# JVM Garbage Collector 深入

1. Java虚拟机栈：

描述的是java方法的执行模型：每个方法执行的时候都会创建一个帧（基于栈的指令集），栈用于存放局部变量，操作数栈，动态链接，方法出口等信息。一个方法的执行过程，就是这个方法对于栈帧的入栈与出栈过程（线程隔离）

1. 堆：

GC主要的工作区域，为了高效的垃圾回收会把对分为更多的子区域（线程共享）

1. 方法区：

存放了每个Class的结构信息，包括常量池，字段描述，方法描述；GC的非主要工作区（线程共享）

1. Object o = new Object()

这个操作会生成2部分的内区域： o这个引用变量，这个是方法内的变量，会存放到JVM 虚拟机栈中（stack）； 真正的Object 实例对象位于Heap中

上述new语句一共消耗12 bytes ，JVM规定引用占4个bytes （stack中），而空对象是八个bytes(在Heap)

1. 方法结束过后，对应Stack中的变量会马上回收，但是Heap中的对象要等到GC来回收
2. JVM垃圾回收（GC）模型
3. 垃圾判断算法
4. GC算法
5. 垃圾回收器的实现和选择
6. 垃圾判断算法
7. 引用计数算法（Reference Counting）
8. 根搜索算法（Root Tracing）
9. 引用计数算法：

给对象添加一个引用计数器，当有一个地方引用它，计数器加加1 ，当引用失效，计数器减1， 任何时刻计数器为0的对象就是不可能在被使用的

引用计数算法无法解决对象循环引用的问题（Objective C 会去规避这个问题）

1. 根搜索算法：

算法基本思路：通过一系列的称为“GC Roots”的点作为起始点向下搜索，当一个对象到GC Roots 没有任何引用链想连时，则证明此对象是不可用的

1. 在VM栈中（栈本地变量）中的引用
2. 方法去中静态变量的引用
3. JNI（既一般说的native 方法）中的引用
4. 当前的商业的JVM 都有实现方法区的GC ,主要回收两部分的内容： 废弃常量和无用类
5. 类回收需要满足以下三个条件：
6. 该类所有的实例都已经被GC，也就是JVM中不存在该类Class的任何实例
7. 加载该类的ClassLoader 已经被GC
8. 该类对应的java.lang.Class 对象没有在任何地方被引用，如不能在任何地方通过反射访问该类的方法
9. 在大量使用反射，动态代理，CDLib 等字节码框架，动态生成JSP以及OSGI这类频繁自定义ClassLoader 的场景都需要JVM 具备类卸载的支持以保证方法区不会溢出
10. JVM常见GC算法
11. 标记-清除算法（Mark-Sweep）
12. 标记-整理算法（Mark-Compact）
13. 复制算法（Copying）
14. 分代算法（Generation）

问题： 分代算法新生代为何使用复制算法，老年代使用标记-整理算法？

1. 绝大多数对象都是朝生夕灭，一般都是在新生代生成对象，每次进行垃圾回收回收的数量比较多，存活的数量相比比较少，而且复制算法会浪费一定的空间；因此复制的对象的有限制不能太多；而老年代大多数都是存活的，复制的成本会比较高
2. 当从from 复制到 to 容不下时，老年代会有分配担保的措施；会把存活的对象转移到老年代。
3. 标记-清除算法

效率不高，扫描的对象较多，堆越大时间越长

会产生碎片，次数越多碎片越多，会发生总体空间够分配，而单独的空间却不够

1. 标记-整理算法

对上一个算法的改良，进行对象移动，碎片的合并，把碎片合并成一个可用于占空间较大的对象；但是更耗时间

1. 复制算法

每次回收都是对整片内存的回收，不会担心内存碎片的问题；但是它需要浪费一半的空间不能用于使用。

复制算法适用于对象存活时间比较短的对象，因为每次GC可以回收大部分对象，效率会比较高

当对象存活时间比较长，复制算法效率就会明显降低

这样使得每次内存回收都会对整个半区进行回收，在分配的时候不用考虑内存的碎片问题，只需要移动堆顶指针就可以进行内存分配

1. 分代算法

会产生新生代与老年代进行分代收集算法；年轻代会有三个区域，一个Eden区域两个survivor区域，对象在Eden区域生成，当Eden区满时，还存活的对象会复制到一个survivor区，当这个survivor区满时，此时存活的对象将会复制到另一个survivor区；此时from 区域可以变成to区域，当对象从from 到 to 时，to区域满时就会把存活的对象复制到老年代中；2个survivor区完全对象可以相互交换。

Eden , from survivor , to survivor 8:1:1 ,当Eden 与 from survivor 中满时会复制到 to survivor 空间；然后交换from 与 to 空间。

内存分配：

* 大多数情况会在Eden区域上分配，偶尔会直接在old区域上分配，取决于GC实现
* 栈上分配原子类型的变量

1. 对象分为四种应用： strong ,soft , weak , phantom 引用，一般的new是强引用；而soft ，weak，phantom都是继承Reference。

两种GC 的模式：19,20 主要这两种

1. Minor GC (scavenge GC) 主要清理新生代的空间

* 新对象生成，Eden空间满
* Eden区大多数会被Minor GC 所回收,算法执行效率会很高，GC时间会很短

1. Full GC 全量的清理；新生代，老年代，方法区都会进行垃圾收集

* 对JVM 进行整理，young old perm(遗弃) meta space
* 触发机制 old满， perm 满，meta space 满，System.gc()
* 效率低，尽量少采用

1. 垃圾收集器并行与并发的含义：

* 并行（parallel）：多个收集器的线程同时工作，但是用户线程处于等待状态
* 并发（concurrent）：指收集器在工作的同时允许用户线程同时工作

并不代表解决了GC停顿的问题，关键步骤还是会停顿，在收集器标记垃圾的时候；在清除垃圾的时候与用户线程同时进行

1. Serial 垃圾收集器
2. 当线程收集器，采用的是复制收集算法；会暂停所用的用户线程STW；
3. new 与 old generation 都可以使用；
4. 新生代使用复制算法，老年代使用mark-compact算法
5. 单线程没有额外的开销，不过是hot spot虚拟机client的缺省默认值
6. Parnew 垃圾收集器
7. 就是serial 的多线程版本，可以同时进行收集，功能跟serial一模一样
8. 这种是虚拟机运行在server模式下，对于多核的CPU 效果会更好，但是对于单核来讲效率反而会下降
9. 可以通过参数 -XX:ParallelGCThreads=4 来指定线程的个数，一般根据CPU核数来设置，server模式下的缺省值
10. Parallel scavenge 垃圾收集器
11. 也是一个多线程版本的收集器，但是它的回收策略与分配规则与parNew不一样
12. 它以吞吐量为最大化（既GC时间占总运行时间最小）为目标
13. 允许较长的STW换取总吞吐量最大化
14. Parallel old 收集器（并行）
15. 老版本吞吐优先的算法，使用多线程，mark-compact 老年代收集器
16. 更注重吞吐量，parallel scavenge + parallel old = 高吞吐量，但是停顿可能不理想
17. Serial old 收集器（串行）
18. 单线程收集器，mark-compact 老年代收集器
19. CMS old 收集器（并发）
20. 枚举根节点：当系统停顿下来后，并不需要一个不漏的检查完所有执行上下文和全局的引用位置，而是虚拟机有办法直接知道哪些地方存放着对象的引用；在Hot Spot 中采用OopMap的数据结构来达到这个目的
21. 安全点：在OopMap的协助下，HotSpot 可以快速准确完成GC Roots 的枚举；会有一个问题：OopMap内容变化的指令非常之多，如果给每一条指令都生成OopMap会产生高昂的内存空间，GC成本就会变高。 实际上HotSpot并没有为每条指令生成OopMap,而是在“特定位置”记录了这些信息，这些特定的位置就叫做安全点（safe point）,程序并非所有位置都能停顿下来执行GC ，必须要达到安全点才能停顿下来执行GC
22. 对于safe point的选定不能太少，以至于让GC停顿的时间太长；也不能过于频繁，会增大运行期的负载；安全点的选定特征是“是否具有让程序长时间执行的特征”为标准去选定；“长时间执行”的最明显的特征就是指令序列复用；例如方法调用，循环转跳，异常转跳等，所以具有这些功能的指令才会具有safe point
23. 对于安全点。另一个需要考虑的问题，在GC发生时，所有的线程都跑到安全点，有两种方案：抢断式中断与主动式中断
24. 抢断式中断： 他不需要线程配合，当GC 执行时会暂停所有的用户线程，然后判断哪些用户线程没有跑到安全点，然后恢复他们跑到安全点
25. 主动式中断： 当GC 需要中断线程的时候，不直接对线程操作，仅仅设置一个标志位，各个线程询问该标志位，当发现标志位为真时，就主动挂起；轮询标志的地方与安全点重合。
26. 安全区域：安全点完美解决了如何进入GC的问题，但是实际情况并不一定；safe point执行保证了程序执行，不太长的时间内就会遇到可进入GC的安全点；但是程序在不执行的时刻呢？或者说CPU 并没有分配时间片段，处于等待，休眠，阻塞的状态下；这时候线程无法响应JVM 的请求，JVM显然也不可能等待线程分配CPU时间，这个时候就得靠安全区域来解决该问题； 线程在执行到safe region区域时，会自动标记自己已经进入到安全区域，那样JVM 在发起GC时就不用管那些已经标识为进入安全区域的用户线程；但是当线程想要离开safe region时需要检查系统是否完成了根节点枚举（或者是整个GC过程）；若完成则就执行，否则则等待直到收到可以离开safe region的信号为止（类似于线程的wait and notify）
27. CMS（concurrent mark sweep）收集器：以获取最短暂定回收时间为目标，应用于b/s架构中
28. 基于“标记-清除”算法，有如下四个步骤：

* 初始标价（CMS initial mark）
* 并发标记（CMS concurrent mark）
* 重新标记（CMS remark）
* 并发清除（CMS concurrent sweep）

1. CMS 收集器主要步骤对应的功能：

* 初始标记，重新标记两个步骤任然会stop the world
* 初始标记只是标记GC Roots 能直接关联到的对象，速度会很快
* 并发标记则是进行 GC Roots tracing 过程
* 重新标记只是为了修复并发标记期间用户程序继续运作导致标志变动的那一部分对象的标记记录，这个阶段停顿时间比初始化标记停顿时间长，但远比并发标记时间短

1. CMS 并发收集，低停顿称之为并发低停顿收集器，但是缺点还是挺多的：

* 对CPU资源非常敏感
* 无法处理浮动垃圾（floating garbage）因此可能会出现“concurrent mode failure ”失败会导致一次full GC 的产生；如果应用中老年代增长的不是太快，可以适当调高参数-XX:CMSInitialingOccupancyFraction的值来提高触发百分比；以便降低内存回收次数来获得更好的系统性能。要是CMS运行期间预留的内存无法满足程序的需要时，则虚拟机启动预备方案，临时启用serial old收集器来进行老年代垃圾回收，但是停顿的时间就会比较长，所以设置-XX:CMSInitialingOccupancyFraction参数过高会导致“concurrent mode failure ”出现，从而影响性能
* 该算法会导致回收会产生大量的碎片，无法找到足够大的连续空间来分配当前的对象，因此会发生FULL GC ；CMS 提供一个参数 -XX:UseCMSCompactAtFullCollection开关参数，默认是开启；用于CMS收集器顶不住时，要进行full gc 时开启碎片整理，内存整理不是并发的，因此会停顿时间不得不边长
* **空间分配担保**： 在发生minor GC时，虚拟机会先检查老年代最大可用连续空间是否大于新生所有对象的总空间，如果这个条件成立，则这次的minor GC 可以确保是安全的。当大量对象在minor GC 后任然存活，就需要老年代进行空间分配担保，把survivor无法容纳的对象直接进入老年代，如果老年代判断到空间剩余不足（根据以往每一晋升到老年代容量的平均值）则进行一次FULL GC

1. CMS收集器收集步骤：

* Initial mark

标记那些直接被GC Roots所引用或者被年轻代对象多引用的对象

* Concurrent mark

遍历老年代，根据上阶段所找到的根进行搜索查找标记所引用对象

* Concurrent preclean

并发运行的过程，对象的引用会发生一些变化，这种情况发生时，JVM会将该对象的区域(carding)标记为dirty 也就是carding mark

在该阶段，能够从dirty对象到达的对象也会被标记，这个标记做完后，会将dirty card清除掉

* Concurrent abortable preclean

这个阶段尽量承担STW 最终标记阶段的工作，这个阶段持续时间要素很多由于是在重复做相同的工作直接满足一些条件

* Concurrent remark
* Concurrent sweep
* Concurrent reset

重试CMS 的数据结构，未下一次GC做准备工作

优点：通常将大量的工作例如标记阶段分散到各个阶段并发执行；以获得较短的停顿；减少STW的时间

1. SATB算法：G1在并发标记阶段使用的增量式的标记算法，一个区对应一个线程
2. G1收集器与CMS收集器的对比： G1区分角色还是按照Eden ,survivor, old 为了动态调整每个代的大小

* G1在压缩空间上占据优势
* G1通过内存空间分成区域方式避免内存碎片
* Eden，survivor，old 区不再固定，内存使用更灵活
* G1可以通过设置预期停顿时间来控制垃圾收集，避免应用雪崩效应
* G1在回收垃圾的时候，做内存合并，而CMS默认在STW时做compact
* G1可以在young 上使用，而CMS 只能在老年代使用

1. G1在运行期的工作模式

* YGC（不同于CMS 只针对old）新生代满的时候
* 并发阶段（gloable concurrent marking 也会执行YGC 标记根 root）四个阶段跟CMS一样
* Mixed GC (YGC , GCM 拿到最高性价比的老年代region)
* Full GC （一般G1出问题）回收赶不上创建的对象

1. 参数控制发生mixGC

* G1HeapWastePercent :在GCM 标记后，看到old gen regions 需要回收的区域，只有当垃圾占比达到这个值就会触发
* G1MixedGCLiveThresholdPercent: old gen region 存活对象占比在此参数下，才会进入到CSet
* G1MixedGCCountTarget: 在 一次GCM 标记之后，最多执行mixed GC的次数
* G1OldCSetRegionThresholdPercent: 一次mixed GC 中能被选入到CSet的最多old gen region 数量
* G1HeapRegionSize:设置region大小
* MaxGCPauseMillis: 设置G1停顿大小
* G1NewSizePercent: 新生代最小值
* G1MaxNewSizePercet:新生代最大值
* ParallelGCThreads: STW期间，并行GC数
* ConcGCThreads: 并把标记阶段，并行执行线程数
* InitiatingHeapOccupancyPercent: 设置触发标记周期的java堆占用率阈值，默认45%

1. Rset的作用：

* 在CMS中，也有RSet的概念，在老年代中有一块区域用来记录指向新生代的引用，这是一种point-out,在进行YOUNG GC时，扫描根时，仅仅只需要扫描这一块区域，而不用扫描整个老年代，扫描老年也阔以同时反过来记录
* 由于新生代有多个，那么我们需要在新生代之间记录引用嘛？这是没有必要的，原因在于每次GC时，所有新生代都会被扫描，所以只需要记录老年代到新生代之间的引用即可

1. G1 Young GC 执行阶段 （STW）

* 根扫描：静态和本地对象被扫描
* 跟新RS: 处理dirty card 队列跟新RS （那些区域的对象指向我这个区域）
* 处理RS： 检测从年轻代指向老年代的对象
* 对象拷贝： 拷贝存活的对象到survivor 或old区域
* 处理引用队列：软引用，弱引用，虚引用

1. Mixed GC 步骤

* 全部并发标记（global concurrent marking）四个步骤跟CMS想象但执行不一样
* 拷贝存活对象（evacuation）

1. 三色标记算法：黑色代表根对象，以及它的子对象都别扫描过（field引用都被扫描完）。灰色表示的是该对象已经被扫描了，但其子对象并没有扫描完；白色则就不可达要被清除。
2. SATB(snapshot at the beginning) 算法：

* 在开始标记的时候生成一个快照，标记存活对象
* 在并发标记的时候所有改变的对象入队（在write barrier里把所有旧的引用所指向的对象都变成非白的）
* 可能存在浮动垃圾

1. 如何找到在GC过程中分配的对象呢？**每个region记录着两个top at mark start （TAMS）指针，分别为prevTAMS 与 nextTAMS 在TAMS以上的对象就是新分配的，因此会隐式marked ;**通过这种方式我们就找到了在GC过程中新分配的对象，并把这些对象认为是活的对象。
2. 漏标的情况以及解决方法：
3. 并发标记时，应用线程给一个黑色对象的引用类型字段赋值了该白色对象
4. 并发标记时，应用线程删除了所有灰色对象到该白色对象的引用

解决方案：

* 第一种情况，利用 post-write barrier 记录所有新增的引用关系，然后根据这些引用关系为跟重新扫描一遍
* 第二种情况，利用pre-write barrier 将所有即将被删除的引用关系的就引用记录下来，最后以这些旧引用为根重新扫描一遍

1. G1通过一个停顿预测模型根据用户配置的停顿时间来选择CSet的大小，从而满足所希望的停顿时间，并不是这个时间越小越好，收集的垃圾越少则会退化成serial GC。
2. **CSet 有两种子模式**，第一种是Young GC 就是所有年轻代里的region ；第二种就是指Mixed GC 中所有年轻代与全局并发标记阶段记录的收益最高的region
3. 初始化阶段会在Young GC期间执行，但是全局并发标志不会执行Mixed GC ，执行mixed GC 不会执行初始化而是执行SATB算法（initial mark ）
4. 关注Evacuation failure 晋升失败，堆空间垃圾较多，因此会进行full GC。