【转】Spark源码分析之-deploy模块

**原文地址：http://jerryshao.me/architecture/2013/04/30/Spark%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90%E4%B9%8B-deploy%E6%A8%A1%E5%9D%97/**

### Background

在前文Spark源码分析之-scheduler模块中提到了Spark在资源管理和调度上采用了Hadoop YARN的方式：外层的资源管理器和应用内的任务调度器；并且分析了Spark应用内的任务调度模块。本文就Spark的外层资源管理器-deploy模块进行分析，探究Spark是如何协调应用之间的资源调度和管理的。

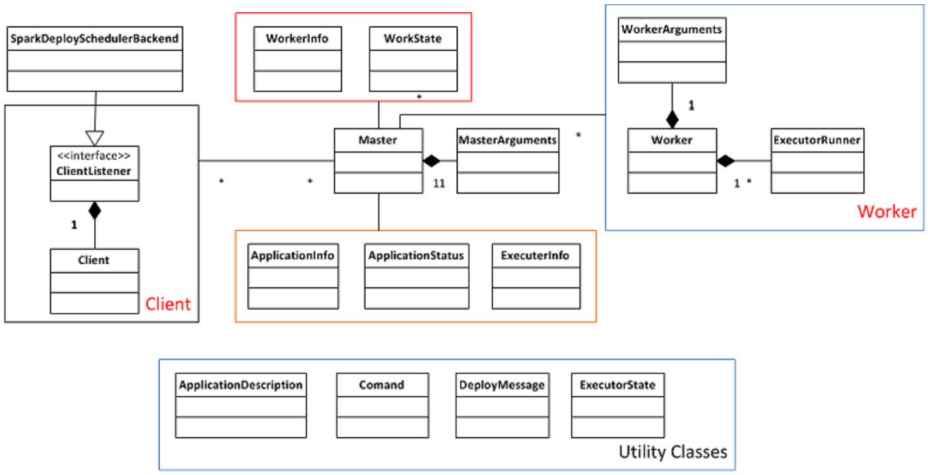
Spark最初是交由Mesos进行资源管理，为了使得更多的用户，包括没有接触过Mesos的用户使用Spark，Spark的开发者添加了Standalone的部署方式，也就是deploy模块。因此deploy模块只针对不使用Mesos进行资源管理的部署方式。

# Deploy模块整体架构

**deploy**模块主要包含3个子模块：**master**, **worker**, **client**。他们继承于Actor，通过actor实现互相之间的通信。

- **Master**：master的主要功能是接收worker的注册并管理所有的worker，接收client提交的application，(FIFO)调度等待的application并向worker提交。
- **Worker**：worker的主要功能是向master注册自己，根据master发送的application配置进程环境，并启动StandaloneExecutorBackend。
- **Client**：client的主要功能是向master注册并监控application。当用户创建SparkContext时会实例化SparkDeploySchedulerBackend，而实例化SparkDeploySchedulerBackend的同时就会启动client，通过向client传递启动参数和application有关信息，client向master发送请求注册application并且在slave node上启动StandaloneExecutorBackend。

下面来看一下deploy模块的类图：




# Deploy模块通信消息

Deploy模块并不复杂，代码也不多，主要集中在各个子模块之间的消息传递和处理上，因此在这里列出了各个模块之间传递的主要消息：

- **client** to **master**
  1. RegisterApplication (向master注册application)

昵称：vincent_hv
园龄：10个月
粉丝：7
关注：1
+加关注

< 2013年10月 >

| 日 | 一 | 二 | 三 | 四 | 五 | 六 |
|---|---|---|---|---|---|---|
| 29 | 30 | 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 | 25 | 26 |
| 27 | 28 | 29 | 30 | 31 | 1 | 2 |
| 3 | 4 | 5 | 6 | 7 | 8 | 9 |

搜索
找找看
谷歌搜索

常用链接
我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

最新随笔
1. linux解压zip乱码解决方案
2. 全能系统监控工具dstat
3. 【转】linux sar命令详解
4. 【原】gnome3增加自定义程序快捷方式
5. 【原】Ubuntu13.04安装、卸载Gnome3.8
6. 【原】安装、卸载、查看软件时常用的命令
7. 【原】中文Ubuntu主目录下的文档文件夹改回英文
8. 【原】Ubuntu ATI/Intel双显卡 驱动安装
9. 【原】Ubuntu 12.04 ATI显卡设置双屏显示
10. 【转】Hadoop vs Spark性能对比

随笔分类
Android(8)
Hadoop(2)
Java(20)
JVM(3)
Linux(23)
others(1)
Scala(5)
Spark(20)
数据结构与算法(2)

- **master** to **client**
    1. RegisteredApplication (作为注册application的reply，回复给client)
    2. ExecutorAdded (通知client worker已经启动了Executor环境，当向worker发送LaunchExecutor后通知client)
    3. ExecutorUpdated (通知client Executor状态已经发生变化了，包括结束、异常退出等，当worker向master发送ExecutorStateChanged后通知client)
- **master** to **worker**
    1. LaunchExecutor (发送消息启动Executor环境)
    2. RegisteredWorker (作为worker向master注册的reply)
    3. RegisterWorkerFailed (作为worker向master注册失败的reply)
    4. KillExecutor (发送给worker请求停止executor环境)
- **worker** to **master**
    1. RegisterWorker (向master注册自己)
    2. Heartbeat (定期向master发送心跳信息)
    3. ExecutorStateChanged (向master发送Executor状态改变信息)

# Deploy模块代码详解

Deploy模块相比于scheduler模块简单，因此对于deploy模块的代码并不做十分细节的分析，只针对application的提交和结束过程做一定的分析。

## Client提交application

Client是由SparkDeploySchedulerBackend创建被启动的，因此client是被嵌入在每一个application中，只为这个applicator所服务，在client启动时首先会先master注册application：

```
def start() {
  // Just launch an actor; it will call back into the listener.
  actor = actorSystem.actorOf(Props(new ClientActor))
}
override def preStart() {
  logInfo("Connecting to master " + masterUrl)
  try {
    master = context.actorFor(Master.toAkkaUrl(masterUrl))
    masterAddress = master.path.address
    master ! RegisterApplication(appDescription) //向master注册application
    context.system.eventStream.subscribe(self, classOf[RemoteClientLifeCycleEvent])
    context.watch(master)  // Doesn't work with remote actors, but useful for testing
  } catch {
    case e: Exception =>
      logError("Failed to connect to master", e)
      markDisconnected()
      context.stop(self)
  }
}
```

Master在收到RegisterApplication请求后会把application加到等待队列中，等待调度：

```
case RegisterApplication(description) => {
  logInfo("Registering app " + description.name)
  val app = addApplication(description, sender)
  logInfo("Registered app " + description.name + " with ID " + app.id)
  waitingApps += app
  context.watch(sender)  // This doesn't work with remote actors but helps for testing
  sender ! RegisteredApplication(app.id)
  schedule()
}
```

Master会在每次操作后调用schedule()函数，以确保等待的application能够被及时调度。

在前面提到deploy模块是资源管理模块，那么Spark的deploy管理的是什么资源，资源以什么单位进行调度的呢？在当前版本的Spark中，集群的cpu数量是Spark资源管理的一个标准，每个提交的application都会标明自己所需要的资源数(也就是cpu的core数)，Master以FIFO的方式管理所有的application请求，当资源数量满足当前任务执行需求的时候该任务就会被调度，否则就继续等待，当然如果master能给予当前任务部分资源则也会启动该application。schedule()函数实现的就是此功能。

```
def schedule() {
  if (spreadOutApps) {
    for (app <- waitingApps if app.coresLeft > 0) {
      val usableWorkers = workers.toArray.filter(_.state == WorkerState.ALIVE)
                                 .filter(canUse(app, _)).sortBy(_.coresFree).reverse
      val numUsable = usableWorkers.length
      val assigned = new Array[Int](numUsable) // Number of cores to give on each node
      var toAssign = math.min(app.coresLeft, usableWorkers.map(_.coresFree).sum)
      var pos = 0
      while (toAssign > 0) {
        if (usableWorkers(pos).coresFree - assigned(pos) > 0) {
          toAssign -= 1
          assigned(pos) += 1
        }
        pos = (pos + 1) % numUsable
      }
      // Now that we've decided how many cores to give on each node, let's actually give
them
      for (pos <- 0 until numUsable) {
        if (assigned(pos) > 0) {
          val exec = app.addExecutor(usableWorkers(pos), assigned(pos))
          launchExecutor(usableWorkers(pos), exec, app.desc.sparkHome)
          app.state = ApplicationState.RUNNING
        }
      }
    }
  } else {
    // Pack each app into as few nodes as possible until we've assigned all its cores
    for (worker <- workers if worker.coresFree > 0 && worker.state == WorkerState.ALIVE)
{
      for (app <- waitingApps if app.coresLeft > 0) {
        if (canUse(app, worker)) {
          val coresToUse = math.min(worker.coresFree, app.coresLeft)
          if (coresToUse > 0) {
            val exec = app.addExecutor(worker, coresToUse)
            launchExecutor(worker, exec, app.desc.sparkHome)
            app.state = ApplicationState.RUNNING
          }
        }
      }
    }
  }
}
```

当application得到调度后就会调用launchExecutor()向worker发送请求，同时向client汇报状态：

```
def launchExecutor(worker: WorkerInfo, exec: ExecutorInfo, sparkHome: String) {
  worker.addExecutor(exec)
  worker.actor ! LaunchExecutor(exec.application.id, exec.id, exec.application.desc, exec
.cores, exec.memory, sparkHome)
  exec.application.driver ! ExecutorAdded(exec.id, worker.id, worker.host, exec.cores, ex
ec.memory)
}
```

至此client与master的交互已经转向了master与worker的交互，worker需要配置application启动环境

```
case LaunchExecutor(appId, execId, appDesc, cores_, memory_, execSparkHome_) =>
  val manager = new ExecutorRunner(
    appId, execId, appDesc, cores_, memory_, self, workerId, ip, new File(execSparkHome_)
, workDir)
  executors(appId + "/" + execId) = manager
  manager.start()
```

```
  coresUsed += cores_
  memoryUsed += memory_
  master ! ExecutorStateChanged(appId, execId, ExecutorState.RUNNING, None, None)
```

Worker在接收到LaunchExecutor消息后创建ExecutorRunner实例，同时汇报master executor环境启动。

ExecutorRunner在启动的过程中会创建线程，配置环境，启动新进程：

```
def start() {
  workerThread = new Thread("ExecutorRunner for " + fullId) {
    override def run() { fetchAndRunExecutor() }
  }
  workerThread.start()
  // Shutdown hook that kills actors on shutdown.
  ...
}
def fetchAndRunExecutor() {
  try {
    // Create the executor's working directory
    val executorDir = new File(workDir, appId + "/" + execId)
    if (!executorDir.mkdirs()) {
      throw new IOException("Failed to create directory " + executorDir)
    }
    // Launch the process
    val command = buildCommandSeq()
    val builder = new ProcessBuilder(command: _*).directory(executorDir)
    val env = builder.environment()
    for ((key, value) <- appDesc.command.environment) {
      env.put(key, value)
    }
    env.put("SPARK_MEM", memory.toString + "m")
    // In case we are running this from within the Spark Shell, avoid creating a "scala"
    // parent process for the executor command
    env.put("SPARK_LAUNCH_WITH_SCALA", "0")
    process = builder.start()
    // Redirect its stdout and stderr to files
    redirectStream(process.getInputStream, new File(executorDir, "stdout"))
    redirectStream(process.getErrorStream, new File(executorDir, "stderr"))
    // Wait for it to exit; this is actually a bad thing if it happens, because we expect
 to run
    // long-lived processes only. However, in the future, we might restart the executor a
 few
    // times on the same machine.
    val exitCode = process.waitFor()
    val message = "Command exited with code " + exitCode
    worker ! ExecutorStateChanged(appId, execId, ExecutorState.FAILED, Some(message),
                                  Some(exitCode))
  } catch {
    case interrupted: InterruptedException =>
      logInfo("Runner thread for executor " + fullId + " interrupted")
    case e: Exception => {
      logError("Error running executor", e)
      if (process != null) {
        process.destroy()
      }
      val message = e.getClass + ": " + e.getMessage
      worker ! ExecutorStateChanged(appId, execId, ExecutorState.FAILED, Some(message), N
one)
    }
  }
}
```

在ExecutorRunner启动后worker向master汇报ExecutorStateChanged，而master则将消息重新pack成为ExecutorUpdated发送给client。

至此整个application提交过程基本结束，提交的过程并不复杂，主要涉及到的消息的传递。

## Application的结束

由于各种原因(包括正常结束，异常返回等)会造成application的结束，我们现在就来看看applicatoin结束的整个

流程。

application的结束往往会造成client的结束，而client的结束会被master通过Actor检测到，master检测到后会调用removeApplication()函数进行操作：

```
def removeApplication(app: ApplicationInfo) {
  if (apps.contains(app)) {
    logInfo("Removing app " + app.id)
    apps -= app
    idToApp -= app.id
    actorToApp -= app.driver
    addressToWorker -= app.driver.path.address
    completedApps += app   // Remember it in our history
    waitingApps -= app
    for (exec <- app.executors.values) {
      exec.worker.removeExecutor(exec)
      exec.worker.actor ! KillExecutor(exec.application.id, exec.id)
    }
    app.markFinished(ApplicationState.FINISHED)  // TODO: Mark it as FAILED if it failed
    schedule()
  }
}
```

removeApplicatoin()首先会将application从master自身所管理的数据结构中删除，其次它会通知每一个work，请求其KillExecutor。worker在收到KillExecutor后调用ExecutorRunner的kill()函数：

```
case KillExecutor(appId, execId) =>
  val fullId = appId + "/" + execId
  executors.get(fullId) match {
    case Some(executor) =>
      logInfo("Asked to kill executor " + fullId)
      executor.kill()
    case None =>
      logInfo("Asked to kill unknown executor " + fullId)
  }
```

在ExecutorRunner内部，它会结束监控线程，同时结束监控线程所启动的进程，并且向worker汇报ExecutorStateChanged：

```
def kill() {
  if (workerThread != null) {
    workerThread.interrupt()
    workerThread = null
    if (process != null) {
      logInfo("Killing process!")
      process.destroy()
      process.waitFor()
    }
    worker ! ExecutorStateChanged(appId, execId, ExecutorState.KILLED, None, None)
    Runtime.getRuntime.removeShutdownHook(shutdownHook)
  }
}
```

Application结束的同时清理了master和worker上的关于该application的所有信息，这样关于application结束的整个流程就介绍完了，当然在这里我们对于许多异常处理分支没有细究，但这并不影响我们对主线的把握。

# End

至此对于deploy模块的分析暂告一个段落。deploy模块相对来说比较简单，也没有特别复杂的逻辑结构，正如前面所说的deploy模块是为了能让更多的没有部署Mesos的集群的用户能够使用Spark而实现的一种方案。

当然现阶段看来还略微简陋，比如application的调度方式(FIFO)是否会造成小应用长时间等待大应用的结束，是否有更好的调度策略；资源的衡量标准是否可以更多更合理，而不单单是cpu数量，因为现实场景中有的应用是dis

k intensive，有的是network intensive，这样就算cpu资源有富余，调度新的application也不一定会很有意义。

总的来说作为Mesos的一种简单替代方式，deploy模块对于推广Spark还是有积极意义的。

分类: Spark



« 上一篇：【转】Spark源码分析之-Storage模块
» 下一篇：【转】Spark源码分析之-scheduler模块