpu密集型而内存占用比较少的app
        // 在这种情况下，spreadOutApps设为False，
        // 于是，继续下一轮的循环
        // 在该Worker上分配executor
        if (spreadOutApps) {
          keepScheduling = false
        }
      }
    }
    freeWorkers = freeWorkers.filter(canLaunchExecutor)
  }
  assignedCores
}
```

接下来看下该函数的内部函数canLaunchExecutor：

```
def canLaunchExecutor(pos: Int): Boolean = {
// 条件1 ：若集群剩余core >= spark.executor.cores
  val keepScheduling = coresToAssign >= minCoresPerExecutor
// 条件2： 若该Worker上的剩余core >= spark.executor.cores
  val enoughCores = usableWorkers(pos).coresFree - assignedCores(pos) >= minCoresPerExecutor


  // 条件3： 若设置了spark.executor.cores
  // 或者 该Worker还未分配executor
  val launchingNewExecutor = !oneExecutorPerWorker || assignedExecutors(pos) == 0
  if (launchingNewExecutor) {
    val assignedMemory = assignedExecutors(pos) * memoryPerExecutor
    // 条件4：若该Worker上的剩余内存 >= spark.executor.memory
    val enoughMemory = usableWorkers(pos).memoryFree - assignedMemory >= memoryPerExecutor
    // 条件5： 若分配了该executor后，
    // 总共分配的core数量 <= spark.cores.max
    val underLimit = assignedExecutors.sum + app.executors.size < app.executorLimit
    //若满足 条件3，
    //且满足 条件1，条件2，条件4，条件5
    //则返回True
    keepScheduling && enoughCores && enoughMemory && underLimit
  } else {
    //若不满足 条件3，
    //即一个worker只有一个executor
    //且满足 条件1，条件2
    //也返回True。
    // 返回后，不会增加 assignedExecutors
    keepScheduling && enoughCores

```

```
      }
    }
```

通过以上源码，我们可以清楚看到，Spark1.4以后新的逻辑实现其实就是将分配单位从原来的一个core，变为了一个executor（即spark.executor.cores）。而若一个worker上只有一个executor（即没有设置spark.executor.cores），那么就按照原来的逻辑实现。

值得我注意的是：

```
    //直到worker上的executor被分配完
    while (freeWorkers.nonEmpty)
```

一个app会尽可能的***使用掉集群的所有资源，所以设置spark.cores.max参数是非常有必要的！***

Master.allocateWorkerResourceToExecutors

现在我们回到上述提到的Master.startExecutorsOnWorkers中，深入allocateWorkerResourceToExecutors：

```
  private def allocateWorkerResourceToExecutors(
      app: ApplicationInfo,
      assignedCores: Int,
      coresPerExecutor: Option[Int],
      worker: WorkerInfo): Unit = {
    // 该work上的executor数量
    // 若没设置 spark.executor.cores
    // 则为1
    val numExecutors = coresPerExecutor.map { assignedCores / _ }.getOrElse(1)
    // 分配给一个executor的core数量
    // 若没设置 spark.executor.cores
    // 则为该worker上所分配的所有core是数量
    val coresToAssign = coresPerExecutor.getOrElse(assignedCores)
    for (i <- 1 to numExecutors) {
      //创建该executor信息
      //并把它加入到app信息中
      //并返回executor信息
      val exec = app.addExecutor(worker, coresToAssign)
      //启动
      launchExecutor(worker, exec)
      app.state = ApplicationState.RUNNING
    }
  }
```

要注意的是

```
app.state = ApplicationState.RUNNING
```

这句代码***并不是将该app从waitingApp队列中去除***。若在该次资源调度中该app并没有启动足够的executor，等到集群资源变化时，会再次资源调度，在waitingApp中遍历到该app，其coresLeft > 0。

```
for (app <- waitingApps if app.coresLeft > 0)
```

我们这里做一个实验：

- 我们的实验集群是4*8核的集群：

State	Cores
ALIVE	8 (0 Used)
ALIVE	8 (0 Used)
ALIVE	8 (0 Used)
ALIVE	8 (0 Used)

- 第1个app，我们申请4个executor，该executor为4个core：

```
spark-shell --master spark://cdh03:7077 --total-executor-cores 4 --executor-cores 4
```

可以看到集群资源：

State	Cores
ALIVE	8 (0 Used)
ALIVE	8 (0 Used)
ALIVE	8 (4 Used)
ALIVE	8 (0 Used)

app1的executor：

Executor Summary

ExecutorID	Worker
0	worker-20170102151129-

- 第2个app，我们申请4个executor，该executor为6个core：

```
spark-shell --master spark://cdh03:7077 --total-executor-cores 24 --executor-cores 6
```

可以看到集群资源：

State	Cores
ALIVE	8 (6 Used)
ALIVE	8 (6 Used)
ALIVE	8 (4 Used)
ALIVE	8 (6 Used)

app2的executor：

Executor Summary

ExecutorID	Worker
2	worker-20170102151232-
1	worker-20170102151120-
0	worker-20170102151055-

我们可以看到，Spark只为app2分配了3个executor。

- 当我们把app1退出

会发现集群资源状态：

State	Cores
ALIVE	8 (6 Used)
ALIVE	8 (6 Used)
ALIVE	8 (6 Used)
ALIVE	8 (6 Used)

app2的executor：

Executor Summary

ExecutorID	Worker
2	worker-20170102151232
1	worker-20170102151120
3	worker-20170102151129
0	worker-20170102151055

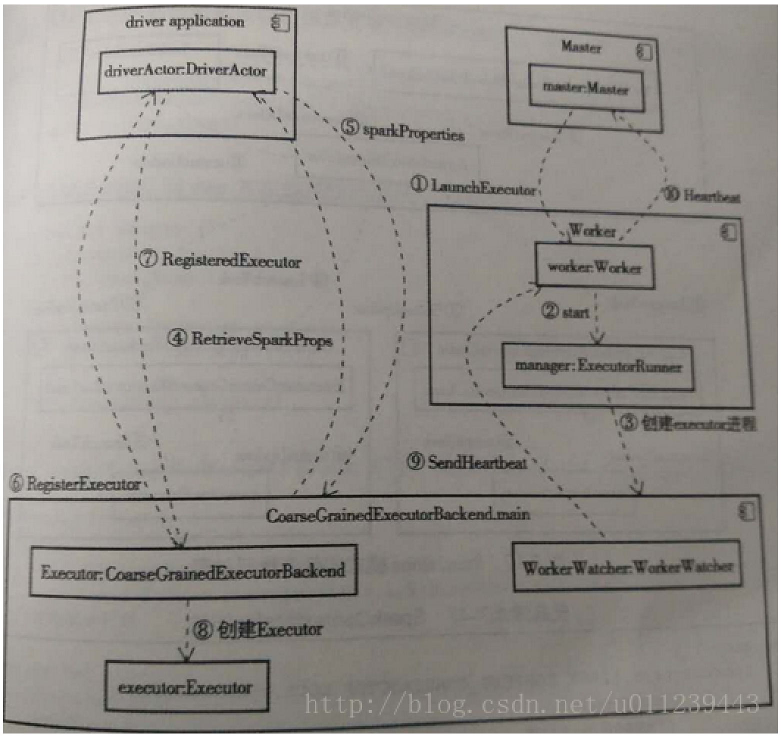
会发现新增加了一个“ worker-20170102151129”的executor。

其实，只要集群中的app没结束，它们都会在waitingApps中，当该app结束时，才会将这个app从waitingApps中移除

```
  def removeApplication(app: ApplicationInfo, state: ApplicationState.Value) {
***
      waitingApps -= app
***
}
```

物理资源调度与启动Executor

接下来，我们就来讲逻辑上资源调度完后，该如何物理上资源调度，即启动Executor。



调用栈如下：

- Master.launchExecutor
- Worker.receive
  - ExecutorRunner.start
    - ExecutorRunner.fetchAndRunExecutor
- CoarseGrainedExecutorBackend.main
  - CoarseGrainedExecutorBackend.run
    - CoarseGrainedExecutorBackend.onStart
- CoarseGrainedSchedulerBackend.DriverEndpoint.receiveAndReply
- CoarseGrainedExecutorBackend.receive

Master.launchExecutor

```
private def launchExecutor(worker: WorkerInfo, exec: ExecutorDesc): Unit = {
  logInfo("Launching executor " + exec.fullId + " on worker " + worker.id)
  //在worker信息中加入executor信息
  worker.addExecutor(exec)
  //给worker发送LaunchExecutor信号
  worker.endpoint.send(LaunchExecutor(masterUrl,
    exec.application.id, exec.id, exec.application.desc, exec.cores, exec.memory))
  //给driver发送ExecutorAdded信号
  exec.application.driver.send(
    ExecutorAdded(exec.id, worker.id, worker.hostPort, exec.cores, exec.memory))
}
```

Worker.receive

worker接收到LaunchExecutor信号后处理：

```
case LaunchExecutor(masterUrl, appId, execId, appDesc, cores_, memory_) =>
  if (masterUrl != activeMasterUrl) {
    logWarning("Invalid Master (" + masterUrl + ") attempted to launch executor.")
  } else {
    try {
      logInfo("Asked to launch executor %s/%d for %s".format(appId, execId, appDesc.name))

      // 创建executor的工作目录
      // shuffle持久化结果会存在这个目录下
      // 节点应每块磁盘大小尽可能相同
      // 并在配置中在每块磁盘上都设置SPARK_WORKER_DIR，
      // 以增加IO性能
      val executorDir = new File(workDir, appId + "/" + execId)
      if (!executorDir.mkdirs()) {
        throw new IOException("Failed to create directory " + executorDir)
      }

      // 为app创建本地dir
      // app完成后，此目录会被删除
      val appLocalDirs = appDirectories.getOrElse(appId,
        Utils.getOrCreateLocalRootDirs(conf).map { dir =>
          val appDir = Utils.createDirectory(dir, namePrefix = "executor")
          Utils.chmod700(appDir)
          appDir.getAbsolutePath()
        }.toSeq)
      appDirectories(appId) = appLocalDirs
      //创建ExecutorRunner
      val manager = new ExecutorRunner(
        appId,
        execId,
        appDesc.copy(command = Worker.maybeUpdateSSLSettings(appDesc.command, conf)),
        cores_,
        memory_,
        self,
        workerId,
        host,
        webUi.boundPort,
        publicAddress,
        sparkHome,
        executorDir,
        workerUri,
        conf,
        appLocalDirs, ExecutorState.RUNNING)
      executors(appId + "/" + execId) = manager
      //启动ExecutorRunner
      manager.start()
      coresUsed += cores_
      memoryUsed += memory_
      // 向Master发送ExecutorStateChanged信号
      sendToMaster(ExecutorStateChanged(appId, execId, manager.state, None, None))
    } catch {
      case e: Exception =>
        logError(s"Failed to launch executor $appId/$execId for ${appDesc.name}.", e)
        if (executors.contains(appId + "/" + execId)) {
          executors(appId + "/" + execId).kill()
          executors -= appId + "/" + execId
        }
        sendToMaster(ExecutorStateChanged(appId, execId, ExecutorState.FAILED,
          Some(e.toString), None))
    }
  }
```

ExecutorRunner.start

接下来我们深入看下ExecutorRunner

```
private[worker] def start() {
//创建worker线程
  workerThread = new Thread("ExecutorRunner for " + fullId) {
    override def run() { fetchAndRunExecutor() }
  }
  //启动worker线程
  workerThread.start()
  // 创建Shutdownhook线程
  // 用于worker关闭时，杀掉executor
  shutdownHook = ShutdownHookManager.addShutdownHook { () =>
    if (state == ExecutorState.RUNNING) {
      state = ExecutorState.FAILED
    }
    killProcess(Some("Worker shutting down")) }
}
```

ExecutorRunner.fetchAndRunExecutor

workerThread执行主要是调用fetchAndRunExecutor，下面我们来看下这个方法：

```
private def fetchAndRunExecutor() {
  try {
    // 创建进程builder
    val builder = CommandUtils.buildProcessBuilder(appDesc.command, new SecurityManager(conf),
      memory, sparkHome.getAbsolutePath, substituteVariables)
    val command = builder.command()
    val formattedCommand = command.asScala.mkString("\"", "\" \"", "\"")
    logInfo(s"Launch command: $formattedCommand")
    //创建进程builder执行目录
    builder.directory(executorDir)
    //为进程builder设置环境变量
    builder.environment.put("SPARK_EXECUTOR_DIRS", appLocalDirs.mkString(File.pathSeparator))
    builder.environment.put("SPARK_LAUNCH_WITH_SCALA", "0")

    val baseUrl =
      if (conf.getBoolean("spark.ui.reverseProxy", false)) {
        s"/proxy/$workerId/logPage/?appId=$appId&executorId=$execId&logType="
      } else {
        s"http://$publicAddress:$webUiPort/logPage/?appId=$appId&executorId=$execId&logType="
      }
    builder.environment.put("SPARK_LOG_URL_STDERR", s"${baseUrl}stderr")
    builder.environment.put("SPARK_LOG_URL_STDOUT", s"${baseUrl}stdout")

    //启动进程builder，创建进程
    process = builder.start()
    val header = "Spark Executor Command: %s\n%s\n\n".format(
      formattedCommand, "=" * 40)

    // 重定向它的stdout和stderr到文件中
    val stdout = new File(executorDir, "stdout")
    stdoutAppender = FileAppender(process.getInputStream, stdout, conf)

    val stderr = new File(executorDir, "stderr")
    Files.write(header, stderr, StandardCharsets.UTF_8)
    stderrAppender = FileAppender(process.getErrorStream, stderr, conf)

    // 等待进程退出。
    // 当driver通知该进程退出
    // executor会退出并返回0或者非0的exitCode
```

```
      val exitCode = process.waitFor()
      state = ExecutorState.EXITED
      val message = "Command exited with code " + exitCode
      // 给Worker发送ExecutorStateChanged信号
      worker.send(ExecutorStateChanged(appId, execId, state, Some(message), Some(exitCode)))
    } catch {
      case interrupted: InterruptedException =>
        logInfo("Runner thread for executor " + fullId + " interrupted")
        state = ExecutorState.KILLED
        killProcess(None)
      case e: Exception =>
        logError("Error running executor", e)
        state = ExecutorState.FAILED
        killProcess(Some(e.toString))
    }
  }
}
```

CoarseGrainedExecutorBackend.main

builder start的是CoarseGrainedExecutorBackend实例进程，我们看下它的主函数：

```
def main(args: Array[String]) {
  var driverUrl: String = null
  var executorId: String = null
  var hostname: String = null
  var cores: Int = 0
  var appId: String = null
  var workerUrl: Option[String] = None
  val userClassPath = new mutable.ListBuffer[URL]()
  // 设置参数
  var argv = args.toList
  while (!argv.isEmpty) {
    argv match {
      case ("--driver-url") :: value :: tail =>
        driverUrl = value
        argv = tail
      case ("--executor-id") :: value :: tail =>
        executorId = value
        argv = tail
      case ("--hostname") :: value :: tail =>
        hostname = value
        argv = tail
      case ("--cores") :: value :: tail =>
        cores = value.toInt
        argv = tail
      case ("--app-id") :: value :: tail =>
        appId = value
        argv = tail
      case ("--worker-url") :: value :: tail =>
        workerUrl = Some(value)
        argv = tail
      case ("--user-class-path") :: value :: tail =>
        userClassPath += new URL(value)
        argv = tail
      case Nil =>
      case tail =>
        System.err.println(s"Unrecognized options: ${tail.mkString(" ")}")
        printUsageAndExit()
    }
  }

  if (driverUrl == null || executorId == null || hostname == null || cores <= 0 ||
    appId == null) {
```

```
      printUsageAndExit()
    }
    //调用run方法
    run(driverUrl, executorId, hostname, cores, appId, workerUrl, userClassPath)
    System.exit(0)
  }
```

CoarseGrainedExecutorBackend.run

```
  private def run(
      driverUrl: String,
      executorId: String,
      hostname: String,
      cores: Int,
      appId: String,
      workerUrl: Option[String],
      userClassPath: Seq[URL]) {

    Utils.initDaemon(log)

    SparkHadoopUtil.get.runAsSparkUser { () =>
      Utils.checkHost(hostname)

      val executorConf = new SparkConf
      val port = executorConf.getInt("spark.executor.port", 0)
      val fetcher = RpcEnv.create(
        "driverPropsFetcher",
        hostname,
        port,
        executorConf,
        new SecurityManager(executorConf),
        clientMode = true)
      val driver = fetcher.setupEndpointRefByURI(driverUrl)
      // 给driver发送RetrieveSparkAppConfig信号,
      // 并根据返回的信息创建属性
      val cfg = driver.askWithRetry[SparkAppConfig](RetrieveSparkAppConfig)
      val props = cfg.sparkProperties ++ Seq[(String, String)](("spark.app.id", appId))
      fetcher.shutdown()

      // 根据这些属性来创建SparkEnv
      val driverConf = new SparkConf()
      for ((key, value) <- props) {
        if (SparkConf.isExecutorStartupConf(key)) {
          driverConf.setIfMissing(key, value)
        } else {
          driverConf.set(key, value)
        }
      }
      if (driverConf.contains("spark.yarn.credentials.file")) {
        logInfo("Will periodically update credentials from: " +
          driverConf.get("spark.yarn.credentials.file"))
        SparkHadoopUtil.get.startCredentialUpdater(driverConf)
      }

      val env = SparkEnv.createExecutorEnv(
        driverConf, executorId, hostname, port, cores, cfg.ioEncryptionKey, isLocal = false)

      // 创建CoarseGrainedExecutorBackend Endpoint
      env.rpcEnv.setupEndpoint("Executor", new CoarseGrainedExecutorBackend(
        env.rpcEnv, driverUrl, executorId, hostname, cores, userClassPath, env))
        // 创建WorkerWatcher Endpoint
        // 用来给worker发送心跳,
        // 告诉worker 这个进程还活着
      workerUrl.foreach { url =>
```

```
        env.rpcEnv.setupEndpoint("WorkerWatcher", new WorkerWatcher(env.rpcEnv, url))
      }
      env.rpcEnv.awaitTermination()
      SparkHadoopUtil.get.stopCredentialUpdater()
    }
  }
```

CoarseGrainedExecutorBackend.onStart

new CoarseGrainedExecutorBackend 会调用CoarseGrainedExecutorBackend.onStart：

```
  override def onStart() {
    logInfo("Connecting to driver: " + driverUrl)
    rpcEnv.asyncSetupEndpointRefByURI(driverUrl).flatMap { ref =>
      //向driver端发送RegisterExecutor信号
      driver = Some(ref)
      ref.ask[Boolean](RegisterExecutor(executorId, self, hostname, cores, extractLogUrls))
    }(ThreadUtils.sameThread).onComplete {
      case Success(msg) =>
      case Failure(e) =>
        exitExecutor(1, s"Cannot register with driver: $driverUrl", e, notifyDriver = false)
    }(ThreadUtils.sameThread)
  }
```

CoarseGrainedSchedulerBackend.DriverEndpoint.receiveAndReply

```
      case RegisterExecutor(executorId, executorRef, hostname, cores, logUrls) =>
        if (executorDataMap.contains(executorId)) {
          executorRef.send(RegisterExecutorFailed("Duplicate executor ID: " + executorId))
          context.reply(true)
        } else {
          // 设置executor信息
          val executorAddress = if (executorRef.address != null) {
              executorRef.address
            } else {
              context.senderAddress
            }
          logInfo(s"Registered executor $executorRef ($executorAddress) with ID $executorId")
          addressToExecutorId(executorAddress) = executorId
          totalCoreCount.addAndGet(cores)
          totalRegisteredExecutors.addAndGet(1)
          val data = new ExecutorData(executorRef, executorRef.address, hostname,
            cores, cores, logUrls)
          CoarseGrainedSchedulerBackend.this.synchronized {
            executorDataMap.put(executorId, data)
            if (currentExecutorIdCounter < executorId.toInt) {
              currentExecutorIdCounter = executorId.toInt
            }
            if (numPendingExecutors > 0) {
              numPendingExecutors -= 1
              logDebug(s"Decremented number of pending executors ($numPendingExecutors left)")
            }
          }
          //向executor端发送RegisteredExecutor信号
          executorRef.send(RegisteredExecutor)
          context.reply(true)
          listenerBus.post(
            SparkListenerExecutorAdded(System.currentTimeMillis(), executorId, data))
          makeOffers()
        }
```

makeOffers()所做的逻辑，在《深入理解Spark 2.1 Core （三）：任务调度器的原理与源码分析 》里已经讲解过。主要是调度任务，并想executor发送任务。

CoarseGrainedExecutorBackend.receive

CoarseGrainedExecutorBackend接收到来自driver的RegisteredExecutor信号后：

```
    case RegisteredExecutor =>
      logInfo("Successfully registered with driver")
      try {
      //创建executor
        executor = new Executor(executorId, hostname, env, userClassPath, isLocal = false)
      } catch {
        case NonFatal(e) =>
          exitExecutor(1, "Unable to create executor due to " + e.getMessage, e)
      }
```

至此，Executor就成功的启动了！