Spark启动流程及SparkContext创建

spark启动流程：

sbin/start-all.sh -> start-master.sh -> start-slaves.sh

sbin/start-master.sh -> 先读取变量 sbin/spark-daemon.sh start org.apache.spark.deploy.master.Master 1 --ip $SPARK\_MASTER\_IP --port $SPARK\_MASTER\_PORT --webui-port $SPARK\_MASTER\_WEBUI\_PORT

sbin/spark-daemon.sh -> /bin/spark-class $command "$@"

/bin/spark-class -> exec "$RUNNER" -cp "$CLASSPATH" $JAVA\_OPTS "$@"

-------------------------------------------------------------------------------------------------------------

spark提交任务的过程

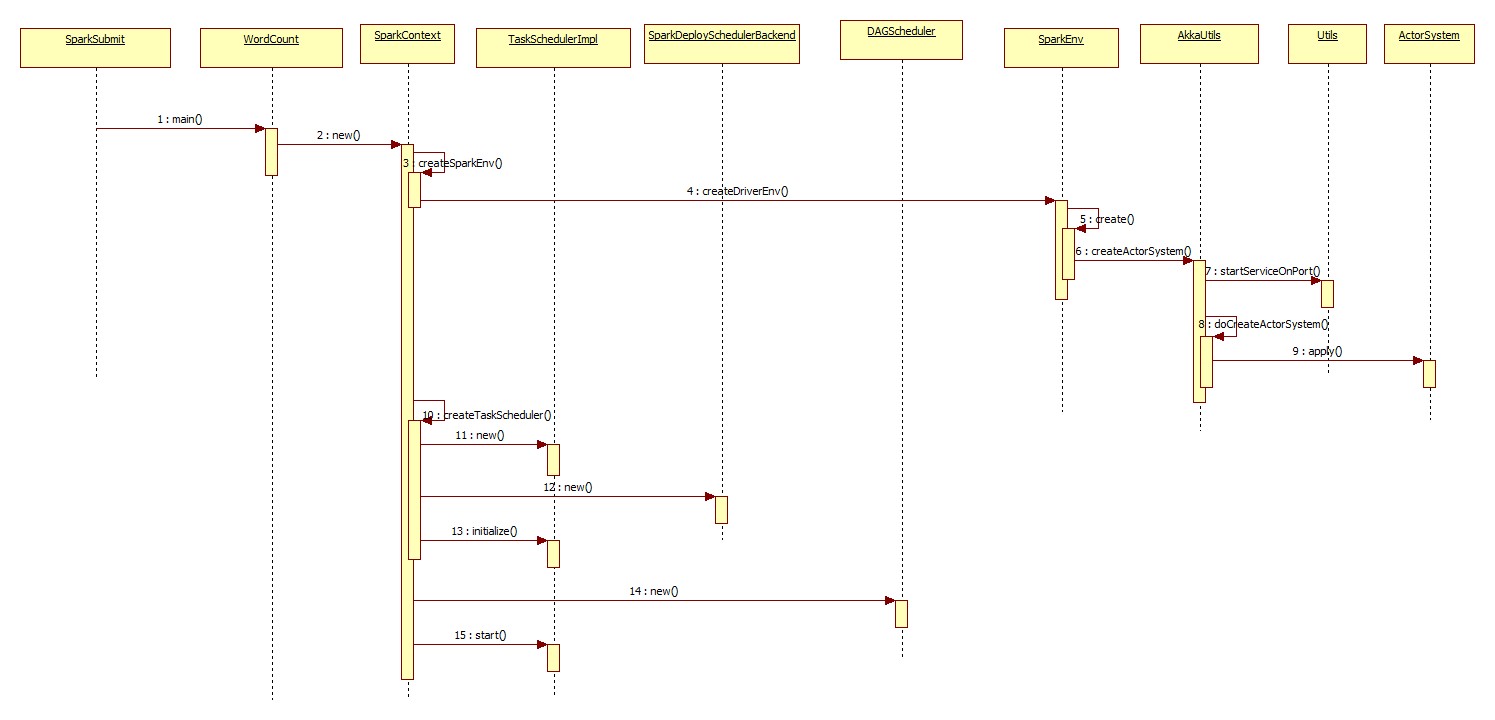
bin/spark-submit --class cn.itcast.spark.WordCount --master spark://node-1.itcast.cn:7077 --executor-memory 2g --total-executor-cores 4

exec "$SPARK\_HOME"/bin/spark-class org.apache.spark.deploy.SparkSubmit -> exec "$RUNNER" -cp "$CLASSPATH" $JAVA\_OPTS "$@"

重点来开一下spark-class org.apache.spark.deploy.SparkSubmit -》submit -》 doRunMain （args class cn.itcast.spark.WordCount ...）

--> Class.forName通过反射调用自定义类的main方法（只有一个进程）

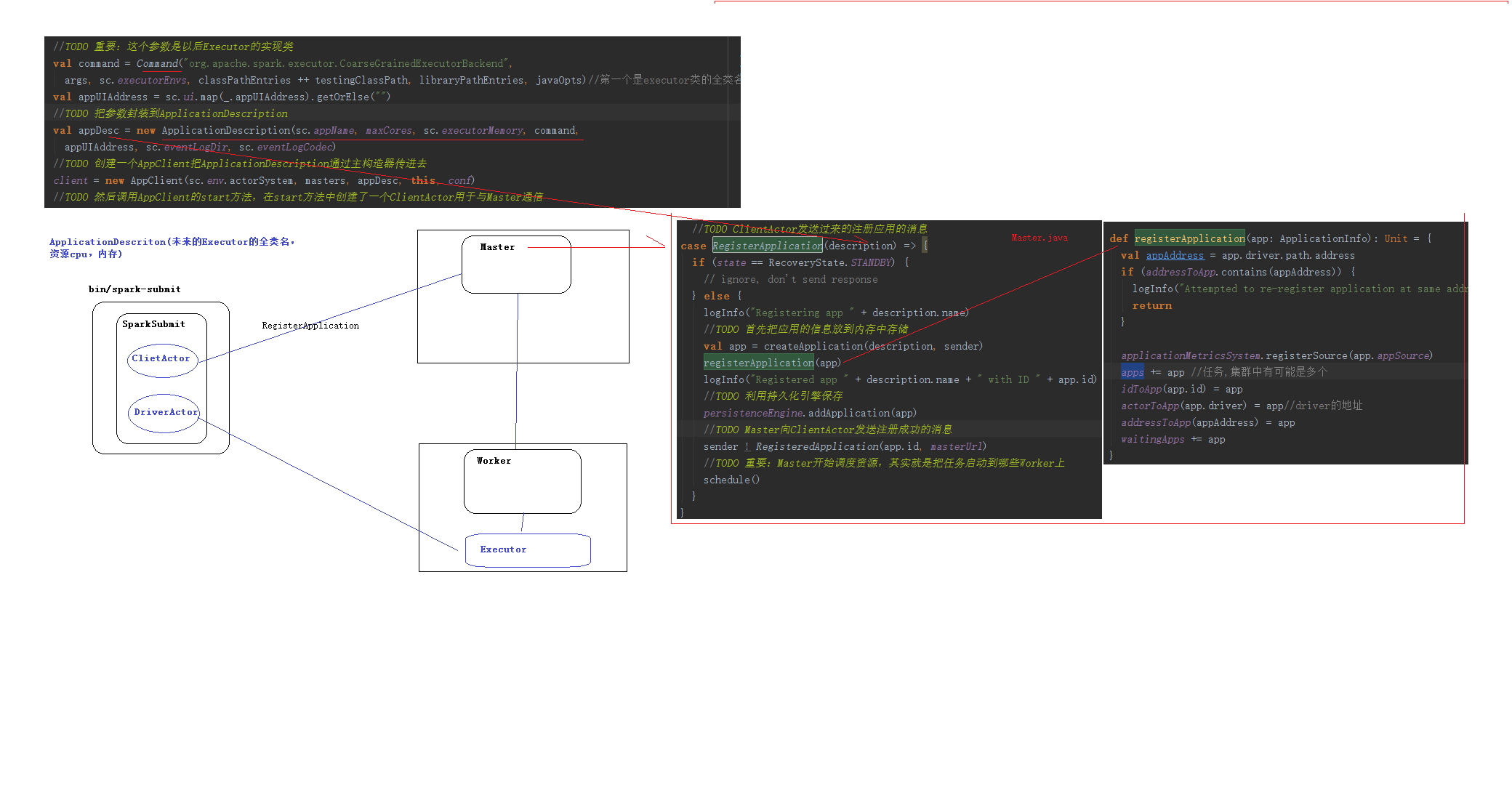
SparkContext创建：



conf = new SparkConf().setAppName("WordCount")

//非常重要，SparkContext是通向spark集群的入口，sparkContext实例在SparkSubmit(Driver) -> 跟Master建立连接（以后要RPC通信），

创建DAGScheduler->TaskScheduler



/\*\*

\* 很重要：SparkContext是Spark提交任务到集群的入口

\* 我们看一下SparkContext的主构造器

\* 1.调用createSparkEnv方法创建SparkEnv，里面有一个非常重要的对象ActorSystem

\* 2.创建TaskScheduler -> 根据提交任务的URL进行匹配 -> TaskSchedulerImpl -> SparkDeploySchedulerBackend(里面有两个Actor)

\* 3.创建DAGScheduler

\* 4.taskScheduler.start()

\*\*\*、

val sc = new SparkContext(conf) -> 执行柱构造器 ->1、创建了一个SparkEnv(创建ActorSystem)2、创建了TaskScheduler 3、场景DAGScheduler 启动TaskScheduler(调用start方法)

创建SparkEnv:调用createSparkEnv ->createDriverEnv ->create(判断Driver还是Executor)-> AkkaUtils.createActorSystem->doCreateActorSystem->ActorSystem

*// We need to register "HeartbeatReceiver" before "createTaskScheduler" because Executor will  
// retrieve "HeartbeatReceiver" in the constructor. (SPARK-6640)*

*/****TODO 通过ActorSystem创建了一个Actor，这个心跳是Executors和DriverActor的心跳***

*\_heartbeatReceiver* = env.rpcEnv.setupEndpoint(  
 HeartbeatReceiver.*ENDPOINT\_NAME*, **new** HeartbeatReceiver(**this**))  
  
*// Create and start the scheduler***val** (sched, ts) = SparkContext.createTaskScheduler(**this**, master)  
*\_schedulerBackend* = sched  
*\_taskScheduler* = ts  
*\_dagScheduler* = **new** DAGScheduler(**this**) */****TODO 传说中的DAGScheduler出现了，用于切分成Stage,然后在转换成TaskSet给TaskScheduler再提交给Executor***

*\_heartbeatReceiver*.ask[Boolean](TaskSchedulerIsSet)  
  
*// start TaskScheduler after taskScheduler sets DAGScheduler reference in DAGScheduler's  
// constructor  
\_taskScheduler*.start()

*/\*\*  
 \* Create a task scheduler based on a given master URL.  
 \* Return a 2-tuple of the scheduler backend and the task scheduler.  
 \*/*

**private def** createTaskScheduler(

**case** SPARK\_REGEX(sparkUrl) =>  
 **val** scheduler = **new** TaskSchedulerImpl(sc)  
 **val** masterUrls = sparkUrl.split(**","**).map(**"spark://"** + \_)  
 **val** backend = **new** SparkDeploySchedulerBackend(scheduler, sc, masterUrls)  
 scheduler.initialize(backend)  
 (backend, scheduler)

）

*// Start executors with a few necessary configs for registering with the scheduler***val** sparkJavaOpts = Utils.*sparkJavaOpts*(*conf*, SparkConf.*isExecutorStartupConf*)  
**val** javaOpts = sparkJavaOpts ++ extraJavaOpts

*//****TODO 重要：这个参数是以后Executor的实现类***

**val** command = *Command*(**"org.apache.spark.executor.CoarseGrainedExecutorBackend"**,  
 args, sc.*executorEnvs*, classPathEntries ++ testingClassPath, libraryPathEntries, javaOpts)  
**val** appUIAddress = sc.*ui*.map(\_.appUIAddress).getOrElse(**""**)

*/****TODO 把参数封装到ApplicationDescription***

**val** appDesc = **new** ApplicationDescription(sc.*appName*, *maxCores*, sc.*executorMemory*, command,  
 appUIAddress, sc.*eventLogDir*, sc.*eventLogCodec*)

*//****TODO 创建一个AppClient把ApplicationDescription通过主构造器传进去***

*client* = **new** AppClient(sc.*env*.actorSystem, masters, appDesc, **this**, *conf*)

*//****TODO 然后调用AppClient的start方法，在start方法中创建了一个ClientActor用于与Master通信***

*client*.start()

**在start函数里面会*首先调用父类的start方法来创建DriverActor***

***然后调用AppClient的start方法，在start方法中创建了一个ClientActor用于与Master通信*将各个参数封装appDesc传给AppClient：**

***driverActor* = actorSystem.actorOf(*Props*(new DriverActor(properties)), name = CoarseGrainedSchedulerBackend.*ACTOR\_NAME*)**

**actorSystem.actorOf(*Props*(new ClientActor))；……………**

**override def** preStart() {  
 *context*.system.eventStream.subscribe(*self*, *classOf*[RemotingLifecycleEvent])  
 **try** {  
 registerWithMaster()

**//***//****TODO ClientActor向Master注册***

**向master注册clientActor(负责与master通信)，一个DriverActor（负责与executor通信）**

*//****TODO 循环所有Master地址，跟Master建立连接*val** actor = *context*.actorSelection(masterAkkaUrl)

*//****TODO 拿到了Master的一个引用，然后向Master发送注册应用的请求，所有的参数都封装到appDescription***

actor ! *RegisterApplication*(appDescription)

//在master中***TODO ClientActor发送过来的注册应用的消息***

**case** *RegisterApplication*(description) => {  
 **if** (*state* == RecoveryState.*STANDBY*) {  
 *// ignore, don't send response* } **else** {  
 logInfo(**"Registering app "** + description.name)

*//****TODO 首先把应用的信息放到内存中存储***  
 **val** app = createApplication(description, sender)

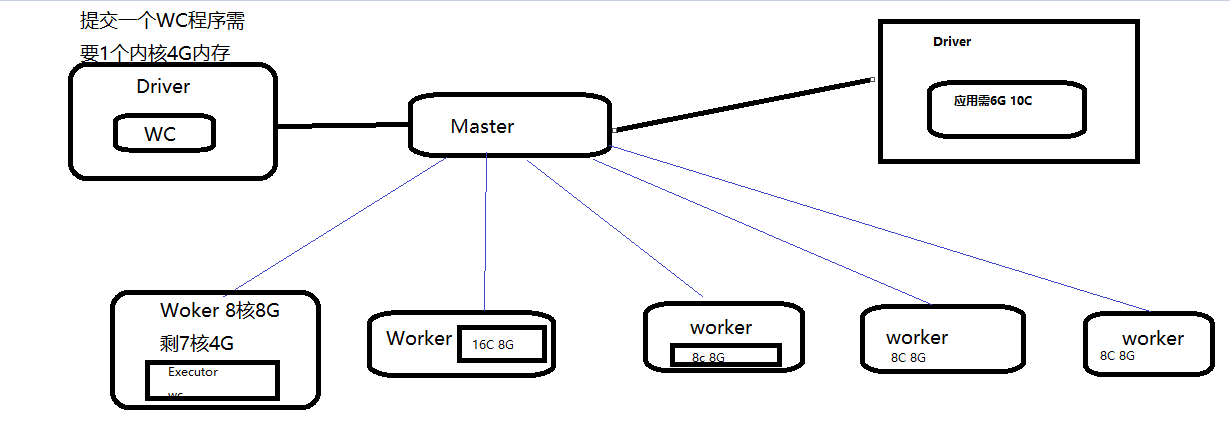
registerApplication(app)//**将封装信description以及driver ：sender封装成application,并且将application及driver保存起来**   
 logInfo(**"Registered app "** + description.name + **" with ID "** + app.id)

*//****TODO 利用持久化引擎保存***  
 *persistenceEngine*.addApplication(app)

*//****TODO Master向ClientActor发送注册成功的消息***

sender ! *RegisteredApplication*(app.id, *masterUrl*)

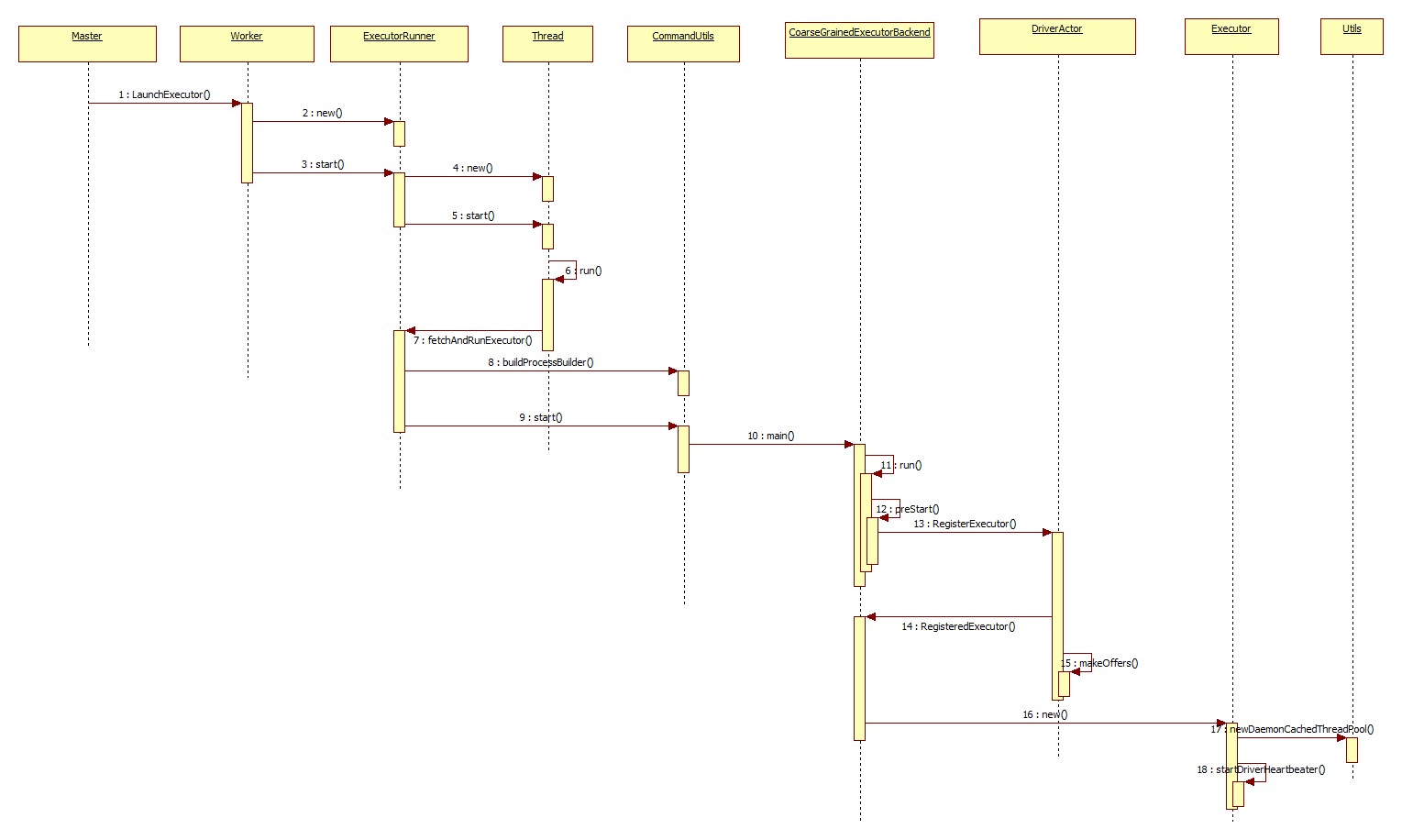
*/****TODO 重要：Master开始调度资源，其实就是把任务启动到哪些Worker上***  
  **schedule()**/**/很重要，资源的调度**  
 }  
}



两种资源调度方式：尽量集中；尽量打散

*// 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))

*//****TODO Master发送消息让Worker启动Executor***  
 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* }  
 }  
 }  
 }  
 }  
}



**def** launchExecutor(worker: WorkerInfo, exec: ExecutorDesc) {  
 logInfo(**"Launching executor "** + exec.fullId + **" on worker "** + worker.id)  
 *//****TODO 记录该Worker使用的资源*** worker.addExecutor(exec)  
 *//****TODO Master发送消息给Worker，把参数通过case class传递给Worker，让他启动Executor，*** worker.actor ! *LaunchExecutor*(*masterUrl*,  
 exec.application.id, exec.id, exec.application.desc, exec.cores, exec.memory)  
 *//****TODO Master向ClientActor发送消息，告诉它Executor已经启动了*** exec.application.driver ! *ExecutorAdded*(  
 exec.id, worker.id, worker.hostPort, exec.cores, exec.memory)  
}

//在woker中：***TODO Master发送给Worker的消息，让Worker启动Executor，LaunchExecutor是一个Case Class，里面封装了以后要启动的Executor的信息***

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

………..

*//****TODO 创建一个ExecutorRunner，将参数都放到其中，然后在通过他启动Executor*val** manager = **new** ExecutorRunner(  
 appId,  
 execId,  
 appDesc.copy(command = Worker.maybeUpdateSSLSettings(appDesc.command, conf)),  
 cores\_,  
 memory\_,  
 *self*,  
 *workerId*,  
 host,  
 *webUi*.boundPort,  
 *publicAddress*,  
 *sparkHome*,  
 executorDir,  
 *akkaUrl*,  
 conf,  
 appLocalDirs, ExecutorState.*LOADING*)  
*//****TODO 把ExecutorDI -> Executor放到一个Map中了****executors*(appId + **"/"** + execId) = manager  
*//****TODO 调用ExecutorRunner的start方法来启动Executor java子进程***manager.start()//见下面start

*coresUsed* += cores\_  
*memoryUsed* += memory\_

//TODO向Master告知worker启动状态变化  
*master* ! *ExecutorStateChanged*(appId, execId, manager.state, None, None)

**def** start() {  
 *//****TODO 先创建一个线程对象，然后通过一个线程来启动一个Java子进程*** *workerThread* = **new** Thread(**"ExecutorRunner for "** + *fullId*) {  
 **override def** run() { fetchAndRunExecutor() }  
 }  
 *//****TODO 调用线程对象的start方法 -> 线程对象的run方法*** *workerThread*.start()  
 *// Shutdown hook that kills actors on shutdown.  
 shutdownHook* = **new** Thread() {  
 **override def** run() {  
 killProcess(*Some*(**"Worker shutting down"**))  
 }  
 }  
 Runtime.*getRuntime*.addShutdownHook(*shutdownHook*)  
}

*//****TODO 线程对象调用该方法启动Java子进程*def** fetchAndRunExecutor() {  
 **try** {  
 *// Launch the process  
 //****TODO 启动子进程* val** builder = CommandUtils.*buildProcessBuilder*(appDesc.command, memory,  
 sparkHome.getAbsolutePath, substituteVariables)  
 **val** command = builder.command()  
 logInfo(**"Launch command: "** + command.mkString(**"\""**, **"\" \""**, **"\""**))

……

*//****TODO 真正启动一个Java子进程 -> CoarseGrainedExecutorBackend的main方法****process* = builder.start()  
**val** header = **"Spark Executor Command: %s\n%s\n\n"**.format(  
 command.mkString(**"\""**, **"\" \""**, **"\""**), **"="** \* 40)

……

//TODO wokerexecutor通知woker启动状态

worker ! *ExecutorStateChanged*(appId, execId, state, *Some*(message), *Some*(exitCode))

*//****TODO 启动Executor子进程的入口*def** main(args: Array[String]) {

//TODO DriverActor url，executor与DriverActor通信  
 **var driverUrl**: String = **null  
 var** executorId: String = **null  
 var** hostname: String = **null**

……..

*//****TODO 调用RUN方法****run*(driverUrl, executorId, hostname, cores, appId, workerUrl, userClassPath)

Def run{

……….

*//****TODO 在Executor里创建ActorSystem*val** (fetcher, \_) = AkkaUtils.*createActorSystem*(  
 **"driverPropsFetcher"**,  
 hostname,  
 port,  
 executorConf,  
 **new** SecurityManager(executorConf))  
*//****TODO 跟Driver建立连接*val** driver = fetcher.actorSelection(driverUrl)  
**val** timeout = AkkaUtils.*askTimeout*(executorConf)  
**val** fut = Patterns.*ask*(driver, RetrieveSparkProps, timeout)  
**val** props = Await.*result*(fut, timeout).asInstanceOf[Seq[(String, String)]] ++  
 *Seq*[(String, String)]((**"spark.app.id"**, appId))  
fetcher.shutdown()

…………

*//****TODO CoarseGrainedExecutorBackend真正进行通信的Actor***env.actorSystem.actorOf(  
 *Props*(*classOf*[CoarseGrainedExecutorBackend],  
 driverUrl, executorId, sparkHostPort, cores, userClassPath, env),  
 name = **"Executor"**)  
workerUrl.foreach { url =>  
 env.actorSystem.actorOf(*Props*(*classOf*[WorkerWatcher], url), name = **"WorkerWatcher"**)  
}  
env.actorSystem.awaitTermination()

}

**private**[spark] **class** CoarseGrainedExecutorBackend(

……

*//****TODO CoarseGrainedExecutorBackend的生命周期方法*override def** preStart() {  
 logInfo(**"Connecting to driver: "** + driverUrl)  
 *//****TODO 跟Driver建立连接*** *driver* = *context*.actorSelection(driverUrl)  
 *//****TODO Executor向DriverActor发送消息，来注册Exectuor*** *driver* ! *RegisterExecutor*(executorId, hostPort, cores, extractLogUrls)  
 *context*.system.eventStream.subscribe(*self*, *classOf*[RemotingLifecycleEvent])  
}

}

//在DriverActor中：

**def** receiveWithLogging = {  
 *//****TODO Executor向DriverActor发送的消息* case** *RegisterExecutor*(executorId, hostPort, cores, logUrls) =>  
 Utils.*checkHostPort*(hostPort, **"Host port expected "** + hostPort)  
 **if** (*executorDataMap*.contains(executorId)) {  
 sender ! *RegisterExecutorFailed*(**"Duplicate executor ID: "** + executorId)  
 } **else** {  
 logInfo(**"Registered executor: "** + sender + **" with ID "** + executorId)  
 *//****TODO DriverActor向Executor发送消息，告诉Executor注册成功*** sender ! RegisteredExecutor

………………..

*listenerBus*.post(  
 *SparkListenerExecutorAdded*(System.*currentTimeMillis*(), executorId, data))  
*//****TODO 重要 查看是否有任务需要提交（DriverActor -> Executor）***makeOffers()

}

在executor中：

**override def** receiveWithLogging = {  
 *//****TODO DirverActor发送给Executor的消息，告诉她已经注册成功* case** RegisteredExecutor =>  
 logInfo(**"Successfully registered with driver"**)  
 **val** (hostname, \_) = Utils.*parseHostPort*(hostPort)  
 *//****TODO 创建了一个Executor实例，用来执行业务逻辑*** *executor* = **new Executor**(executorId, hostname, env, userClassPath,

}

*/****TODO 是Executor的主构造器***

*//****TODO 初始化线程池*val** *threadPool* = Utils.*newDaemonCachedThreadPool*(**"Executor task launch worker"**)

…….

*// Create an actor for receiving RPCs from the driver***private val** *executorActor* = env.actorSystem.actorOf(  
 *Props*(**new** ExecutorActor(executorId)), **"ExecutorActor"**)

……….

*//****TODO Executor向DriverActor发送心跳***startDriverHeartbeater()

*//在Master中：****TODO Worker发送给Master的消息，告诉Master Executor已经启动*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)  
 exec.*state* = state  
 **if** (state == ExecutorState.*RUNNING*) { appInfo.resetRetryCount() }  
 exec.application.driver ! *ExecutorUpdated*(execId, state, message, exitStatus)  
 **if** (ExecutorState.*isFinished*(state)) {  
 *// Remove this executor from the worker and app* logInfo(**s"Removing executor $**{exec.fullId} **because it is $**state**"**)  
 appInfo.removeExecutor(exec)  
 exec.worker.removeExecutor(exec)  
  
 **val** normalExit = exitStatus == *Some*(0)  
 *// Only retry certain number of times so we don't go into an infinite loop.* **if** (!normalExit) {  
 **if** (appInfo.incrementRetryCount() < ApplicationState.*MAX\_NUM\_RETRY*) {  
 schedule()  
 } **else** {  
 **val** execs = appInfo.*executors*.values  
 **if** (!execs.exists(\_.*state* == ExecutorState.*RUNNING*)) {  
 logError(**s"Application $**{appInfo.desc.name} **with ID $**{appInfo.id} **failed "** +  
 **s"$**{appInfo.retryCount} **times; removing it"**)  
 removeApplication(appInfo, ApplicationState.*FAILED*)  
 }  
 }  
 }  
 }  
 }

*//在appclien中：****TODO Master发送给ClientActor的消息，告诉ClientActor Master已经向Worker发送了启动Executor的消息*case** *ExecutorAdded*(id: Int, workerId: String, hostPort: String, cores: Int, memory: Int) =>  
 **val** fullId = *appId* + **"/"** + id  
 logInfo(**"Executor added: %s on %s (%s) with %d cores"**.format(fullId, workerId, hostPort,  
 cores))  
 *master* ! *ExecutorStateChanged*(*appId*, id, ExecutorState.*RUNNING*, None, None)  
 listener.executorAdded(fullId, workerId, hostPort, cores, memory)

**RDD提交及state划分流程：**

**def** saveAsTextFile(path: String) {  
 *// https://issues.apache.org/jira/browse/SPARK-2075  
 //  
 // NullWritable is a `Comparable` in Hadoop 1.+, so the compiler cannot find an implicit  
 // Ordering for it and will use the default `null`. However, it's a `Comparable[NullWritable]`  
 // in Hadoop 2.+, so the compiler will call the implicit `Ordering.ordered` method to create an  
 // Ordering for `NullWritable`. That's why the compiler will generate different anonymous  
 // classes for `saveAsTextFile` in Hadoop 1.+ and Hadoop 2.+.  
 //  
 // Therefore, here we provide an explicit Ordering `null` to make sure the compiler generate  
 // same bytecodes for `saveAsTextFile`.* **val** nullWritableClassTag = *implicitly*[ClassTag[NullWritable]]  
 **val** textClassTag = *implicitly*[ClassTag[Text]]  
 *//****TODO rdd在用saveAsTextFile方法时还会产生一个RDD* val** r = **this**.mapPartitions { iter =>  
 **val** text = **new** Text()  
 iter.map { x =>  
 text.set(x.toString)  
 (NullWritable.*get*(), text)  
 }  
 }  
 *//****TODO saveAsHadoopFile*** RDD.*rddToPairRDDFunctions*(r)(nullWritableClassTag, textClassTag, **null**)  
 .saveAsHadoopFile[TextOutputFormat[NullWritable, Text]](path)  
}

**def** saveAsHadoopFile(

………

*//****TODO 准备一下HDFS的参数***hadoopConf.set(**"mapred.output.format.class"**, outputFormatClass.getName)  
**for** (c <- codec) {  
 hadoopConf.setCompressMapOutput(**true**)  
 hadoopConf.set(**"mapred.output.compress"**, **"true"**)  
 hadoopConf.setMapOutputCompressorClass(c)  
 hadoopConf.set(**"mapred.output.compression.codec"**, c.getCanonicalName)  
 hadoopConf.set(**"mapred.output.compression.type"**, CompressionType.*BLOCK*.toString)

….

*//****TODO saveAsHadoopDataset***saveAsHadoopDataset(hadoopConf)

}

**def** saveAsHadoopDataset(conf: JobConf) {

………

*//****TODO 创建了一个HDFS的流*val** writer = **new** SparkHadoopWriter(hadoopConf)  
writer.preSetup()

*//****TODO 这是一个函数(Lazy)，Iterator把一个分区的数据传过去*val** writeToFile = (context: TaskContext, iter: Iterator[(K, V)]) => {  
 **val** config = wrappedConf.value  
 *// Hadoop wants a 32-bit task attempt ID, so if ours is bigger than Int.MaxValue, roll it  
 // around by taking a mod. We expect that no task will be attempted 2 billion times.* **val** taskAttemptId = (context.taskAttemptId % Int.*MaxValue*).toInt  
  
 **val** (outputMetrics, bytesWrittenCallback) = initHadoopOutputMetrics(context)  
  
 writer.setup(context.stageId, context.partitionId, taskAttemptId)  
 writer.open()  
 **var** recordsWritten = 0L  
 **try** {  
 **while** (iter.hasNext) {  
 **val** record = iter.next()  
 writer.write(record.\_1.asInstanceOf[AnyRef], record.\_2.asInstanceOf[AnyRef])  
  
 *// Update bytes written metric every few records* maybeUpdateOutputMetrics(bytesWrittenCallback, outputMetrics, recordsWritten)  
 recordsWritten += 1  
 }  
 } **finally** {  
 writer.close()  
 }  
 writer.commit()  
 bytesWrittenCallback.foreach { fn => outputMetrics.setBytesWritten(fn()) }  
 outputMetrics.setRecordsWritten(recordsWritten)  
}  
  
*//****TODO 开始提交任务,self就是DAG中的最后一个RDD，这个RDD通过依赖关系进行切分Stage***self.context.runJob(self, writeToFile)  
writer.commitJob()

}

*//****TODO 将最后一个RDD和一个函数传入到该方法中*def** runJob[T, U: ClassTag](rdd: RDD[T], func: (TaskContext, Iterator[T]) => U): Array[U] = {  
 runJob(rdd, func, 0 until rdd.partitions.size, **false**)  
}

**def** runJob[T, U: ClassTag](

……………….

*//****TODO 传说中的DAGScheduler出现了，用于切分成Stage,然后在转换成TaskSet给TaskScheduler再提交给Executor****dagScheduler*.runJob(rdd, cleanedFunc, partitions, callSite, allowLocal,  
 resultHandler, *localProperties*.get)  
*progressBar*.foreach(\_.finishAll())  
rdd.doCheckpoint()

}

*//****TODO DAGScheduler的runJob方法，切分stage*def** runJob[T, U: ClassTag](

………

*//****TODO 调用submitJob方法并返回一个回调器*val** waiter = submitJob(rdd, func, partitions, callSite, allowLocal, resultHandler, properties)

}

**def** submitJob[T, U](

………………..

**val** waiter = **new** JobWaiter(**this**, jobId, partitions.size, resultHandler)  
*//****TODO 先将数据封装到事件中然后放入到eventProcessLoop的阻塞队列中****eventProcessLoop*.post(*JobSubmitted*(  
 jobId, rdd, func2, partitions.toArray, allowLocal, callSite, waiter, properties))  
waiter

}

**def** post(event: E): Unit = {  
 *eventQueue*.put(event)  
}

//在eventQueue中的onReicve函数的子类DAGSchedulerEventProcessLoop中：

*//****TODO 通过模式匹配判断事件的类型*override def** onReceive(event: DAGSchedulerEvent): Unit = event **match** {  
 *//****TODO 提交计算任务* case** *JobSubmitted*(jobId, rdd, func, partitions, allowLocal, callSite, listener, properties) =>  
 *//****TODO 调用dagScheduler的handleJobSubmitted方法处理*** dagScheduler.handleJobSubmitted(jobId, rdd, func, partitions, allowLocal, callSite,  
 listener, properties)

*//****TODO 用切stage*private**[scheduler] **def** handleJobSubmitted(

……………………….

*//****TODO 重要：该方法用于划分Stage***finalStage = newStage(finalRDD, partitions.size, None, jobId, callSite)

…………….

*/****TODO 集群模式*** *jobIdToActiveJob*(jobId) = job  
 *activeJobs* += job  
 finalStage.*resultOfJob* = *Some*(job)  
 **val** stageIds = *jobIdToStageIds*(jobId).toArray  
 **val** stageInfos = stageIds.flatMap(id => *stageIdToStage*.get(id).map(\_.*latestInfo*))  
 listenerBus.post(  
 *SparkListenerJobStart*(job.jobId, jobSubmissionTime, stageInfos, properties))  
 *//****TODO 开始提交Stage*** submitStage(finalStage)  
}

}

*//****TODO 用于创建Stage*private def** newStage(  
 rdd: RDD[\_],  
 numTasks: Int,  
 shuffleDep: Option[ShuffleDependency[\_, \_, \_]],  
 jobId: Int,  
 callSite: CallSite)  
 : Stage =  
{  
 *//****TODO 获取他的父Stage* val** parentStages = getParentStages(rdd, jobId)  
 **val** id = *nextStageId*.getAndIncrement()  
 **val** stage = **new** Stage(id, rdd, numTasks, shuffleDep, parentStages, jobId, callSite)  
 *stageIdToStage*(id) = stage  
 updateJobIdStageIdMaps(jobId, stage)  
 stage  
}

*//****TODO 用户获取父Stage*private def** getParentStages(rdd: RDD[\_], jobId: Int): List[Stage] = {  
 **val** parents = **new** HashSet[Stage]  
 **val** visited = **new** HashSet[RDD[\_]]  
 *// We are manually maintaining a stack here to prevent StackOverflowError  
 // caused by recursively visiting* **val** waitingForVisit = **new** Stack[RDD[\_]]  
 **def** visit(r: RDD[\_]) {  
 **if** (!visited(r)) {  
 visited += r  
 *// Kind of ugly: need to register RDDs with the cache here since  
 // we can't do it in its constructor because # of partitions is unknown* **for** (dep <- r.dependencies) {  
 dep **match** {  
 **case** shufDep: ShuffleDependency[\_, \_, \_] =>  
 *//****TODO 把宽依赖传进去，获得父Stage*** parents += getShuffleMapStage(shufDep, jobId)  
 **case** \_ =>  
 waitingForVisit.push(dep.rdd)  
 }  
 }  
 }  
 }  
 waitingForVisit.push(rdd)  
 **while** (!waitingForVisit.isEmpty) {  
 visit(waitingForVisit.pop())  
 }  
 parents.toList  
}

**private def** getShuffleMapStage(shuffleDep: ShuffleDependency[\_, \_, \_], jobId: Int): Stage = {  
 *shuffleToMapStage*.get(shuffleDep.*shuffleId*) **match** {  
 **case** *Some*(stage) => stage  
 **case** None =>  
 *// We are going to register ancestor shuffle dependencies* registerShuffleDependencies(shuffleDep, jobId)  
 *// Then register current shuffleDep* **val** stage =  
 *//****TODO 创建服父Stage*** newOrUsedStage(  
 shuffleDep.rdd, shuffleDep.rdd.partitions.size, shuffleDep, jobId,  
 shuffleDep.rdd.*creationSite*)  
 *shuffleToMapStage*(shuffleDep.*shuffleId*) = stage  
  
 stage  
 }  
}

**private def** newOrUsedStage(  
 rdd: RDD[\_],  
 numTasks: Int,  
 shuffleDep: ShuffleDependency[\_, \_, \_],  
 jobId: Int,  
 callSite: CallSite)  
 : Stage =  
{//递归调用newStage了  
 **val** stage = **newStage**(rdd, numTasks, *Some*(shuffleDep), jobId, callSite)

…….

}

*/\*\* Submits stage, but first recursively submits any missing parents. \*/  
//****TODO 根据最后一个Stage，递归的提交Stage*private def** submitStage(stage: Stage) {  
 **val** jobId = activeJobForStage(stage)  
 **if** (jobId.isDefined) {  
 logDebug(**"submitStage("** + stage + **")"**)  
 **if** (!*waitingStages*(stage) && !*runningStages*(stage) && !*failedStages*(stage)) {  
 *//****TODO 获取他的父Stage* val** missing = getMissingParentStages(stage).sortBy(\_.id)  
 logDebug(**"missing: "** + missing)  
 *//判断父Stage是否为空，为空就意味着他是第一个Stage* **if** (missing == *Nil*) {  
 logInfo(**"Submitting "** + stage + **" ("** + stage.rdd + **"), which has no missing parents"**)  
 *//****TODO 开始提交最前面的Stage*** submitMissingTasks(stage, jobId.get)  
 } **else** {  
 *//****TODO 有父Stage，就递归提交* for** (parent <- missing) {  
 submitStage(parent)  
 }  
 *waitingStages* += stage  
 }  
 }

*//****TODO DAG提交Stage个TaskScheduler*private def** submitMissingTasks(stage: Stage, jobId: Int) {

…………….

*//****TODO 创建多少个Task*val** tasks: Seq[Task[\_]] = **if** (stage.*isShuffleMap*) {  
 partitionsToCompute.map { id =>  
 **val** locs = getPreferredLocs(stage.rdd, id)  
 **val** part = stage.rdd.partitions(id)  
 *//ShuffleMapTask用于拉取上游的数据* **new** ShuffleMapTask(stage.id, taskBinary, part, locs)  
 }  
} **else** {  
 **val** job = stage.*resultOfJob*.get  
 partitionsToCompute.map { id =>  
 **val** p: Int = job.partitions(id)  
 **val** part = stage.rdd.partitions(p)  
 **val** locs = getPreferredLocs(stage.rdd, p)  
 *//ResultTask将计算结果写入HDFS、NoSQL。。。* **new** ResultTask(stage.id, taskBinary, part, locs, id)  
 }

}

………..

*//****TODO 调用taskScheduler的submitTasks方法来提交TaskSet***taskScheduler.submitTasks(  
 **new** TaskSet(tasks.toArray, stage.id, stage.newAttemptId(), stage.jobId, properties))  
stage.*latestInfo*.*submissionTime* = *Some*(clock.getTimeMillis())

}

*/****TODO 该方法用于提交Tasks*override def** submitTasks(taskSet: TaskSet) {  
 **val** tasks = taskSet.tasks

…………………

*//****TODO 向DriverActor发消息的任务*** *backend*.reviveOffers()  
}

**override def** reviveOffers() {  
 *driverActor* ! ReviveOffers  
}

*//****TODO 调用makeOffers向Executor提交Task*case** ReviveOffers =>  
 makeOffers()

**def** makeOffers() {  
 *//****TODO 调用launchTask向Executor提交Task*** launchTasks(scheduler.resourceOffers(*executorDataMap*.map { **case** (id, executorData) =>  
 **new** WorkerOffer(id, executorData.executorHost, executorData.freeCores)  
 }.toSeq))  
}

**def** launchTasks(tasks: Seq[Seq[TaskDescription]]) {

……………

*//****TODO 序列化Task*val** serializedTask = ser.serialize(task)

…………..

*//****TODO 向Executor发送序列化好的Task***executorData.executorActor ! *LaunchTask*(**new** SerializableBuffer(serializedTask))

}

*//****TODO DirverActor发送给Executor的消息，让Executor启动计算任务*case** *LaunchTask*(data) =>  
 **if** (*executor* == **null**) {  
 logError(**"Received LaunchTask command but executor was null"**)  
 System.*exit*(1)  
 } **else** {  
 *//获得序列化器* **val** ser = env.closureSerializer.newInstance()  
 *//****TODO 反序列化Task* val** taskDesc = ser.deserialize[TaskDescription](data.value)  
 logInfo(**"Got assigned task "** + taskDesc.taskId)  
 *//****TODO 将反序列化后的Task放到线程池里面*** *executor*.launchTask(**this**, taskId = taskDesc.taskId, attemptNumber = taskDesc.attemptNumber,  
 taskDesc.name, taskDesc.serializedTask)  
 }

*//****TODO 启动Task*def** launchTask(  
 context: ExecutorBackend,  
 taskId: Long,  
 attemptNumber: Int,  
 taskName: String,  
 serializedTask: ByteBuffer) {  
 *//****TODO 创建一个TaskRunner对象，把Task的信息封装到TaskRunner里面* val** tr = **new** TaskRunner(context, taskId = taskId, attemptNumber = attemptNumber, taskName,  
 serializedTask)  
 *runningTasks*.put(taskId, tr)  
 *//****TODO 把TaskRunner丢到线程池中*** *threadPool*.execute(tr)  
}

*//****TODO 执行Task真正的业务逻辑*override def** run() {

……….

*//获取序列化器***val** ser = env.closureSerializer.newInstance()

……..

*//****TODO 反序列化****task* = ser.deserialize[Task[Any]](taskBytes, Thread.*currentThread*.getContextClassLoader)

…

*//****TODO 调用Task的run方法*val** value = *task*.run(taskAttemptId = taskId, attemptNumber = attemptNumber)

……..

}

*//****TODO 执行Task***runTask(*context*)

**override def** runTask(context: TaskContext): U = {  
 *// Deserialize the RDD and the func using the broadcast variables.  
 //得到一个序列化器* **val** ser = SparkEnv.*get*.closureSerializer.newInstance()  
 *//反序列化Task，得到RDD和作用在RDD上的函数* **val** (rdd, func) = ser.deserialize[(RDD[T], (TaskContext, Iterator[T]) => U)](  
 ByteBuffer.*wrap*(taskBinary.value), Thread.*currentThread*.getContextClassLoader)  
  
 *metrics* = *Some*(context.taskMetrics)  
 *//开始调用这个函数* func(context, rdd.iterator(partition, context))  
}