Java 进程 JVM 参数调优指导

内存合理设置

一个 Java 进程在 32 位 OS 上理论可分配 2^32（4G）内存，但事实上由于 OS 限制一般能够可用的内存也就是 1.5G 左右；在 64 位 OS 上，理论可分配 2^64（无限）内存，也就是说只要 OS 可用内存足够，都可以分配给它，无限制。这里很容易产生了两个主观判断：64 位 JVM 性能比 32 位的好，Java 的进程内存越大越好！

可惜，这两个都是错觉。下面是一段，摘自 Sun 官方说明：

"**What are the performance characteristics of 64-bit versus 32-bit VMs?**

Generally, the benefits of being able to address larger amounts of memory come with a small performance loss in 64-bit VMs versus running the same application on a 32-bit VM. ***This is due to the fact that every native pointer in the system takes up 8 bytes instead of 4. The loading of this extra data has an impact on memory usage which translates to slightly slower execution depending on how many pointers get loaded during the execution of your Java program.*** The good news is that with AMD64 and EM64T platforms running in 64-bit mode, the Java VM gets some additional registers which it can use to generate more efficient native instruction sequences. These extra registers increase performance to the point where there is often no performance loss at all when comparing 32 to 64-bit execution speed. The performance difference comparing an application running on a 64-bit platform versus a 32-bit platform on SPARC is on the order of 10-20% degradation when you move to a 64-bit VM. On AMD64 and EM64T platforms this difference ranges from 0-15% depending on the amount of pointer accessing your application performs. "

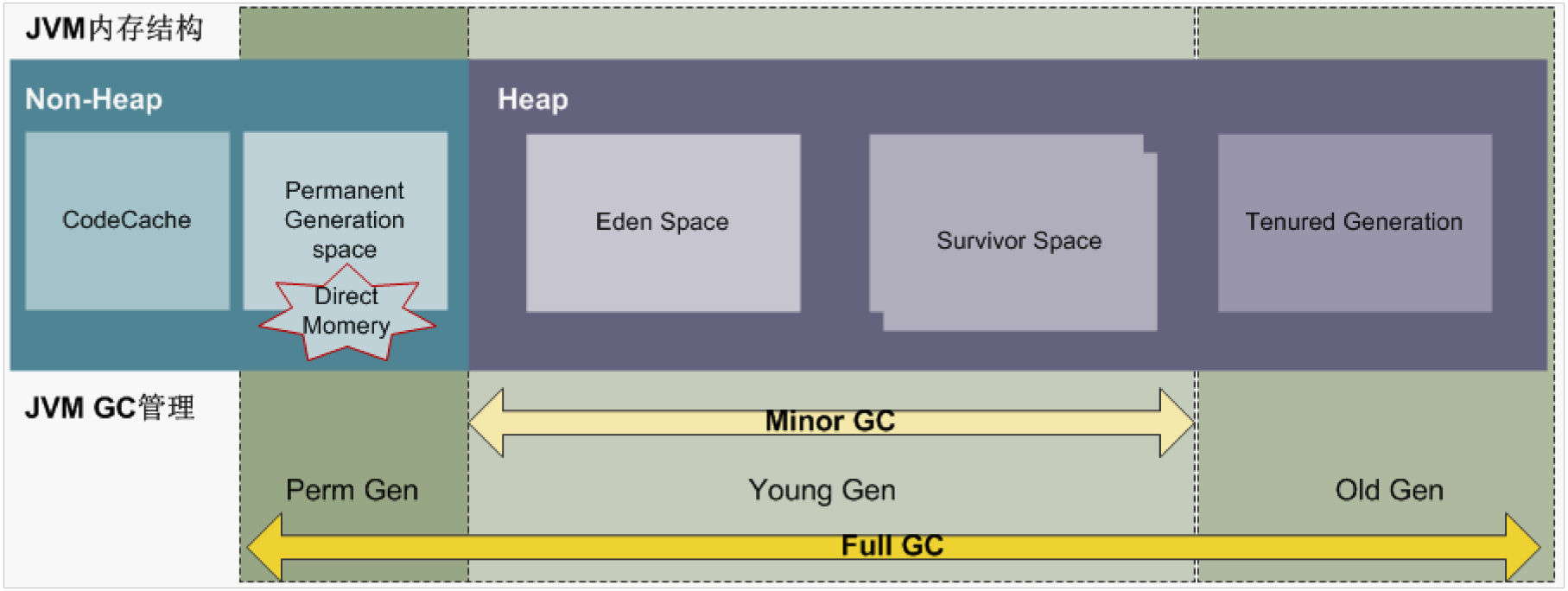
可见，用 64 位 JVM 后是性能是变慢了，在 SUN 自己的 SPARC 芯片性能最大可降低 20%，AMD 最大是 15%，Intel 虽然没数据，但可断定不可能比 32 位快。那是否不要采用 64 位 OS 了？这不是本文关注的范围，这里就不讨论了。但是我们有一个选择就是，在 64 位 OS 下，可以使用 32 位的 JVM。这个开关是：

```
Usage: java [-options] class [args...]
           (to execute a class)
   or  java [-options] -jar jarfile [args...]
           (to execute a jar file)

where options include:
    -d32          use a 32-bit data model if available

    -d64          use a 64-bit data model if available
    -server       to select the "server" VM
                  The default VM is server.
...
```

注意的是，由于没有具体测试数据支持，这里并不是要求大家在 64 位 OS 环境下不要采用 64 位 JVM，只是建议如果你的应用性能要求很高，而调优又实在无计可施的时候，可换成 32 位 JVM 试试看而已 O(∩_∩)O~。

显然，JVM 内存过大对寻址计算和垃圾回收周期都不是什么好事，而且造成浪费。例如一台普通的 PC 服务器内存资源只要 8G，光跑 Java 进程就有 10 几个，可见资源有限，合理分配才是关键。那么如何设置 JVM 参数，让进程内存和性能都达到

最优呢？这需要对自己应用的容量有个估算（如：正常时、最大并发时，消耗多少线程和内存等），设置多组参数进行反复测试，通过 JDK1.6 自带的 VisualVM 观察，得出最优结果。这里涉及到 Java JVM 内存管理机制有一定的认识。



从上示意图（仅是结构组成示意，图上各组件比例只是方便绘图，不代表真实比例）可见，从一个 JVM 进程内存分两大部分：

- Non-Heap

各中文技术论坛对此部分说法有点众说纷纭，我们还是参考官方的吧：

“Non-heap memory includes a method area shared among all threads and memory required for the internal processing or optimization for the Java VM. It stores per-class structures such as a runtime constant pool, field and method data, and the code for methods and constructors. The method area is logically part of the heap but, depending on the implementation, a Java VM may not garbage collect or compact it. Like the heap memory, the method area may be of a fixed or variable size. The memory for the method area does not need to be contiguous.

In addition to the method area, a Java VM may require memory for internal processing or optimization which also belongs to non-heap memory. For example, the Just-In-Time (JIT) compiler requires memory for storing the native machine code translated from the Java VM code for high performance”

CodeCache	代码缓冲区，用于 JIT 编译和保存本地代码(Native code)所需的内存。通过“-XX:ReservedCodeCacheSize”参数可以指定大小，默认值是 48M；如果你的应用用到了 JNI 技术存在较多的调用本地代码的情况，可考虑把此参数调大一点，如：-XX:ReservedCodeCacheSize=32m；另外，值得注意的是，CodeCache 分配的内存，GC 是不会管理的，也就是说多大就是多大永远不会被垃圾回收。所以，盲目设大此参数，只会浪费内存。

Permanent Generation Space	在 Non-Heap 中，除了 CodeCache 外所有的如类加载信息、类结构信息、类常量、方法等等，都放在这个 Perm 中，从垃圾管理角度来说，这个区属于持久代。所谓持久代，就是 JVM 把那些持久不变东西放在这里，例如：类定义元数据、常量数据等等。当然这也不是绝对的，特别是动态加载类的时候，还是有可能产生垃圾的。可通过“-XX:PermSize”和“-XX:MaxPermSize”两个参数调整。默认最大是 64M。对于一个应用来而言，一般来说，用到类及其相关的类库是固定的，这也意味着 Perm 大小只要设置满足这些类库正常装载就可以了，其大小固定不变。但是，如果你的应用存在类库的动态加载（例如，用 cglib 类库），就要根据你的应用具体情况来设当调整 MaxPermSize 大小了。这个内存区平时是不会进行垃圾回收的，只有 JVM Full GC 时才会进行垃圾回收。
Direct Memory	这个区比较特殊，官方文档没有过多论述。由于它也是在 Full GC 的时候才能回收，我们也就把它归为 Perm Gen 的一部分。使用 java.nio.ByteBuffer 类的 allocateDirect 方法分配的内存，NIO 基本都是采取 direct momery 来交互数据（详见：http://docs.oracle.com/javase/6/docs/api/java/nio/ByteBuffer.html），它可以通过参数“-XX:MaxDirectMemorySize”来指派最大值；如果“MaxDirectMemorySize”没有设置，默认值是最大可用堆空间（**Xmx-survivor**）。另外，每次 ByteBuffer.allocateDirect 方法时，都会判

	断是否达到最大值，如达到则会自己调用 System.gc()清理内存，如果用户添加了 JVM 参数 -XX:+DisableExplicitGC。这个调用会变成空调用，就会出现内存溢的风险。 所以对于采取 NIO 通讯的应用 Direct Momory 大小值得关注，建议合理手动配置 “-XX:MaxDirectMemorySize”值以免 directMemory 与 Xmx 值一样，造成没必要的内存浪费。若进程由于 directMemory 大小影响进程不稳定，建议此进程采取不定时调用 System.gc()来尝试回收 directMemory 空间。

- Heap

官方对 Heap 区的定义是：

“Heap memory is the runtime data area from which the Java VM allocates memory for all class instances and arrays.The heap may be of a fixed or variable size. The garbage collector is an automatic memory management system that reclaims heap memory for objects.”

说白就是我们创建的对象实例和计算数据都存在 Heap 区，是我们 JVM 内存调优的重点关注的区域。为了内存管理的需要，Heap 分成：Eden、Survivor 和 Tenured 三种类型区域，其基本的管理过程是：创建的新对象开始放在 Eden 区，Minor GC 后剩余的对象放入 Survivor；Survivor 里面对象过了一段时间好，依然没有被回收的话，那么这些对象就放入 Tenured 区；当 Tenured 区满了以后，就会触发 Full GC 来全局回收空间。

从 GC 管理角度，Heap 分成 Young Gen（年轻代），它包括 1 个 Eden 和 2 个 Survivor 区，建立年轻代的目的是尽可能快的回收那些生命周期短的对象，所以针对这个代，特别建立一个 Minor GC 来专门对待；另外一个就是 Old Gen（年老代），它包括 1 个 Tenured 区，那些在年轻代经过多次 Minor GC 就不能回收的对象就往年老代扔，等年老代的 Tenured 区满了，JVM 就会进行 Full GC 处理。值得说明的是，Full GC 作用范围包括：年轻代、年老代和 Non-Heap 的持久代（Perm Gen）。由于 Full GC 一般较为耗时（内存越大，GC 耗时越多），而且在 GC 过程中整个 JVM 会临时终止服务，所以***为了高性能，合理调整年轻代、年老代的大小，让对象尽量在年轻代就被 Minor GC 回收，减少 Full GC 出现的机会***。

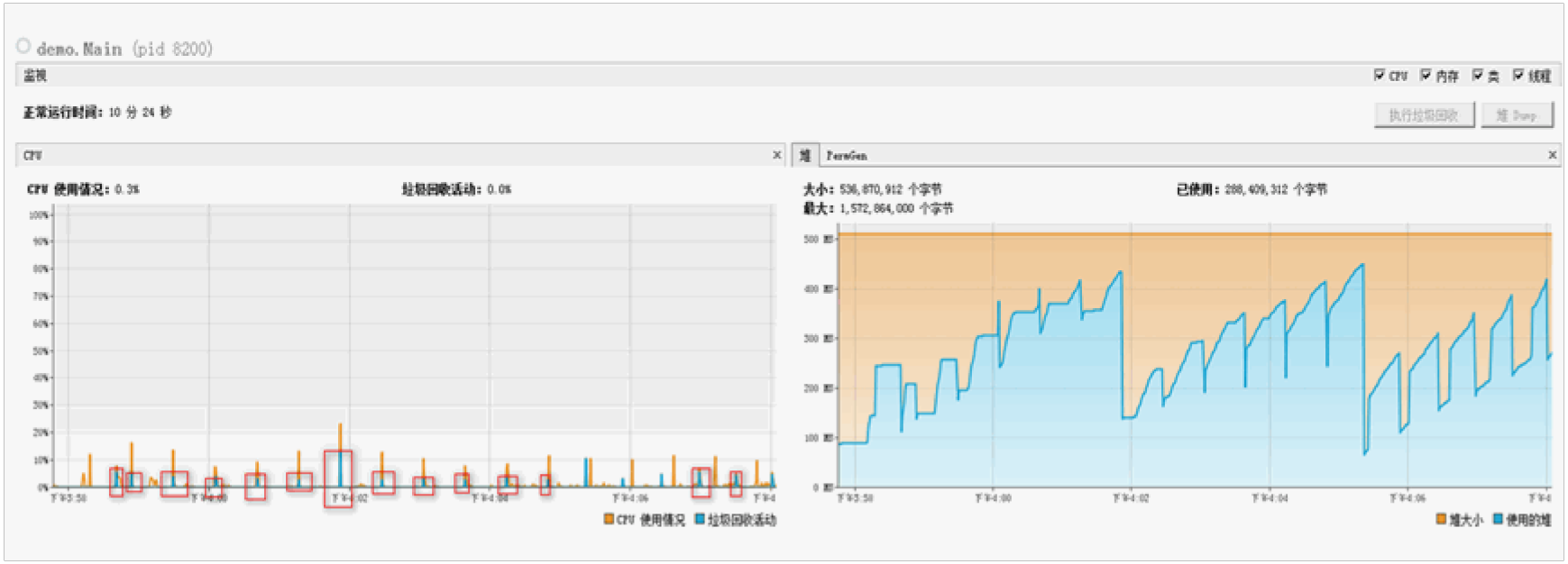
针对 Heap 调优 JVM 参数主要有：

参数	说明
-Xms	Heap Size 最小值，太小内存不够，JVM 性能会浪费在申请内存上，最好设置为应用正常时的消耗的内存大小

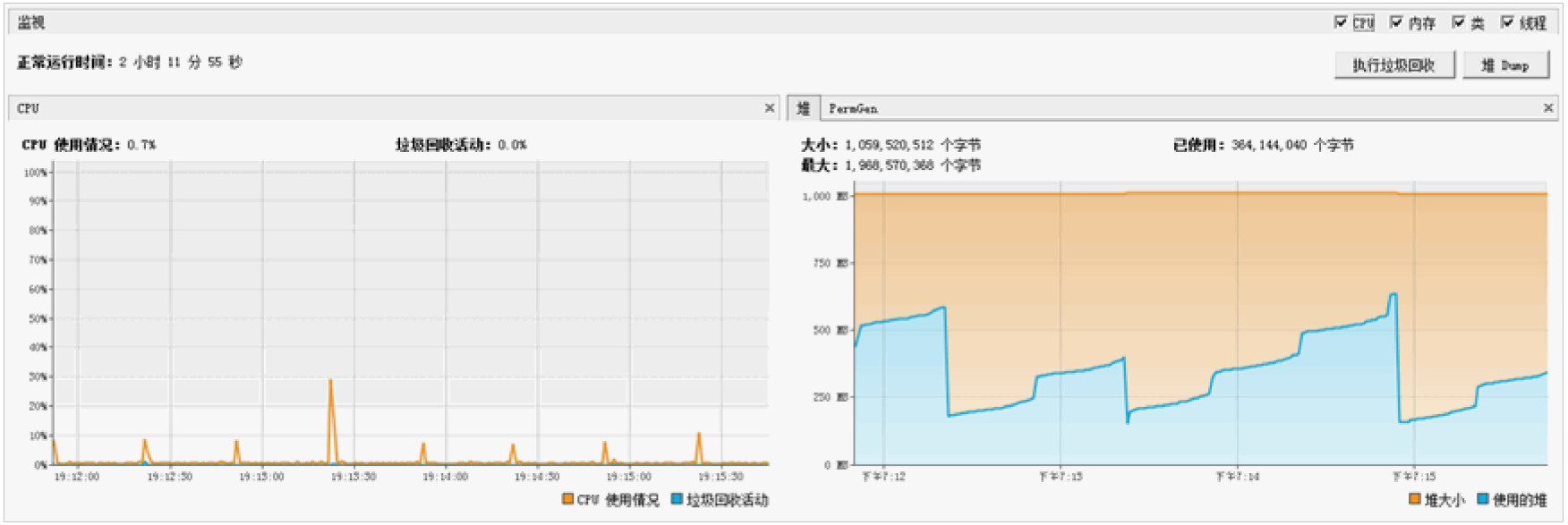
	有些文章推荐把 Xms 值配成与 Xmx 相等，说是可以避免每次垃圾回收完成后 JVM 重新分配内存带来的性能消耗。 不过这里也是值得探讨的。把内存一下子弄成最大值，相当于一下子掩埋了应用的特点，因为大多数正常情况下，应用是无需开辟这么多内存来工作的，这造成 OS 资源浪费，大内存空间无疑也增加了每次 GC 的时间，从而降低应用的性能；另外 JVM 参数“-XX:MinHeapFreeRatio”默认是打开的，是 40%。此参数会在 GC 回收垃圾后，保证重新分配 40%的可用内存给应用程序使用，这个值已经相当大了，所以设置 Xms=Xmx 就显得没有多少必要了。
-Xmx	Heap Size 最大值，太大浪费内存，计算寻址过大降低性能。设置为应用最大峰值时所消耗的内存略大为佳
-Xmn	设置 YoungGen（年轻代的大小），OldGen=Xmx-Xmn。年轻代大小很关键，如果过大，可能导致 minor gc 耗时过长，影响性能；如果过小，新创建的对象放不下，导致拷贝到年老区的动作频繁，同样影响性能。另外，我们还希望对象尽量在年轻代 minor gc 过程中就回收，以免堆积在年老代中，靠 full gc 来回收。我们推荐值是 Xmn=1/3Xmx，就是占 heap 的 1/3。当然这也是推荐值，每个应用的特点不一样，有些应用缓存数据多，每次请求处理过程中创建的对象少，那么它的年轻代可设置小点；若每次请求需要创建较多的对象，那么年轻代就设置大点。最好设置多组参数，通过 VisualVM（本地直接连上去，远程需要先用-Dcom.sun.management.jmxremote.port=9090 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false 参数打开，才

能连上去）等工具来观察，找出最佳值。

举例：年轻代设置不当，容易导致频繁的 gc，工作 cpu 与 gc cpu 叠加，影响性能，见下图



设置得当的参考下图：

	
-XX:NewSize	设置年轻代初始值
-XX:MaxNewSize	设置年轻代的最大值，显然“-XX:NewSize”和“-XX:MaxNewSize”相对-Xmn 来说更加灵活。由于年轻代的大小很关键，强行设置一个大小，有时候不能反映应用的特点。因为应用它的场景也是多变的，反映在年轻代就是有时希望小点，有时希望大点，所以配置一个范围，让 JVM 根据应用特点自动设置不失为一种不错的做法。
-XX:NewRatio	设置年轻代和年老代的比值。如:为 3，表示年轻代与年老代比值为 1：3，年轻代占整个年轻代年老代和的 1/4 通过比例来设置，可以看作是 Xmn 参数设置的另外形式。
-XX:SurvivorRatio	年轻代中 Eden 区与两个 Survivor 区的比值。注意 Survivor 区有两个。如：3，表示 Eden：Survivor=3：2，一个 Survivor

	区占整个年轻代的 1/5。这个参数是针对年轻代的内存分配进行调优。我们知道，新创建的对象会放入 Eden 区中，minor gc 后不能回收的对象，由 Eden 区移到 Survivor 区。如果你的应用对这个过程性能也是很敏感的话，那么 Eden 和 Survivor 的两个内存区的大小比例就很关键了。

上面综合论述了 JVM 的 Non-Heap 和 Heap 构成及其相关的调优参数，这里给出两个配置示例，供大家参考：

- 配置 1

配置	java –server -Xms512m -Xmx1024m -Xmn342m -XX:MaxPermSize=80m -XX:MaxDirectMemorySize=256m
说明	-server，采取 Server 模式执行 -Xms512m，Heap 内存最小大小 512M（JVM 初始化会分配 512M 内存） -Xmx1024m，Heap 内存最大值是 1024m -Xmn342m，Heap 内存中年轻代的大小是 342m，为 Xmx 的 1/3 -XX:MaxPermSize=80m，Non-Heap 内存中的持久代最大值 80m -XX:MaxDirectMemorySize=256m，Non-Heap 中直接内存最大值 256m，应用使用 NIO，要考虑管理直接内存大小

■ 配置 2

配置	java –server -Xms512m -Xmx1024m -XX:NewSize=291m -XX:MaxNewSize=391m -XX:MaxPermSize=80m -XX:MaxDirectMemorySize=256m
说明	基本配置与配置 1 一致，只是调整了年轻代的配置策略，指定一个范围，让 JVM 自己管理。 -XX:NewSize=291m，年轻代初始化大小 291m -XX:MaxNewSize=391m，年轻代最大值 391m

垃圾收集器选择

从上面的章节的分析，我们知道 JVM 的垃圾回收有 Minor GC 和 Full GC 两个作用范围不一样的 GC 类型行为。JVM 执行 GC 究竟是如何识别哪些对象是垃圾以及如何实现垃圾回收的呢？这涉及到各种 GC 算法，如：基本回收策略、针对各个内存代收集方式等等，详细请参考，本文只针对 JVM 公开的垃圾收集器参数进行讨论。

JVM 提供了 3 中垃圾收集器：

垃圾收集器	说明
串行收集器	串行收集使用单线程处理所有垃圾回收工作，实现容易，效率比较高。但是无法使用多处理器的优势，所以此收集适合单处理器机器。当然，此收集器也可以用在小数据量（100M 左右）情况下的多处理器机器上。 用“-XX:+UseSerialGC”参数启动
并行收集器	并行收集使用多线程处理垃圾回收工作，速度快，效率高。在多 CPU 能体现优势。 对年轻代进行并行垃圾回收，因此可以减少垃圾回收时间。使用“-XX:+UseParallelGC”参数启动。JDK1.6 年老代默认使用单线程进行垃圾回收 ,可以使用“-XX:+UseParallelOldGC”参数让年老代也使用并行收集器。 使用-XX:ParallelGCThreads=<N>设置并行垃圾回收的线程数。此值可以设置与机器处理器数量相等。 相关配置还有： （1）最大垃圾回收暂停:指定垃圾回收时的最长暂停时间，通过-XX:MaxGCPauseMillis=<N>指定。<N>为毫秒.如果指定了此值的话，堆大小和垃圾回收相关参数会进行调整以达到指定值。设定此值可能会减少应用的吞吐量。 （2）吞吐量:吞吐量为垃圾回收时间与非垃圾回收时间的比值，通过-XX:GCTimeRatio=<N>来设定，公式为 1/（1+N）。例如，-XX:GCTimeRatio=19 时，表示 5%的时间用于垃圾回收。默认情况为 99，即 1%的时间用于垃圾

	回收。
并发收集器	对串行收集和并行收集来说，在执行 GC 时，会出现所谓的“Stop the world”，也就是 JVM 会临时中止，只有 GC 程序运行。如果你应用无法接受，那么可以使用并发收集器。“-XX:+UseConcMarkSweepGC”参数启动。 另外，使用并发收集器值得注意时。由于在垃圾回收过程中，不中止，这也意味着边回收，程序会生产新垃圾。这可能出现一个问题，垃圾还没有回收完，堆就消耗满，结果出现“Concurrent Mode Failure”错误。解决这个问题，通过“-XX:CMSInitiatingOccupancyFraction”参数来设置还有多少剩余堆空间时开始执行并发收集。

关于 GC 收集器优化选择，***JVM1.5 以后就变得比较智能(GC ergonomics)根据应用和 OS 特点来合理选择相关的 GC 收集器了，如果你观察你应用在默认的情况下，性能能满足要求，就没有必要考虑 GC 收集器选择优化了***。一般原则是吞吐量优先的应用选择并行收集器，响应时间优先的应用选择并发收集器。但这两种应用本身的界定有时候也是含糊不清的，还是还得具体问题具体分析。

下面给出配置示例给大家参考：

- 配置 1

配置	java -server -Xms512m -Xmx1024m -Xmn342m -XX:MaxPermSize=80m -XX:MaxDirectMemorySize=256m

	-XX:+UseParallelGC -XX:ParallelGCThreads=2
说明	-XX:+UseParallelGC，年轻代采取并行垃圾收集器（年老代还是使用串行收集器） -XX:+UseParallelGC，配置并行收集器的线程数，即：同时有多少个线程进行垃圾回收工作。 如果 OS 只有一个 JVM 的话，此值比较优的配置是与处理器数目相等；但 OS 如果存在多个 JVM 的话，就要考虑这 多个 JVM 之间在垃圾回收是竞争 CPU 资源的情况。存在这样的场景的话，一般设置为 2 就可以了。

- 配置 2

配置	java -server -Xms512m -Xmx1024m -Xmn342m -XX:MaxPermSize=80m -XX:MaxDirectMemorySize=256m -XX:+UseParallelGC -XX:ParallelGCThreads=2 -XX:+UseParallelOldGC
说明	-XX:+UseParallelOldGC，年老区也采取并行垃圾收集器处理

关于线程

在 Java 中一个线程与一个线程栈（Stack）对应，堆（Heap）为所有线程共享的。栈可以看作是执行计算逻辑单元，里面存储的都是与当前计算逻辑相关的临时数据，它包括：局部变量、运行状态、方法返回值等等，而堆存储是对象数据。如果栈中用来计算的临时数据大于 Java 默认分配的大小（JDK1.4 以前（含 1.4）为 256kb，JDK1.5 以后为 1M），就会抛出 java.lang.StackOverflowError 异常错误。通过“-Xss”参数来调整每个线程栈的大小。

对一个 Java 进程的所占用的资源来说 n*stack 与 heap 是大小竞争的。也就是减少 Heap 大小或减少“-Xss”大小都可以增加进程所支持的线程数量。如果你的应用是一个消息中间件或代理服务器的话，线程一般只负责消息传递的工作，其在 Stack 所用到数据是有限的，此时可以调小“-Xss”值，如“-Xss128k”，这样进程支持线程更多，支持的并发量更大。

在 JVM 涉及到线程的调优场景是：在高并发下，容易出现多线程同时创建对象的情况，我们知道这些新对象都是在 Heap 的年轻代的 Eden 区中被创建的，然而在每个对象创建的过程中，内存是被加锁的，这就意味着在一个拥有多 CPU 的 OS 上，多线程并发创建对象会造成竞争，从而影响性能。

JVM 提供的解决方案是，允许每个线程拥有一小段的 Eden 私有区（**T**hread **L**ocal **A**llocation **B**uffer），相关参数如下：

-XX:+UseTLAB，启动上述功能，默认启动（JDK1.4 及使用-client 参数的默认不开启）

-XX:TLABSize=<size in kb>，设置 buffer 的大小，由于 JDK1.5 以上的 JVM 有所谓的 GC ergonomics 智能，这个大小 JVM 会根据每个线程具体情况自动设置。如果非要手动调整的话，建议你先打开“-XX:+PrintTLAB”分析后，再作决定。

Java 进程的调优需要在性能、可靠和可用性三者之间取得平衡，对 JVM 调优而已，要确保内存合理分配，GC 回收与业务工作错开，尽量减少 GC 对正常业务的影响。编写一个后台程序，在系统业务空闲时，进行 Full GC 回收资源是个有效的策略。

参考资料

http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html

http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html

http://www.google.com.hk/search?q=jvm%B5%F7%D3%C5

http://www.oracle.com/technetwork/java/hotspotfaq-138619.html#64bit_performance

欢迎交流：余浩东，yuhaodong@gmail.com

<the end>