简要：

本篇博文主要讨论的内容如下;

1. Executor工作原理
2. ExecutorBackend注册源码解密
3. Executor实例化及具体如何工作

**前置知识：**

**为啥Worker不直接启动Executor，而是启动另一个线程来启动Executor？**

1. **Worker是管理当前机器资源的，当前机器的资源发生改变的时候，会将消息汇报给Master的。**
2. **Spark集群中会有很多应用程序，很多应用程序你就需要很多Executor，如果你不是为每个Executor启动一个进程，而是让这些所有的Executor都在Worker里面，那岂不是一个程序崩溃了，其他程序也崩溃了。**

**AppClient：代表应用应用程序本身；**

**一：Spark Executor工作原理**

1. 需要特别注意是在CoarseGrainedExecutorBackend启动时向Driver注册Executor其实质是注册ExecutorBackend实例，和Executor无关。

2．**CoarseGrainedExecutorBackend是Executor运行所在的进程名称**，**CoarseGrainedExecutorBackend本身并不会完成具体任务的计算，Executor才会完成计算。**Executor才是处理Task的对象，Executor内部是通过线程池的方式来完成Task的计算的；

3．**CoarseGrainedExecutorBackend是一个进程，里面有一个**Executor对象，CoarseGrainedExecutorBackend和Executor是一一对应的；

4．CoarseGrainedExecutorBackend是一个消息通信体(其实现了ThreadSafeRpcEndPoint)。**可以发送消息给Driver并可以接受Driver中发过来的指令**，**例如启动Task等**；

5. 在Driver进程有两个至关重要的Endpoint:

a) ClientEndpoint: 主要负责向Master注册当前的程序,是AppClient的内部成员；

b) DriverEndpoint:这是整个程序运行时候的驱动器,例如接收CoarseGrainedExecutorBackend的注册，是CoarseGrainedSchedulerBackend的内部成员。

**Executor内幕原理和运行流程解密**

1. CoarseGrainedExecutorBackend通过onStart()发送RegisterExecutor向Driver注册。这里面的ref也就相当于Driver。

**override def** onStart() {  
 logInfo("Connecting to driver: " + driverUrl)  
 rpcEnv.asyncSetupEndpointRefByURI(driverUrl).flatMap { ref =>  
 // This is a very fast action so we can use "ThreadUtils.sameThread"  
 *driver* = *Some*(ref)  
 ref.ask[RegisterExecutorResponse](  
 *RegisterExecutor*(executorId, self, hostPort, cores, extractLogUrls))

RegisterExecutor此时的命名是有问题的，因为向Driver端注册并不是注册Executor，而是注册ExecutorBackend.

1. Driver要接受ExecutorBackend的注册，也就是接受CoarseGrainedExecutorBackend发来的请求。在DriverEndpoint中会接收到RegisterExecutor信息并完成在Driver的注册。

**override def** receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {  
  
 **case** *RegisterExecutor*(executorId, executorRef, hostPort, cores, logUrls) =>  
 **if** (*executorDataMap*.contains(executorId)) {  
 context.reply(*RegisterExecutorFailed*("Duplicate executor ID: " + executorId))

**接收到请求之后Driver是怎么办的？**

在Driver中通过ExecutorData封装并注册ExecutorBackend的信息到Driver的内存数据结构executorMapData中；

**private val** *executorDataMap* = **new** HashMap[String, ExecutorData]

**为啥是ExecutorBackend**？

**因为CoarseGrainedExecutorBackend是继承ThreadSafeRpcEndpoint，而Executor没有继承ThreadSafeRpcEndpoint，因此在Driver接收参数里面有RpcEndpoint，肯定是CoarseGrainedExecutorBackend发过来的。与Driver通信请求注册的是CoarseGrainedExecutorBackend，并不是Executor。**

**override def** receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {  
 **case** *RegisterExecutor*(executorId, executorRef, hostPort, cores, logUrls) =>  
 **if** (*executorDataMap*.contains(executorId)) {  
 context.reply(*RegisterExecutorFailed*("Duplicate executor ID: " + executorId)) //ID不可重复注册  
 } **else** {  
 // If the executor's rpc env is not listening for incoming connections, `hostPort`  
 // will be null, and the client connection should be used to contact the executor.  
 **val** executorAddress = **if** (executorRef.address != **null**) {  
 executorRef.address  
 } **else** {  
 context.senderAddress  
 }

实际在执行的时候DriverEndpoint会把信息吸入CoarseGrainedSchedulerBackend的内存数据结构executorMapData，所以说最终注册给了CoarseGrainedSchedulerBackend，也就是说，CoarseGrainedSchedulerBackend掌握了当前程序分配的所有的ExecutorBackend进程，而在每个ExecutorBackend进行实例中会通过Executor对象来负责具体Task的运行。在运行的时候使用synchronized关键字来保证executorMapData安全的并发写操作。

**为什么说注册信息其实给了**CoarseGrainedSchedulerBackend**？**

executorMapData属于CoarseGrainedSchedulerBackend的成员，在注册的时候，把信息注册给CoarseGrainedSchedulerBakend的成员的数据结构中。

**synchronized关键字**？

因为集群中会有很多ExecutorBackend向Driver注册，为了避免写冲突，所以用同步代码块。

// This must be synchronized because variables mutated  
// in this block are read when requesting executors  
CoarseGrainedSchedulerBackend.**this**.synchronized {  
 *executorDataMap*.put(executorId, data)  
 **if** (*numPendingExecutors* > 0) {  
 *numPendingExecutors* -= 1  
 logDebug(s"Decremented number of pending executors (**$***numPendingExecutors* left)")  
 }  
}

1. Driver会在Executor注册成功后，返回RegisteredExecutor信息给

CoarseGrainedExecutorBackend

// Note: some tests expect the reply to come after we put the executor in the map  
 context.reply(*RegisteredExecutor*(executorAddress.host))  
 *listenerBus*.post(  
 *SparkListenerExecutorAdded*(System.*currentTimeMillis*(), executorId, data))  
 makeOffers()  
}

1. CoarseGrainedExecutorBackend收到DriverEndpoint发送过来的RegisteredExecutor消息后会启动Executor实例对象，而Executor实例对象是事实上负责真正Task计算的；

**override def** receive: PartialFunction[Any, Unit] = {  
 **case** *RegisteredExecutor*(hostname) =>  
 logInfo("Successfully registered with driver")  
 *executor* = **new** Executor(executorId, hostname, env, userClassPath, isLocal = **false**)

其在实例化的时候会实例化一个线程池来准备Task的计算。

// Start worker thread pool  
**private val** *threadPool* = ThreadUtils.*newDaemonCachedThreadPool*("Executor task launch worker")  
**private val** *executorSource* = **new** ExecutorSource(*threadPool*, executorId)

1. 创建的threadPool中以多线程并发执行和线程复用的方式来高效的执行Spark发过来的Task。

public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory) {  
 return new ThreadPoolExecutor(0, Integer.*MAX\_VALUE*,  
 60L, TimeUnit.*SECONDS*,  
 new SynchronousQueue<Runnable>(),  
 threadFactory);  
}

**现在线程池准备好了，那么就等任务了。**

1. **当Driver发送过来Task的时候，其实是发送给了CoarseGrainedExecutorBackend这个RpcEndpoint,而不是直接发送给了Executor(Executor由于不是消息消息循环体，所以永远也无法直接接受远程发过来的信息*)；attemptNumber executor可以重试的次数***

**case** *LaunchTask*(data) =>  
 **if** (*executor* == **null**) {  
 logError("Received LaunchTask command but executor was null")  
 System.*exit*(1)  
 } **else** {  
 **val** taskDesc = *ser*.deserialize[TaskDescription](data.value) //反序列化  
 logInfo("Got assigned task " + taskDesc.taskId)  
 *executor*.launchTask(**this**, taskId = taskDesc.taskId, attemptNumber = taskDesc.attemptNumber,  
 taskDesc.name, taskDesc.serializedTask)  
 }

ExecutorBackend在收到Driver中发送过来的消息后会提供调用LaunchTask来交给Executor去执行；

LaunchTask来执行任务，接收到Task执行的命令之后，会首先把Task封装在TaskRunner里面，然后交给线程池中的线程处理。

**def** launchTask(  
 context: ExecutorBackend,  
 taskId: Long,  
 attemptNumber: Int,  
 taskName: String,  
 serializedTask: ByteBuffer): Unit = {  
 **val** tr = **new** TaskRunner(context, taskId = taskId, attemptNumber = attemptNumber, taskName,

1. TaskRunner是Java中的Runnable的接口的具体实现，在真正工作的时候会交给线程池中的线程去运行，此时会调用Run方法来执行Task。

**class** TaskRunner(  
 execBackend: ExecutorBackend,  
 **val** taskId: Long,  
 **val** attemptNumber: Int,  
 taskName: String,  
 serializedTask: ByteBuffer)  
 **extends** Runnable {

1. TaskRunner在调用run方法的时候会调用Task的run方法，而Task的run方法会调用runTask。

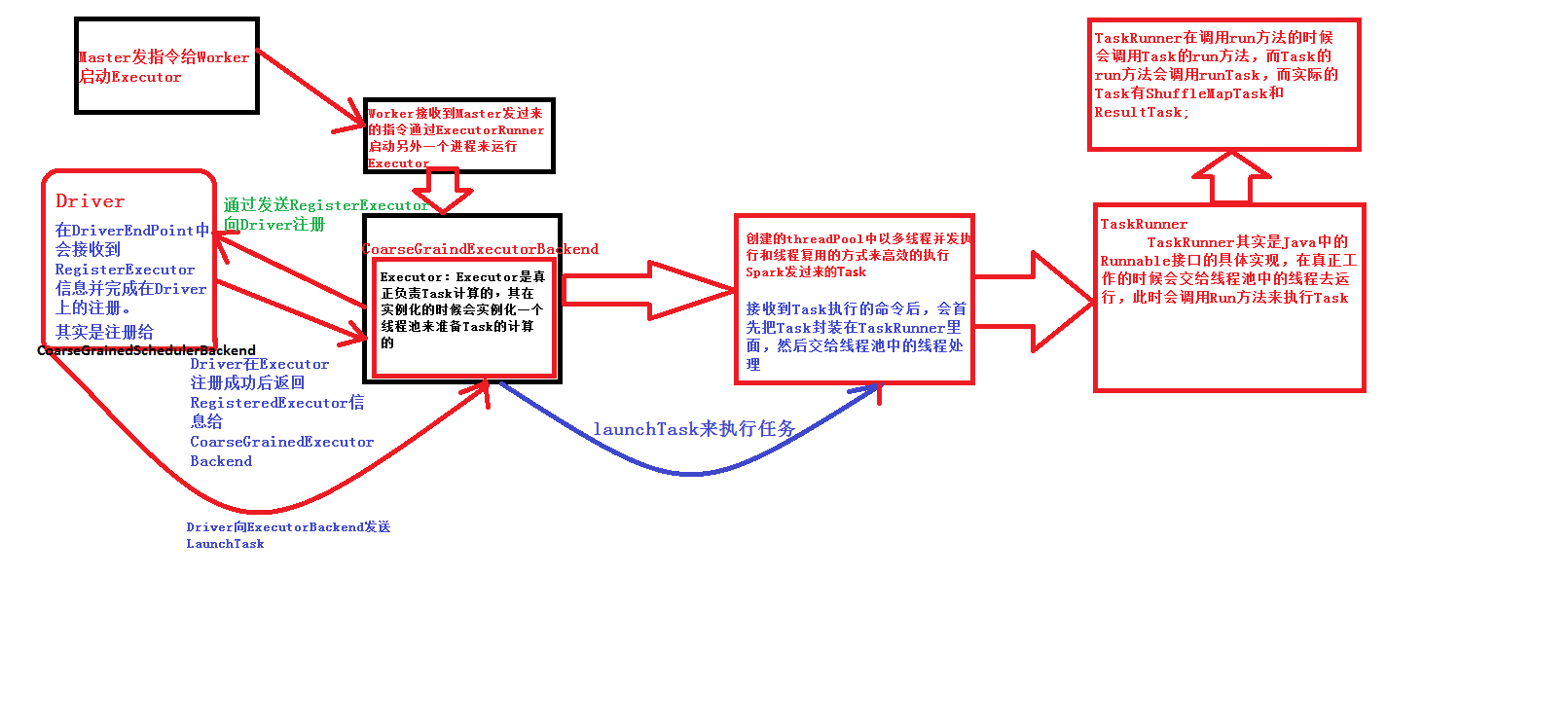
**val** res = *task*.run(  
 taskAttemptId = taskId,  
 attemptNumber = attemptNumber,  
 metricsSystem = env.metricsSystem)  
threwException = **false**res

Task的run方法：

**final def** run(  
 taskAttemptId: Long,  
 attemptNumber: Int,  
 metricsSystem: MetricsSystem)

**try** {  
 (runTask(*context*), *context*.collectAccumulators())

**总结流程图如下：**

****