简要：

本篇博文主要讨论的内容如下;

1. Taskscheduler工作原理
2. Taskscheduler源码解密

前置知识：

一：TaskScheduler原理解密

1. DAGScheduler在提交TaskSet给底层调度器的时候是面向接口TaskScheduler的，这符合面向对象中依赖抽象而不依赖具体的原则。带来底层资源调度器的可插拔性，导致Spark可以运行在众多的资源调度器上，例如Standalone,Yarn,Mesos,Local,EC2,其他自定义的资源调度器。在standalone的模式下我们聚焦于TaskSchedulerImpl.
2. 在SparkContext实例化的时候通过createTaskScheduler来创建TaskSchedulerImpl和SparkDeploySchedulerBackend。

**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)

在TaskSchedulerImpl的initialize方法中把SparkDeploySchedulerBackend传进来从而赋值给TaskSchedulerImpl的backend：在TaskSchedulerImpl调用start方法的时候会调用backend.start方法，在start方法中会最终注册应用程序。

1. TaskScheduler的核心任务是提交Taskset到集群运算并汇报结构。
2. 为TaskSet创建和维护一个TaskSetManager并追踪任务的本地性以及错误信息。
3. 遇到Straggle任务会放到其他节点进行重试。
4. 向DAGScheduler汇报执行情况，包括在Shuffle输出lost的时候报告fetch failed错误等信息。
5. TaskScheduler内部会握有SchedulerBackend，从Standalone的模式来讲具体实现是SparkDeploySchedulerBackend.

SparkDeploySchedulerBackend的核心功能？

1. 向Master注册当前应用程序。
3. SparkDeploySchedulerBackend在启动的时候构造了AppClient实例并在该实例start的时候启动了ClientEndpoint这个消息循环体，ClientEndpoint在启动的时候会向Master注册当前程序。而SparkDeploySchedulerBackend的父类CoarseGrainedSchedulerBackend在start的时候会实例化类型为DriverEndPoint(这就是我们程序运行时候的经典的对象Driver)的消息循环体，SparkDeploySchedulerBackend专门负责收集Worker上的资源信息的，当ExecutorBackend启动的时候会发送RegisteredExecutor信息向DriverEndpoint注册,此时SparkDeploySchedulerBackend就掌握了当前应用程序拥有的计算资源，TaskScheduler就是通过SparkDeploySchedulerBackend拥有的计算资源来具体运行Task的。
4. SparkContext,DAGScheduler, TaskSchedulerImpl,SparkDeploySchedulerBackend在应用程序启动的时候只实例化一次，应用程序存在期间始终存在这些对象.

**大总结：**

**在SparkContext实例化的时候，调用createTaskScheduler来创建TaskSchedulerImpl和SparkDeploySchedulerBackend,同时在SparkContext实例化的时候会调用TaskSchedulerImpl的start，在start方法中会调用SparkDeploySchedulerBackend的start，在该start方法中会创建AppClient对象并调用AppClient对象的start方法，在该start方法中会创建ClientEndpoint,在创建ClientEndpoint会传入Command来指定具体为当前应用程序启动的Executor进程的入口类的名称CoarseGrainedExecutorBackend，然后ClientEndpoint启动并通过tryRegisterMaster来注册当前的应用程序到Master中，Master接受到注册信息后如果可以运行程序，则会为该程序生成Job ID并通过schedule来分配计算资源，具体计算资源的分配是通过应用程序的运行方式，Memory,cores等配置信息来决定的，最后Master会发送指令给Worker，Worker中为当前应用程序分配计算资源时会首先分配ExecutorRunner，ExecutorRunner内部会通过Thread的方式构建ProcessBuilder来启动另外一个JVM进程，这个JVM进程启动时加载的main方法所在的类的名称，就是在创建ClientEndpoint时传入的Command来指定具体名称为CoarseGrainedExecutorBackend的类，此时JVM在通过ProcessBuilder启动的时候获得了CoarseGrainedExecutorBackend后，加载并调用其中的main方法。在main方法中会实例化CoarseGrainedExecutorBackend本身这个消息循环体，而CoarseGrainedExecutorBackend在实例化的时候会通过回调onStart向DriverEndpoint发送RegisterExecutor来注册当前的CoarseGrainedExecutorBackend，此时DriverEndpoint收到该注册信息并保持在了SparkDeploySchedulerBackend实例的内存数据结构中，这样Driver就获得了计算资源。**

**总流程源码解析：**

1. 在SparkContext实例化的时候，调用createTaskScheduler来创建TaskSchedulerImpl和SparkDeploySchedulerBackend。

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

1. 在TaskSchedulerImpl的initialize方法中把SparkDeploySchedulerBackend传进来从而赋值给TaskSchedulerImpl的backend：

scheduler.initialize(backend)  
 (backend, scheduler)

在TaskSchedulerImpl调用start方法的时候会调用backend.start方法，其实就是 SparkDeploySchedulerBackend的start方法。

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

1. SparkDeploySchedulerBackend的start方法被调用后，SparkDeploySchedulerBackend会创建AppClient实例，而AppClient的start方法又会被调用。

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

1. 先看一下command这个变量，它指定了具体为当前应用程序启动Executor进程的入口类为**CoarseGrainedExecutorBackend。**

**val** command = *Command*("org.apache.spark.executor.CoarseGrainedExecutorBackend",  
 args, sc.*executorEnvs*, classPathEntries ++ testingClassPath, libraryPathEntries, javaOpts)

AppClient的start方法被调用。

**val** appDesc = **new** ApplicationDescription(sc.appName, *maxCores*, sc.executorMemory,  
 command, appUIAddress, sc.eventLogDir, sc.eventLogCodec, coresPerExecutor)  
*client* = **new** AppClient(sc.env.rpcEnv, masters, appDesc, **this**, *conf*)  
*client*.start()

1. **ClientEndpoint实例就会被创建。**

**def** start() {  
 // Just launch an rpcEndpoint; it will call back into the listener.  
 *endpoint*.set(rpcEnv.setupEndpoint("AppClient", **new** ClientEndpoint(rpcEnv)))  
}

**我们接着看，ClientEndpoint类中具体发生了什么？**

registerWithMaster(1)向master注册当前的应用程序。

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

**Master是接收到ClientEndpoint注册请求之后做了什么呢？**

从下面的源码可以看到，在Master的receive方法中，则会为该程序生成Job ID并通过schedule来分配计算资源，具体计算资源的分配是通过应用程序的运行方式，Memory,cores等配置信息来决定的。

**case** *RegisterApplication*(description, driver) => {  
 // *TODO Prevent repeated registrations from some driver* **if** (*state* == RecoveryState.*STANDBY*) {  
 // ignore, don't send response  
 } **else** {  
 logInfo("Registering app " + description.name)  
 **val** app = createApplication(description, driver)  
 registerApplication(app)  
 logInfo("Registered app " + description.name + " with ID " + app.id)  
 *persistenceEngine*.addApplication(app)  
 driver.send(*RegisteredApplication*(app.id, self))  
 schedule()  
 }

**到此程序就就已经注册完成了，那么接下来就Master就要发指令给Worker了。**

1. **最后Master会发送指令给Worker，Worker中为当前应用程序分配计算资源时会首先分配ExecutorRunner，ExecutorRunner内部会通过Thread的方式构建ProcessBuilder来启动另外一个JVM进程，这个JVM进程启动时加载的main方法所在的类的名称，就是在创建ClientEndpoint时传入的Command来指定具体名称为CoarseGrainedExecutorBackend的类，此时JVM在通过ProcessBuilder启动的时候获得了CoarseGrainedExecutorBackend。**

**对上述语句进行源码详解：**

Worker中为当前应用程序分配计算资源时会首先分配ExecutorRunner

**val** manager = **new** ExecutorRunner(

启动ExecutorRunner的线程。

manager.start()

ExecutorRunner里面的start方法，调用fetchAndRunExecutor方法

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

然后buildProcessBuilder方法被调用。

**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")

buildProcessBuilder方法的返回类型是ProcessBuilder，并且在实际的方法里面创建了ProcessBuilder的实例。并且将command传入到了ProcessBuilder里面。

**def** buildProcessBuilder(  
 command: Command,  
 securityMgr: SecurityManager,  
 memory: Int,  
 sparkHome: String,  
 substituteArguments: String => String,  
 classPaths: Seq[String] = *Seq*[String](),  
 env: Map[String, String] = sys.*env*): ProcessBuilder = {  
 **val** localCommand = *buildLocalCommand*(  
 command, securityMgr, substituteArguments, classPaths, env)  
 **val** commandSeq = *buildCommandSeq*(localCommand, memory, sparkHome)  
 **val** builder = **new** ProcessBuilder(commandSeq: \_\*)

将command的消息复制给了this.command.

public ProcessBuilder(String... command) {  
 this.command = new ArrayList<>(command.length);  
 for (String arg : command)  
 this.command.add(arg);  
}

我们继续来看ProcessBuilder的start方法

此时将我们传入的command以数组的方式接收，数组名为cmdarray

public Process start() throws IOException {  
 // Must convert to array first -- a malicious user-supplied  
 // list might try to circumvent the security check.  
 String[] cmdarray = command.toArray(new String[command.size()]);  
 cmdarray = cmdarray.clone();

其中start方法里面返回ProcessImpl.start()

try {  
 return ProcessImpl.*start*(cmdarray,  
 environment,  
 dir,  
 redirects,  
 redirectErrorStream);

注释说的很明白，ProcessBuilder.start()也就是通过ProcessImpl.start()来创建一个JVM进程的。

因为将command的消息传入到了ProcessImpl.start()，并且，前面已经介绍过。

/\* This class is for the exclusive use of ProcessBuilder.start() to  
 \* create new processes.

final class ProcessImpl extends Process {  
 private static final sun.misc.JavaIOFileDescriptorAccess *fdAccess* = sun.misc.SharedSecrets.*getJavaIOFileDescriptorAccess*();

**对于上述讨论小结一下：**

**上述对ExecutorRunner内部如何通过Thread的方式构建ProcessBuilder来启动另外一个JVM进程进行了讨论，这个JVM进程启动的时候，运行CoarseGrainedExecutorBackend的main方法，注意这里的CoarseGrainedExecutorBackend是独立的一个进程。**

1. 在main方法中会实例化CoarseGrainedExecutorBackend本身这个消息循环体，而CoarseGrainedExecutorBackend在实例化的时候会通过回调onStart向DriverEndpoint发送RegisterExecutor来注册当前的CoarseGrainedExecutorBackend，此时DriverEndpoint收到该注册信息并保持在了SparkDeploySchedulerBackend实例的内存数据结构中，这样Driver就获得了计算资源。

**此步骤的详细说明在：**

**http://blog.csdn.net/snail\_gesture/article/details/50652938**