JVM GC

- 对象存活性判断
  - 1 引用计数法
    - 给对象添加一个引用计数器，有引用+1，无引用-1，为0时无引用，JVM并不使用该方法，且会出现循环引用 的问题
  - 2 可达性分析算法
    - 不同于引用记数，可达性分析采用一系列GC Roots的对象向下搜索产生对象链，对某个对象不可达，则该对象可回收
    - 可作为GC Roots的对象有哪些？
      - 虚拟机栈(帧栈中的本地变量表)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(Native方法)JNI引用的对象
    - "引用"类型：引用关系着对象的存活
      - 强引用(Strong Reference)：能直接访问对象;强引用存在，则对象不能被回收，JVM宁愿抛出OOM也不回收;强引用可能导致内在泄露；
      - 软引用(Soft Refernce):比强引用弱一点，堆空间不足时被回收
      - 弱引用(Weak Reference):比软引用弱一点，垃圾回收时，只要发现弱引用就回收
      - 虚引用(Phantom Reference):引用强度最弱的一种，无法从虚引用来获得强对象，必须与引用队列一同使用
  - 3 finalize()可使对象复活
    - 如果对象被判定为有必要去执行finalize()方法，就会被放入F-Queue中，稍后会由虚拟建立的低优先级线程去执行
    - 虚拟机的线程只是有可能去执行，并不表示一定会执行
    - finalize()只会执行一次，是对象复活的最后一次机会
  - 4 方法区(永久代)回收
    - 废弃常量判定
      - 如常量池中的常量，其他类(接口)，方法，字段的符号引用等等，无引用来引用它们就可判定
    - 无用的类(3个条件及指定虚拟机参数来判定)
      - 该类的所有实例都已经被回收，也就是堆中不存在任何该类的实例
      - 加载该类的ClassLoader已经被回收
      - 该类的对应的java.lang.Class对象没有在任何地方引用，无法在任何地方通过反射访问该类的方法
- 垃圾收集算法
  - 1 标记-清除法
    - "标记"对象和"清除"对象，标记使用采用的就是对象引用的可达性判定。缺点：两阶段效率都不高，会产生大量的不连续空间碎片
  - 2 复制算法
    - 内存分为相同两块，使用时使用一块空间，当用内内存用完了，将存活对象复制到另一块空间。缺点：内存少了一半。如hotspot的新生代
  - 3 标记-整理算法
    - 基本上与"标记-清除"法相同，但最后一步不是""清除"，而是将存活对象移到内存的一端，再清理回收。该算法属于老年代算法
  - 4 分代收集算法
    - 根据对象的存活周期将内存划分成不同的几块，根据不同代的特点进行回收。一般分成新生和老年代，新生代用复制算法，老年代用"标记-清除/整理"法
- HotSpot的算法实现
  - 1 对象回收判定和垃圾收集在HotSpot上的实现
    - 1.枚举根节点：使用OopMap数据结构来快速完成GC Root的枚举
    - 2.安全点：因为在对指令生成OopMap时会使引用变得混乱而导致GC成本变大，采用安全点SafePoint让程序在点停顿
      - 如何到达该停顿点
    - 3.安全Region:当程序不执行(sleep或blocked)状态，进行该区域进行GC
      - 抢占式中断和主动式中断
- GC Collectors(垃圾回收器)
  - 1 Serial收集器
    - 单线程：虚拟机使用一个CPU（或一条线程）去进行垃圾回收，特点：无线程交互,简单高效
    - 新生，老年代都会产生STW（stop the world）:暂停所有用户线程,虚拟机自发执行并完成
    - 在新生代采用复制算法,在老年代采用标记-整理算法
    - JVM Client模式下默认使用Serial收集器(桌面应用比较合适)
  - 2 ParNew收集器
    - Serial收集器的多线程版本:控制参数，算法，内存分配，回收策略等与Serial基本一致
    - 在多CPU情况下效果优于单CPU效果（单CPU下效果绝对不会优于多CPU效果）,默认回收线程数=CPU数量
    - 老年代使用CMS,新生代只能使用Serial或ParNew,使用-XX:+UseConcMarkSeepGC后，ParNew默认为新生代回收器
  - 3 Parallel Scavenge收集器
    - 新生代收集器,使用复制算法，并行多线程收集器
    - "吞吐量优先"收集器：不关注于垃圾回收时用户线程的停顿时间，而是关注于一个可控制的吞吐量
    - 吞吐量 = CPU用于运行用户代码的时间/(运行用户代码时间 + 垃圾回收时间)，JVM运行100个单位时间,垃圾回收1 个单位时间，吞吐率为99%
    - -XX:MaxGCPauseMillis=<size>：控制最大垃圾回收停顿时间，-XX:GCTimeRatio=<size>：设置吞吐量大小
    - -XX:+UseAdaptiveSizePolicy：用于设置GC自适应策略，若使用该参数和Parallel Scavenge收集器，JVM会自适应优化内存
  - 4 Serial Old收集器
    - Serial的老年代版本(PS MarkSweep)，单线程且使用"标记-整理"算法，主要意义是给JVM Client模式使用
    - JVM Server模式下2个作用: (1).在jdk1.5及以前与Parallel Scavenge搭配使用.(2)作为CMS的后备方案，在Concurrent Mode Failure时使用
  - 5 Parallel Old收集器(JDK1.6开始)
    - Parallel Scavenge的老年代版本,多线程且使用"标记-整理"算法
    - Parallel Scavenge无法与CMS组合使用，而Parallel Scavenge+Parallel Old可兼顾吞吐与CPU资源敏感
  - 6 CMS收集器(jdk1.5发布)
    - 一种以获取最短回收停顿时间为目标的收集器,"标记-清除"算法，主要针对于老年代
    - 回收过程（标记-清除）:
      - （1）初始标记(CMS initial mark): 暂停所有的其他线程，并记录下直接与root相连的对象(会STW)
      - （2）并发标记(CMS concurrent mark):做GC Root Tracing的过程
      - （3）重新标记(CMS remark)：修正并发标记阶段的标记数据，当前阶段会STW
      - （4）并发清除(CMS concurrent sweep):开启用户线程，同时GC线程开始对为标记的区域做清扫
    - 缺点:
      - （1）CMS对CPU资源非常敏感：在并发阶段，CMS战用一部分线程会占用一部分CPU资源而使系统变慢
      - （2）CMS无法处理浮动垃圾：并发标记阶段用户线程还在执行，会出现新的垃圾，只能留着下次CMS回收
      - （3）CMS所使用的算法"标记-清除"会产生大量碎片，产生许多碎片时，再分配大对象时会触发FULL GC
  - 7 G1收集器(Garbage First)
    - 特点:
      - 并行与并发：利用多核CPU来缩短STW时间
      - 分代收集：提供分代收集，不需要与其他收集器配合就能管理整个堆
      - 空间整合：将堆分成多个Region区，采用"标记-整理"，标记+复制算法不会产生碎片
      - 可预测的停顿:降低STW的同时还能建立预测STW模型
    - 运作步骤：
      - 初始标记
      - 并发标记
      - 最终标记
      - 这三个阶段与CMS相同
      - 筛选回收：对各个Region区进行回收，回收依据是成本和价值
- 内存分配和回收策略
  - 1 内存分配策略
    - 对象优先在Eden上分配
    - 大对象直接进入老年代
      - 大量连续内存空间的Java对象(如长字符串或大数组)，会导致新生代现个S区或Eden区大量对象复制
      - 利用-XX:PretenureSizeThreshold = <size>,大于size的对象直接进行老年代
    - 长期存活的对象进行老年代
      - -XX:MaxTenuringThreshold = <size>,size为对象进入老年代的阀值，默认15
      - 年龄增长判定：对象在Eden出生并经过一次Minor GC后仍存活并能在S区被容纳，年龄+1；每经过一次Minor GC 年龄+1
    - 同时还使用"动态对象年龄判断"和"空间分配担保"方式来配合内存分配
  - 2 回收策略：依靠垃圾回收体系