[Spark On YARN内存分配](http://blog.csdn.net/wisgood/article/details/51436546)

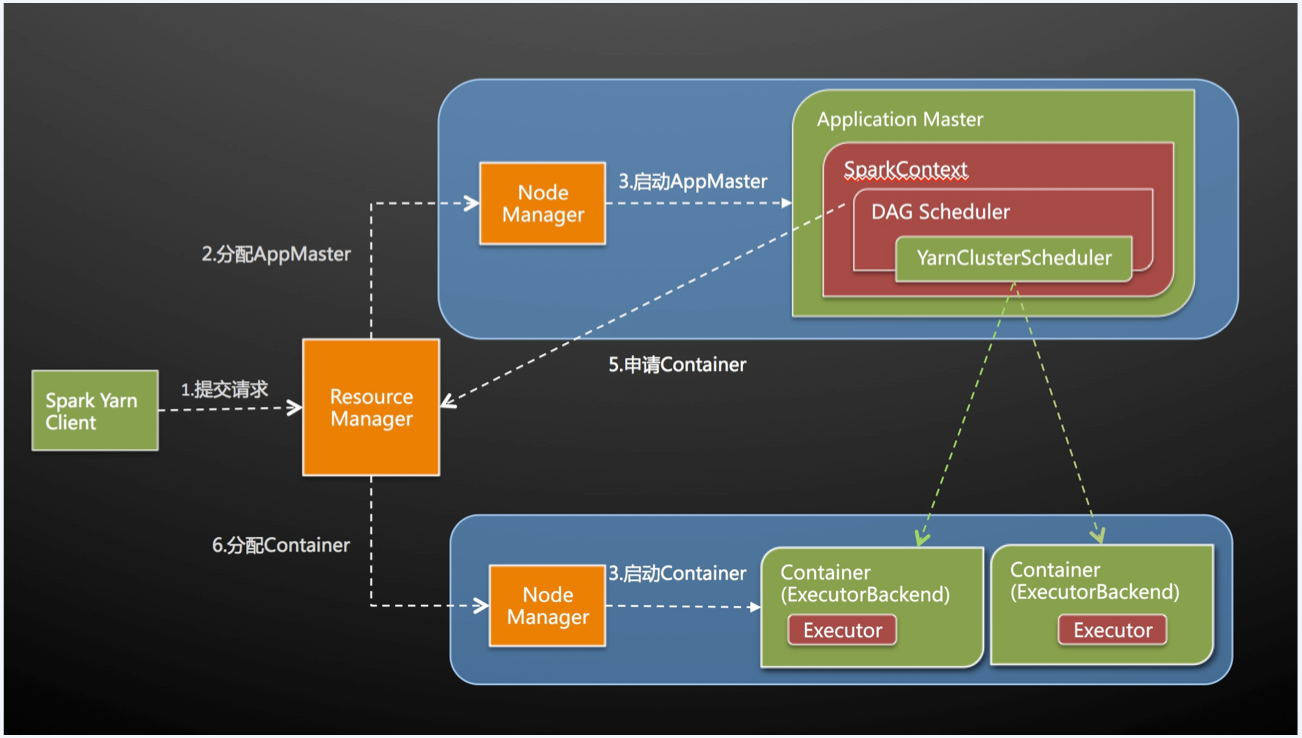
本文主要了解**[Spark](http://lib.csdn.net/base/10" \o "Apache Spark知识库" \t "_blank)** On YARN部署模式下的内存分配情况，因为没有深入研究Spark的源代码，所以只能根据日志去看相关的源代码，从而了解“为什么会这样，为什么会那样”。

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](http://blog.javachen.com/2015/06/07/spark-configuration.html" \t "_blank)。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：**(1.5.2版本中，比例0.07其实是0.1)**

* 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](http://blog.javachen.com/2015/06/05/yarn-memory-and-cpu-configuration.html)：

* 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**](http://lib.csdn.net/base/20)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-hdpnamenn:51568 --num-executors 4 --num-executors 4 --executor-memory 3g --executor-memory 3g --executor-cores 4 --executor-cores 4 --name Spark Pi

15/06/08 13:57:04 DEBUG YarnClientSchedulerBackend: [actor] handled message (24.52531 ms) ReviveOffers from Actor[akka://sparkDriver/user/CoarseGrainedScheduler#864850679]

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 capability of the cluster (106496 MB per container)

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\_COMMON\_HOME/\*<CPS>$HADOOP\_COMMON\_HOME/lib/\*<CPS>$HADOOP\_HDFS\_HOME/\*<CPS>$HADOOP\_HDFS\_HOME/lib/\*<CPS>$HADOOP\_MAPRED\_HOME/\*<CPS>$HADOOP\_MAPRED\_HOME/lib/\*<CPS>$HADOOP\_YARN\_HOME/\*<CPS>$HADOOP\_YARN\_HOME/lib/\*<CPS>$HADOOP\_MAPRED\_HOME/share/hadoop/mapreduce/\*<CPS>$HADOOP\_MAPRED\_HOME/share/hadoop/mapreduce/lib/\*<CPS>:/usr/lib/spark/lib/spark-assembly.jar::/usr/lib/hadoop/lib/\*:/usr/lib/hadoop/\*:/usr/lib/hadoop-hdfs/lib/\*:/usr/lib/hadoop-hdfs/\*:/usr/lib/hadoop-mapreduce/lib/\*:/usr/lib/hadoop-mapreduce/\*:/usr/lib/hadoop-yarn/lib/\*:/usr/lib/hadoop-yarn/\*:/usr/lib/hive/lib/\*:/usr/lib/flume-ng/lib/\*:/usr/lib/paquet/lib/\*:/usr/lib/avro/lib/\*

15/06/08 13:57:07 DEBUG Client: SPARK\_DIST\_CLASSPATH -> :/usr/lib/spark/lib/spark-assembly.jar::/usr/lib/hadoop/lib/\*:/usr/lib/hadoop/\*:/usr/lib/hadoop-hdfs/lib/\*:/usr/lib/hadoop-hdfs/\*:/usr/lib/hadoop-mapreduce/lib/\*:/usr/lib/hadoop-mapreduce/\*:/usr/lib/hadoop-yarn/lib/\*:/usr/lib/hadoop-yarn/\*:/usr/lib/hive/lib/\*:/usr/lib/flume-ng/lib/\*:/usr/lib/paquet/lib/\*:/usr/lib/avro/lib/\*

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\_0001

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/.sparkStaging/application\_1433742899916\_0001/spark-assembly-1.3.0-cdh5.4.0-hadoop2.6.0-cdh5.4.0.jar#\_\_spark\_\_.jar

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: 8020 file: "/user/root/.sparkStaging/application\_1433742899916\_0001/spark-assembly-1.3.0-cdh5.4.0-hadoop2.6.0-cdh5.4.0.jar" } size: 97237208 timestamp: 1433743027399 type: FILE visibility: PRIVATE

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.enabled=true' '-Dspark.executor.instances=4' '-Dspark.executor.memory=3g' '-Dspark.executor.cores=4' '-Dspark.driver.port=51568' '-Dspark.serializer=org.apache.spark.serializer.KryoSerializer' '-Dspark.driver.appUIAddress=http://bj03-bi-pro-hdpnamenn:4040' '-Dspark.executor.id=<driver>' '-Dspark.kryo.classesToRegister=scala.collection.mutable.BitSet,scala.Tuple2,scala.Tuple1,org.apache.spark.mllib.recommendation.Rating' '-Dspark.driver.maxResultSize=8g' '-Dspark.jars=file:/usr/lib/spark/lib/spark-examples-1.3.0-cdh5.4.0-hadoop2.6.0-cdh5.4.0.jar' '-Dspark.driver.memory=2g' '-Dspark.eventLog.dir=hdfs://mycluster:8020/user/spark/applicationHistory' '-Dspark.app.name=Spark Pi' '-Dspark.fileserver.uri=http://X.X.X.X:49172' '-Dspark.tachyonStore.folderName=spark-81ae0186-8325-40f2-867b-65ee7c922357' -Dspark.yarn.app.container.log.dir=<LOG\_DIR> org.apache.spark.deploy.yarn.ExecutorLauncher --arg 'bj03-bi-pro-hdpnamenn:51568' --executor-memory 3072m --executor-cores 4 --num-executors 4 1> <LOG\_DIR>/stdout 2> <LOG\_DIR>/stderr

15/06/08 13:57:07 DEBUG Client: ===============================================================================

从Will allocate AM [**Container**](http://lib.csdn.net/base/4), 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](http://lib.csdn.net/base/17" \o "Java EE知识库" \t "_blank)**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)

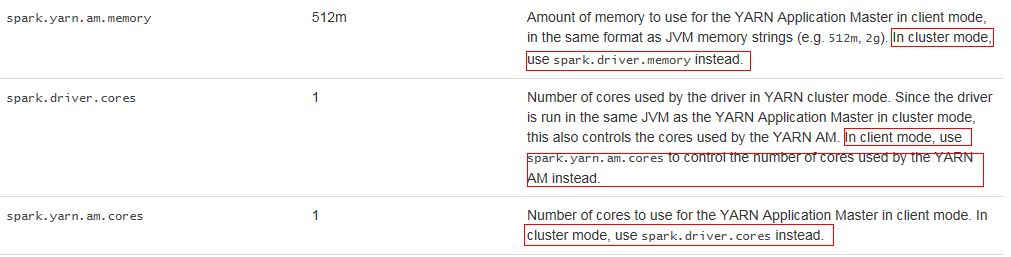
.map(**\_**.toInt)

.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-hdpnamenn:51568 --num-executors 4 --num-executors 4 --executor-memory 3g --executor-memory 3g --executor-cores 4 --executor-cores 4 --name Spark Pi

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 **\**

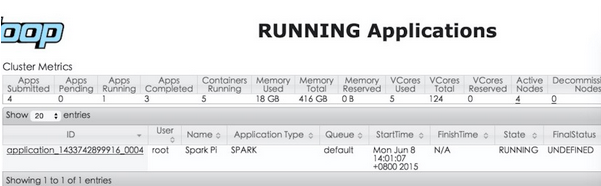
--conf spark.yarn.am.memory=1024m **\**

/usr/lib/spark/lib/spark-examples-1.3.0-cdh5.4.0-hadoop2.6.0-cdh5.4.0.jar **\**

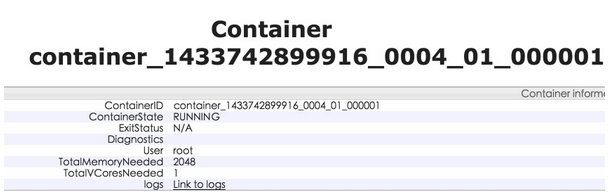
100000

打开YARN管理界面，可以看到：

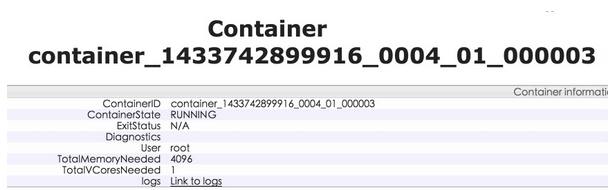
a. Spark Pi 应用启动了5个Container，使用了18G内存、5个CPU core



b. YARN为AM启动了一个Container，占用内存为2048M



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



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。

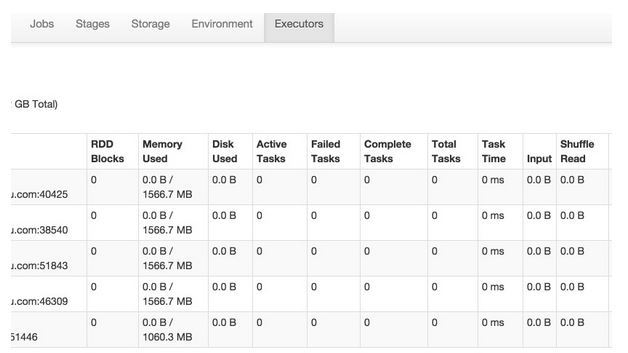
注意：

* 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](http://ip:4040/) ，可以看到driver和Executor中内存的占用情况：



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?](http://www.wdong.org/wordpress/blog/2015/01/08/spark-on-yarn-where-have-all-my-memory-gone/" \t "_blank)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:OnOutOfMemoryError=kill %p -Xms3072m -Xmx3072m -Djava.io.tmpdir=/data/yarn/local/usercache/root/appcache/application\_1433742899916\_0069/container\_1433742899916\_0069\_01\_000003/tmp -Dspark.driver.port=60235 -Dspark.yarn.app.container.log.dir=/data/yarn/logs/application\_1433742899916\_0069/container\_1433742899916\_0069\_01\_000003 org.apache.spark.executor.CoarseGrainedExecutorBackend --driver-url akka.tcp://sparkDriver@bj03-bi-pro-hdpnamenn:60235/user/CoarseGrainedScheduler --executor-id 2 --hostname X.X.X.X --cores 4 --app-id application\_1433742899916\_0069 --user-class-path file:/data/yarn/local/usercache/root/appcache/application\_1433742899916\_0069/container\_1433742899916\_0069\_01\_000003/\_\_app\_\_.jar

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：

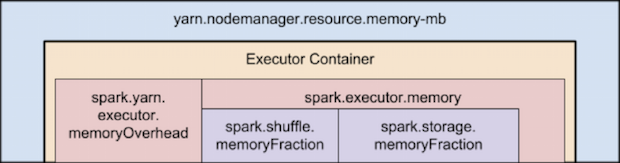
spark.executor.extraJavaOptions -Dcom.sun.management.jmxremote.port=1099 -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote.authenticate=false

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)](http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/)：



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！