Spark on Yarn 之Executor内存管理

本文1、2、3节介绍了Spark 内存相关之识，第4节描述了常见错误类型及产生原因并给出了解决方案。

1堆内和堆外内存规划

Executor 的内存管理建立在 JVM 的内存管理之上，Spark 对 JVM 的空间（Heap+Off-heap）进行了更为详细的分配，以充分利用内存。同时，Spark 引入了Off-heap（TungSten）内存模式，使之可以直接在工作节点的系统内存中开辟空间，进一步优化了内存的使用（可以理解为是独立于JVM托管的Heap之外利用c-style的malloc从os分配到的memory。由于不再由JVM托管，通过高效的内存管理，可以避免JVM object overhead和Garbage collection的开销）。运行于Executor中的Task同时可使用JVM和Off-heap两种模式的内存。

JVM OnHeap内存： 大小由”--executor-memory”(即 spark.executor.memory)参数指定。Executor中运行的并发任务共享JVM堆内内存。

JVM OffHeap内存：大小由“spark.yarn.executor.memoryOverhead”参数指定，主要用于JVM自身，字符串等开销。

Off-heap模式：默认情况下Off-heap模式的内存并不启用，可以通过“spark.memory.offHeap.enabled”参数开启，并由spark.memory.offHeap.size指定堆外内存的大小（占用overhead空间）。

备注：我们现在未启用Off-heap模式的内存，因此，只介绍JVM模式的Executor内存管理。

2 Executor内存划分

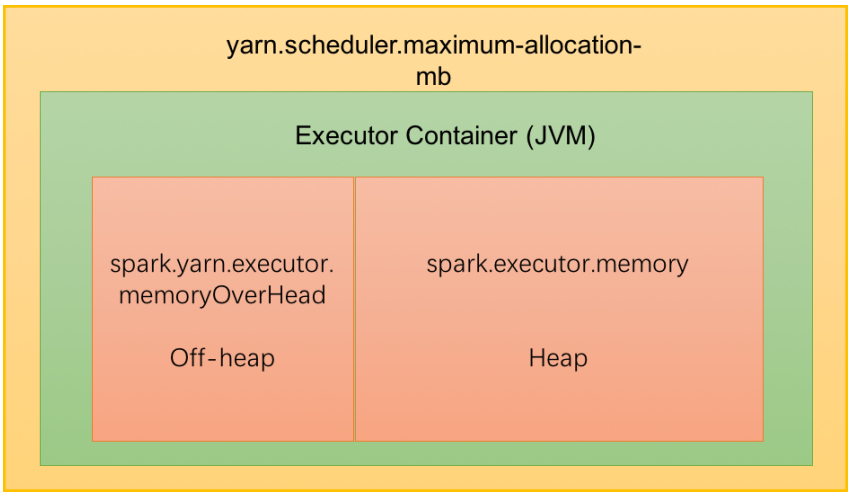
2.1 Executor可用内存总量

如下图所示，Yarn集群管理模式中，Spark 以Executor Container的形式在NodeManager中运行，其可使用的内存上限由（yarn.scheduler.maximum-allocation-mb，公司集群现默认为15G）限制。

如前所述，Executor的内存由“spark.executor.memory”设定的Heap内存和“spark.yarn.executor.memoryOverhead”设定的Off-heap内存组成。因此，对现有Yarn集群，存在：

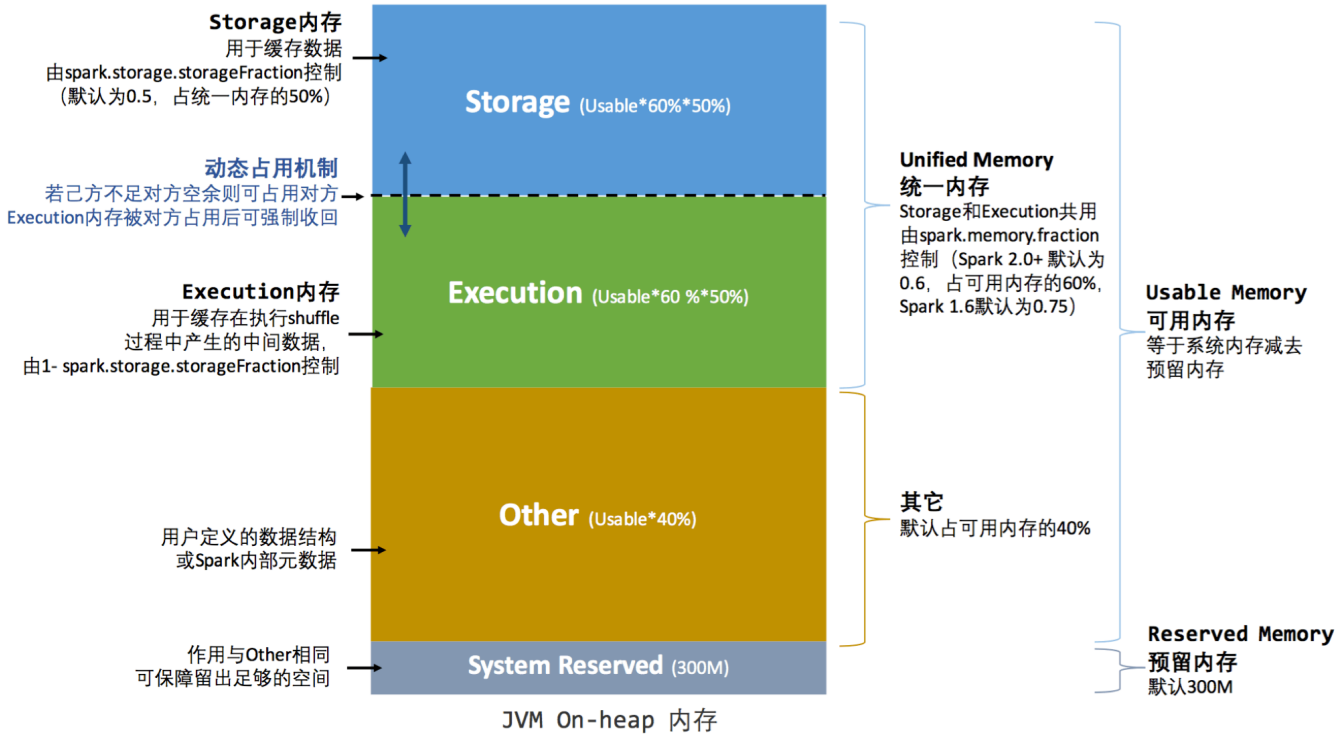
“spark.executor.memory”+ “spark.yarn.executor.memoryOverhead”<= 15G

若总量大于15G，则，会导致Executor申请失败；若运行过程中，实际使用内存超过上限阈值，Executor进程会被Yarn终止掉（kill）。



2.2 Heap

“spark.executor.memory”指定的内存为JVM最大分配的堆内存（“-xmx”），Spark为了更高效的使用这部分内存，对这部分内存进行了细分，下图（备注：此图源于互联网）对基于spark 2.0 （1.6）对堆内存分配比例进行了描述：



其中：

1) Reserved Memory保留内存，系统默认值为300，一般无需改动，不用关心此部分内存。 但如果Executor分配的内存小于 1.5 * 300 = 450M时，Executor将无法执行。

2) Storage Memory 存储内存

用于缓存Shuffle过程中生成的数据及RDD的缓存数据。由上图可知，Spark 2+中，初始状态下，Storage及Execution Memory均约占系统总内存的30%（1 * 0.6 * 0.5 = 0.3）。在Unified Memory中，这两部分内存可以相互借用，为了方便描述我们记spark.storage.storageFraction为storageRegionSize。当计算内存不足时，可以改造storageRegionSize中未使用部分，且StorageMemory需要存储内存时也不可被抢占； 若实际StorageMemory使用量超过storageRegionSize，那么当计算内存不足时，可以改造(StorageMemory - storageRegionSize)部分，而storageRegionSize部分不可被抢占。

备注： Unified Memory中，shuffle.memoryFraction, storage.unrollFraction 等参数无需在指定。

2.3 Java Off-heap (Memory Overhead)

Executor 中，另一块内存为由“spark.yarn.executor.memoryOverhead”指定的Java Off-heap内存，此部分内存主要是创建Java Object时的额外开销，Native方法调用，线程栈， NIO Buffer等开销。此部分为用户代码及Spark 不可操作的内存，不足时可通过调整参数解决，无需过多关注。

3 任务内存管理（Task Memory Manager）

Executor中任务以线程的方式执行，各线程共享JVM的资源，任务之间的内存资源没有强隔离（任务没有专用的Heap区域）。因此，可能会出现这样的情况：先到达的任务可能占用较大的内存，而后到的任务因得不到足够的内存而挂起。

在Spark任务内存管理中，使用HashMap存储任务与其消耗内存的映射关系。每个任务可占用的内存大小为潜在可使用计算内存的1/2n - 1/n ，当剩余内存为小于1/2n时，任务将被挂起，直至有其他任务释放执行内存，而满足内存下限1/2n，任务被唤醒，其中n为当前Executor中活跃的任务数。

任务执行过程中，如果需要更多的内存，则会进行申请，如果，存在空闲内存，则自动扩容成功，否则，将抛出OutOffMemroyError。

备注，潜在可使用计算内存为：初始计算内存+可抢占存储内存

4 内存调整方案

如下图所示Spark 管理的内存主要为 M_1 和 M_2，其中与任务处理数据直接使用的内存为 M_2，**每个任务可使用最大内存量约为（M_2 / 正在运行的任务数量）.**

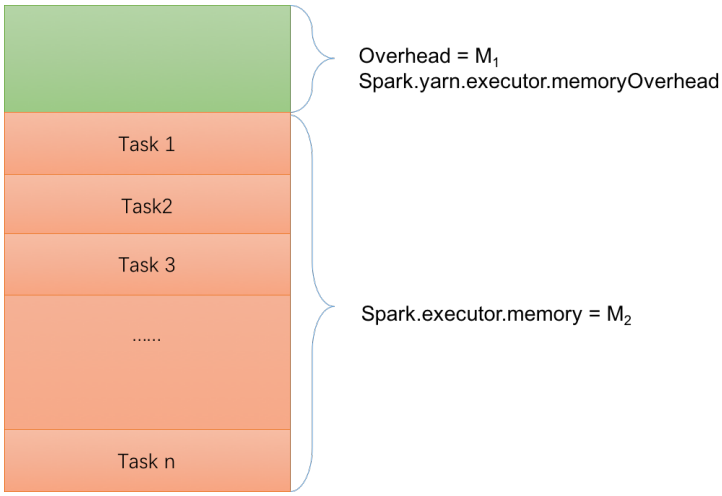
Executor中可同时运行的任务数由Executor分配的CPU的核数N 和每个任务需要的CPU核心数C决定。其中:

N = spark.executor.cores

C = spark.task.cpus

则，Executor的最大任务并行度可表示为 TP = N / C. 其中,C值与应用类型有关，大部分应用使用默认值1即可，因此，影响Executor中最大任务并行度的主要因素是N.

依据Task的内存使用特征，前文所述的Executor内存模型可以简单抽象为下图所示模型：



其中， Executor 向yarn申请的总内存可表示为： $M = M_1 + M_2$.

4.1 错误类型及调整方案

4.1.1 Executor OOM类错误 （错误代码 137、143等）

该类错误一般是由于Heap（M_2）已达上限，Task需要更多的内存，而又得不到足够的内存而导致。因此，解决方案要从增加每个Task的内存使用量，满足任务需求 或 降低单个Task的内存消耗量，从而使现有内存可以满足任务运行需求两个角度出发。因此存在如下解决方案：

1） 增加单个task的内存使用量

〈1〉增加最大Heap值， 即 上图中M_2 的值，使每个Task可使用内存增加。

操作方法：在提交脚本中添加 --conf spark.executor.memory=12g〈设置一个更大的值〉 (注：若$M = M_1 + M_2$ 已达到15g 请参考下面解决方案)

〈2〉降低Executor的可用Core的数量 N ，使Executor中同时运行的任务数减少，在总资源不变的情况下，使每个Task获得的内存相对增加。

操作方法：在提交脚本中添加 --executor-cores=3 〈比原来小的值〉 或 --conf spark.executor.cores=3 〈比原来小的值〉

2） 降低单个Task的内存消耗量

可从配制方式和调整应用逻辑两个层面进行优化

配制方式

减少每个Task处理的数据量，可降低Task的内存开销，在Spark中，每个partition对应一个处理任务Task，因此，在数据总量一定的前提下，可以通过增加partition数量的方式来减少每个Task处理的数据量,从而降低Task的内存开销。针对不同的Spark应用类型，存在不同的partition调整参数如下：

〈1〉 P = spark.default.parallism (非SQL应用)　　操作方法： 在提交脚本中添加 --conf spark.default.parallism=〈比原来大的值 ， 依数据量估算〉

〈2〉 P = spark.sql.shuffle.partitions (SQL 应用)　　操作方法： 在提交脚本中添加 --conf spark.sql.shuffle.partitions=〈比原来大的值 ， 依数据量估算〉

通过增加P的值，可在一定程度上使Task现有内存满足任务运行

注：当调整一个参数不能解决问题时，上述方案应进行协同调整 例如： --conf spark.executor.memory=12g --conf spark.executor.cores=3 --conf spark.default.parallism=〈更大的值〉 ……

若Driver端发生OOM，则：增加内存 或 调整代码（多为大量数据回传driver端所致，或map任务过多，导致大量shuffle结果上报driver端）

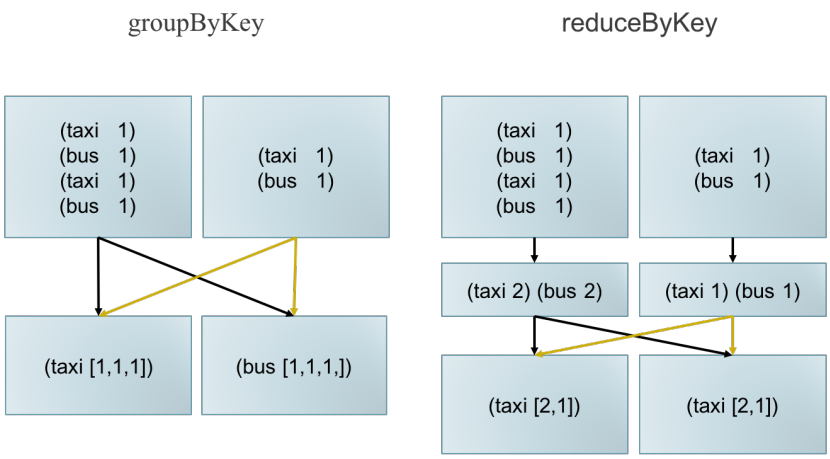
若应用shuffle阶段 spill严重，则可以通过调整“spark.shuffle.spill.numElementsForceSpillThreshold”的值，来限制spill使用的内存大小，比如设置（1500000），该值太大不足以解决OOM问题，若太小，则spill会太频繁，影响集群性能，因此，要依据负载类型进行合理伸缩（此处，可设法引入动态伸缩机制，待后续处理）。

调整应用逻辑

Executor OOM 一般发生Shuffle阶段，该阶段需求计算内存较大，且应用逻辑对内存需求有较大影响，下面举例就行说明：

〈1〉 groupByKey 转换为 reduceByKey

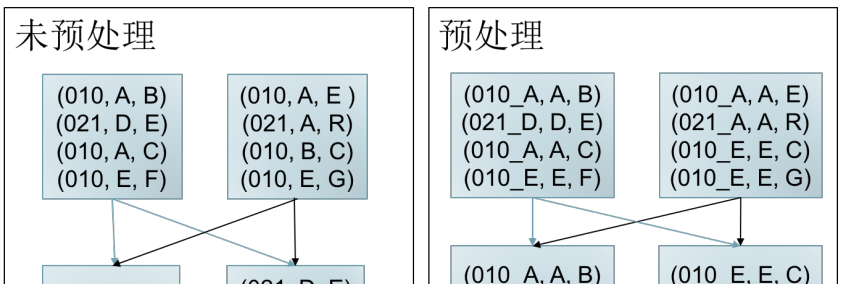
一般情况下，groupByKey能实现的功能使用reduceByKey均可实现，而ReduceByKey存在Map端的合并，可以有效减少传输带宽占用及Reduce端内存消耗。

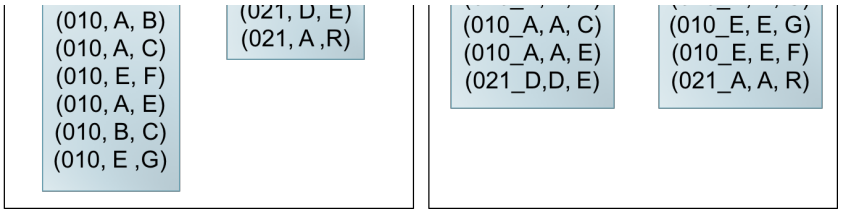


〈2〉 数据预处理，防止数据倾斜（ Data Skew）

Data Skew是指任务间处理的数据量存大较大的差异。

如左图所示，key 为010的数据较多，当发生shuffle时，010所在分区存在大量数据，不仅拖慢Job执行（Job的执行时间由最后完成的任务决定）。 而且导致010对应Task内存消耗过多，可能导致OOM. 而右图，经过预处理（加盐，此处仅为举例说明问题，解决方法不限于此）可以有效减少Data Skew导致 的问题





，

注：上述举例仅为说明调整应用逻辑可以在一定程序上解决OOM问题，解决方法不限于上述举例

4.1.2 Beyond…… memory, killed by yarn.

出现该问题原因是由于实际使用内存上限超过申请的内存上限而被Yarn终止掉了，首先说明Yarn中Container内存监控机制：

Container进程的内存使用量：以Container进程为根的进程树中所有进程的内存使用总量。

Container被杀死的判断依据：进程树总内存（物理内存或虚拟内存）使用量超过向Yarn申请的内存上限值，则认为该Container使用内存超量，可以被“杀死”。

因此，对该异常的分析要从是否存在子进程两个角度出发。

〈1〉不存在子进程

根据Container进程杀死的条件可知，在不存在子进程时，出现killed by yarn问题是于由Executor(JVM)进程自身内存超过向Yarn申请的内存总量M 所致。由于未出现4.1.1节所述的OOM异常，因此可判定其为 M_1(Overhead)不足，依据Yarn内存使用情况有如下两种方案：

1）如果，M未达到Yarn单个Container允许的上限时，可仅增加M_1，从而增加M；如果，M达到Yarn单个Container允许的上限时，增加 M_1， 降低 M2.

操作方法：在提交脚本中添加 --conf spark.yarn.executor.memoryOverhead=3072(或更大的值，比如4096等) --conf spark.executor.memory = 10g 或 更小的值，注意二者之各要小于15g,否则伸请资源将被yarn拒绝。

2）减少可用的Core的数量 N，使并行任务数减少，从而减少Overhead开销

操作方法：在提交脚本中添加 --executor-cores=3 〈比原来小的值〉 或 --conf spark.executor.cores=3 〈比原来小的值〉

3）增加数量Partition的量也可以在一定程度上减少Overhead开销

〈2〉存在子进程

Spark 应用中Container以Executor（JVM进程）的形式存在，因此根进程为Executor，而Spark 应用向Yarn申请的总资源$M = M_1 + M_2$，都是以Executor（JVM）进程（非进程树）可用资源的名义申请的。申请的资源并非一次性全量分配给JVM使用，而是先为JVM分配初始值，随后内存不足时再按比率不断进行扩容，直致达到Container监控的最大内存使用量M 。当Executor中启动了子进程（调用shell等）时，子进程占用的内存（记为 S） 就被加入Container进程树，此时就会影响Executor实际可使用内存资源（Executor进程实际可使用资源为：M - S），然而启动JVM时设置的可用最大资源为M， 且JVM进程并不会感知Container中留给自己的使用量已被子进程占用，因此，当JVM使用量达到 M - S，还会继续开劈内存空间，这就会导致Executor进程树使用的总内存量大于M 而被Yarn 杀死。

典形场景有： PySpark（Spark已做内存限制，一般不会占用过大内存）、自定义Shell调用

解决方案：

PySpark场景：

1）如果，M未达到Yarn单个Container允许的上限时，可仅增加M_1，从而增加M；如果，M达到Yarn单个Container允许的上限时，增加 M_1， 降低 M2.

2）减少可用的Core的数量 N，使并行任务数减少，从而减少Overhead开销

3） 增加数量Partition的量也可以在一定程度上减少Overhead开销

自定义Shell 场景：（OverHead不足为假象）

4）调整子进程可用内存量，（通过单机测试，内存控制在Container监控内存以内，且为Spark保留内存等留有空间）。

操作方法同4.1.2 <1>中所述