Life is short, you need Spark!

从零开始

不需要任何基础，带领您无痛入门 Spark

云计算分布式大数据 Spark 实战高手之路

王家林著

Spark 亚太研究院系列丛书

伴随着大数据相关技术和产业的逐步成熟，继 Hadoop 之后，Spark 技术以其无可比拟的优势，发展迅速，将成为替代 Hadoop 的下一代云计算、大数据核心技术。

本书特点

- 云计算分布式大数据 Spark 实战高手之路三部曲之第一部
- 网络发布版为图文并茂方式，边学习，边演练
- 不需要任何前置知识，从零开始，循序渐进

本书作者

Spark 亚太研究院院长和首席专家，中国目前唯一的移动互联网和云计算大数据集大成者。在 Spark、Hadoop、Android 等方面有丰富的源码、实务和性能优化经验。彻底研究了 Spark 从 0.5.0 到 0.9.1 共 13 个版本的 Spark 源码，并已完成 2014 年 5 月 31 日发布的 Spark1.0 源码研究。

Hadoop 源码级专家，曾负责某知名公司的类 Hadoop 框架开发工作，专注于 Hadoop 一站式解决方案的提供，同时也是云计算分布式大数据处理的最早实践者之一。

Android 架构师、高级工程师、咨询顾问、培训专家。

通晓 Spark、Hadoop、Android、HTML5，迷恋英语播音和健美。

“真相会使你获得自由。”

—— 耶稣《圣经》约翰 8:32KJV

“所有人类的不幸都来源于不能直面事实。”

—— 释迦摩尼

“道法自然”

—— 老子《道德经》第 25 章

《云计算分布式大数据 Spark 实战高手之路》

系列丛书三部曲

《云计算分布式大数据 **Spark** 实战高手之路---从零开始》：

不需要任何基础，带领您无痛入门 Spark 并能够轻松处理 Spark 工程师的日常编程工作，内容包括 Spark 集群的构建、Spark 架构设计、RDD、Shark/SparkSQL、机器学习、图计算、实时流处理、Spark on Yarn、JobServer、Spark 测试、Spark 优化等。

《云计算分布式大数据 **Spark** 实战高手之路---高手崛起》：

大话 Spark 源码，全世界最有情趣的源码解析，过程中伴随诸多实验，解析 Spark 1.0 的任何一句源码！更重要的是，思考源码背后的问题场景和解决问题的设计哲学和实现招式。

《云计算分布式大数据 **Spark** 实战高手之路---高手之巅》：

通过当今主流的 Spark 商业使用方法和最成功的 Hadoop 大型案例让您直达高手之巅，从此一览众山小。

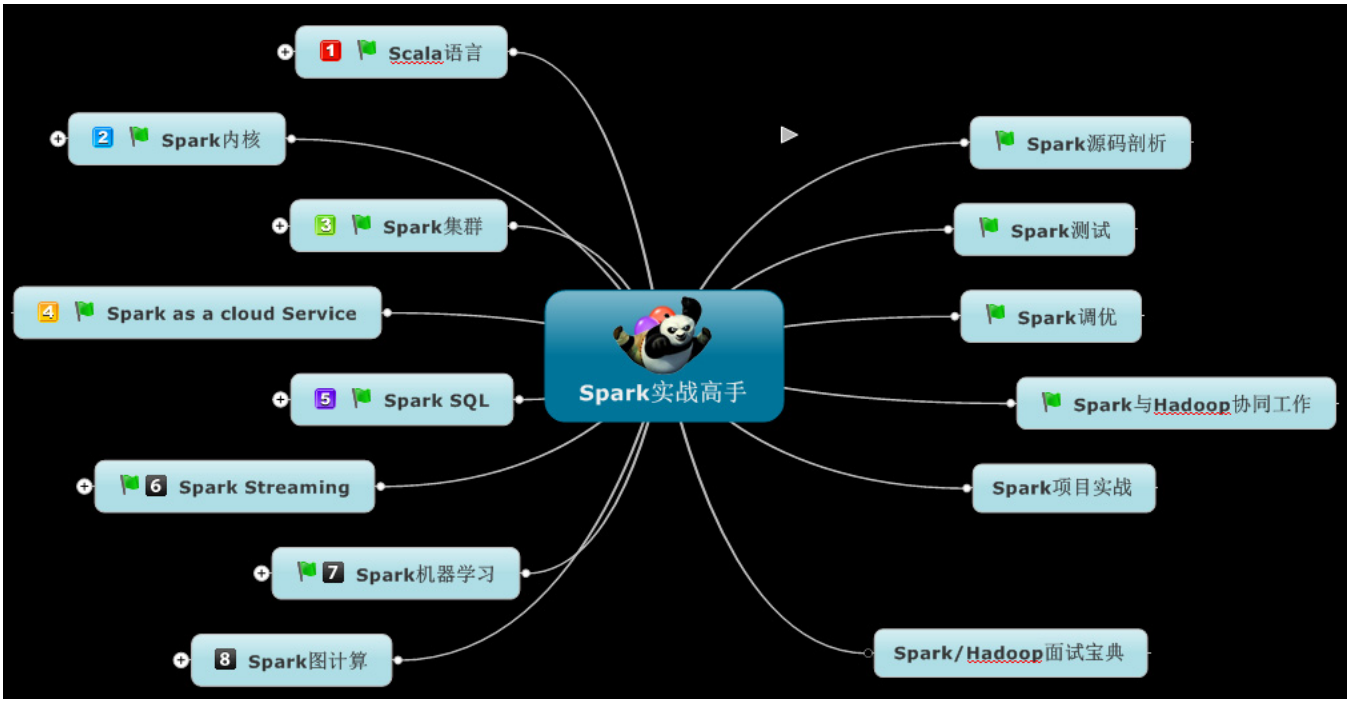
www.sparkinchina.com
TEL: 4006-998-758

《前言》

Spark采用一个统一的技术堆栈解决了云计算大数据的如流处理、图技术、机器学习、NoSQL查询等方面的所有核心问题，具有完善的生态系统，这直接奠定了其一统云计算大数据领域的霸主地位；

要想成为Spark高手，需要经历六大阶段

Spark 实战高手之核心技能点



第一阶段：熟练的掌握Scala语言

1. Spark 框架是采用 Scala 语言编写的，精致而优雅。要想成为 Spark 高手，你就必须阅读 Spark 的源代码，就必须掌握 Scala,；
2. 虽然说现在的 Spark 可以采用多语言 Java、Python 等进行应用程序开发，但是最快速的和支持最好的开发 API 依然并将永远是 Scala 方式的 API，所以你必须掌握 Scala 来编写复杂的和高性能的 Spark 分布式程序；
3. 尤其要熟练掌握 Scala 的 trait、apply、函数式编程、泛型、逆变与协变等；

推荐课程：" 精通Spark的开发语言：Scala最佳实践"

第二阶段：精通Spark平台本身提供给开发者API

1. 掌握Spark中面向RDD的开发模式 掌握各种transformation和action函数的使用；
2. 掌握 Spark 中的宽依赖和窄依赖以及 lineage 机制；
3. 掌握 RDD 的计算流程，例如 Stage 的划分、Spark 应用程序提交给集群的基本过程和 Worker 节点基础的工作原理等

推荐课程： "18 小时内掌握Spark：把云计算大数据速度提高 100 倍以上!"

第三阶段：深入Spark内核

此阶段主要是通过 Spark 框架的源码研读来深入 Spark 内核部分：

1. 通过源码掌握 Spark 的任务提交过程；
2. 通过源码掌握 Spark 集群的任务调度；
3. 尤其要精通 DAGScheduler、TaskScheduler 和 Worker 节点内部的工作的每一步的细节；

推荐课程： “Spark 1.0.0 企业级开发动手：实战世界上第一个Spark 1.0.0 课程，涵盖Spark 1.0.0 所有的企业级开发技术”

第四阶级:掌握基于Spark上的核心框架的使用

Spark 作为云计算大数据时代的集大成者，在实时流处理、图技术、机器学习、NoSQL 查询等方面具有显著的优势，我们使用 Spark 的时候大部分时间都是在使用其上的框架例如 Shark、Spark Streaming 等：

1. Spark Streaming 是非常出色的实时流处理框架 ,要掌握其 DStream、transformation 和 checkpoint 等；
2. Spark 的离线统计分析功能，Spark 1.0.0 版本在 Shark 的基础上推出了 Spark SQL，离线统计分析的功能的效率有显著的提升，需要重点掌握；
3. 对于 Spark 的机器学习和 GraphX 等要掌握其原理和用法；

推荐课程： “Spark企业级开发最佳实践”

第五阶级:做商业级别的Spark项目

通过一个完整的具有代表性的 Spark 项目来贯穿 Spark 的方方面面 ,包括项目的架构设计、用到的技术的剖析、开发实现、运维等，完整掌握其中的每一个阶段和细节，这样就可以让您以后可以从容面对绝大多数 Spark 项目。

推荐课程： “Spark架构案例鉴赏：Conviva、Yahoo！、优酷土豆、网易、腾讯、淘宝等公司的实际Spark案例”

第六阶级：提供Spark解决方案

1. 彻底掌握 Spark 框架源码的每一个细节；
2. 根据不同的业务场景的需要提供 Spark 在不同场景的下的解决方案；
3. 根据实际需要，在 Spark 框架基础上进行二次开发，打造自己的 Spark 框架；

推荐课程： “精通Spark：Spark内核剖析、源码解读、性能优化和商业案例实战”

《第四章：Spark 内核揭秘》

掌握 Spark 内核是精通 Spark 的关键，也是驾驭 Spark 的精髓所在。

基于 Spark 内核，Spark 构建起了一体化多元化的大数据处理流水线，在一个技术堆栈中即可以同时完成批处理、实时流处理、交互式查询、机器学习、图计算以及这些子框架之间数据和 RDD 算子的无缝共享与互操作。

可以说，Spark 内核是每个想彻底掌握 Spark 的人员的必修课，通过对内核的探索，我们对整个 Spark 的运行机制会了如指掌，这对 Spark 的大规模应用、性能优化、系统自定义开发 Spark 系统都是至关重要的。

本章首先带你初探 Spark 内核，接着通过源码游离 Spark 内核中，然后通过源码细致解析 Spark 作业的全生命周期，最后分享 Spark 性能优化，内容组织循序渐进，希望助力诸位 Spark 爱好者能够掌握 Spark 内核。

Spark 内核揭秘共分六个部分：

- 第一部分：Spark 内核初探
- 第二部分：Spark 内核核心源码解析
- 第三部分：Job 全生命周期源码解读
- 第四部分： 解读 RDD 源码
- 第五部分： Driver、 Master、 Woker
- 第六部分：Spark 性能优化

本讲是 **Spark** 内核揭秘的第五部分：**Driver**、 **Master**、 **Slave**，具体内容如下所示：

- 1，Driver 中 AppClient 全程源码解析
- 2，AppClient 注册 Master 源码解析：
- 3，Worker 中 Executor 启动源代码解析；

不需任何前置知识，从零开始，循序渐进，成为 Spark 高手！

目录

一.Driver 中 AppClient 源码解析

首先从 SparkContext 中 TaskScheduler 实例的创建开始：

```
// Create and start the scheduler
private[spark] var taskScheduler = SparkContext.createTaskScheduler(this, master)
private val heartbeatReceiver = env.actorSystem.actorOf(
  Props(new HeartbeatReceiver(taskScheduler)), "HeartbeatReceiver")
@volatile private[spark] var dagScheduler: DAGScheduler = _
try {
  dagScheduler = new DAGScheduler(this)
} catch {
  case e: Exception => throw
    new SparkException("DAGScheduler cannot be initialized due to %s".format(e.getMessage))
}

// start TaskScheduler after taskScheduler sets DAGScheduler reference in DAGScheduler's
// constructor
taskScheduler.start()
```

进入 start 代码内部：

```
/**
 * Low-level task scheduler interface, currently implemented exclusively by TaskSchedulerImpl.
 * This interface allows plugging in different task schedulers. Each TaskScheduler schedulers tasks
 * for a single SparkContext. These schedulers get sets of tasks submitted to them from the
 * DAGScheduler for each stage, and are responsible for sending the tasks to the cluster, running
 * them, retrying if there are failures, and mitigating stragglers. They return events to the
 * DAGScheduler.
 */
private[spark] trait TaskScheduler {

  def rootPool: Pool

  def schedulingMode: SchedulingMode

  def start(): Unit
```

进入其实现者 TaskSchedulerImpl 内部：

```
override def start() {
  backend.start()

  if (!isLocal && conf.getBoolean("spark.speculation", false)) {
    logInfo("Starting speculative execution thread")
    import sc.env.actorSystem.dispatcher
    sc.env.actorSystem.scheduler.schedule(SPECULATION_INTERVAL milliseconds,
          SPECULATION_INTERVAL milliseconds) {
      Utils.tryOrExit { checkSpeculatableTasks() }
    }
  }
}
```

可以发现在 start 具体实现的内部首先是有个 backend.start 方法：

```
/**
 * A backend interface for scheduling systems that allows plugging in different ones under
 * TaskSchedulerImpl. We assume a Mesos-like model where the application gets resource offers as
 * machines become available and can launch tasks on them.
 */
private[spark] trait SchedulerBackend {
  def start(): Unit
  def stop(): Unit
  def reviveOffers(): Unit
  def defaultParallelism(): Int

  def killTask(taskId: Long, executorId: String, interruptThread: Boolean)
    throw new UnsupportedOperationException
  def isReady(): Boolean = true
}
```

其最终具体的实现类为：

```
private[spark] class SparkDeploySchedulerBackend(
    scheduler: TaskSchedulerImpl,
    sc: SparkContext,
    masters: Array[String])
  extends CoarseGrainedSchedulerBackend(scheduler, sc.env.actorSystem)
  with AppClientListener
  with Logging {

  var client: AppClient = null
  var stopping = false
  var shutdownCallback : (SparkDeploySchedulerBackend) => Unit = _

  val maxCores = conf.getOption("spark.cores.max").map(_.toInt)
  val totalExpectedCores = maxCores.getOrElse(0)

  override def start() {
    super.start()

    // The endpoint for executors to talk to us
    val driverUrl = "akka.tcp://spark@%s:%s/user/%s".format(
      conf.get("spark.driver.host"), conf.get("spark.driver.port"),
      CoarseGrainedSchedulerBackend.ACTOR_NAME)
```

```
  val args = Seq(driverUrl, "{{EXECUTOR_ID}}", "{{HOSTNAME}}", "{{CORES}}", "{{WORKER_URL}}")
  val extraJavaOpts = sc.conf.getOption("spark.executor.extraJavaOptions")
    .map(Utils.splitCommandString).getOrElse(Seq.empty)
  val classPathEntries = sc.conf.getOption("spark.executor.extraClassPath").toSeq.flatMap { cp =>
    cp.split(java.io.File.pathSeparator)
  }
  val libraryPathEntries =
    sc.conf.getOption("spark.executor.extraLibraryPath").toSeq.flatMap { cp =>
      cp.split(java.io.File.pathSeparator)
    }

  // Start executors with a few necessary configs for registering with the scheduler
  val sparkJavaOpts = Utils.sparkJavaOpts(conf, SparkConf.isExecutorStartupConf)
  val javaOpts = sparkJavaOpts ++ extraJavaOpts
  val command = Command("org.apache.spark.executor.CoarseGrainedExecutorBackend",
    args, sc.executorEnvs, classPathEntries, libraryPathEntries, javaOpts)
  val appDesc = new ApplicationDescription(sc.appName, maxCores, sc.executorMemory, command,
    sc.ui.appUIAddress, sc.eventLogger.map(_.logDir))

  client = new AppClient(sc.env.actorSystem, masters, appDesc, this, conf)
  client.start()
}
```

从代码中可以看出，我们把 CoarseGrainedExecutorBackend 封装成 command，然后交给 appDesc，接着交给了 Appclient，此时的 AppClient 就是客户端程序！

Appclient 会调用 start 方法：

```
def start() {
  // Just launch an actor; it will call back into the listener.
  actor = actorSystem.actorOf(Props(new ClientActor))
}
```

此时启动了 ClientActor：

```
class ClientActor extends Actor with ActorLogReceive with Logging {
  var master: ActorSelection = null
  var alreadyDisconnected = false  // To avoid calling listener.disconnected() multiple times
  var alreadyDead = false  // To avoid calling listener.dead() multiple times
  var registrationRetryTimer: Option[Cancellable] = None

  override def preStart() {
    context.system.eventStream.subscribe(self, classOf[RemotingLifecycleEvent])
    try {
      registerWithMaster()
    } catch {
      case e: Exception =>
        logWarning("Failed to connect to master", e)
        markDisconnected()
        context.stop(self)
    }
  }

  def tryRegisterAllMasters() {
    for (masterUrl <- masterUrls) {
      logInfo("Connecting to master " + masterUrl + "...")
      val actor = context.actorSelection(Master.toAkkaUrl(masterUrl))
      actor ! RegisterApplication(appDescription)
```

二.AppClient 注册 Master

注册 Master 有两种，一种是 registerWithMaster 方法，一种是 tryRegisterAllMasters 方法，前者是单 Master 的情况，后者是多 Master，一般情况下是满足 HA 机制，我们看一下 registerWithMaster 方法：

```
def registerWithMaster() {
  tryRegisterAllMasters()
  import context.dispatcher
  var retries = 0
  registrationRetryTimer = Some {
    context.system.scheduler.schedule(REGISTRATION_TIMEOUT, REGISTRATION_TIMEOUT) {
      Utils.tryOrExit {
        retries += 1
        if (registered) {
          registrationRetryTimer.foreach(_.cancel())
        } else if (retries >= REGISTRATION_RETRIES) {
          markDead("All masters are unresponsive! Giving up.")
        } else {
          tryRegisterAllMasters()
        }
      }
    }
  }
}
```

此时会发生 tryRegisterAllMasters 方法：

```
def tryRegisterAllMasters() {
  for (masterUrl <- masterUrls) {
    logInfo("Connecting to master " + masterUrl + "...")
    val actor = context.actorSelection(Master.toAkkaUrl(masterUrl))
    actor ! RegisterApplication(appDescription)
  }
}
```

此时通过 Akka 通过消息机制发送消息给 Master 来注册程序，RegisterApplication 是一个 case class，来封装消息：

```
// AppClient to Master

case class RegisterApplication(appDescription: ApplicationDescription)
  extends DeployMessage
```

我们进入 Master 的源代码：

```
private[spark] class Master(
    host: String,
    port: Int,
    webUiPort: Int,
    val securityMgr: SecurityManager)
  extends Actor with ActorLogReceive with Logging {

  import context.dispatcher   // to use Akka's scheduler.schedule()

  val conf = new SparkConf
```

看一下接受客户端发送过来消息 RegisterApplication 的代码如下所示：

SparkContext.scala | Master.scala | TaskSchedulerImpl.scala

```
    case RegisterApplication(description) => {
      if (state == RecoveryState.STANDBY) {
        // ignore, don't send response
      } else {
        logInfo("Registering app " + description.name)
        val app = createApplication(description, sender)
        registerApplication(app)
        logInfo("Registered app " + description.name + " with ID " + app.id)
        persistenceEngine.addApplication(app)
        sender ! RegisteredApplication(app.id, masterUrl)
        schedule()
      }
    }
```

此时首先使用 ApplicationInfo 构建一些准备信息，然后会导致 registerApplication 代码的调用：

```
def createApplication(desc: ApplicationDescription, driver: ActorRef): ApplicationInfo = {
  val now = System.currentTimeMillis()
  val date = new Date(now)
  new ApplicationInfo(now, newApplicationId(date), desc, date, driver, defaultCores)
}

def registerApplication(app: ApplicationInfo): Unit = {
  val appAddress = app.driver.path.address
  if (addressToWorker.contains(appAddress)) {
    logInfo("Attempted to re-register application at same address: " + appAddress)
    return
  }

  applicationMetricsSystem.registerSource(app.appSource)
  apps += app
  idToApp(app.id) = app
  actorToApp(app.driver) = app
  addressToApp(appAddress) = app
  waitingApps += app
}
```

代码中就是一个注册应用的过程。

接着在 Master 的消息响应中会调用 schedule 方法：

```
/**
 * Schedule the currently available resources among waiting apps. This method will be called
 * every time a new app joins or resource availability changes.
 */
private def schedule() {
  if (state != RecoveryState.ALIVE) { return }

  // First schedule drivers, they take strict precedence over applications
  val shuffledWorkers = Random.shuffle(workers) // Randomization helps balance drivers
  for (worker <- shuffledWorkers if worker.state == WorkerState.ALIVE) {
    for (driver <- List(waitingDrivers: _*)) { // iterate over a copy of waitingDrivers
      if (worker.memoryFree >= driver.desc.mem && worker.coresFree >= driver.desc.cores) {
        launchDriver(worker, driver)
        waitingDrivers -= driver
      }
    }
  }
```

可以看到 schedule 方法中首先要启动 Driver 程序，也就是有 main 函数的程序，然后在 schedule 中会调度 Worker 的过程：

```
// Right now this is a very simple FIFO scheduler. We keep trying to fit in the first app
// in the queue, then the second app, etc.
if (spreadOutApps) {
  // Try to spread out each app among all the nodes, until it has all its cores
  for (app <- waitingApps if app.coresLeft > 0) {
    val usableWorkers = workers.toArray.filter(_.state == WorkerState.ALIVE)
      .filter(canUse(app, _)).sortBy(_.coresFree).reverse
    val numUsable = usableWorkers.length
    val assigned = new Array[Int](numUsable) // Number of cores to give on each node
    var toAssign = math.min(app.coresLeft, usableWorkers.map(_.coresFree).sum)
    var pos = 0
    while (toAssign > 0) {
      if (usableWorkers(pos).coresFree - assigned(pos) > 0) {
        toAssign -= 1
        assigned(pos) += 1
      }
      pos = (pos + 1) % numUsable
    }
    // Now that we've decided how many cores to give on each node, let's actually give them
    for (pos <- 0 until numUsable) {
      if (assigned(pos) > 0) {
        val exec = app.addExecutor(usableWorkers(pos), assigned(pos))
        launchExecutor(usableWorkers(pos), exec)
        app.state = ApplicationState.RUNNING
      }
    }
  }
} else {
  // Pack each app into as few nodes as possible until we've assigned all its cores
  for (worker <- workers if worker.coresFree > 0 && worker.state == WorkerState.ALIVE) {
    for (app <- waitingApps if app.coresLeft > 0) {
      if (canUse(app, worker)) {
        val coresToUse = math.min(worker.coresFree, app.coresLeft)
        if (coresToUse > 0) {
          val exec = app.addExecutor(worker, coresToUse)
          launchExecutor(worker, exec)
          app.state = ApplicationState.RUNNING
        }
      }
    }
  }
}
```

改代码会导致 launchExecutor 代码的执行：

```
def launchExecutor(worker: WorkerInfo, exec: ExecutorInfo) {
  logInfo("Launching executor " + exec.fullId + " on worker " + worker.id)
  worker.addExecutor(exec)
  worker.actor ! LaunchExecutor(masterUrl,
    exec.application.id, exec.id, exec.application.desc, exec.cores, exec.memory)
  exec.application.driver ! ExecutorAdded(
    exec.id, worker.id, worker.hostPort, exec.cores, exec.memory)
}
```

在 launchExecutor 内部 Master 发送消息给 Worker 节点，消息为 LaunchExecutor：

```
case class LaunchExecutor(
    masterUrl: String,
    appId: String,
    execId: Int,
    appDesc: ApplicationDescription,
    cores: Int,
    memory: Int)
  extends DeployMessage
```

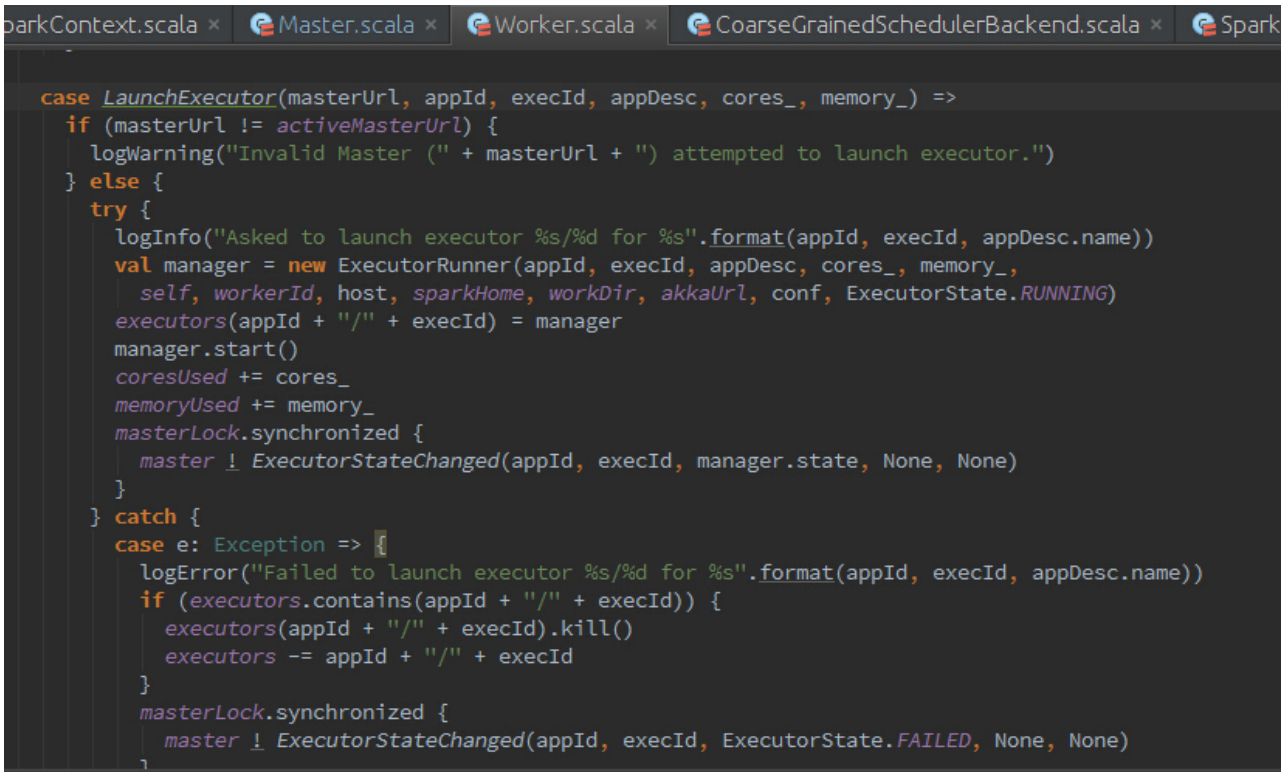
三.Worker 中 Executor 启动过程源代码解析

进入 Worker 源代码：

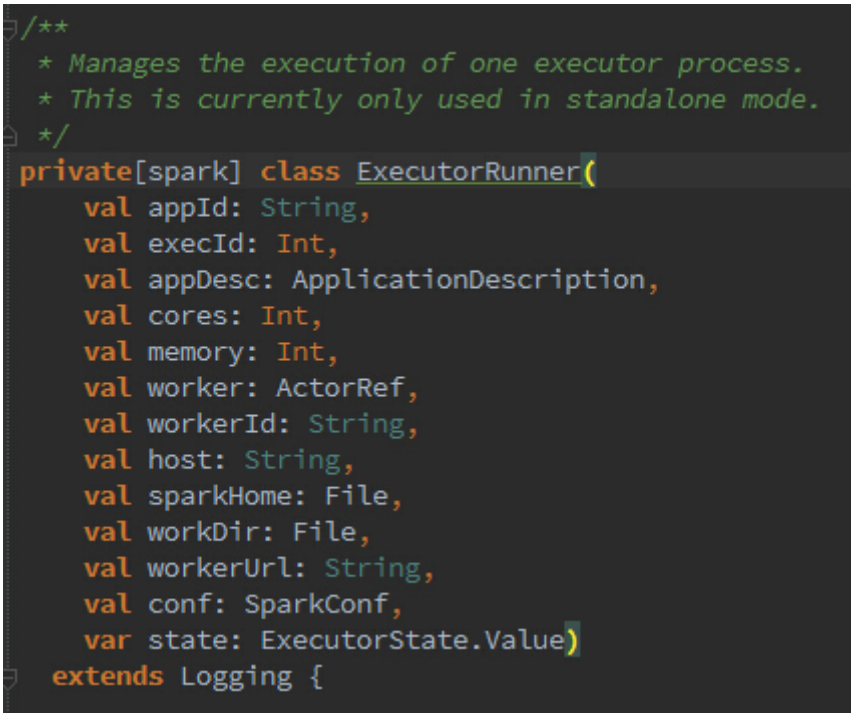
```
/**
  * @param masterUrls Each url should look like spark://host:port.
  */
private[spark] class Worker(
    host: String,
    port: Int,
    webUiPort: Int,
    cores: Int,
    memory: Int,
    masterUrls: Array[String],
    actorSystemName: String,
    actorName: String,
    workDirPath: String = null,
    val conf: SparkConf,
    val securityMgr: SecurityManager)
  extends Actor with ActorLogReceive with Logging {
  import context.dispatcher
```

可以看出 Worker 本身是 Akka 中的一个 Actor。

我们看一下 Worker 对 LaunchExecutor 消息的处理：



从源代码可以看出 Worker 节点上要分配 CPU 和 Memory 给新的 Executor，首先需要创建一个 ExecutorRunner：



ExecutorRunner 是用于维护 executor 进程的：

```
// NOTE: This is now redundant with the automated shut-down enforced by the Executor. It might
// make sense to remove this in the future.
var shutdownHook: Thread = null

def start() {
  workerThread = new Thread("ExecutorRunner for " + fullId) {
    override def run() { fetchAndRunExecutor() }
  }
  workerThread.start()
  // Shutdown hook that kills actors on shutdown.
  shutdownHook = new Thread() {
    override def run() {
      killProcess(Some("Worker shutting down"))
    }
  }
  Runtime.getRuntime.addShutdownHook(shutdownHook)
}

/**
 * Kill executor process, wait for exit and notify worker to update resource status.
 *
```

其中最重要的方法是 fetchAndRunExecutor：

```
/**
 * Download and run the executor described in our ApplicationDescription
 */
def fetchAndRunExecutor() {
  try {
    // Create the executor's working directory
    val executorDir = new File(workDir, appId + "/" + execId)
    if (!executorDir.mkdirs()) {
      throw new IOException("Failed to create directory " + executorDir)
    }

    // Launch the process
    val command = getCommandSeq
    logInfo("Launch command: " + command.mkString("\"", "\" \"", "\""))
    val builder = new ProcessBuilder(command: _*).directory(executorDir)
    val env = builder.environment()
    for ((key, value) <- appDesc.command.environment) {
      env.put(key, value)
    }
    // In case we are running this from within the Spark Shell, avoid creating
    // parent process for the executor command
    env.put("SPARK_LAUNCH_WITH_SCALA", "0")
    process = builder.start()
    val header = "Spark Executor Command: %s\n%s\n\n".format(
      command.mkString("\"", "\" \"", "\""), "=" * 40)
```

```
    // Redirect its stdout and stderr to files
    val stdout = new File(executorDir, "stdout")
    stdoutAppender = FileAppender(process.getInputStream, stdout, conf)

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

    // Wait for it to exit; executor may exit with code 0 (when driver instructs it to shutdown)
    // or with nonzero exit code
    val exitCode = process.waitFor()
    state = ExecutorState.EXITED
    val message = "Command exited with code " + exitCode
    worker ! 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))
    }
```

至此，Worker 节点上的 Executor 启动运行。

Spark 亚太研究院

Spark 亚太研究院，提供 Spark、Hadoop、Android、Html5、云计算和移动互联网一站式解决方案。以帮助企业规划、部署、开发、培训和使用为核心，并规划和实施人才培训完整路径，提供源码研究和应用技术训练。

近期活动及相关课程

1、决战云计算大数据时代 Spark 亚太研究院 100 期公益大奖堂

每周四晚上 20：00—21:00

课程介绍：http://edu.51cto.com/course/course_id-1659.html#showDesc

报名参与：http://ke.qq.com/cgi-bin/courseDetail?course_id=6167

2、大数据 Spark 实战高手之路—熟练掌握 Scala 语言视频课程

国内第一个 Scala 视频学习课程！

成为 Spark 高手必备技能，必修课程！

现在购买，即可享受套餐优惠！

链接地址：http://edu.51cto.com/pack/view/id-124.html

近期公开课：

《决胜大数据时代：Hadoop、Yarn、Spark 企业级最佳实践》

集大数据领域最核心三大技术：Hadoop 方向 50%：掌握生产环境下、源码级别下的 Hadoop 经验，解决性能、集群难点问题；Yarn 方向 20%：掌握最佳的分布式集群资源管理框架，能够轻松使用 Yarn 管理 Hadoop、Spark 等；Spark 方向 30%：未来统一的大数据框架平台，剖析 Spark 架构、内核等核心技术，对未来转向 SPARK 技术，做好技术储备。课程内容落地性强，即解决当下问题，又有助于驾驭未来。

开课时间：9 月 26—28 日 上海、10 月 26—28 日北京、11 月 1—3 日深圳

咨询电话：4006-998-758

QQ 交流群： 1 群：317540673 （已满）
2 群 297931500

微信公众号：spark-china