垃圾收集器与内存分配策略

- 概述
  - 哪些内存需要回收?
  - 什么时候回收?
  - 如何回收?
- 对象已死吗
  - 引用计数法
    - 计数器为0的对象不可能在被使用
      - 缺点：很难解决对象之间相互循环引用的问题
  - 可达性分析算法
    - 基本思想：通过一系列称为 GC Roots 的对象做为起始点
      - 向下搜索，搜索所走过的路径称为引用链
      - 一个对象到GC Roots没有任何引用链相连，证明此对象不可用
    - GC Roots 对象包括
      - 虚拟机栈中(本地变量表)引用的对象；方法区中类静态属性引用的对象及常量引用的对象；本地方法栈中 JNI 引用的对象
  - 在谈引用
    - 判断对象是否存活都与引用有关
    - 类型：强引用、软引用、弱引用、虚引用
      - 强引用指代码程序中普遍存在的，类似 new Object(), 强引用存在，永远不会被回收
      - 软引用描述一些还有用但非必需的对象，SoftReference 类实现
        - 发生内存溢出之前，会先回收此类对象
      - 弱引用强度比软引用更弱一些，弱引用关联的对象只能生存到下一次垃圾收集发生之前，WeakReFerence类实现
      - 虚引用称为幽灵引用或者幻影引用，PhantomReference 类实现
        - 目的是被回收时收到一个系统通知
  - 生存还是死亡
    - 对象死亡，至少经历二次标记过程
      - 没有发现与 GC Roots 相连接的引用链，第一次被标记且进行一次筛选，筛选条件此对象是否有必要执行 finalize()方法
      - 有必要执行 finalize 方法，会放到 F-Queue 的队列中，稍后 GC 堆 F-Queue 中的对象进行第二次小规模的标记
        - 如果对象想在这个时候拯救自己，只要重新与引用链上的对象建立关联即可，第二次标记会被移出即将回收的集合
  - 回收方法区
    - 垃圾收集二部分：废弃常量和无用的类
      - 废弃常量：没有对象引用常量池中的常量，常量就会被回收
      - 无用的类
        - 该类所有的实例都已经被回收，java堆中不存在该类的任何实例
        - 加载该类的 ClassLoader 已经被回收
        - 该类对象的 Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法
- 垃圾收集算法
  - 标记-清除算法
    - 标记阶段：标记所需要回收的对象
    - 清除阶段：统一回收所有被标记对象
    - 最基础的算法，主要不足
      - 效率不高
      - 会产生大量不连续的内存碎片
  - 复制算法
    - 解决标记清除效率问题
      - 每次都是对半区进行回收，不用考虑内存碎片等复杂情况，按顺序分配内存，实现简单，运行高效
    - 将可用内存按容量划分为大小相等的两块，每次使用其中一块
      - 一块内存用完，将还存活的对象复制到另一块，然后把已使用的内存空间一次清理掉
  - 标记-整理算法
    - 根据老年代的特点，过程与标记清除算法一样，后续步骤不是直接回收对象
    - 让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存
  - 分代收集算法
    - 根据对象存活周期的不同将内存划分为几块
      - 新生代
        - 大批对象死去，选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集
      - 老年代
        - 存活率高，没有额外空间对他分配担保，使用标记整理或者标记清除算法进行回收
- HotSpot 的算法实现
  - 枚举根节点
    - 可达性分析工作必须在一个能确保一致性的快照中进行，一致性指整个系统看起来像冻结在某个时间点上，不会出现对象引用关系还在不断变化的情况
    - 这点是导致GC进行时必须停顿所有 JAVA 执行线程的其中一个重要原因
    - 即使号称不会发生停顿的 CMS收集器，枚举根节点也必须要停顿的
    - 使用一组称为 OopMap 的数据结构达到得知哪些地方存放在对象引用
    - Sun 将这件事情称为"Stop The World"
  - 安全点-safepoint
    - 程序执行时并非在所有地方都能停顿下来开始 GC，只有在到达安全点时才能暂停
    - 如何在GC发生时让所有线程都"跑"到最近的安全点上在停顿下来
      - 抢先式中断
        - 不需要线程的执行代码主动配合
      - 主动式中断
        - 当 GC 需要中断线程的时候，不直接对线程操作，而是设置一个标志，发现标志为真自己中断挂起
  - 安全区域-safe region
    - 在一段代码片段之中，引用关系不会发生变化，任意地方开始 GC 都是安全的
    - 线程无法响应 JVM 的中断请求，走到安全的地方中断挂起，JVM 不可能等待线程重新被分配 CPU 时间，需要安全区域来解决，典型例子 线程处于Sleep、Blocked 状态
- 垃圾收集器
  - 新生代收集器
    - Serial 收集器
      - 最基本、最悠久的，单线程收集器，复制算法
      - 简单高效，单个 CPU 的环境下，没有线程交互开销，专心做垃圾收集
    - ParNew收集器
      - Serial 收集器多线程版本，许多Server模式虚拟机首选的新生代收集器
      - 多 CPU 数量，对于 GC 时系统资源的有效利用还是很有好处的
    - Parallel Scavenge收集器
      - Parallel-并行，指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态
      - 目的达到可控的吞吐量，可以自适应调节，提供合适的的停顿时间或者最大的吞吐量
  - 老年带收集器
    - Serial old收集器
      - Serial的l收集器老年代版本，采用"标记-整理"算法
    - Parallel old收集器
      - Parallel Scavenge收集器老年代版本，JDK1.6开始提供
    - CMS 收集器
      - concurrent-并发，指用户线程与垃圾收集器同时执行（不一定是并行，有可能交替执行），用户程序继续运行，而垃圾收集器程序运行在另一个 CPU 上
      - Concurrent Mark Sweep收集器以获取最短回收停顿时间为目标， 非常适合互联网活 BS 系统
      - 基于标记-清除算法实现，运作过程分为4个步骤
        - 初始标记-CMS initial mark
          - 标记 GC roots 能直接关联到的对象
        - 并发标记-CMS concurrent mark
        - 重新标记-CMS remark
          - 修正并发标记期间因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录
        - 并发清除-CMS concurrent sweep
      - 并发收集，低停顿
      - 明显缺点
        - 对 CPU 资源非常敏感，并发阶段，会导致程序变慢，总吞吐量会降低
        - 无法处理浮动垃圾，可能出现 concurrent mode failure失败而导致另一次 Full GC 的产生
        - 收集结束时会有大量空间碎片产生
  - G1收集器
    - 当今收集器技术发展的最前沿成果之一，面向服务端应用
    - 具备特点
      - 并行与并发，利用多核优势，缩短 stop the world停顿的时间，可以使用并发方式让 java 程序继续执行
      - 分代收集
      - 空间整合，整体基于标记-整理算法，局部基于复制算法，运行期间不会产生内存碎片，有利于程序长时间运行
      - 可预测停顿，建立了可预测的停顿时间模型
    - 将整个 JAVA 堆划为多个大小相等的独立区域，新生代与老年代不再是物理隔离，都是一部分 region 的集合。优先回收价值最大的 region
    - 运作步骤
      - 初始标记-initial Marking
        - 标记 GC Roots 能直接关联到的对象，并且修改 TAMS（Next Top at Mark Start） 的值
      - 并发标记
        - 从 GC Root 开始对堆中对象进行可达性分析，找出存活的对象，耗时较长，可与用户程序并发执行
      - 最终标记
        - 为了修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，这阶段需要停顿线程，可并行执行
      - 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
  - 理解 GC 日志
    - GC 发生时间、垃圾收集停顿类型，GC 发生区域、GC 前内存区域已使用容量、内存区域 GC 所占用时间
  - 垃圾收集器参数总结
- 内存分配与回收策略
  - 对象优先在 Eden 分配
    - 当 Eden 区没有总够空间进行分配时，虚拟机将进行一次 Minor GC（新生代 GC）
  - 大对象直接进入老年代
    - 大对象指的是需要大量连续内存空间的 JAVA 对象 ，典型大对象是那种长的字符串级数组
  - 长期存活的对象将进入老年代
    - 虚拟机给每个对象定义了一个对象年龄 age 计数器
    - 对象再 Eden 出生并经过第一次Mnior GC 后仍存活，并且能被 Survivor 容纳的话，，会被移到 Survivor 空间，年龄设为1
    - 对象再 Survivor 区每熬过一次 Minor GC，年龄增加1，年龄达到15（默认值），会晋升为老年代
  - 动态对象年龄判断
    - 如果在 Survivor 空间中相同年龄所有对象大小的总和大于 Survivor 空间的一半，年龄大于或者等于该年龄的对象直接进入老年代
  - 空间分配担保
    - 虚拟机先检查老年代最大可用的连续空间是否大于新生代所有对象总空间
      - 成立，Minor GC可以确保是安全的
      - 不成立，虚拟机会查看是否设置允许担保失败
        - 如果允许，检查老年代最大可用的连续空间是否大于历次晋升到老年代的对象的平均大小
          - 大于，尝试 Minor GC
          - 小于，进行 Full GC
        - 如果不允许，进行一次 Full GC

XMind:ZEN 试用模式