简介：

本篇博文主要讨论的内容如下：

1. Spark天堂之门
2. SparkContext使用案例鉴赏
3. SparkContext内幕
4. SparkContext源码解密

一：Spark天堂之门

1. Spark程序在运行的时候分为Driver和Executors两部分。
2. Spark的程序编写是基于SparkContext的，具体来说包含两方面。
   * 1. Spark编程的核心基础是RDD，是由SparkContext来最初创建(**第一个RDD一定是由SparkContext来创建的);**
     2. Spark程序的调度优化也是基于SparkContext;
3. **Spark程序的注册时通过SparkContext实例化时候生产的对象来完成的(其实是SchedularBackend来注册程序的)**
4. **Spark程序运行的时候要通过Cluster Manager获取具体的计算资源，计算资源的获取也是通过SparkContext产生的对象来申请的(其实是SchedularBackend来获取计算资源的)**

**总结：SparkContext开启了天堂之门：Spark程序是通过SparkContext发布到Spark集群的；**

**SparkContext导演天堂世界：Spark程序的运行都在SparkContext为核心 调度器的指挥下进行的。**

**SparkContext关闭天堂之门：SparkContext崩溃或者结束的时候整个Spark程序也结束了！！**

二：SparkContext案例鉴赏

三：SparkContext天堂内幕

1. SparkContex构建的顶级三大核心对象：DAGSchedular,TaskSchedular,ShedulerBackend,其中：
2. DAGScheduler是面向Job的Stage的高层调度器。
3. TaskScheduler是一个接口，根据具体的Cluster Manager的不同会有不同的实现，Standalone模式下具体的实现是TaskSchedulerImpl。
4. SchedulerBackend是一个接口，根据具体的Cluster Manager的不同会有不同的实现，Standalone模式下具体的实现是SparkDeploySchedulerBackend。作用是分配当前可用的资源，具体就是向当前等待分配计算资源的Task分配计算资源(即Executor)，并且在分配的Executor上启动Task，完成计算的调度过程。4

// Create and start the scheduler

val (sched, ts) = SparkContext.createTaskScheduler(this, master)

\_schedulerBackend = sched

\_taskScheduler = ts

\_dagScheduler = new DAGScheduler(this)

\_heartbeatReceiver.ask[Boolean](TaskSchedulerIsSet)

// start TaskScheduler after taskScheduler sets DAGScheduler reference in DAGScheduler's

// constructor

\_taskScheduler.start()

createTaskScheduler：

TaskSchedulerImpl是底层调度器的核心和灵魂，创建TaskSchedulerImpl的时候必须要给它创建一个schedulerbackend

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scheduler.initialize调用的时候会创建SchedulerPool

def initialize(backend: SchedulerBackend) {

this.backend = backend

// temporarily set rootPool name to empty

rootPool = new Pool("", schedulingMode, 0, 0)

schedulableBuilder = {

schedulingMode match {

case SchedulingMode.FIFO =>

new FIFOSchedulableBuilder(rootPool)

case SchedulingMode.FAIR =>

new FairSchedulableBuilder(rootPool, conf)

}

}

SparkDeploySchedulerBackend有三大核心功能：

负责与Master链接注册当前程序。

接收集群中为当前应用程序而分配的计算资源Executor的注册并管理Executors。

负责发送Task到具体的Executor执行。

补充说明的是：SparkDeploySchedulerBackend是被TaskSchedulerImpl来管理的！

// start TaskScheduler after taskScheduler sets DAGScheduler reference in DAGScheduler's

// constructor

\_taskScheduler.start()

**当通过SparkDeploySchedulerBackend向我们的集群注册的时候会把command传递给master ，让Worker去启动具体的进程资源的时候，具体进程的主类名称叫：**CoarseGrainedExecutorBackend

val command = Command("org.apache.spark.executor.CoarseGrainedExecutorBackend",

args, sc.executorEnvs, classPathEntries ++ testingClassPath, libraryPathEntries, javaOpts)

**当通过SparkDeploySchedulerBackend注册程序给Master的时候会把上述command提交给Master，Master发指令给Worker去启动Executor所在的进程的时候加载的main方法所在的入口类就是command中的CoarseGrainedExecutorBackend，当然你可以实现自己的ExecutorBackened，在CoarseGrainedExecutorBackend中启动Executor（Executor是先注册再实例化），Executor通过线程池并发执行Task。**

注册的时候是通过registerMasterThreadPool线程池中的一个线程去注册，注册给master，所有这里面偶master引用，send一个消息给master

for (masterAddress <- masterRpcAddresses) yield {

registerMasterThreadPool.submit(new Runnable {

override def run(): Unit = try {

if (registered.get) {

return

}

logInfo("Connecting to master " + masterAddress.toSparkURL + "...")

val masterRef =

rpcEnv.setupEndpointRef(Master.SYSTEM\_NAME, masterAddress, Master.ENDPOINT\_NAME)

masterRef.send(RegisterApplication(appDescription, self))

// AppClient to Master

case class RegisterApplication(appDescription: ApplicationDescription, driver: RpcEndpointRef)

private[spark] case class ApplicationDescription(

name: String,

maxCores: Option[Int],

memoryPerExecutorMB: Int,

command: Command,

appUiUrl: String,

eventLogDir: Option[URI] = None,

// short name of compression codec used when writing event logs, if any (e.g. lzf)

eventLogCodec: Option[String] = None,

coresPerExecutor: Option[Int] = None,

user: String = System.getProperty("user.name", "<unknown>")) {

override def toString: String = "ApplicationDescription(" + name + ")"

}

Master

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()

}

}