Spark源码分析之Worker

Spark支持三种模式的部署：YARN、Standalone以及Mesos。本篇说到的Worker只有在Standalone模式下才有。Worker节点是Spark的工作节点，用于执行提交的作业。我们先从Worker节点的启动开始介绍。

Spark中Worker的启动有多种方式，但是最终调用的都是org.apache.spark.deploy.worker.Worker类，启动Worker节点的时候可以传很多的参数：内存、核、工作目录等。如果你不知道如何传递，没关系，help一下即可：

```
[wyp@iteblog spark]$ ./bin/spark-class org.apache.spark.deploy.worker.Worker -h
Spark assembly has been built with Hive, including Datanucleus jars on classpath
Usage: Worker [options] <master>

Master must be a URL of the form spark://hostname:port

Options:
  -c CORES, --cores CORES  Number of cores to use
  -m MEM, --memory MEM     Amount of memory to use (e.g. 1000M, 2G)
  -d DIR, --work-dir DIR   Directory to run apps in (default: SPARK_HOME/work)
  -i HOST, --ip IP         Hostname to listen on (deprecated, please use --host or -h)
  -h HOST, --host HOST     Hostname to listen on
  -p PORT, --port PORT     Port to listen on (default: random)
  --webui-port PORT        Port for web UI (default: 8081)
```

从上面的输出我们可以看出Worker的启动支持多达7个参数！这样每个都这样输入岂不是很麻烦？其实，我们不用担心，Worker节点启动地时候将先读取conf/spark-env.sh里面的配置，这些参数配置的解析都是由Worker中的WorkerArguments类进行解析的。如果你没有设置内存，那么将会把Worker启动所在机器的所有内存（会预先留下1G内存给操作系统）分给Worker，具体的代码实现如下：

```
def inferDefaultMemory(): Int = {
  val ibmVendor = System.getProperty("java.vendor").contains("IBM")
  var totalMb = 0
  try {
    val bean = ManagementFactory.getOperatingSystemMXBean()
    if (ibmVendor) {
      val beanClass = Class.forName("com.ibm.lang.management.OperatingSystemMXBean")
      val method = beanClass.getDeclaredMethod("getTotalPhysicalMemory")
      totalMb = (method.invoke(bean).asInstanceOf[Long] / 1024 / 1024).toInt
    } else {
```

```
      val beanClass = Class.forName("com.sun.management.OperatingSystemMXBean")
      val method = beanClass.getDeclaredMethod("getTotalPhysicalMemorySize")
      totalMb = (method.invoke(bean).asInstanceOf[Long] / 1024 / 1024).toInt
    }
  } catch {
    case e: Exception => {
      totalMb = 2*1024
      System.out.println("Failed to get total physical memory. Using " + totalMb + " MB")
    }
  }
  // Leave out 1 GB for the operating system, but don't return a negative memory size
  math.max(totalMb - 1024, 512)
}
```

同样，如果你没设置cores，那么Spark将会获取你机器的所有可用的核作为参数传进去。解析完参数之后，将运行preStart函数，进行一些启动相关的操作，比如判断是否已经向Master注册过，创建工作目录，启动Worker的WEB UI，向Master进行注册等操作，如下：

```
override def preStart() {
  assert(!registered)
  logInfo("Starting Spark worker %s:%d with %d cores, %s RAM".format(
    host, port, cores, Utils.megabytesToString(memory)))
  logInfo("Spark home: " + sparkHome)
  createWorkDir()
  context.system.eventStream.subscribe(self, classOf[RemotingLifecycleEvent])
  webUi = new WorkerWebUI(this, workDir, Some(webUiPort))
  webUi.bind()
  registerWithMaster()

  metricsSystem.registerSource(workerSource)
  metricsSystem.start()
}
```

Worker向Master注册的超时时间为20秒，如果在这20秒内没有成功地向Master注册，那么将会进行重试，重试的次数为3，如过重试的次数大于等于3，那么将无法启动Worker，这时候，你就该看看你的网络环境或者你的Master是否存在问题了。
Worker在运行的过程中将会触发许多的事件，比如：RegisteredWorker、SendHeartbeat、WorkDirCleanup以及MasterChanged等等，收到不同的事件，Worker进行不同的操作。比如，如果需要运行一个作业，Worker将会启动一个或多个ExecutorRunner，具体的代码可参见receiveWithLogging函数：

```
override def receiveWithLogging = {
  case RegisteredWorker(masterUrl, masterWebUiUrl) =>

  case SendHeartbeat =>
  case WorkDirCleanup =>

  case MasterChanged(masterUrl, masterWebUiUrl) =>

  case Heartbeat =>

  case RegisterWorkerFailed(message) =>

  case LaunchExecutor(masterUrl, appId, execId, appDesc, cores_, memory_) =>

  case ExecutorStateChanged(appId, execId, state, message, exitStatus) =>

  case KillExecutor(masterUrl, appId, execId) =>

  case LaunchDriver(driverId, driverDesc) => {


  case KillDriver(driverId) => {

  case DriverStateChanged(driverId, state, exception) => {


  case x: DisassociatedEvent if x.remoteAddress == masterAddress =>

  case RequestWorkerState => {
}
```

上面的代码是经过处理的，其实receiveWithLogging 方法是从ActorLogReceive继承下来的。
当Worker节点Stop的时候，将会执行postStop函数，如下：

```
override def postStop() {
  metricsSystem.report()
  registrationRetryTimer.foreach(_.cancel())
  executors.values.foreach(_.kill())
  drivers.values.foreach(_.kill())
  webUi.stop()
  metricsSystem.stop()
}
```

杀掉所有还未执行完的executors、drivers等，操作。这方法也是从Actor继承下来的。

本文只是简单地介绍了Worker节点的一些环境，启动等相关的代码，关于它如何和Master通信；如何启动Executor；如何启动Driver都没有涉及，如果你想更好地了解Worker的运行情况，请参见Worker相关的代码吧。