Spark On YARN内存分配

时间 2015-06-09 00:00:00 JavaChen's Blog
原文 http://blog.javachen.com/2015/06/09/memory-in-spark-on-yarn.html
主题 Spark YARN

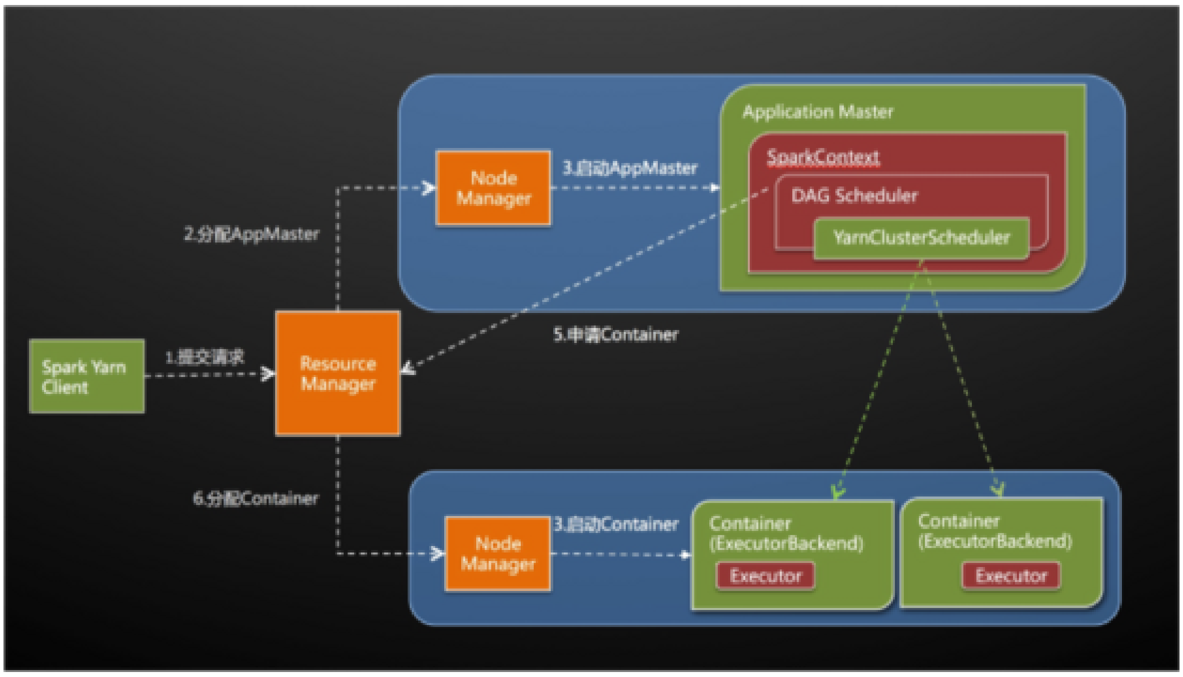
本文主要了解Spark On YARN部署模式下的内存分配情况，因为没有深入研究Spark的源代码，所以只能根据日志去看相关的源代码，从而了解“为什么会这样，为什么会那样”。

说明

按照Spark应用程序中的driver分布方式不同，Spark on YARN有两种模式： `yarn-client` 模式、 `yarn-cluster` 模式。

当在YARN上运行Spark作业，每个Spark executor作为一个YARN容器运行。Spark可以使得多个Tasks在同一个容器里面运行。

下图是yarn-cluster模式的作业执行图，图片来源于网络：



关于Spark On YARN相关的配置参数，请参考Spark配置参数。本文主要讨论内存分配情况，所以只需要关注以下几个内心相关的参数：

- spark.driver.memory ：默认值512m
- spark.executor.memory ：默认值512m
- spark.yarn.am.memory ：默认值512m
- spark.yarn.executor.memoryOverhead ：值为 executorMemory * 0.07, with minimum of 384
- spark.yarn.driver.memoryOverhead ：值为 driverMemory * 0.07, with minimum of 384
- spark.yarn.am.memoryOverhead ：值为 AM memory * 0.07, with minimum of 384

注意：

- --executor-memory/spark.executor.memory 控制 executor 的堆的大小，但是 JVM 本身也会占用一定的堆空间，比如内部的 String 或者直接 byte buffer， spark.yarn.XXX.memoryOverhead 属性决定向 YARN 请求的每个 executor 或dirver或am 的额外堆内存大小，默认值为 max(384, 0.07 * spark.executor.memory)
- 在 executor 执行的时候配置过大的 memory 经常会导致过长的GC延时，64G是推荐的一个 executor 内存大小的上限。
- HDFS client 在大量并发线程时存在性能问题。大概的估计是每个 executor 中最多5个并行的 task 就可以占满写入带宽。

另外，因为任务是提交到YARN上运行的，所以YARN中有几个关键参数，参考YARN的内存和CPU配置：

- yarn.app.mapreduce.am.resource.mb ：AM能够申请的最大内存，默认值为1536MB
- yarn.nodemanager.resource.memory-mb ：nodemanager能够申请的最大内存，默认值为8192MB
- yarn.scheduler.minimum-allocation-mb ：调度时一个container能够申请的最小资源，默认值为1024MB
- yarn.scheduler.maximum-allocation-mb ：调度时一个container能够申请的最大资源，默认值为8192MB

测试

Spark集群测试环境为：

- master：64G内存，16核cpu

- worker：128G内存，32核cpu
- worker：128G内存，32核cpu
- worker：128G内存，32核cpu
- worker：128G内存，32核cpu

注意：YARN集群部署在Spark集群之上的，每一个worker节点上同时部署了一个NodeManager，并且YARN集群中的配置如下：

```
<property>
    <name>yarn.nodemanager.resource.memory-mb</name>
    <value>106496</value> <!-- 104G -->
  </property>
  <property>
    <name>yarn.scheduler.minimum-allocation-mb</name>
    <value>2048</value>
  </property>
  <property>
    <name>yarn.scheduler.maximum-allocation-mb</name>
    <value>106496</value>
  </property>
  <property>
    <name>yarn.app.mapreduce.am.resource.mb</name>
    <value>2048</value>
  </property>
```

将spark的日志基本调为DEBUG，并将log4j.logger.org.apache.hadoop设置为WARN建设不必要的输出，修改/etc/spark/conf/log4j.properties：

```
# Set everything to be logged to the console
log4j.rootCategory=DEBUG, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd HH:mm:ss} %p %c{1}: %m%n

# Settings to quiet third party logs that are too verbose
log4j.logger.org.eclipse.jetty=WARN
log4j.logger.org.apache.hadoop=WARN
log4j.logger.org.eclipse.jetty.util.component.AbstractLifeCycle=ERROR
log4j.logger.org.apache.spark.repl.SparkIMain$exprTyper=INFO
log4j.logger.org.apache.spark.repl.SparkILoop$SparkILoopInterpreter=INFO
```

接下来是运行测试程序，以官方自带的SparkPi例子为例， 下面主要测试client模式，至于cluster模式请参考下面的过程 。运行下面命令：

```
spark-submit --class org.apache.spark.examples.SparkPi \
  --master yarn-client  \
  --num-executors 4 \
  --driver-memory 2g \
  --executor-memory 3g \
  --executor-cores 4 \
  /usr/lib/spark/lib/spark-examples-1.3.0-cdh5.4.0-hadoop2.6.0-cdh5.4.0.jar \
  100000
```

观察输出日志（无关的日志被略去）：

```
15/06/08 13:57:01 INFO SparkContext: Running Spark version 1.3.0
15/06/08 13:57:02 INFO SecurityManager: Changing view acls to: root
15/06/08 13:57:02 INFO SecurityManager: Changing modify acls to: root
```

```
15/06/08 13:57:03 INFO MemoryStore: MemoryStore started with capacity 1060.3 MB

15/06/08 13:57:04 DEBUG YarnClientSchedulerBackend: ClientArguments called with: --arg bj03-bi-pro-hdpnamen
15/06/08 13:57:04 DEBUG YarnClientSchedulerBackend: [actor] handled message (24.52531 ms) ReviveOffers from
15/06/08 13:57:05 INFO Client: Requesting a new application from cluster with 4 NodeManagers
15/06/08 13:57:05 INFO Client: Verifying our application has not requested more than the maximum memory capa
15/06/08 13:57:05 INFO Client: Will allocate AM container, with 896 MB memory including 384 MB overhead
15/06/08 13:57:05 INFO Client: Setting up container launch context for our AM

15/06/08 13:57:07 DEBUG Client: ===============================================================================
15/06/08 13:57:07 DEBUG Client: Yarn AM launch context:
15/06/08 13:57:07 DEBUG Client:     user class: N/A
15/06/08 13:57:07 DEBUG Client:     env:
15/06/08 13:57:07 DEBUG Client:         CLASSPATH -> <CPS>/__spark__.jar<CPS>$HADOOP_CONF_DIR<CPS>$HADOOP_CO
15/06/08 13:57:07 DEBUG Client:         SPARK_DIST_CLASSPATH -> :/usr/lib/spark/lib/spark-assembly.jar::/usr
15/06/08 13:57:07 DEBUG Client:         SPARK_YARN_CACHE_FILES_FILE_SIZES -> 97237208
15/06/08 13:57:07 DEBUG Client:         SPARK_YARN_STAGING_DIR -> .sparkStaging/application_1433742899916_00
15/06/08 13:57:07 DEBUG Client:         SPARK_YARN_CACHE_FILES_VISIBILITIES -> PRIVATE
15/06/08 13:57:07 DEBUG Client:         SPARK_USER -> root
15/06/08 13:57:07 DEBUG Client:         SPARK_YARN_MODE -> true
15/06/08 13:57:07 DEBUG Client:         SPARK_YARN_CACHE_FILES_TIME_STAMPS -> 1433743027399
15/06/08 13:57:07 DEBUG Client:         SPARK_YARN_CACHE_FILES -> hdfs://mycluster:8020/user/root/.sparkStag
15/06/08 13:57:07 DEBUG Client:     resources:
15/06/08 13:57:07 DEBUG Client:         __spark__.jar -> resource { scheme: "hdfs" host: "mycluster" port: 8
15/06/08 13:57:07 DEBUG Client:     command:
15/06/08 13:57:07 DEBUG Client:         /bin/java -server -Xmx512m -Djava.io.tmpdir=/tmp '-Dspark.eventLog.e
15/06/08 13:57:07 DEBUG Client: ===============================================================================
```

从 `Will allocate AM container, with 896 MB memory including 384 MB overhead` 日志可以看到，AM占用了 `896 MB` 内存，除掉 `384 MB` 的overhead内存，实际上只有 `512 MB` ，即 `spark.yarn.am.memory` 的默认值，另外可以看到YARN集群有4个NodeManager，每个container最多有106496 MB内存。

Yarn AM launch context启动了一个Java进程，设置的JVM内存为 `512m` ，见 `/bin/java -server -Xmx512m` 。

这里为什么会取默认值呢？查看打印上面这行日志的代码，见 org.apache.spark.deploy.yarn.Client：

```
private def verifyClusterResources(newAppResponse: GetNewApplicationResponse): Unit = {
  val maxMem = newAppResponse.getMaximumResourceCapability().getMemory()
  logInfo("Verifying our application has not requested more than the maximum " +
    s"memory capability of the cluster ($maxMem MB per container)")
  val executorMem = args.executorMemory + executorMemoryOverhead
  if (executorMem > maxMem) {
    throw new IllegalArgumentException(s"Required executor memory (${args.executorMemory}" +
      s"+$executorMemoryOverhead MB) is above the max threshold ($maxMem MB) of this cluster!")
  }
  val amMem = args.amMemory + amMemoryOverhead
  if (amMem > maxMem) {
    throw new IllegalArgumentException(s"Required AM memory (${args.amMemory}" +
      s"+$amMemoryOverhead MB) is above the max threshold ($maxMem MB) of this cluster!")
  }
  logInfo("Will allocate AM container, with %d MB memory including %d MB overhead".format(
    amMem,
    amMemoryOverhead))
}
```

args.amMemory来自ClientArguments类，这个类中会校验输出参数：

```
private def validateArgs(): Unit = {
  if (numExecutors <= 0) {
    throw new IllegalArgumentException(
      "You must specify at least 1 executor!\n" + getUsageMessage())
  }
  if (executorCores < sparkConf.getInt("spark.task.cpus", 1)) {
    throw new SparkException("Executor cores must not be less than " +
      "spark.task.cpus.")
  }
  if (isClusterMode) {
    for (key <- Seq(amMemKey, amMemOverheadKey, amCoresKey)) {
      if (sparkConf.contains(key)) {
        println(s"$key is set but does not apply in cluster mode.")
      }
    }
    amMemory = driverMemory
    amCores = driverCores
  } else {
    for (key <- Seq(driverMemOverheadKey, driverCoresKey)) {
      if (sparkConf.contains(key)) {
        println(s"$key is set but does not apply in client mode.")
      }
    }
    sparkConf.getOption(amMemKey)
      .map(Utils.memoryStringToMb)
      .foreach { mem => amMemory = mem }
    sparkConf.getOption(amCoresKey)
      .map(_.toInt)
      .foreach { cores => amCores = cores }
  }
}
```

从上面代码可以看到当 isClusterMode 为true时，则args.amMemory值为driverMemory的值；否则，则从 spark.yarn.am.memory 中取，如果没有设置该属性，则取默认值512m。isClusterMode 为true的条件是 userClass 不为空， def isClusterMode: Boolean = userClass != null ，即输出参数需要有 --class 参数，而从下面日志可以看到ClientArguments的输出参数中并没有该参数。

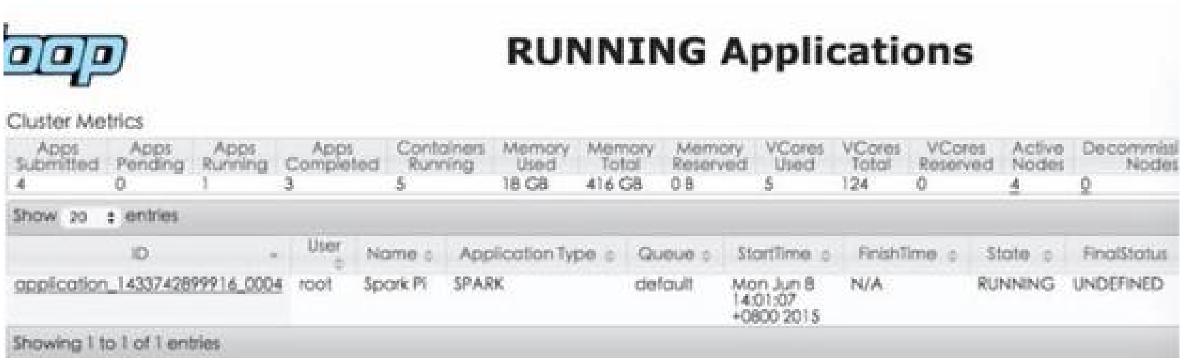
```
15/06/08 13:57:04 DEBUG YarnClientSchedulerBackend: ClientArguments called with: --arg bj03-bi-pro-hdpnamenr
```

故，要想设置AM申请的内存值，要么使用cluster模式，要么在client模式中，是有 --conf 手动设置 spark.yarn.am.memory 属性，例如：

```
spark-submit --class org.apache.spark.examples.SparkPi \
  --master yarn-client  \
  --num-executors 4 \
  --driver-memory 2g \
  --executor-memory 3g \
  --executor-cores 4 \
  --conf spark.yarn.am.memory=1024m \
  /usr/lib/spark/lib/spark-examples-1.3.0-cdh5.4.0-hadoop2.6.0-cdh5.4.0.jar \
  100000
```

打开YARN管理界面，可以看到：

a. Spark Pi 应用启动了5个Container，使用了18G内存、5个CPU core



RUNNING Applications

Cluster Metrics

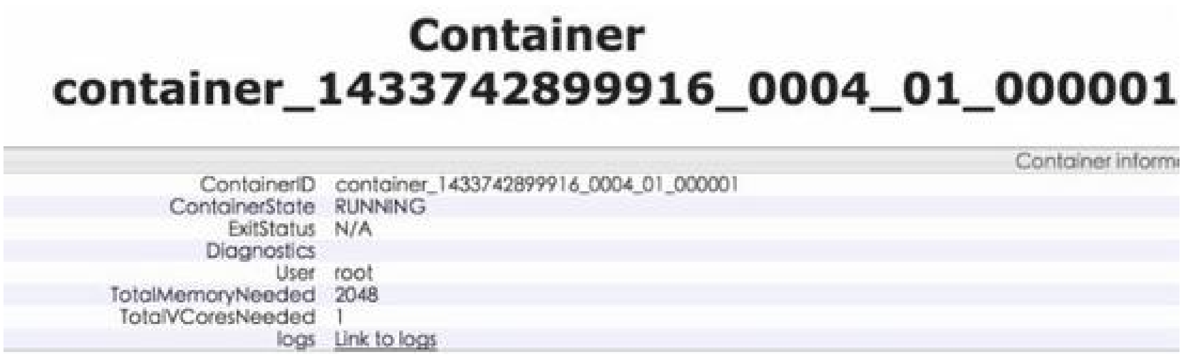
Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissi Nodes
4	0	1	3	5	18 GB	416 GB	0 B	5	124	0	4	0

Show 20 entries

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus
application_1433742899916_0004	root	Spark Pi	SPARK	default	Mon Jun 8 14:01:07 +0800 2015	N/A	RUNNING	UNDEFINED

Showing 1 to 1 of 1 entries

b. YARN为AM启动了一个Container，占用内存为2048M

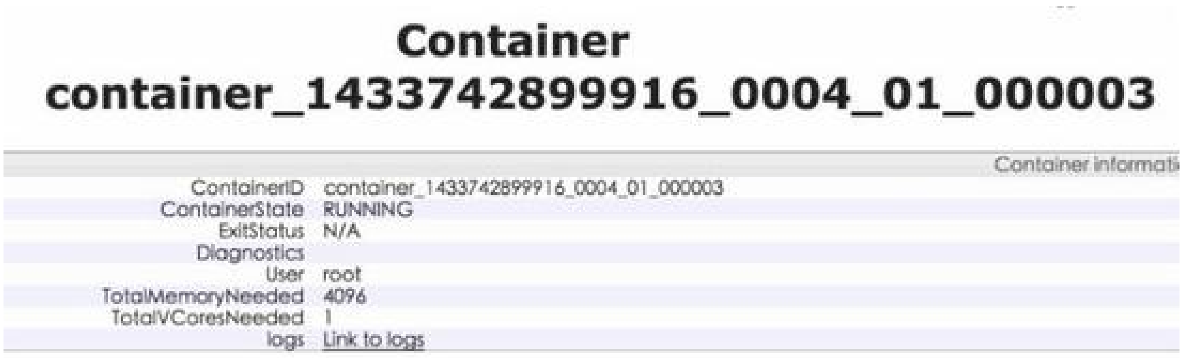


Container container_1433742899916_0004_01_000001

Container inform

ContainerID	container_1433742899916_0004_01_000001
ContainerState	RUNNING
ExitStatus	N/A
Diagnostics	
User	root
TotalMemoryNeeded	2048
TotalVCoresNeeded	1
logs	Link to logs

c. YARN启动了4个Container运行任务，每一个Container占用内存为4096M



Container container_1433742899916_0004_01_000003

Container informati

ContainerID	container_1433742899916_0004_01_000003
ContainerState	RUNNING
ExitStatus	N/A
Diagnostics	
User	root
TotalMemoryNeeded	4096
TotalVCoresNeeded	1
logs	Link to logs

为什么会是 `2G +4G *4=18G` 呢？第一个Container只申请了2G内存，是因为我们的程序只为AM申请了512m内存，而 `yarn.scheduler.minimum-allocation-mb` 参数决定了最少要申请2G内存。至于其余的Container，我们设置了executor-memory内存为3G，为什么每一个Container占用内存为4096M呢？

为了找出规律，多测试几组数据，分别测试并收集executor-memory为3G、4G、5G、6G时每个executor对应的Container内存申请情况：

- executor-memory=3g：2G+4G * 4=18G
- executor-memory=4g：2G+6G * 4=26G
- executor-memory=5g：2G+6G * 4=26G
- executor-memory=6g：2G+8G * 4=34G

关于这个问题，我是查看源代码，根据org.apache.spark.deploy.yarn.ApplicationMaster -> YarnRMClient -> YarnAllocator的类查找路径找到YarnAllocator中有这样一段代码：

```
// Executor memory in MB.
protected val executorMemory = args.executorMemory
// Additional memory overhead.
protected val memoryOverhead: Int = sparkConf.getInt("spark.yarn.executor.memoryOverhead",
  math.max((MEMORY_OVERHEAD_FACTOR * executorMemory).toInt, MEMORY_OVERHEAD_MIN))
// Number of cores per executor.
protected val executorCores = args.executorCores
// Resource capability requested for each executors
private val resource = Resource.newInstance(executorMemory + memoryOverhead, executorCores)
```

因为没有具体的去看YARN的源代码，所以这里猜测Container的大小是根据 `executorMemory + memoryOverhead` 计算出来的，大概的规则是每一个Container的大小必须为 `yarn.scheduler.minimum-allocation-mb` 值的整数倍，当 `executor-memory=3g` 时， `executorMemory + memoryOverhead` 为3G+384M=3456M，需要申请的Container大小为 `yarn.scheduler.minimum-allocation-mb` * 2 =4096m=4G，其他依此类推。

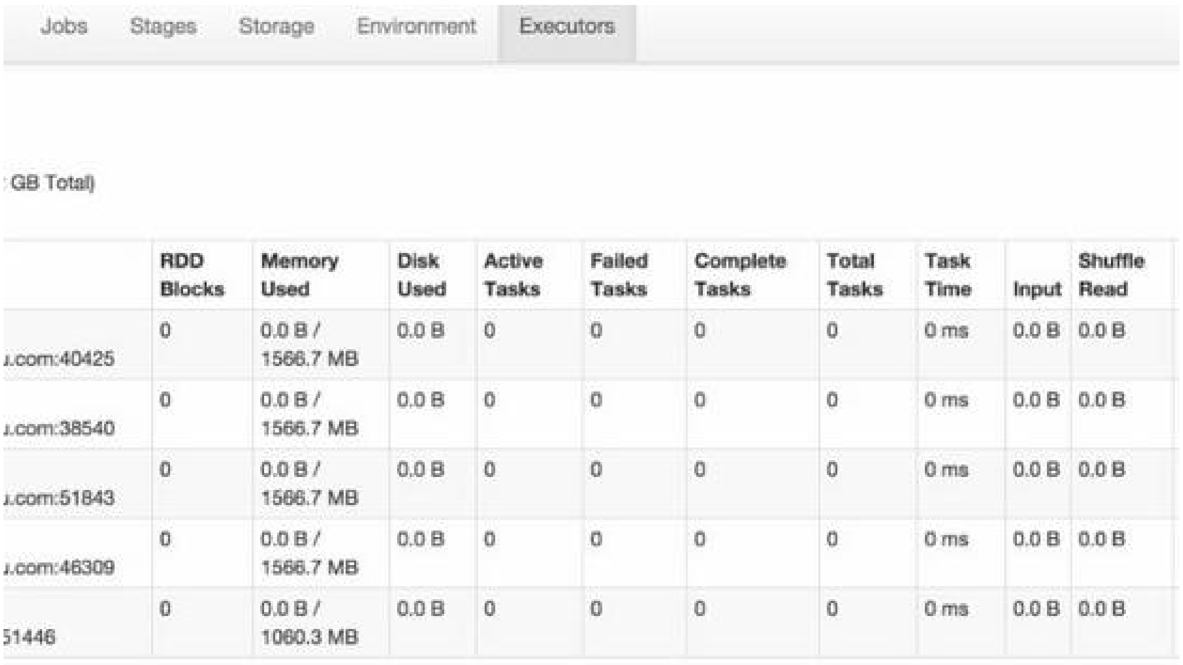
注意：

- Yarn always rounds up memory requirement to multiples of `yarn.scheduler.minimum-allocation-mb` , which by default is 1024 or 1GB.
- Spark adds an `overhead` to `SPARK_EXECUTOR_MEMORY/SPARK_DRIVER_MEMORY` before asking Yarn for the amount.

另外，需要注意memoryOverhead的计算方法，当executorMemory的值很大时，memoryOverhead的值相应会变大，这个时候就不是384m了，相应的Container申请的内存值也变大了，例如：当executorMemory设置为90G时，memoryOverhead值为 `math.max(0.07 * 90G, 384m)=6.3G` ，其对应的Container申请的内存为98G。

回头看看给AM对应的Container分配2G内存原因，512+384=896，小于2G，故分配2G，你可以在设置 `spark.yarn.am.memory` 的值之后再来观察。

打开Spark的管理界面 http://ip:4040 ，可以看到driver和Executor中内存的占用情况：



	RDD Blocks	Memory Used	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read
.com:40425	0	0.0 B / 1566.7 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B
.com:38540	0	0.0 B / 1566.7 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B
.com:51843	0	0.0 B / 1566.7 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B
.com:46309	0	0.0 B / 1566.7 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B
51446	0	0.0 B / 1060.3 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B

从上图可以看到Executor占用了1566.7 MB内存，这是怎样计算出来的？参考 [Spark on Yarn: Where Have All the Memory Gone?](#) 这篇文章，totalExecutorMemory的计算方式为：

```
 //yarn/common/src/main/scala/org/apache/spark/deploy/yarn/YarnSparkHadoopUtil.scala
  val MEMORY_OVERHEAD_FACTOR = 0.07
  val MEMORY_OVERHEAD_MIN = 384

 //yarn/common/src/main/scala/org/apache/spark/deploy/yarn/YarnAllocator.scala
  protected val memoryOverhead: Int = sparkConf.getInt("spark.yarn.executor.memoryOverhead",
    math.max((MEMORY_OVERHEAD_FACTOR * executorMemory).toInt, MEMORY_OVERHEAD_MIN))
......
      val totalExecutorMemory = executorMemory + memoryOverhead
      numPendingAllocate.addAndGet(missing)
      logInfo(s"Will allocate $missing executor containers, each with $totalExecutorMemory MB " +
        s"memory including $memoryOverhead MB overhead")
```

这里我们给executor-memory设置的3G内存，memoryOverhead的值为 `math.max(0.07 * 3072, 384)=384` ，其最大可用内存通过下面代码来计算：

```
//core/src/main/scala/org/apache/spark/storage/BlockManager.scala
/** Return the total amount of storage memory available. */
private def getMaxMemory(conf: SparkConf): Long = {
  val memoryFraction = conf.getDouble("spark.storage.memoryFraction", 0.6)
  val safetyFraction = conf.getDouble("spark.storage.safetyFraction", 0.9)
  (Runtime.getRuntime.maxMemory * memoryFraction * safetyFraction).toLong
}
```

即，对于executor-memory设置3G时，executor内存占用大约为 3072m * 0.6 * 0.9 = 1658.88m，注意：实际上是应该乘以 `Runtime.getRuntime.maxMemory` 的值，该值小于3072m。

上图中driver占用了1060.3 MB，此时driver-memory的值是位2G，故driver中存储内存占用为：2048m * 0.6 * 0.9 =1105.92m，注意：实际上是应该乘以 `Runtime.getRuntime.maxMemory` 的值，该值小于2048m。

这时候，查看worker节点CoarseGrainedExecutorBackend进程启动脚本：

```
$ jps
46841 Worker
21894 CoarseGrainedExecutorBackend
9345
21816 ExecutorLauncher
43369
24300 NodeManager
38012 JournalNode
36929 QuorumPeerMain
22909 Jps

$ ps -ef|grep 21894
nobody   21894 21892 99 17:28 ?        00:04:49 /usr/java/jdk1.7.0_71/bin/java -server -XX:OnOutOfMemoryErro
```

可以看到每个CoarseGrainedExecutorBackend进程分配的内存为3072m，如果我们想查看每个executor的jvm运行情况，可以开启jmx。在/etc/spark/conf/spark-defaults.conf中添加下面一行代码：

```
spark.executor.extraJavaOptions -Dcom.sun.management.jmxremote.port=1099 -Dcom.sun.management.jmxremote.ssl=
```

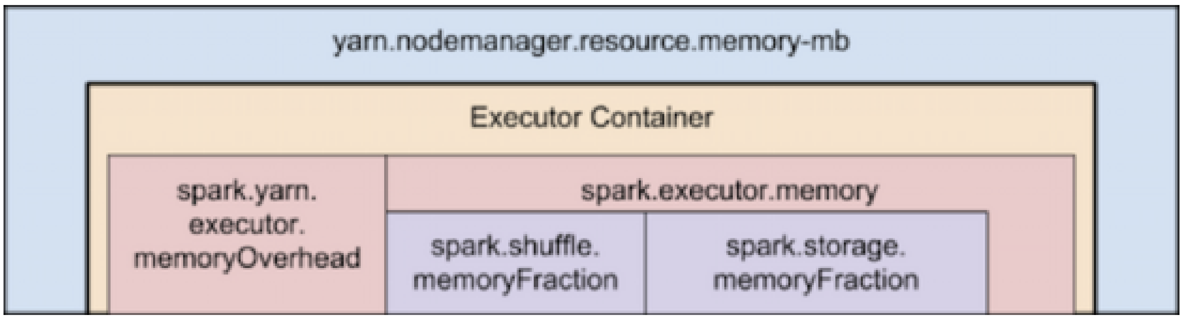
然后，通过jconsole监控jvm堆内存运行情况，这样方便调试内存大小。

总结

由上可知，在client模式下，AM对应的Container内存由 `spark.yarn.am.memory` 加上 `spark.yarn.am.memoryOverhead` 来确定，executor加上spark. `yarn.executor.memoryOverhead` 的值之后确定对应Container需要申请的内存大小，driver和executor的内存加上 `spark.yarn.driver.memoryOverhead` 或 `spark.yarn.executor.memoryOverhead` 的值之后再乘以0.54确定storage memory内存大小。在YARN中，Container申请的内存大小必须为

yarn.scheduler.minimum-allocation-mb 的整数倍。

下面这张图展示了Spark on YARN 内存结构，图片来自 How-to: Tune Your Apache Spark Jobs (Part 2) ：



yarn.nodemanager.resource.memory-mb
Executor Container
spark.yarn.executor.memoryOverhead
spark.executor.memory
spark.shuffle.memoryFraction
spark.storage.memoryFraction

至于cluster模式下的分析，请参考上面的过程。希望这篇文章对你有所帮助！

分享

收藏 纠错

翻译公司收费标准

安卓程序员工资

软件工程师待遇

顺义别墅

推荐文章

- 1. Storm运维调优笔记（7）——Pyleus设置拓扑worker数量
- 2. PaaS时代来临，运维人要做哪些准备工作
- 3. Spark编译与打包
- 4. Storm运维实战调优（6）——使用Pyleus提交的Topology如何修改任务..
- 5. 直传文件到Azure Storage的Blob服务中
- 6. Pig 实现关键词匹配

我来评几句

请输入评论内容...

登录后评论

已发表评论数(0)

相关站点