**第二十八课：Spark天堂之门解密**

**本期内容：**

    1、Spark天堂之门

    2、SparkContext使用案例借鉴

    3、SparkContext内幕

    4、SparkContext源码解密

**一、Spark天堂之门**

    1、Spark程序在执行的时候分为Driver和Executor两部分；

    2、Spark程序的编写核心基础是RDD，具体包含两部分：

        a）是由SparkContext来最初创建第一个RDD，一定是由SparkContext来创建的；

        b）Spark程序的调度优化也是基于SparkContext；

    3、Spark程序的注册要通过SparkContext实例化时候产生的对象来完成（其实是由SchedulerBackend注册程序，申请计算资源）；

    4、Spark程序运行的时候要通过ClusterManager获得具体的资源，计算资源的获取也是由SparkContext产生的对象来申请的（其实是 SchedulerBackend来获取资源的，SchedulerBackend是由SparkContext实例化的时候产生的，也就是说在构造SparkContext的时候产生）；

        从调度的层面讲调度优化也是基于SparkContext的，从程序注册的角度讲也是基于SparkContext，从程序获取计算资源的角度讲也是基于SparkContext获取计算资源，只要SparkContext关闭，程序也就结束运行。

    5、SparkContext崩溃或者结束的整个Spark程序也结束啦！

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

          SparkContext导演天堂世界：SPark程序的运行都是在SparkContext为核心的调度器下指挥下进行的；

          SparkContext关闭天堂之门：SparkContext崩溃或者结束整个Spark程序结束了；

**二、SparkContext内幕天堂揭秘**

    1、SparkContext构建的顶级三大核心对象：DAGScheduler、TaskScheduler、SchedulerBackend，其中：

   a）DAGScheduler是面向Stage的高层调度器；

   b）TaskScheduler是一个接口，更加具体的Cluster Manager的不同会有不同的实现，Standalone模式下具体的实现是**TaskSchedulerlmpl**；

   c）SchedulerBackend是一个接口，更加具体的Cluster Manager的不同会有不同的实现，Standalone模式下具体的实现是**SparkDeploySchedulerBackend**；

    2、从整个程序运行的角度来讲，SparkContext包含四大核心对象：DAGScheduler、TaskScheduler、SchedulerBackend、MapOutputTrackerMaster。

    SparkContext默认构造器在SparkContext实例化的时候需要对，构造器中的参数赋值。

// Create and start the scheduler

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

\_schedulerBackend = sched

\_taskScheduler = ts //task

\_dagScheduler = new DAGScheduler(this)

\_heartbeatReceiver.ask[Boolean](TaskSchedulerIsSet)

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

// constructor

\_taskScheduler.start()

    createTaskScheduler在SparkContext的默认构造器中，所以当实例化SparkContext的时候需要被调用，**调用的时候返回了SchedulerBackend和\_taskScheduler的实例**，基于该内容有创建了DAGScheduler，DAGScheduler管理TaskScheduler。

    createTaskScheduler中采用了Local模式LOCAL\_N\_REGEX、Local多线程模式、Local多线程模式重试、Cluster模式、Yarn模式、Mesos模式，默认情况下作业失败了就失败了，不会去重试：

/\*\*

\* Create a task scheduler based on a given master URL.

\* Return a 2-tuple of the scheduler backend and the task scheduler.

\* 在SparkContext实例化的时候被调用

\*/

private def createTaskScheduler(

sc: SparkContext,

master: String): (SchedulerBackend, TaskScheduler) = {

import SparkMasterRegex.\_

// When running locally, don't try to re-execute tasks on failure.

val MAX\_LOCAL\_TASK\_FAILURES = 1

    下面为**Standalone模式：**

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)

    从上述源码可以看出传进的sparkURL创建TaskSchedulerImpl对象，**TaskSchedulerImpl是底层调度器的核心**，在创建TaskSchedulerImpl后，以TaskSchedulerImpl为参数创建了一个SchedulerBackend对象，而SchedulerBackend进行initialize的时候将SchedulerBackend传入，具体传入的是SparkDeploySchedulerBackend，这里Scheduler的initialize方法如下：

//先进先出

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 => //先进先出的调度模式

new FIFOSchedulableBuilder(rootPool)

case SchedulingMode.FAIR =>

new FairSchedulableBuilder(rootPool, conf)

}

}

schedulableBuilder.buildPools()

}

    在Scheduler.initialize调用的时候会创建SchedulerPool调度池，一个任务有两种方式。

**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计算资源Executor的注册并管理Executor；**
* **负责发送Task到具体的Executor执行；**

**注：把Task发送给Executor是通过SchedulerBackend来完成的**

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

    SparkContext的下一步进行TaskSchedulerImpl.start()，然后导致SparkDeploySchedulerBackend.start()，

override def start() {

backend.start()//实质上是SchedulerBackend.start();

if (!isLocal && conf.getBoolean("spark.speculation", false)) {

logInfo("Starting speculative execution thread")

speculationScheduler.scheduleAtFixedRate(new Runnable {

override def run(): Unit = Utils.tryOrStopSparkContext(sc) {

checkSpeculatableTasks()

}

}, SPECULATION\_INTERVAL\_MS, SPECULATION\_INTERVAL\_MS, TimeUnit.MILLISECONDS)

}

}

        start方法中主要是Backend.start()，实质是SchedulerBackend.start()。

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main（Executor进程所在的入口类）方法的所在的入口类就是command中的CoarseGrainedExecutorBackend，**当然这里可以实现自己的ExecutorBackend**，在CoarseGrainedExecutorBackend中启动Executor（Executor先注册），Executor通过线程池并发执行的方式执行Task。

      下面是CoaseGrainedExecutorBackend中的run()方法，其中有driverUrl、executorId、hostname、cores、APPId等参数：

val executorConf = new SparkConf

val port = executorConf.getInt("spark.executor.port", 0) //获得executor的port

val fetcher = RpcEnv.create(

"driverPropsFetcher",

hostname,

port,

executorConf,

new SecurityManager(executorConf), //传入SparkConf的实例

clientMode = true)

val driver = fetcher.setupEndpointRefByURI(driverUrl)

val props = driver.askWithRetry[Seq[(String, String)]](RetrieveSparkProps) ++

Seq[(String, String)](("spark.app.id", appId))

fetcher.shutdown()

      下面代码中使用从driver上发送的属性创建了SparkEnv，然后以键值对的方式循环遍历从driver传入的参数，通过SparkConf设置该属性（Standalone模式），然后通过SparkEnv调用createExecutorEnv，主要实现了对executor的port设置，创建了CoaseGrainedExecutorBackend（new 出该main方法的对象实例）的对象进行RPC通信。

val env = SparkEnv.createExecutorEnv(

driverConf, executorId, hostname, port, cores, isLocal = false)

// SparkEnv will set spark.executor.port if the rpc env is listening for incoming

// connections (e.g., if it's using akka). Otherwise, the executor is running in

// client mode only, and does not accept incoming connections.

val sparkHostPort = env.conf.getOption("spark.executor.port").map { port =>

hostname + ":" + port

}.orNull

env.rpcEnv.setupEndpoint("Executor", new CoarseGrainedExecutorBackend(

env.rpcEnv, driverUrl, executorId, sparkHostPort, cores, userClassPath, env))

      启动CoaseGrainedExecutorBackend的时候需要注册Executor，当收到driver级别的Executor注册信息后，才会实例化Executor的对象，只有当Executor注册成功的时候才会实例化。

override def receive: PartialFunction[Any, Unit] = {

case RegisteredExecutor(hostname) =>

logInfo("Successfully registered with driver")

executor = new Executor(executorId, hostname, env, userClassPath, isLocal = false)

      当TaskSchedulerImpl调用start方法后导致SparkDeploySchedulerBackend调用start方法，而Executor要向SparkDeploySchedulerBackend注册，在SparkDeploySchedulerBackend的start方法中实例化了APPClient对象：

val coresPerExecutor = conf.getOption("spark.executor.cores").map(\_.toInt)

//每个Executor中使用多少个cores可以在Spark.env中配置

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()

launcherBackend.setState(SparkAppHandle.State.SUBMITTED)

waitForRegistration()

launcherBackend.setState(SparkAppHandle.State.RUNNING)

        上面new除了AppClient的实例对象，AppClient是一个接口，可以和集群进行通信，其参数包含了masterURL和应用程序信息（因为要注册应用程序给集群），并且是集群时间的监听器。

/\*\*

\* Interface allowing applications to speak with a Spark deploy cluster. Takes a master URL,

\* an app description, and a listener for cluster events, and calls back the listener when various

\* events occur.

\*

\* @param masterUrls Each url should look like spark://host:port.

\*/

private[spark] class AppClient(

rpcEnv: RpcEnv,

masterUrls: Array[String],

appDescription: ApplicationDescription,

listener: AppClientListener,

conf: SparkConf)

extends Logging {

      在AppClient中有很重要的类ClientEndpoint，在ClientEndpoint实例化启动的start中主要是registerWithMaster：

override def onStart(): Unit = {

try {

registerWithMaster(1)

} catch {

case e: Exception =>

logWarning("Failed to connect to master", e)

markDisconnected()

stop()

}

}

在registerWithMaster中进行注册：

private def registerWithMaster(nthRetry: Int) {

registerMasterFutures.set(tryRegisterAllMasters())

registrationRetryTimer.set(registrationRetryThread.scheduleAtFixedRate(new Runnable {

其内部使用tryRegisterAllMasters进行注册：

/\*\*

\* Register with all masters asynchronously and returns an array `Future`s for cancellation.

\*/

private def tryRegisterAllMasters(): Array[JFuture[\_]] = {

for (masterAddress <- masterRpcAddresses) yield {

registerMasterThreadPool.submit(new Runnable {

 tryRegisterAllMasters注册Application的时候是通过Thread完成的。面继续跟进代码：

masterRef.send(RegisterApplication(appDescription, self))

进行跟进查看注册Application的内容：

private[spark] case class ApplicationDescription(

name: String,

maxCores: Option[Int],

memoryPerExecutorMB: Int,

command: Command,

appUiUrl: String, //在web控制台显示的Application信息

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>")) {

      上面我们看到了tryRegisterAllMasters中是通过Thread的方式向Master注册的，那么我们在Master中可以看到发送的具体注册Application的信息。

我们在Master的receive方法中看到了通过模式匹配的方式包括了RegisterApplication，接受注册，分配资源。

**三、总结**

      上面我们通过源码的方式详细剖析了提交应用程序后，通过driver向Master注册Application的过程，当注册完成Application后Master发送指令给Worker启动Executor，并向driver中的SparkDeploySchedulerBackend注册Executor的信息。

      SparkContext中还包含了TaskScheduler面向Stage的高层调度器，将action触发的job中的rdd划分为若干个Stage。

      SparkContext中也包括SparkUI，背后是Jetty服务，支持通过web的方式访问程序的状态。