在关键的业务系统里，除了继续追求技术人员最爱的高吞吐与低延时之外，系统的稳定性与出现问题时排查的便捷性也很重要。
这是本文的一个原则，后面也会一次又一次的强调，所以与网上其他的文章略有不同，请调优高手和运维老大们多指引。

前言1，资料

学**开源项目的启动脚本是个不错的主意，比如Cassandra家的， 附送一篇解释它的文章。
JVM调优的"标准参数"的各种陷阱 R大的文章，在JDK6时写的，期待更新。
偶然翻到Linkedin工程师的一篇文章。
更偶然翻到的一份不错的参数列表。

前言2， -XX:+PrintFlagsFinal打印参数值

当你在网上兴冲冲找到一个可优化的参数时，先用-XX:+PrintFlagsFinal看看，它可能已经默认打开了，再找到一个，还是默认打开了...
JDK7与JDK8，甚至JDK7中的不同版本，有些参数值都不一样，所以不要轻信网上任何文章，一切以生产环境同版本的JDK打出来的为准。
经常以类似下面的语句去查看参数，偷懒不起应用，用-version代替。
有些参数设置后会影响其他参数，所以查看时也把它带上。

```
java -server -Xmx1024m -Xms1024m -XX:+UseConcMarkSweepGC -XX:+PrintFlagsFinal -version| grep ParallelGCThreads
```

前言3，关于默认值

JDK8会默认打开-XX:+TieredCompilation多层编译，而JDK7则不会。
JDK7u40以后的版本会默认打开-XX:+OptimizeStringConcat优化字符串

拼接，而之前的则不打开。

对于这些参数，我的建议是顺势而为，JDK在那个版本默认打开不打开总有它的理由。安全第一，没有很好的因由，不要随便因为网上某篇文章的推荐(包括你现在在读的这篇)就去设置。

1. 性能篇

先写一些不那么常见的，后面再来老生常谈。

1.1 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到未锁定线程的状态，详见 JVM的Stop The World，安全点，黑暗的地底世界，可以看到不少RevokeBiasd的纪录，像GC一样，会Stop The World的干活，虽然只是很短很短的停顿，但对于多线程并发的应用，取消掉它反而有性能的提升和延时的极微的缩短，所以Cassandra就取消了它。

1.2 -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int 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从48,000提升到50,000，足足4%的影响。详见Java Integer(-128~127)值的==和equals比较产生的思考

1.3 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，比如不会在晋升新生代到老生代时才去访问页面使得GC停顿时间加长。不过这选项对大堆才会更有感觉一点。

1.4 -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。用此参数可以禁止JVM写statistics数据，代价是jps, jstat 用不了，只能用JMX取数据。有时用JMX取新生代老生代使用百分比还真没jstat方便。详见The Four Month Bug: JVM statistics cause garbage collection pauses

1.5 -Djava.security.egd=file:/dev/./urandom

此江湖偏方原用于Tomcat显式使用SHA1PRNG算法时，初始因子从/dev/random读取导致堵塞。而使用此设置后，额外效果是默认的SecureRandom算法也变成SHA1了。 SHA1PRNG 比 NativePRNG消耗小一半，synchronized的代码少一半，所以没特殊安全要求的话建议用SHA1。详见 SecureRandom的江湖偏方与真实效果

1.6 不建议的参数

1. **-XX:+AggressiveOpts**是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，还是不要了。
2. Linkined那种黑科技，先要解锁VMOptions才能配置的就更不用说了，比如

    -XX:+UnlockDiagnosticVMOptions -XX:
    ParGCCardsPerStrideChunk=32768

3. 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，不要乱改了。
4. **-XX:+UseFastAccessorMethods**，JDK6的优化，据说在多层编译下还慢了，所以是默认关闭的。
5. **-server**，在64位linux中，你想设成-client都不行的，所以写了也是白写。

1.7 可选参数

1. **-Djava.awt.headless=true**，如果服务器上没有屏幕，键盘，鼠标，又需要用到它们的时候，详见在 Java SE 平台上使用 Headless 模式

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，导致有些方法一直是个温热，可能永远都达不到C2编译的1万次的阀值。

3. **-XX:-TieredCompilation**，禁止JDK8默认的多层编译，在某些情况下因为有些方法C1编译后C2不再编译，多层编译反而比C2编译慢，如果发现此情况可进行禁止。

2. GC篇

2.1 GC策略

为了稳健，还是8G以下的堆还是CMS好了，G1的细节实现起来难度太大，从理论提出到现在都做了六七年了。

CMS真正可设的东西也不多，详见JVM实用参数（七）CMS收集器

1.基本配置

-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（留点处理的时间），所以设置让它75%就开始跑了，早点开始也能避免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-XX:+UseCMSInitiatingOccupancyOnly，否则75只被用来做开始的参考值，后面还是JVM自己算。

2. **-XX:MaxTenuringThreshold=2**，这是GC里改动效果最明显的一个参数了。对象在Survivor区熬过多少次Young GC后晋升到年老代，JDK7里看起来默认是6，跑起来好像变成了15。

Young GC是最大的应用停顿来源，而新生代里GC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长期对象赶紧晋升到年老代，别呆着。

用-XX:+PrintTenuringDistribution观察下，如果后面几代都差不多，就可以设小，比如JMeter里是2。而我们的两个系统里一个设了2，一个设了6。

3. **-XX:+ExplicitGCInvokesConcurrent**， 但不要**-XX:+DisableExplicitGC**， 比如Netty之堆外内存扫盲篇，可见禁了system.gc() 未必是好事，只要自己的代码里没有调它，也没用什么特别烂的类库，真有人调了总有调的原因。-XX+ExplicitGCInvokesConcurrent 则在full gc时，并不全程停顿，依然只在ygc和两个remark阶段停顿，详见JVM源码分析之SystemGC完全解读

4. **-XX: ParallelRefProcEnabled** , 默认为false，并行的处理Reference对象，如WeakReference，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。

2.2 GC里不建议设的参数

1. **-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。它会增加CMS remark的暂停时间，所以如果新类加载并不频繁，这个参数还是不开的好。

2. 用了CMS，新生代收集默认就是**-XX:+UseParNewGC**，不用自己设。

3. 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ParallelGCThreads＝18， ConcGCThreads＝5。除了一些不在乎停顿时间的后台辅助程序会特意把它减少，平时不建议动。

4. **-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。如果看到GC日志里remark阶段的时间超长，可以打开此项看看有没有效果，否则还是不要打开了，白白多了次YGC。

5. **-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 前面设置的75%老生代时触发的CMS GC，只在System.gc()，老生代达到100%，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。

2.3 内存大小的设置

这些关于大小的参数，给人感觉是最踏实可控的。

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，再大了注意GC时间。

2. **-Xmn or -XX:NewSize and -XX:MaxNewSize or -XX:NewRatio**, JDK默认新生代占堆大小的1/3， 个人喜欢把对半分， 增大新生代的大小，能减少GC的频率（但也会加大每次GC的停顿时间），主要是看老生代里没多少长期对象的话，占2/3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(但如果想设置新生代比老生代大就只能用-Xmn)。

3. **-XX: PermSize=128m -XX:MaxPermSize=512m （JDK7）**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，并设一个更大的Max值以求保险。

4. **-XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=512m（JDK8）**，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。

2.4 其他内存大小等可选设置

1. **-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要晋升到老生代，还是从GC Log里看实际情况了。
2. **-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，有人喜欢设小点节约内存开更多线程。但反正内存够也就不必要设小，有人喜欢再设大点，特别是有JSON解析之类的递归调用时不能设太小。
3. **-XX:MaxDirectMemorySize**，堆外内存/直接内存的大小，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。
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。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的大小，JDK7下的48M一般够了，也可以把它设大点，反正内存多。

2.5 GC日志

1.基本配置

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

详见JVM实用参数（八）GC日志，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，还是留一份用来排查好。

到后来，又发现如果遇上高IO的情况，如果GC的时候，操作系统正在flush pageCache 到磁盘，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**，它的名字没起好，它除了打印清晰的GC停顿时间外，还可以打印其他的停顿时间，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什么的停顿，再临时加上"-XX:+PrintSafepointStatistics -XX:PrintSafepointStatisticsCount=1"找原因。

3. GC日志默认会在重启后清空，但有人担心长期运行不重启的应用会把文件弄得很大，有"-XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=1M"的参数可以让日志滚动起来。但重启后的文件名太混乱太让人头痛，所以还是不加。

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 -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。

3.2 -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，而真正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。

3.3 coredump与 -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=${MYLOGDIR}/hs_err_%p.log
```

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

3.4 -XX:+HeapDumpOnOutOfMemoryError

在Out Of Memory，JVM快死快死掉的时候，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。如果指向文件，而文件已存在，反而不能写入。

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

3.5 JMX

```
-Dcom.sun.management.jmxremote.port=${MY_JMX_PORT} -
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。

4. 小结

4.1 性能相关

-XX:-UseBiasedLocking -XX:-UseCounterDecay -
XX:AutoBoxCacheMax=20000 -XX:+PerfDisableSharedMem -
XX:+AlwaysPreTouch -Djava.security.egd=file:/dev/./urandom

4.2 内存大小相关(JDK7)

-Xms4096m -Xmx4096m -Xmn2048m -
XX:MaxDirectMemorySize=4096m-XX: PermSize=256m -
XX:MaxPermSize=512m -XX:ReservedCodeCacheSize=240M

4.3 CMS GC 相关

-XX:+UseConcMarkSweepGC -
XX:CMSInitiatingOccupancyFraction=75 -
XX:+UseCMSInitiatingOccupancyOnly -
XX:MaxTenuringThreshold=6 -XX:+ExplicitGCInvokesConcurrent -
XX:+ParallelRefProcEnabled

4.4 GC 日志 相关

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

4.5 异常 日志 相关

-XX:-OmitStackTraceInFastThrow -
XX:ErrorFile=${LOGDIR}/hs_err_%p.log -
XX:+HeapDumpOnOutOfMemoryError -
XX:HeapDumpPath=${LOGDIR}/

4.6 JMX相关