**1. 性能篇**

#### **1.1 建议的性能参数**

**1. 取消偏向锁： -XX:-UseBiasedLocking**

JDK1.6开始默认打开的偏向锁，会尝试把锁赋给第一个访问它的线程，取消同步块上的synchronized原语。如果始终只有一条线程在访问它，就成功略过同步操作以获得性能提升。

但一旦有第二条线程访问这把锁，JVM就要撤销偏向锁恢复之前的状态，如果打开安全点日志，可以看到不少RevokeBiasd的纪录，像GC一样Stop The World的干活，虽然只是很短的停顿，但对于多线程并发的应用，取消掉它反而有性能的提升，所以Cassandra就取消了它。

**2. 加大Integer Cache： -XX:AutoBoxCacheMax=20000**

Integer i=3;这语句有着 int自动装箱成Integer的过程，JDK默认只缓存 -128 ~ +127的Integer 和 Long，超出范围的数字就要即时构建新的Integer对象。设为20000后，我们应用的QPS有足足4%的提升。为什么是2万呢，因为-XX:+AggressiveOpts里也是这个值。详见《Java Integer(-128~127)值的==和equals比较产生的思考》。

**3. 启动时访问并置零内存页面： -XX:+AlwaysPreTouch**

启动时就把参数里说好了的内存全部舔一遍，可能令得启动时慢上一点，但后面访问时会更流畅，比如页面会连续分配，比如不会在晋升老生代时才去访问页面使得GC停顿时间加长。ElasticSearch和Cassandra都打开了它。

**4. SecureRandom生成加速： -Djava.security.egd=file:/dev/./urandom**

此江湖偏方原因是Tomcat的SecureRandom显式使用SHA1PRNG算法时，初始因子默认从/dev/random读取会存在堵塞。额外效果是SecureRandom的默认算法也变成更合适的SHA1了。详见 《SecureRandom的江湖偏方与真实效果》

#### **1.2 可选的性能参数**

**1. -XX:+PerfDisableSharedMem**

Cassandra家的一个参数，一直没留意，直到发生高IO时的JVM停顿。原来JVM经常会默默的在/tmp/hperf 目录写上一点statistics数据，如果刚好遇到PageCache刷盘，把文件阻塞了，就不能结束这个Stop the World的安全点了。

禁止JVM写statistics数据的代价，是jps和jstat 用不了，只能用JMX，而JMX取新老生代的使用百分比还真没jstat方便，VJTools的vjmxcli弥补了这一点。详见《The Four Month Bug: JVM statistics cause garbage collection pauses》

**2. -XX:-UseCounterDecay**

禁止JIT调用计数器衰减。默认情况下，每次GC时会对调用计数器进行砍半的操作，导致有些方法一直温热，永远都达不到触发C2编译的1万次的阀值。

**3. -XX:-TieredCompilation**

多层编译是JDK8后默认打开的比较骄傲的功能，先以C1静态编译，采样足够后C2编译。

但我们实测，性能最终略降2%，可能是因为有些方法C1编译后C2不再编译了。应用启动时的偶发服务超时也多了，可能是忙于编译。所以我们将它禁止了，但记得打开前面的-XX:-UseCounterDecay，避免有些温热的方法永远都要解释执行。

#### **1.3 不建议的性能参数**

**1. -XX:+AggressiveOpts**

一些还没默认打开的优化参数集合, -XX:AutoBoxCacheMax是其中的一项。但如前所述，关键系统里不建议打开。虽然通过-XX:+AggressiveOpts 与 **-**XX:-AggressiveOpts 的对比，目前才改变了三个参数，但为免以后某个版本的JDK里默默改变更多激进的配置，还是不要打开了。

2. JIT Compile相关的参数，函数调用多少次之后开始编译的阀值，内联函数大小的阀值等等，不要乱改。

3. -server，在64位多核的linux中，你想设成**-client**都不行的，所以写了也是白写。

### **2. 内存与GC篇**

#### **2.1 GC策略**

为了稳健，还是8G以下的堆还是CMS好了，G1现在虽然是默认了，但其实在小堆里的表现也没有比CMS好，还是JDK11的ZGC引人期待。

**1.CMS基本写法**

-XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSInitiatingOccupancyOnly

因为我们的监控系统会通过JMX监控内存达到90%的状况，所以设置让它75%就开始跑了，早点开始也能减少Full GC等意外情况(概念重申，这种主动的CMS GC，和JVM的老生代、永久代、堆外内存完全不能分配内存了而强制Full GC是不同的概念)。  
为了让这个设置生效，还要设置**UseCMSInitiatingOccupancyOnly**，否则75％只被用来做开始的参考值，后来还是JVM自己算。

**2. -XX:MaxTenuringThreshold=2**

这是改动效果最明显的一个参数了。对象在Survivor区最多熬过多少次Young GC后晋升到年老代，JDK8里CMS 默认是6，其他如G1是15。

Young GC是最大的应用停顿来源，而YGC后存活对象的多少又直接影响停顿的时间，所以如果清楚Young GC的执行频率和应用里大部分临时对象的最长生命周期，可以把它设的更短一点，让其实不是临时对象的新生代对象赶紧晋升到年老代，别呆着。

用**-XX:+PrintTenuringDistribution**观察下，如果后面几代的大小总是差不多，证明过了某个年龄后的对象总能晋升到老生代，就可以把晋升阈值设小，比如JMeter里2就足够了。

**3. -XX:+ExplicitGCInvokesConcurrent 但不要加-XX:+DisableExplicitGC**

让full gc时使用CMS算法，不是全程停顿，必选。

但像R大说的，System GC是保护机制（如堆外内存满时清理它的堆内引用对象），禁了system.gc() 未必是好事，只要没用什么特别烂的类库，真有人调了总有调的原因，所以不应该加这个烂大街的参数。

**4.** **-XX:+**ParallelRefProcEnabled 和 **-XX:+**CMSParallelInitialMarkEnabled

并行的处理Reference对象，如WeakReference，默认为false，除非在GC log里出现Reference处理时间较长的日志，否则效果不会很明显，但我们总是要JVM尽量的并行，所以设了也就设了。同理还有-XX:+CMSParallelInitialMarkEnabled，JDK8已默认开启，但小版本比较低的JDK7甚至不支持。

**5. ParGCCardsPerStrideChunk**

Linkined的黑科技， 2016版不建议打开，后来发现有些场景的确能减少YGC时间，详见《难道他们说的都是真的》，简单说就是影响YGC时扫描老生代的时间，默认值256太小了，但32K也未必对，需要自己试验。

-XX:+UnlockDiagnosticVMOptions -XX:ParGCCardsPerStrideChunk=1024

#### **2.2 可选的GC参数**

**1. 并发收集线程数**

ParallelGCThreads＝8+( Processor - 8 ) ( 5/8 )；  
ConcGCThreads = (ParallelGCThreads + 3)/4

比如双CPU，六核，超线程就是24个处理器，小于8个处理器时ParallelGCThreads按处理器数量，大于时按上述公式YGC线程数＝18， CMS GC线程数＝5。

CMS GC线程数的公式太怪，也有人提议简单改为YGC线程数的1/2。

一些不在乎停顿时间的后台辅助程序，比如日志收集的logstash，建议把它减少到2，避免在GC时突然占用太多CPU核，影响主应用。

而另一些并不独占服务器的应用，比如旁边跑着一堆sidecar的，也建议减少YGC线程数。

一个真实的案例，24核的服务器，默认18条YGC线程，因为旁边有个繁忙的Service Mesh Proxy在跑着，这18条线程并不能100%的抢到CPU，出现了不合理的慢GC，把线程数降低到12条反而更快了。 所以那些贪心的把YGC线程数＝CPU 核数的，通常弄巧成拙。

**2. -XX:-CMSClassUnloadingEnabled**

在CMS中清理永久代中的过期的Class而不等到Full GC，JDK7默认关闭而JDK8打开。看自己情况，比如有没有运行动态语言脚本如Groovy产生大量的临时类。它有时会大大增加CMS GC的暂停时间。所以如果新类加载并不频繁，这个参数还是显式关闭的好。

**3. -XX:+CMSScavengeBeforeRemark**

默认为关闭，在CMS remark前，先执行一次minor GC将新生代清掉，这样从老生代的对象引用到的新生代对象的个数就少了，停止全世界的CMS remark阶段就短一些。但如果打开了，会让一次YGC紧接着一次CMS GC，使得停顿的总时间加长了。

又一个真实案例，CMS GC的时间和新生代的使用量成比例，新生代较小时很快完成，新生代快满时CMS GC的停顿时间超过2秒，这时候就还是打开了划算。

#### **2.3 不建议的GC参数**

**1. -XX:+UseParNewGC**

用了CMS，新生代收集默认就是它，不用自己设。

**2. -XX:CMSFullGCsBeforeCompaction**  
默认为0，即每次Full GC都对老生代进行碎片整理压缩。Full GC 不同于 老生代75%时触发的CMS GC，只在老生代达到100%，堆外内存满，老生代碎片过大无法分配空间给新晋升的大对象这些特殊情况里发生，所以设为每次都进行碎片整理是合适的，详见此贴里R大的解释。

**3.-XX:+GCLockerInvokesConcurrent**

我们犯过的错，不是所有Concurrent字样的参数都是好参数，加上之后，原本遇上JNI GCLocker只需要补偿YGC就够的，变成要执行YGC ＋ CMS GC了。

#### **2.4 内存大小的设置**

其实JVM除了显式设置的**-Xmx**堆内存，还有一堆其他占内存的地方(堆外内存，线程栈，永久代，二进制代码cache)，在容量规划的时候要留意。

关键业务系统的服务器上内存一般都是够的，所以尽管设得宽松点。

**1. -Xmx, -Xms,**  
堆内存大小，2～4G均可。

**2. -Xmn or -XX:NewSize or -XX:NewRatio**

JDK默认新生代占堆大小的1/3， 个人喜欢把对半分， 因为增大新生代能减少GC的频率，如果老生代里没多少长期对象的话，占2/3通常太多了。可以用-Xmn 直接赋值(等于-XX:NewSize and -XX:MaxNewSize同值的缩写)，或把NewRatio设为1来对半分。

**3. -XX: PermSize=128m -XX:MaxPermSize=512m （JDK7）**  
**-XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=512m（JDK8）**

现在的应用有Hibernate/Spring这些闹腾的家伙AOP之后类都比较多，可以一开始就把初始值从64M设到128M（否则第一次自动扩张会造成大约3秒的JVM停顿），并设一个更大的Max值以求保险。

JDK8的永生代几乎可用完机器的所有内存，同样设一个128M的初始值，512M的最大值保护一下。

#### **2.5 其他内存大小的设置**

**1. -Xss**

在堆之外，线程占用栈内存，默认每条线程为1M（以前是256K）。存放方法调用出参入参的栈，局部变量，标量替换后掉局部变量等，有人喜欢把它设回256k如Cassandra，节约内存并开更多线程，有人则会在遇到错误后把它再设大点，特别是有很深的JSON解析之类的递归调用时。

**2. -XX:SurvivorRatio**

新生代中每个存活区的大小，默认为8，即1/10的新生代 1/(SurvivorRatio+2)，有人喜欢设小点省点给新生代，但要避免太小使得存活区放不下临时对象而被迫晋升到老生代，还是从GC日志里看实际情况了。

**3. -XX:MaxDirectMemorySize**

堆外内存的最大值，默认为Heap区总内存减去一个Survivor区的大小，详见《Netty之堆外内存扫盲篇》，如果肯定用不了这么多，也可以把它主动设小，来获得一个比较清晰内存占用预估值，特别是在容器里。

**4. -XX:ReservedCodeCacheSize**

JIT编译后二进制代码的存放区，满了之后就不再编译，对性能影响很大。 JDK7默认不开多层编译48M，开了96M，而JDK8默认开多层编译240M。可以在JMX里看看CodeCache的占用情况，也可以用VJTools里的vjtop来看，JDK7下默认的48M可以设大点，不抠这么点。

### **3. 监控篇**

JVM输出的各种日志，如果未指定路径，通常会生成到运行应用的相同目录，为了避免有时候在不同的地方执行启动脚本，一般将日志路径集中设到一个固定的地方。

#### **3.1 监控建议配置**

**1. -XX:+PrintCommandLineFlags**

运维有时会对启动参数做一些临时的更改，将每次启动的参数输出到stdout，将来有据可查。  
打印出来的是命令行里设置了的参数以及因为这些参数隐式影响的参数，比如开了CMS后，-XX:+UseParNewGC也被自动打开。

**2. -XX:-OmitStackTraceInFastThrow**

为异常设置StackTrace是个昂贵的操作，所以当应用在相同地方抛出相同的异常N次(两万?)之后，JVM会对某些特定异常如NPE，数组越界等进行优化，不再带上异常栈。此时，你可能会看到日志里一条条Nul Point Exception，而之前输出完整栈的日志早被滚动到不知哪里去了，也就完全不知道这NPE发生在什么地方，欲哭无泪。 所以，将它禁止吧，ElasticSearch也这样干。

#### **3.2 Crash文件**

**1. -XX:ErrorFile**

JVM crash时，hotspot 会生成一个error文件，提供JVM状态信息的细节。如前所述，将其输出到固定目录，避免到时会到处找这文件。文件名中的%p会被自动替换为应用的PID

-XX:ErrorFile=${LOGDIR}/hs\_err\_%p.log

**2. coredump**

当然，更好的做法是生成coredump，从CoreDump能够转出Heap Dump 和 Thread Dump 还有crash的地方，非常实用。

在启动脚本里加上 ulimit -c unlimited或其他的设置方式，如果有root权限，设一下输出目录更好

echo "/{MYLOGDIR}/coredump.%p" > /proc/sys/kernel/core\_pattern

什么？你不知道coredump有什么用？看来你是没遇过JVM Segment Fault的幸福人。

**3. -XX:+HeapDumpOnOutOfMemoryError(可选)**

在Out Of Memory，JVM快死掉的时候，输出Heap Dump到指定文件。不然开发很多时候还真不知道怎么重现错误。

路径只指向目录，JVM会保持文件名的唯一性，叫java\_pid${pid}.hprof。因为如果指向文件，而文件已存在，反而不能写入。

-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=${LOGDIR}/

但在容器环境下，输出4G的HeapDump，在普通硬盘上会造成20秒以上的硬盘IO跑满，也是个十足的恶邻，影响了同一宿主机上所有其他的容器。

#### **3.3 GC日志**

JDK9完全不一样了，这里还是写JDK7/8的配置。

**1.基本配置**

-Xloggc:/dev/shm/gc-myapp.log -XX:+PrintGCDateStamps -XX:+PrintGCDetails

有人担心写GC日志会影响性能，但测试下来实在没什么影响，GC问题是Java里最常见的问题，没日志怎么行。

后来又发现如果遇上高IO的情况，GC时操作系统正在flush pageCache 到磁盘的话，也可能导致GC log文件被锁住，从而让GC结束不了。所以把它指向了/dev/shm 这种内存中文件系统，避免这种停顿，详见《Eliminating Large JVM GC Pauses Caused by Background IO Traffic》

用PrintGCDateStamps而不是PrintGCTimeStamps，打印可读的日期而不是时间戳。

**2. -XX:+PrintGCApplicationStoppedTime**

这是个非常非常重要的参数，它的名字没起好，它除了打印清晰的完整的GC停顿时间外，还可以打印其他的JVM停顿时间，比如取消偏向锁，class 被agent redefine，code deoptimization等等，有助于发现一些原来没想到的问题，建议也加上。如果真的发现了一些不知是什么的停顿，需要打印安全点日志找原因。

**3. -XX:+PrintGCCause**

打印产生GC的原因，比如AllocationFailure什么的，在JDK8已默认打开，JDK7要显式打开一下。

**4. -XX:+PrintPromotionFailure**

打开了就知道是多大的新生代对象晋升到老生代失败从而引发Full GC时的。

**5. GC日志滚动与备份**

GC日志默认会在重启后清空，有人担心长期运行的应用会把文件弄得很大，所以"-XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=1M"的参数可以让日志滚动起来。但真用起来重启后的文件名太混乱太让人头痛，GC日志再大也达不到哪里去，所以我们没有加滚动，并且自行在启动脚本里对旧日志做备份。

#### **3.4 安全点日志**

如果GC日志里有非GC的JVM停顿时间，你得打出安全点日志来知道详情，详见《 JVM的Stop The World，安全点，黑暗的地底世界》

|  |  |
| --- | --- |
| 1 | -XX:+PrintSafepointStatistics -XX:PrintSafepointStatisticsCount=1 -XX:+UnlockDiagnosticVMOptions -XX:-DisplayVMOutput -XX:+LogVMOutput -XX:LogFile=/dev/shm/vm-myapp.log |

#### **3.5 JMX**

|  |  |
| --- | --- |
| 1 | -Dcom.sun.management.jmxremote.port=7001 -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false -Djava.rmi.server.hostname=127.0.0.1 |

以上设置，只让本地的Zabbix之类监控软件通过JMX监控JVM，不允许远程访问。

如果应用忘记了加上述参数，又不想改参数重启服务，可以用VJTools的vjmxcli来救急，它能通过PID直接连入目标JVM打开JMX。