Spark源码分析之-deploy模块

architecture (6) (/categories.html#architecture-ref)

cloud [7] (/tags.html#cloud-ref) spark [8] (/tags.html#spark-ref)

30 April 2013

Background

在前文Spark源码分析之-scheduler模块 (http://jerryshao.me/Architecture/2013/04/21/Spark%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90%E4%B9%8B-scheduler%E6%A8%A1%E5%9D%97/)中提到了Spark在资源管理和调度上采用了Hadoop **YARN** (http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html)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。

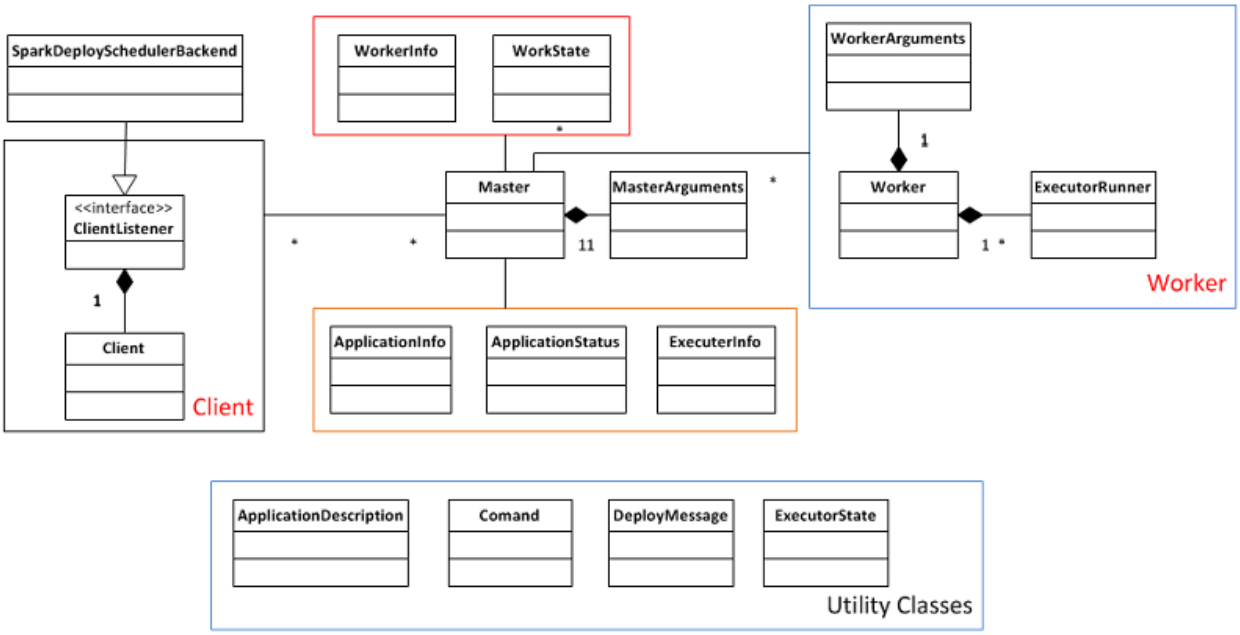
Spark最初是交由**Mesos** (http://incubator.apache.org/mesos/)进行资源管理，为了使得更多的用户，包括没有接触过Mesos的用户使用Spark，Spark的开发者添加了Standalone的部署方式，也就是deploy模块。因此deploy模块只针对不使用Mesos进行资源管理的部署方式。

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)
- **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, exec.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
```

```
41.      worker ! ExecutorStateChanged(appId, execId, ExecutorState.FAILED, Some(message),
42.                                    Some(exitCode))
43.    } catch {
44.      case interrupted: InterruptedException =>
45.        logInfo("Runner thread for executor " + fullId + " interrupted")
46.
47.      case e: Exception => {
48.        logError("Error running executor", e)
49.        if (process != null) {
50.          process.destroy()
51.        }
52.        val message = e.getClass + ": " + e.getMessage
53.        worker ! ExecutorStateChanged(appId, execId, ExecutorState.FAILED, Some(message), None)
54.      }
55.    }
56.  }
```

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：

```
1.  def removeApplication(app: ApplicationInfo) {
2.    if (apps.contains(app)) {
3.      logInfo("Removing app " + app.id)
4.      apps -= app
5.      idToApp -= app.id
6.      actorToApp -= app.driver
7.      addressToWorker -= app.driver.path.address
8.      completedApps += app   // Remember it in our history
9.      waitingApps -= app
10.     for (exec <- app.executors.values) {
11.       exec.worker.removeExecutor(exec)
12.       exec.worker.actor ! KillExecutor(exec.application.id, exec.id)
13.     }
14.     app.markFinished(ApplicationState.FINISHED)  // TODO: Mark it as FAILED if it failed
15.     schedule()
16.   }
17. }
```

`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disk 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。

← Previous (/architecture/2013/04/21/Spark%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90%E4%B9%8B-scheduler%E6%A8%A1%E5%9D%97)

Archive (/archive.html)

Next → (/architecture/2013/05/22/SparkStreaming-Job-Troubleshooting-of-Dependency-Chain)

Best Community Share Login

DjvuLee • 15 days ago
非常感谢lz详细的分析。问一个很小弱的问题，killExecutor的时候，为什么workerThread这个线程只是interrupt，而没有被kill掉？

Reply • Share ›

vincent_hv • 3 months ago
前段时间自己用四台机器做了些小测试，发现数据量由小到大的过程中，四台worker上的内存并不是均分的，差距很大，其中两三台机器的内存会用到2G多（每台机器4G），另外一台可能只用了五六百M。感觉上是在几个worker上计算，内存不够是在向其他worker转移，但是这种理解又违背了并行计算的初衷。比较困惑，如何理解这种情况呢？还有，我的HDFS部署在四台机上，如果我将spark部署在这四台机上后额外多添加第五台做为worker时，当执行的数据量已经让spark的性能急剧下滑，那四台机器的内存用到了近3G时，这第5个worker内存最多也就700M，cpu使用率也很低，也没太多的日志，GC日志也没几条，感觉这第5个worker并没有被使用到。为什么会这样？难道spark不会向空闲worker传输数据并计算吗？

Reply • Share ›

jerryshao Mod ➦ vincent_hv • 3 months ago
准确地说，类似于Hadoop,Spark之类的框架应该是分布式计算框架，而不是并行计算框架，对于这些框架设计的目标是处理大规模的数据，而不是做计算密集型作业，因此这些框架设计时考虑的一个很重要的因素是避免数据在网络传输而影响处理的吞吐量，所以往往是数据在哪个节点上，运算就在那个节点上执行，只有在slot (core)不够情况下数据和运算会放在不同的节点上。

看你的情况，可能是你的数据在hdfs并不是均匀分配，有的节点储存的数据多，有的少，那样数据多的节点处理的任务量就大，占用的内存就多，数据量少或者根本没数据的节点任务就占用很少的内存。内存不是调度任务的指标，只有在该节点slot(hadoop)或是core(Spark)不足的情况下任务才会调度到其他节点上面，所以再添加一台node也没多少意义。并且将任务调度到空闲的node需要把数据同时也传过去，这就违背了这些框架设计的初衷(移动程序而不是移动数据)。

先把Hadoop熟悉了再看Spark的设计理念就能想明白为什么要这样设计了 :-)

Reply • Share ›

vincent_hv • 3 months ago
博主你好，刚无意中看到你在google group上跟大牛们讨论spark提交任务的权限问题，也就是它的安全模式。因为英语不太好只能勉强进行阅读，无法书写交流。所以只能继续麻烦您了，spark_0.7.x版是如何解决多用户操作问题的？可以对指定用户分配资源吗？有自己独立的安全机制吗？刚出来的0.8.x版本对这方面有什么改进和加强吗？

Reply • Share ›

jerryshao Mod ➦ vincent_hv • 3 months ago
0.7没有解决多用户支持，提交的pacth还没有merge进去。这里的多用户支持指的是不同的用户需要以不同的身份向spark cluster提交任务，任务的执行需要以提交任务的用户的身份去执行。Hadoop是有用户访问权限的，Spark要访问Hadoop的话也需要有用户的概念，解决的是这个问题。不能对指定用户分配资源，也没有安全机制。

Reply • Share ›

vincent_hv ➦ jerryshao • 3 months ago
spark的standalone模式下的deploy模块调度是采用FIFO调度等待apps的，我想问的是在这个模式下只有这一中调度方式吗？因为实际运用中任务必然是有优先级的，这时要如

何解决？

Reply • Share ›

jerryshao Mod → vincent_hv • 3 months ago

在app内部，在新发布的0.8中已经有Fair Scheduler的调度方式了，这个feature是我们team做的。但是在app之间仍然只有FIFO的。如果想要app之间是 Fair Scheduler的话可以使用Mesos的fine grain模式。

Reply • Share ›

vincent_hv → jerryshao • 3 months ago

真的很感激您如此及时详尽的回复，之前spark的学习一直是自己瞎琢磨，我本身就是个小白，遇到问题都不知道如何解决，有点自己的理解又不知道与谁讨论，无从证实，挺痛苦的。最近学习过程中网上浏览到很多你以前的帖子，博客园，csdn，google group等，很庆幸能遇到你这样热心的前辈。之后可能还会有不少问题，希望不会影响到你的工作，这些问题您抽空闲时间回复就好，再次感谢您的帮助

Reply • Share ›

vincent_hv • 3 months ago

博主，请问spark计算过程中某个worker发生OOM，这时他是如何处理的？会自动重启这个worker然后继续错误发生前的任务吗？

Reply • Share ›

jerryshao Mod → vincent_hv • 3 months ago

worker应该是不会oom的，oom的应该是worker上的某个task吧？这个task会标记为失败并重新执行，如果重新执行继续失败n次以后这个job就会标记为失败，不再执行，这里的n是可以设置的。

Reply • Share ›

vincent_hv → jerryshao • 3 months ago

对，是task的oom， sorry没说清楚。那task在这个worker失败时，他会将task分配到其他worker上执行吗？这个n是在哪里设置的？

Reply • Share ›

jerryshao Mod → vincent_hv • 3 months ago

task失败之后如何重新调度得要看具体的代码，这一块我也没深究过，分配到哪里是由具体算法决定的，如果你想要了解，可以去看Scheduler这一块的具体实现。这里的n是通过java的property设置的，具体的是spark.task.maxFailures，默认是4。

Reply • Share ›

vincent_hv → jerryshao • 3 months ago

好的。我在官网的Spark Configuration板块并没有看到spark.task.maxFailures这个属性的设置，也就是说官网并没有列出完整的可设置的属性，我去哪能了解到所有的可配置属性呢？

Reply • Share ›

jerryshao Mod → vincent_hv • 3 months ago

Spark的配置现在还是比较乱的，没有所有都列出来的配置，需要从代码中找出来。我们现在在重构config这个模块，目标就是做成像Hadoop那样的清晰的配置方式，估计要到0.9才能正式release。

Reply • Share ›

CATEGORIES

lessons (1) (/categories.html#lessons-ref)

test (1) (/categories.html#test-ref)

architecture (6) (/categories.html#architecture-ref)

functional programming (1) (/categories.html#functional programming-ref)

arhitecture (1) (/categories.html#arhitecture-ref)

LINKS

阮一峰的网络日志 (http://www.ruanyifeng.com/blog/)

刘未鹏 (http://mindhacks.cn/)

酷壳 (http://coolshell.cn/)

BeiYuu.com (http://beiyuu.com/)

MY FAVORITES