HotSpot虚拟机：

那些内存需要回收？

什么时候回收？

如何回收？

1. 回收堆与方法区

方法区：废弃常量和无用的类

废弃常量：例如常量池中一个字符串“abc”，若当前系统中没有任何String对象引用常量池中的“abc”常量，也没有其他地方引用了这个字面量，如果这时候发生垃圾回收，而且必须的话，这个“abc”常量就会被系统清理出常量池。

无用的类：该类所有的实例已经被回收

加载该类的ClassLoader已经被回收

该类对应的java.lang.Class对象没有地方被引用，无法在任何地方通过反射访问该类的方法。

是否对类进行回收，HotSpot虚拟机提供了-Xnoclassgc参数进行控制。因此在大量使用动态代理、反射、CGLib等字节码框架、动态生成JSP的应用中一定要关闭该选项, 开启VM的类卸载功能, 以保证方法区不会溢出.

2、可达性分析算法

判断对象是否存活

可作为GC roots的对象：

虚拟机栈（栈帧中的本地变量表）中引用的对象

方法区中类静态属性引用的对象

方法区中常量引用的对象

本地方法栈中JNI（即一般说的Native方法）引用的对象

1. 垃圾回收算法
2. 标记—清除算法（Mark-Sweep）：首先标记出所有需要回收的对象，在标记完后统一回收所有被标记的对象。

不足：效率问题，标记和清除效率都不高；

空间问题，产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配交大对象时，无法找到足够的连续碎片而不得不提前触发一次垃圾收集动作。

1. 复制算法（Copying）：解决效率问题，将可用内存分为大小相等的两块，每次只是用其中的一块，当这一块的内存用完了，就将还存活的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。内存分配时不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。

不足：将内存缩小了原来的一半。

现在商业虚拟机都采用这种收集算法回收新生代。新生代中对象都是朝生夕死，所以不用按照1:1的比例来分配内存空间，而是将内存分为一块较大的的Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块Survivor。

HotSpot虚拟机默认Eden：Survivor = 8:1，也就是每次新生代可用内存空间为整个新生代的90%。当存活对象超过10%时，即Survivor空间不够用时，需要依赖其他内存（老年代）进行分配担保。

1. 标记—整理算法（Mark-Compat）：标记过程一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活对象都向一端移动，然后清理掉端边界以外的内存。
2. HotSpot算法实现
3. 枚举根节点

当执行系统停顿后，并不需要一个不漏地检查完所有执行上下文和全局的引用变量，虚拟机应当是有办法直接得到哪些地方存在对象引用——OopMap数据结构，在类加载完成的时候，HotSpot就把对象内什么偏移量上是什么类型的数据计算出来

1. 安全点

在OopMap的协助下，HotSpot可以快速准确地完成GC Roots枚举，但一个很现实的问题随之而来,可能导致引用关系变化，或者说OopMap内容变化的指令非常多，如果为每一条指令都生成对应的OopMap，那将会需要大量的额外空间，这样GC的空间成本将会变得很高。

HotSpot也的确没有为每条指令都生成OopMap，只是在“特定的位置”记录了这些信息，这些位置被称为安全点（Safepoint），即程序执行时并非在所有的地方都能停顿下来开始GC，只有在到达安全点时才能暂停。

安全点：选定以是否具有让程序长时间执行的特征来选定，长时间执行的最明显特征就是指令序列复用，例如方法调用、循环跳转、异常跳转。

如何让GC发生时，让所有线程（这里不包括执行JNI调用的线程）都跑到最近的安全点上再停顿下来。HotSpot采用主动式中断,主动式中断的是当GC需要中断线程的时候，不直接对线程操作，仅仅简单地设置一个标志，各个线程执行时主动去轮询这个标志，发现中断标志为真时就自己中断挂起。（轮询标志的地方和安全点重合）

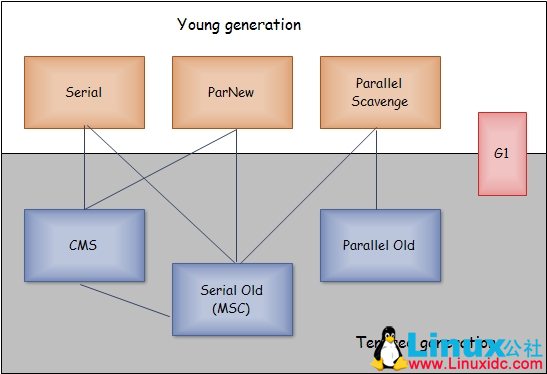
对Java线程中的JNI方法，它们既不是由JVM里的解释器执行的，也不是由JVM的JIT编译器生成的，所以会缺少OopMap信息,HotSpot的解决方法是：所有经过JNI调用边界（调用JNI方法传入的参数、从JNI方法传回的返回值）的引用都必须用“句柄”（handle）包装起来。JNI需要调用Java API的时候也必须自己用句柄包装指针。在这种实现中，JNI方法里写的“jobject”实际上不是直接指向对象的指针，而是先指向一个句柄，通过句柄才能间接访问到对象。这样在扫描到JNI方法的时候就不需要扫描它的栈帧了,只要扫描句柄表就可以得到所有从JNI方法能访问到的GC堆里的对象。

1. 安全区域

程序不执行时如何进入GC，不执行指没有分配CPU时间，如线程处于Sleep或Blocaked状态，这时候线程无法响应KVM的中断请求，JVM也不会等待线程重新分配DPU时间。

安全区域：在一段代码之中，引用关系不会发生变化。在这个区域的任何地方开始GC都是安全的。当执行到安全区域中的代码时，标识自己进入Safe Region，当在这段时间JVM发起GC时，就不用管标识自己为Safe Region状态的线程了。线程要离开安全区域时，要检查是否已经完成了根节点枚举（或者是整个GC过程），如果完成线程继续执行否则必须等待直到收到可以安全离开Safe Region的信号为止。

5 垃圾收集器



5.1 年轻代收集器

1. Serial收集器：复制算法，单线程收集器

虚拟机运行在Client模式下的默认新生代收集器，简单而高效。

1. ParNew收集器：复制算法，多线程收集器，默认开启线程数与CPU数量相同

许多运行在Server模式下的虚拟机首选的新生代收集器

当可用CPU数量增加，GC时系统资源的有效利用

除了Serial，目前只有它可以与CMS收集器配合工作

1. Parallel Scavenge收集器：复制算法，并行的多线程收集器

适合在后台运算而不需要太多交互的任务

自适应调节策略也是Parallel Scavenge与ParNew收集器的一个重要区别

5.2老年代收集器：

1）Serial Old收集器：标记-整理算法，单线程收集器

Client模式下的虚拟机使用

Server模式下：与PS收集器搭配使用（JDK 1.5及以前的版本）

作为CMS收集器的后备预案

1. Parallel Old收集器：标记-整理算法，多线程

注重吞吐量以及COU资源敏感的场合（Parallel +Parallel Old），吞吐量=运行用户代码时间/(运行用户代码时间)

1. CMS收集器：标记-清除，以最短回收停顿时间为目标

重视服务响应速度，希望系统停顿时间最短（互联网站或B/S系统的服务端）

步骤：

初始标记（CMS initial mark）：标记一下GC roots直接关联的对象，速度很快，stop the world

并发标记（CMS concurrent mark）：进行GC roots Tracing的过程

重新标记（CMS remark）：修正并发标记期间因用户线程继续执行而导致标记产生变动的那一部分对象的标记记录，stop the world

并发清除（CMS concurrent sweep）：

优点：并发收集，低停顿

缺点：（1）对CPU资源敏感，CMS默认启动的回收线程数是（CPU数量+3）/4，CPU在4个以上时，并发回收时垃圾回收线程不少于25%的CPU资源，并且随着CPU数量的增加而下降，但是当CPU不足4个时，CMS对用户线程的影响可能很大。

解决办法：增量式并发收集器i-CMS,在并发标记和并发清理时候让GC线程、用户线程交替运行尽量减少GC线程独占资源的时间。实践证明效果一般，不提倡使用。

1. CMS运行期间预留空间无法满足程序需要，即无法处理浮动垃圾，出现Concurrent Model Failure失败而导致另一次Ful l GC的出现。CMS收集器预留一部分空间。JDK 1.5老年代使用68%就会GC，JDK 1.6 92%。

浮动垃圾：并发清理阶段用户线程还在运行，产生新的垃圾无法再本次收集清理掉，只能下一次GC时清理。

解决办法：临时启用Serial Old收集器来进行老年代的垃圾收集，这样停顿时间很长。

1. 产生大量空间碎片。

解决办法：-XX:+UseCMSCompactAtFullCollection开关参数（默认开启），表明CMS收集器顶不住要进行FullGC时开启内存碎片的整理过程，内存整理无法并发，停顿时间很长。

-XX:CMSFullGCsBeforeCompaction参数设置执行多少次不压缩的FullGC后，跟着来一次带压缩（默认值为0，标识每次进入FullGC时都进行碎片正理）

1. G1收集器：面向服务端的垃圾收集器

并行与并发

空间整合

分带手机

可预测停顿

步骤：初始标记

并发标记

最终标记

筛选回收：对各个Region的回收价值和成本进行排序，根据用户期望的GC停顿时间来制定回收计划