先看一下Worker中Driver和Executor注册过程：

Worker本身核心的作用是：管理当前机器的内存和CPU等资源，接受Master的指令来启动Driver，或者启动Executor。

如何启动Driver

如何启动Executor

如果Driver或者Executor有挂掉了，则Master就可以通过schedule再次调度资源。

Worker本身在实际运行的时候作为一个进程。实现RPC通信的。

**extends** ThreadSafeRpcEndpoint **with** Logging {

1. Master通过RPC协议将消息发给Worker，Worker通过receive接收到了Master发过来的消息。

**case** *LaunchDriver*(driverId, driverDesc) => {  
 logInfo(s"Asked to launch driver **$**driverId")  
 **val** driver = **new** DriverRunner(  
 conf,  
 driverId,  
 *workDir*, //工作目录  
 *sparkHome*,  
 driverDesc.copy(command = Worker.*maybeUpdateSSLSettings*(driverDesc.command, conf)),  
 self,  
 *workerUri*,  
 securityMgr)  
 *drivers*(driverId) = driver

**//启动DriverRunner**  
 driver.start()  
  
 *coresUsed* += driverDesc.cores  
 *memoryUsed* += driverDesc.mem  
}

根据DriverId来具体管理DriverRunner。DriverRunner内部通过开辟线程的方式来启动了另外的一个线程。DriverRunner是Driver所在进程中Driver本身的Process

**DriverId DriverRunner**

**val** *drivers* = **new** HashMap[String, DriverRunner]

**DriverRunner:**

1. 管理Driver的执行，包括在Driver失败的时候自动重启，主要是指在standaolone模式

Worker会负责重新启动Driver。Cluster中的Driver失败的时候，如果supervise为true，则启动Driver的Worker会负责重新启动该Driver。

*/\*\*  
 \* Manages the execution of one driver, including automatically restarting the driver on failure.  
 \* This is currently only used in standalone cluster deploy mode.  
 \*/*

1. 创建Driver的工作目录

*/\*\* Starts a thread to run and manage the driver. \*/***private**[worker] **def** start() = {  
 **new** Thread("DriverRunner for " + driverId) {  
 **override def** run() {  
 **try** {  
 **val** driverDir = createWorkingDirectory()

1. createWorkingDirectory()创建Driver的工作目录

*/\*\*  
 \* Creates the working directory for this driver.  
 \* Will throw an exception if there are errors preparing the directory.  
 \*/***private def** createWorkingDirectory(): File = {

*//创建Driver的工作目录*  
 **val** driverDir = **new** File(workDir, driverId)  
 **if** (!driverDir.exists() && !driverDir.mkdirs()) {  
 **throw new** IOException("Failed to create directory " + driverDir)  
 }  
 driverDir  
}

1. 自己写的代码打成Jar包。

**val** localJarFilename = downloadUserJar(driverDir)

下载Jar文件，返回Jar在本地的路径， 将程序打成JAR包上传到HDFS上，这样每台机器均可以从HDFS上下载。

*/\*\*  
 \* Download the user jar into the supplied directory and return its local path.  
 \* Will throw an exception if there are errors downloading the jar.  
 \*/***private def** downloadUserJar(driverDir: File): String = {

*//*  
 **val** jarPath = **new** Path(driverDesc.jarUrl)  
//从HDFS上获取Jar文件。  
 **val** hadoopConf = SparkHadoopUtil.*get*.newConfiguration(conf)  
 **val** destPath = **new** File(driverDir.getAbsolutePath, jarPath.getName)  
 **val** jarFileName = jarPath.getName  
 **val** localJarFile = **new** File(driverDir, jarFileName)  
 **val** localJarFilename = localJarFile.getAbsolutePath  
  
 **if** (!localJarFile.exists()) { // May already exist if running multiple workers on one node  
 logInfo(s"Copying user jar **$**jarPath to **$**destPath")  
 Utils.fetchFile(  
 driverDesc.jarUrl,  
 driverDir,  
 conf,  
 securityManager,  
 hadoopConf,  
 System.*currentTimeMillis*(),  
 useCache = **false**)  
 }  
  
 **if** (!localJarFile.exists()) { // Verify copy succeeded  
 **throw new** Exception(s"Did not see expected jar **$**jarFileName in **$**driverDir")  
 }  
  
 localJarFilename  
}

1. 有些变量在开始的时候是占位符，因为还没有初始化，所以在实际运行的时候要初始化。

**def** substituteVariables(argument: String): String = argument **match** {  
 **case** "{{WORKER\_URL}}" => workerUrl   
 **case** "{{USER\_JAR}}" => localJarFilename //前面已经下载好了。  
 **case** other => other  
}

1. command主要就是构建进程执行类的入口。

// *TODO: If we add ability to submit multiple jars they should also be added here*

// driverDesc.command指定启动的时候运行什么类。**val** builder = CommandUtils.*buildProcessBuilder*(driverDesc.command, securityManager,  
 driverDesc.mem, sparkHome.getAbsolutePath, substituteVariables)

//launchDriver  
 launchDriver(builder, driverDir, driverDesc.supervise)  
}

1. launchDriver的源码如下：
2. 将stdout和stderr重定向到了baseDir之下了，这样就可以通过log去查看之前的执行情况。

**private def** launchDriver(builder: ProcessBuilder, baseDir: File, supervise: Boolean) {  
 builder.directory(baseDir)  
 **def** initialize(process: Process): Unit = {  
 // Redirect stdout and stderr to files  
 **val** stdout = **new** File(baseDir, "stdout")  
 CommandUtils.*redirectStream*(process.getInputStream, stdout)  
  
 **val** stderr = **new** File(baseDir, "stderr")

**//将command格式化一下**  
 **val** formattedCommand = builder.command.asScala.mkString("\"", "\" \"", "\"")  
 **val** header = "Launch Command: %s\n%s\n\n".format(formattedCommand, "=" \* 40)  
 Files.append(header, stderr, UTF\_8)  
 CommandUtils.*redirectStream*(process.getErrorStream, stderr)  
 }  
 runCommandWithRetry(*ProcessBuilderLike*(builder), initialize, supervise)  
}

1. ProcessBuilderLike静态方法:

**private**[deploy] **object** ProcessBuilderLike {

**//apply方法复写了start方法**  
 **def** apply(processBuilder: ProcessBuilder): ProcessBuilderLike = **new** ProcessBuilderLike {  
 **override def** start(): Process = processBuilder.start()  
 **override def** command: Seq[String] = processBuilder.command().asScala  
 }  
}

ProcessBuilderLike源码如下：

// Needed because ProcessBuilder is a final class and cannot be mocked  
**private**[deploy] **trait** ProcessBuilderLike {  
 **def** start(): Process  
 **def** command: Seq[String]  
}

1. 而在runCommandWithRetry方法中：

**//传入**ProcessBuilderLike的接口

**def** runCommandWithRetry(  
 command: ProcessBuilderLike, initialize: Process => Unit, supervise: Boolean): Unit = {  
 // Time to wait between submission retries.  
 **var** waitSeconds = 1  
 // A run of this many seconds resets the exponential back-off.  
 **val** successfulRunDuration = 5  
  
 **var** keepTrying = !*killed* **while** (keepTrying) {  
 logInfo("Launch Command: " + command.command.mkString("\"", "\" \"", "\""))  
  
 synchronized {  
 **if** (*killed*) { **return** }

**//调用**ProcessBuilderLike的start()方法  
 *process* = *Some*(command.start())  
 initialize(*process*.get)  
 }  
  
 **val** processStart = *clock*.getTimeMillis()

**//然后再调用***process*.get.waitFor()来完成启动Driver。  
 **val** exitCode = *process*.get.waitFor()  
 **if** (*clock*.getTimeMillis() - processStart > successfulRunDuration \* 1000) {  
 waitSeconds = 1  
 }

**if** (supervise && exitCode != 0 && !*killed*) {  
 logInfo(s"Command exited with status **$**exitCode, re-launching after **$**waitSeconds s.")  
 *sleeper*.sleep(waitSeconds)  
 waitSeconds = waitSeconds \* 2 // exponential back-off  
 }  
  
 keepTrying = supervise && exitCode != 0 && !*killed  
 finalExitCode* = *Some*(exitCode)  
}

1. 最后，如果Driver的状态有变，则会给自己发条消息。

worker.send(*DriverStateChanged*(driverId, state, *finalException*))

Worker端：

**case** driverStateChanged @ *DriverStateChanged*(driverId, state, exception) => {

**//处理Driver State Changed**  
 handleDriverStateChanged(driverStateChanged)  
}

1. 给Master发消息

**private**[worker] **def** handleDriverStateChanged(driverStateChanged: DriverStateChanged): Unit = {  
 **val** driverId = driverStateChanged.driverId  
 **val** exception = driverStateChanged.exception  
 **val** state = driverStateChanged.state  
 state **match** {  
 **case** DriverState.*ERROR* =>  
 logWarning(s"Driver **$**driverId failed with unrecoverable exception: **$**{exception.get}")  
 **case** DriverState.*FAILED* =>  
 logWarning(s"Driver **$**driverId exited with failure")  
 **case** DriverState.*FINISHED* =>  
 logInfo(s"Driver **$**driverId exited successfully")  
 **case** DriverState.*KILLED* =>  
 logInfo(s"Driver **$**driverId was killed by user")  
 **case** \_ =>  
 logDebug(s"Driver **$**driverId changed state to **$**state")  
 }

**//给master发送消息，告诉master，Driver状态发生变化了。**  
 sendToMaster(driverStateChanged)

1. 然后Master端receive方法是负责接收Worker发消息的。根据Driver状态进行处理。

**case** *DriverStateChanged*(driverId, state, exception) => {  
 state **match** {  
 **case** DriverState.*ERROR* | DriverState.*FINISHED* | DriverState.*KILLED* | DriverState.*FAILED* =>  
 removeDriver(driverId, state, exception)  
 **case** \_ =>  
 **throw new** Exception(s"Received unexpected state update for driver **$**driverId: **$**state")  
 }  
}

1. removeDriver方法：从自己的数据结构中remove掉。

**private def** removeDriver(  
 driverId: String,  
 finalState: DriverState,  
 exception: Option[Exception]) {  
 *drivers*.find(d => d.id == driverId) **match** {  
 **case** *Some*(driver) =>  
 logInfo(s"Removing driver: **$**driverId")  
 *drivers* -= driver  
 **if** (*completedDrivers*.size >= *RETAINED\_DRIVERS*) {  
 **val** toRemove = math.*max*(*RETAINED\_DRIVERS* / 10, 1)  
 *completedDrivers*.trimStart(toRemove)  
 }  
 *completedDrivers* += driver

**//删除持久化引擎，例如Zookeeper持久化数据。**  
 *persistenceEngine*.removeDriver(driver)  
 driver.*state* = finalState  
 driver.*exception* = exception  
 driver.*worker*.foreach(w => w.removeDriver(driver))

**//资源发生了变动，执行下schedule**  
 schedule()  
 **case** None =>  
 logWarning(s"Asked to remove unknown driver: **$**driverId")  
 }  
}

**LaunchExecutor：**

1. 先判断是否此时的路径是是activeMasterUrl.

**case** *LaunchExecutor*(masterUrl, appId, execId, appDesc, cores\_, memory\_) =>  
 **if** (masterUrl != *activeMasterUrl*) {  
 logWarning("Invalid Master (" + masterUrl + ") attempted to launch executor.")

1. 创建Executor的工作目录

// Create the executor's working directory  
**val** executorDir = **new** File(*workDir*, appId + "/" + execId)

1. 启动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  
manager.start()

1. Start()方法通过fetchAndRunExecutor方法启动Executor。

**private**[worker] **def** start() {  
 *workerThread* = **new** Thread("ExecutorRunner for " + *fullId*) {  
 **override def** run() { fetchAndRunExecutor() }  
 }

1. fetchAndRunExecutor源码如下：

*/\*\*  
 \* Download and run the executor described in our ApplicationDescription  
 \*/***private def** fetchAndRunExecutor() {  
 **try** {  
 // Launch the process  
 **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")  
  
 builder.directory(executorDir)  
 builder.environment.put("SPARK\_EXECUTOR\_DIRS", appLocalDirs.mkString(File.*pathSeparator*))  
 // In case we are running this from within the Spark Shell, avoid creating a "scala"  
 // parent process for the executor command  
 builder.environment.put("SPARK\_LAUNCH\_WITH\_SCALA", "0")  
  
 // Add webUI log urls  
 **val** baseUrl =  
 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")  
  
 *process* = builder.start()  
 **val** header = "Spark Executor Command: %s\n%s\n\n".format(  
 formattedCommand, "=" \* 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, 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

*//executor状态改变的时候给Worker发消息。*  
 worker.send(*ExecutorStateChanged*(appId, execId, state, *Some*(message), *Some*(exitCode)))

1. 然后Worker将消息发送给Master

sendToMaster(*ExecutorStateChanged*(appId, execId, manager.state, None, None))

1. Master端处理的时候，还要给Driver发送消息

**case** *ExecutorStateChanged*(appId, execId, state, message, exitStatus) => {  
 **val** execOption = *idToApp*.get(appId).flatMap(app => app.*executors*.get(execId))  
 execOption **match** {  
 **case** *Some*(exec) => {  
 **val** appInfo = *idToApp*(appId)  
 **val** oldState = exec.*state* exec.*state* = state  
  
 **if** (state == ExecutorState.*RUNNING*) {  
 *assert*(oldState == ExecutorState.*LAUNCHING*,  
 s"executor **$**execId state transfer from **$**oldState to RUNNING is illegal")  
 appInfo.resetRetryCount()  
 }  
//给Driver发送消息告诉Driver，Executor状态发生改变了。  
 exec.application.driver.send(*ExecutorUpdated*(execId, state, message, exitStatus))

**Worker原理内幕和流程控制如下图：**

