2017-07-24

JVM垃圾回收算法及回收器详解

引言

本文主要讲述JVM中几种常见的垃圾回收算法和相关的垃圾回收器，以及常见的和GC相关的性能调优参数。

JVM系列文章

1. JVM类加载机制详解
2. JVM内存模型解析

GC Roots

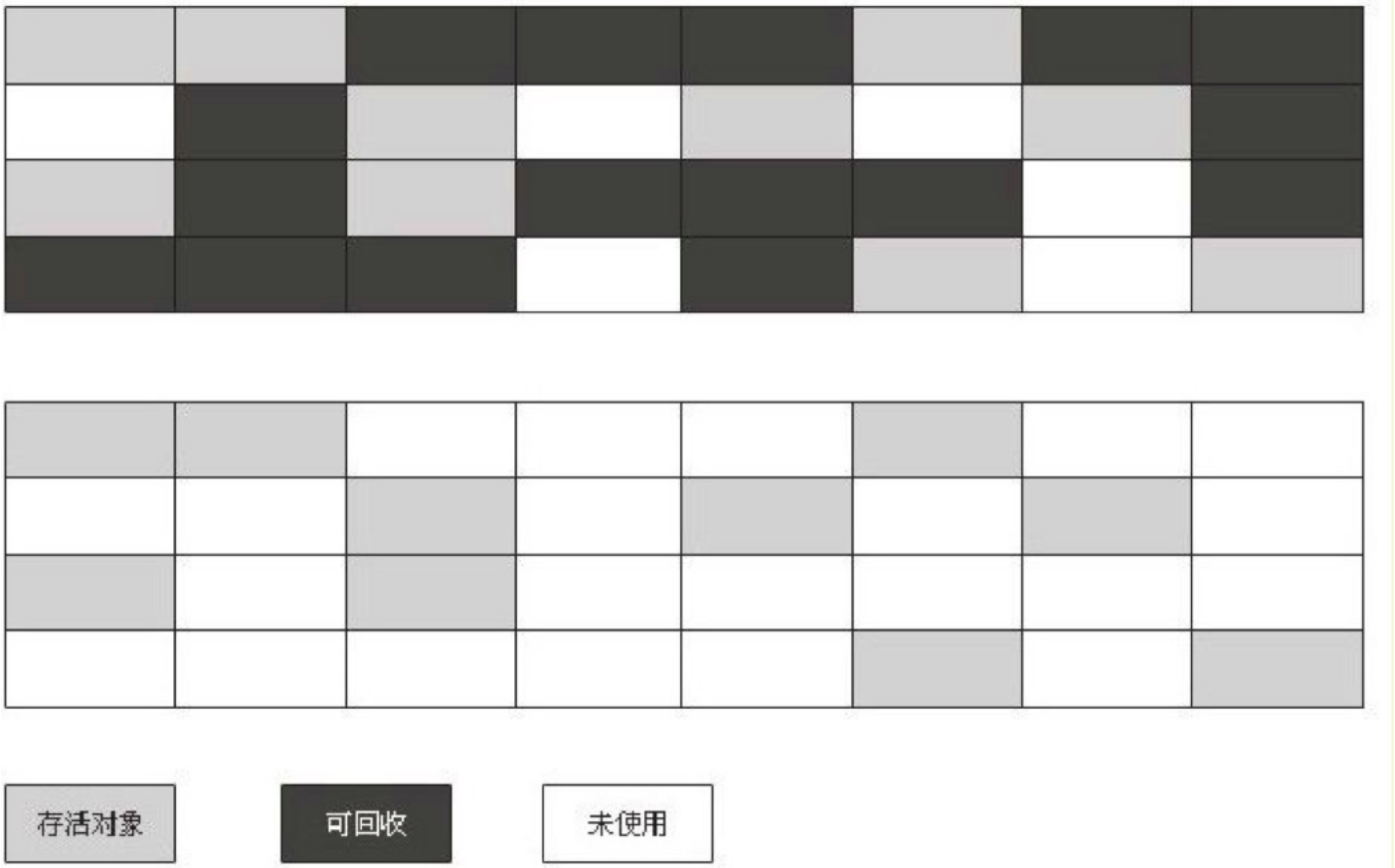
我们先来了解一下在Java中是如何判断一个对象的生死的，有些语言比如Python是采用引用计数来统计的，但是这种做法可能会遇见循环引用的问题，在Java以及C#等语言中是采用GC Roots来解决这个问题。如果一个对象和GC Roots之间没有链接，那么这个对象也可以被视作是一个可回收的对象。

Java中可以被作为GC Roots中的对象有：

1. 虚拟机栈中的引用的对象。
2. 方法区中的类静态属性引用的对象。
3. 方法区中的常量引用的对象。
4. 本地方法栈（jni）即一般说的Native的引用对象。

标记清除

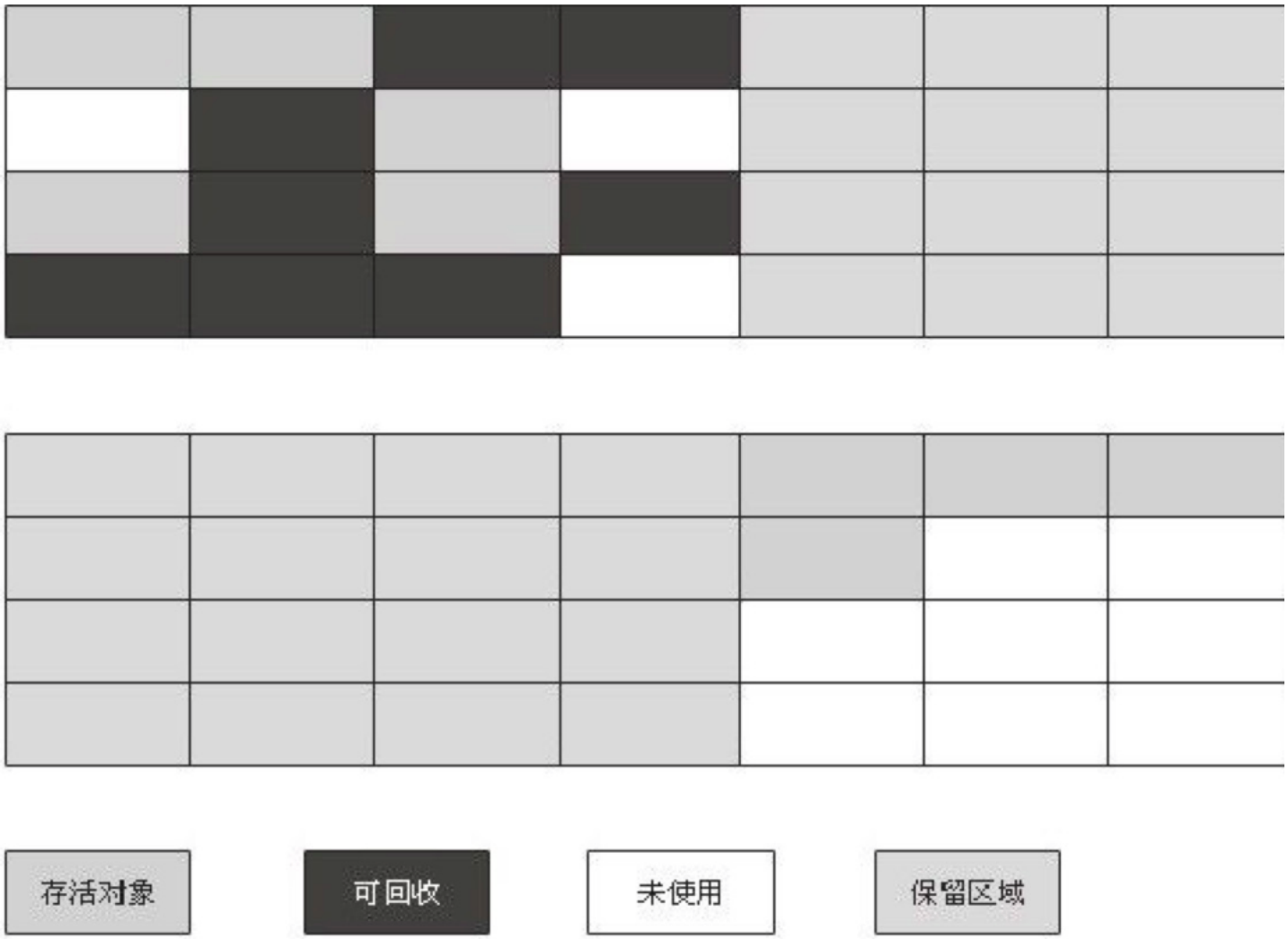
标记-清除算法将垃圾回收分为两个阶段：标记阶段和清除阶段。在标记阶段首先通过根节点，标记所有从根节点开始的对象，未被标记的对象就是未被引用的垃圾对象。然后，在清除阶段，清除所有未被标记的对象。标记清除算法带来的一个问题是会存在大量的空间碎片，因为回收后的空间是不连续的，这样给大对象分配内存的时候可能会提前触发full gc。



标记清除

复制算法

将现有的内存空间分为两快，每次只使用其中一块，在垃圾回收时将正在使用的内存中的存活对象复制到未被使用的内存块中，之后，清除正在使用的内存块中的所有对象，交换两个内存的角色，完成垃圾回收。

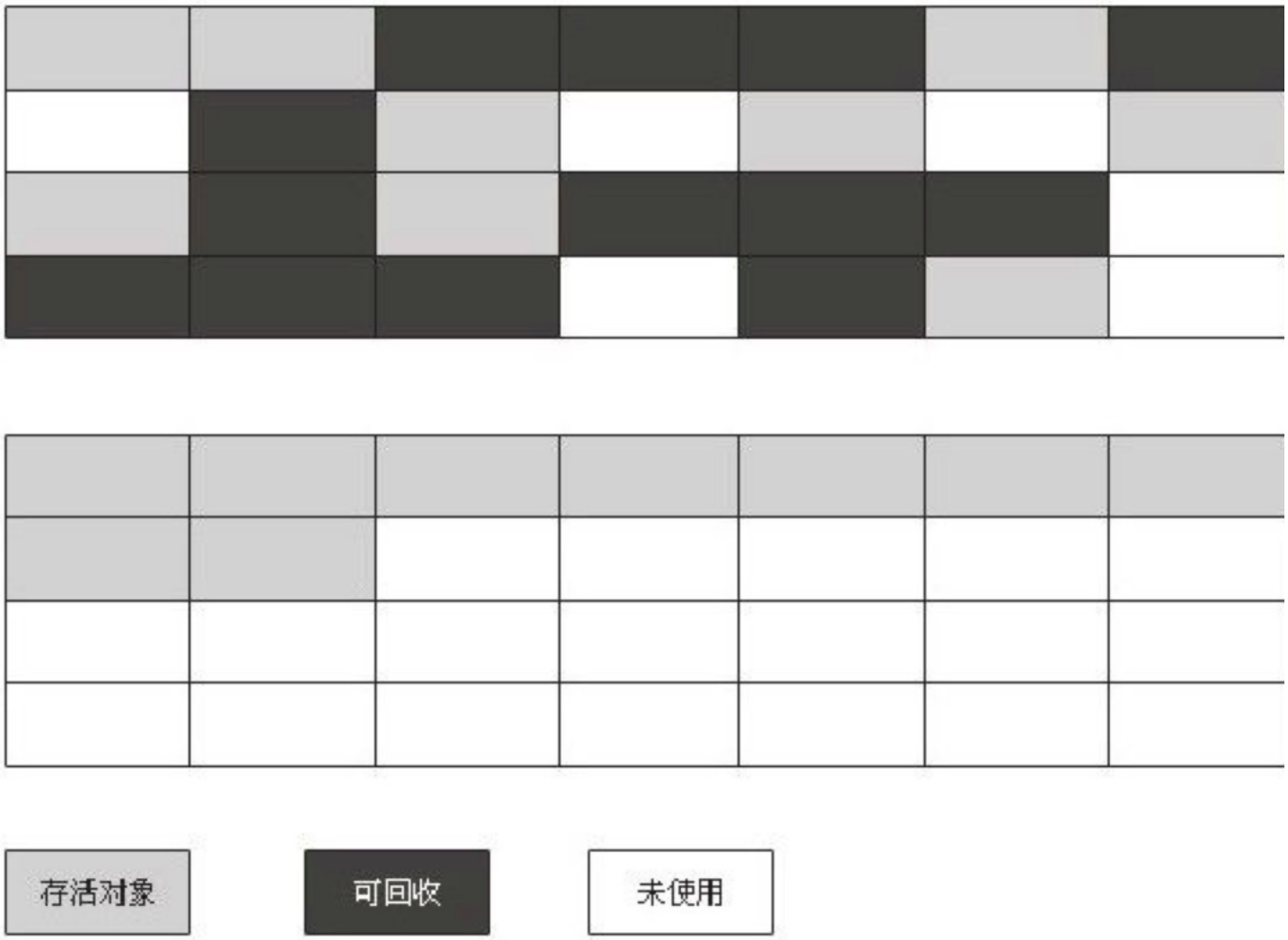


复制算法

现在的商业虚拟机都采用这种收集算法来回收新生代，IBM研究表明新生代中的对象98%是朝夕生死的，所以并不需要按照1:1的比例划分内存空间，而是将内存分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中的一块Survivor。当回收时，将Eden和Survivor中还存活着的对象一次性地拷贝到另外一个Survivor空间上，最后清理掉Eden和刚才用过的Survivor的空间。HotSpot虚拟机默认**Eden和Survivor的大小比例是8:1(可以通过-SurvivorRattio来配置)**，也就是每次新生代中可用内存空间为整个新生代容量的90%，只有10%的内存会被“浪费”。当然，98%的对象可回收只是一般场景下的数据，我们没有办法保证回收都只有不多于10%的对象存活，当Survivor空间不够用时，需要依赖其他内存（这里指老年代）进行分配担保。

标记整理

复制算法的高效性是建立在存活对象少、垃圾对象多的前提下的。这种情况在新生代经常发生，但是在老年代更常见的情况是大部分对象都是存活对象。如果依然使用复制算法，由于存活的对象较多，复制的成本也将很高。



标记整理

标记-压缩算法是一种老年代的回收算法，它在标记-清除算法的基础上做了一些优化。首先也需要从根节点开始对所有可达对象做一次标记，但之后，它并不简单地清理未标记的对象，而是将所有的存活对象压缩到内存的一端。之后，清理边界外所有的空间。这种方法既避免了碎片的产生，又不需要两块相同的内存空间，因此，其性价比比较高。

增量算法

增量算法的基本思想是，如果一次性将所有的垃圾进行处理，需要造成系统长时间的停顿，那么就可以让垃圾收集线程和应用程序线程交替执行。每次，垃圾收集线程只收集一小片区域的内存空间，接着切换到应用程序线程。依次反复，直到垃圾收集完成。使用这种方式，由于在垃圾回收过程中，间断性地还执行了应用程序代码，所以能减少系统的停顿时间。但是，因为线程切换和上下文转换的消耗，会使得垃圾回收的总体成本上升，造成系统吞吐量的下降。

垃圾回收器

Serial收集器

Serial收集器是最古老的收集器，它的缺点是当Serial收集器想进行垃圾回收的时候，必须暂停用户的所有进程，即stop the world。到现在为止，它依然是虚拟机运行在client模式下的默认新生代收集器，与其他收集器相比，对于限定在单个CPU的运行环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾回收自然可以获得最高的单线程收集效率。

Serial Old是Serial收集器的老年代版本，它同样是一个单线程收集器，使用”标记- 整理“算法。这个收集器的主要意义也是被Client模式下的虚拟机使用。在Server模式下，它主要还有两大用途：一个是在JDK1.5及以前的版本中与Parallel Scanvenge收集器搭配使用，另外一个就是作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure的时候使用。

通过指定-UseSerialGC参数，使用Serial + Serial Old的串行收集器组合进行内存回收。

ParNew收集器

ParNew收集器是Serial收集器新生代的多线程实现，注意在进行垃圾回收的时候依然会stop the world，只是相比较Serial收集器而言它会运行多条进程进行垃圾回收。

ParNew收集器在单CPU的环境中绝对不会有比Serial收集器更好的效果，甚至由于存在线程交互的开销，该收集器在通过超线程技术实现的两个CPU的环境中都不能百分之百的保证能超越Serial收集器。当然，随着可以使用的CPU的数量增加，它对于GC时系统资源的利用还是很有好处的。它默认开启的

收集线程数与CPU的数量相同，在CPU非常多（譬如32个，现在CPU动辄4核加超线程，服务器超过32个逻辑CPU的情况越来越多了）的环境下，可以使用-XX:ParallelGCThreads参数来限制垃圾收集的线程数。

-UseParNewGC: 打开此开关后，使用ParNew + Serial Old的收集器组合进行内存回收，这样新生代使用并行收集器，老年代使用串行收集器。

Parallel Scavenge收集器

Parallel是采用复制算法的多线程新生代垃圾回收器，似乎和ParNew收集器有很多的相似的地方。但是Parallel Scanvenge收集器的一个特点是它所关注的目标是吞吐量(Throughput)。所谓吞吐量就是CPU用于运行用户代码的时间与CPU总消耗时间的比值，即吞吐量=运行用户代码时间 / (运行用户代码时间 + 垃圾收集时间)。停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能够提升用户的体验；而高吞吐量则可以最高效率地利用CPU时间，尽快地完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

Parallel Old收集器是Parallel Scavenge收集器的老年代版本，采用多线程和”标记- 整理”算法。这个收集器是在jdk1.6中才开始提供的，在此之前，新生代的Parallel Scavenge收集器一直处于比较尴尬的状态。原因是如果新生代Parallel Scavenge收集器，那么老年代除了Serial Old(PS MarkSweep)收集器外别无选择。由于单线程的老年代Serial Old收集器在服务端应用性能上的”拖累“，即使使用了Parallel Scavenge收集器也未必能在整体应用上获得吞吐量最大化的效果，又因为老年代收集中无法充分利用服务器多CPU的处理能力，在老年代很大而且硬件比较高级的环境中，这种组合的吞吐量甚至还不一定有ParNew加CMS的组合”给力“。直到Parallel Old收集器出现后，”吞吐量优先“收集器终于有了比较名副其实的应用，在注重吞吐量及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge加Parallel Old收集器。

-UseParallelGC: 虚拟机运行在Server模式下的默认值，打开此开关后，使用Parallel Scavenge + Serial Old的收集器组合进行内存回收。-UseParallelOldGC: 打开此开关后，使用Parallel Scavenge + Parallel Old的收集器组合进行垃圾回收

CMS收集器

CMS(Concurrent Mark Swep)收集器是一个比较重要的回收器，现在应用非常广泛，我们重点来看一下，CMS一种获取最短回收停顿时间为目标的收集器，这使得它很适合用于和用户交互的业务。从名字(Mark Swep)就可以看出，CMS收集器是基于**标记清除算法**实现的。它的收集过程分为四个步骤：

1. 初始标记(initial mark)
2. 并发标记(concurrent mark)
3. 重新标记(remark)
4. 并发清除(concurrent sweep)

注意初始标记和重新标记还是会stop the world，但是在耗费时间更长的并发标记和并发清除两个阶段都可以和用户进程同时工作。

不过由于CMS收集器是基于标记清除算法实现的，会导致有大量的空间碎片产生，在为大对象分配内存的时候，往往会出现老年代还有很大的空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前开启一次Full GC。为了解决这个问题，CMS收集器默认提供了一个-XX:+UseCMSCompactAtFullCollection收集开关参数（默认就是开启的），用于在CMS收集器进行FullGC完开启内存碎片的合并整理过程，内存整理的过程是无法并发的，这样内存碎片问题倒是没有了，不过停顿时间不得不变长。虚拟机设计者还提供了另外一个参数-XX:CMSFullGCsBeforeCompaction参数用于设置执行多少次不压缩的FULL GC后跟着来一次带压缩的（默认值为0，表示每次进入Full GC时都进行碎片整理）。

不幸的是，它作为老年代的收集器，却无法与jdk1.4中已经存在的新生代收集器Parallel Scavenge配合工作，所以在jdk1.5中使用cms来收集老年代的时候，新生代只能选择ParNew或Serial收集器中的一个。ParNew收集器是使用-XX:+UseConcMarkSweepGC选项启用CMS收集器之后的默认新生代收集器，也可以使用-XX:+UseParNewGC选项来强制指定它。

由于CMS收集器现在比较常用，下面我们再额外了解一下CMS算法的几个常用参数：

- UseCMSInitatingOccupancyOnly：表示只在到达阈值的时候，才进行 CMS 回收。
- 为了减少第二次暂停的时间，通过-XX:+CMSParallelRemarkEnabled开启并行remark。如果ramark时间还是过长的话，可以开启-XX:+CMSScavengeBeforeRemark选项，强制remark之前开启一次minor gc，减少remark的暂停时间，但是在remark之后也立即开始一次minor gc。
- CMS默认启动的回收线程数目是(ParallelGCThreads + 3)/4，如果你需要明确设定，可以通过-XX:+ParallelCMSThreads来设定，其中-XX:+ParallelGCThreads代表的年轻代的并发收集线程数目。
- CMSClassUnloadingEnabled： 允许对元类数据进行回收。
- CMSInitatingPermOccupancyFraction：当永久区占用率达到这一百分比后，启动 CMS 回收 (前提是-XX:+CMSClassUnloadingEnabled 激活了)。
- CMSIncrementalMode：使用增量模式，比较适合单 CPU。
- UseCMSCompactAtFullCollection参数可以使 CMS 在垃圾收集完成后，进行一次内存碎片整理。内存碎片的整理并不是并发进行的。
- UseFullGCsBeforeCompaction：设定进行多少次 CMS 垃圾回收后，进行一次内存压缩。

一些建议

对于Native Memory:

- 使用了NIO或者NIO框架（Mina/Netty）
- 使用了DirectByteBuffer分配字节缓冲区
- 使用了MappedByteBuffer做内存映射
- 由于Native Memory只能通过FullGC回收，所以除非你非常清楚这时真的有必要，否则不要轻易调用System.gc()。

另外为了防止某些框架中的System.gc调用（例如NIO框架、Java RMI），建议在启动参数中加上-XX:+DisableExplicitGC来禁用显式GC。这个参数有个巨大的坑，如果你禁用了System.gc()，那么上面的3种场景下的内存就无法回收，可能造成OOM，如果你使用了CMS GC，那么可以用这个参数替代：-XX:+ExplicitGCInvokesConcurrent。

此外除了CMS的GC，其实其他针对old gen的回收器都会在对old gen回收的同时回收young gen。

G1收集器

G1收集器是一款面向服务端应用的垃圾收集器。HotSpot团队赋予它的使命是在未来替换掉JDK1.5中发布的CMS收集器。与其他GC收集器相比，G1具备如下特点：

- 并行与并发：G1能更充分的利用CPU，多核环境下的硬件优势来缩短stop the world的停顿时间。
- 分代收集：和其他收集器一样，分代的概念在G1中依然存在，不过G1不需要其他的垃圾回收器的配合就可以独自管理整个GC堆。
- 空间整合：G1收集器有利于程序长时间运行，分配大对象时不会无法得到连续的空间而提前触发一次GC。
  可预测的非停顿：这是G1相对于CMS的另一大优势，降低停顿时间是G1和CMS共同的关注点，能让使用者明确指定在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒。
- 在使用G1收集器时，Java堆的内存布局和其他收集器有很大的差别，它将这个Java堆分为多个大小相等的独立区域，虽然还保留新生代和老年代的概念，但是新生代和老年代不再是物理隔离的了，它们都是一部分Region（不需要连续）的集合。

虽然G1看起来有很多优点，实际上CMS还是主流。

与GC相关的常用参数

除了上面提及的一些参数，下面补充一些和GC相关的常用参数：

- `Xmx`: 设置堆内存的最大值。
- `Xms`: 设置堆内存的初始值。
- `Xmn`: 设置新生代的大小。
- `Xss`: 设置栈的大小。
- `PretenureSizeThreshold`: 直接晋升到老年代的对象大小，设置这个参数后，大于这个参数的对象将直接在老年代分配。
- `MaxTenuringThrehold`: 晋升到老年代的对象年龄。每个对象在坚持过一次Minor GC之后，年龄就会加1，当超过这个参数值时就进入老年代。
- `UseAdaptiveSizePolicy`: 在这种模式下，新生代的大小、eden 和 survivor 的比例、晋升老年代的对象年龄等参数会被自动调整，以达到在堆大小、吞吐量和停顿时间之间的平衡点。在手工调优比较困难的场合，可以直接使用这种自适应的方式，仅指定虚拟机的最大堆、目标的吞吐量 (GCTimeRatio) 和停顿时间 (MaxGCPauseMills)，让虚拟机自己完成调优工作。
- `SurvivorRattio`: 新生代Eden区域与Survivor区域的容量比值，默认为8，代表Eden: Suvivor= 8: 1。
- `XX:ParallelGCThreads`：设置用于垃圾回收的线程数。通常情况下可以和 CPU 数量相等。但在 CPU 数量比较多的情况下，设置相对较小的数值也是合理的。
- `XX:MaxGCPauseMills`：设置最大垃圾收集停顿时间。它的值是一个大于 0 的整数。收集器在工作时，会调整 Java 堆大小或者其他一些参数，尽可能地把停顿时间控制在 MaxGCPauseMills 以内。
- `XX:GCTimeRatio`:设置吞吐量大小，它的值是一个 0-100 之间的整数。假设 GCTimeRatio 的值为 n，那么系统将花费不超过 1/(1+n) 的时间用于垃圾收集。

Contact

GitHub: https://github.com/ziwenxie
Blog: https://www.ziwenxie.site

Java, JVM

Search

Tags

- Algorithm3
- JVM3
- Java9
- Linux4
- MySQL1
- Project2
- Python6
- Tool2

Archives

- 2017-083
- 2017-073
- 2017-064
- 2017-041
- 2017-036
- 2017-013
- 2016-123

Links

- DuckDuckGo
- Arch Linux
- 依云
- 酷壳
- Manolo-辍耕录
- 鱼非鱼-VTEC
- 许哲
- 胖次网盘
- 鸠摩搜书