GC

垃圾指没有任何指针指向的对象，这些对象就是需要被回收的，主要针对堆区和方法区
优点 - 自动内存管理机制，专注业务开发
担忧 - 弱化定位内存问题和解决问题的能力

- **GC 算法**
  1.
  - 标记

    标记出已死亡的对象，即不再被任何存活对象引用的对象
    - 引用计数法

      每个对象保存一个引用计数器属性
      优点：实现简单，便于辨识垃圾对象，判定效率高，回收延迟低
      缺点：单独的计数器存储的空间开销，每次引用更新计数器的时间开销，循环引用缺陷
      - 循环引用

        造成内存泄漏
        解决方法（Python）
        1. 手动解决：在合适时机解除引用关系
        2. 弱引用 weakref
    - GC Roots

      根节点可达性分析，根搜索算法，追踪性垃圾收集
      以 GC Roots 对象集合为起点，从上至下搜索所有连接的对象，这里所走过的路径称为引用链（Reference Chain），不在引用链上的对象即已经死亡的垃圾对象
      优点：实现简单，执行高效，无循环引用问题，避免内存泄漏
      GC Roots 对象： 一组必须活跃的引用
      1. 虚拟机栈（局部变量表）中引用的对象
      2. 方法区中类静态变量引用的对象
      3. 方法区中常量引用的对象
      4. 本地方法栈JNI（Native 方法）引用的对象
      5. 所有被同步锁持有的对象
      6. JVM 内部引用的对象（基本数据类型对应的 Class 对象，常驻异常对象，如 NullPointerException，OOM），系统类加载器）
      7. JVM 内部的 JMXBean，JVMTI 中的回调，本地代码缓存等

      总结：除堆空间外，虚拟机栈、本地方法栈、方法区、字符串常量池等地方对堆进行引用的，都是 GC Roots 对象
      如果是分代回收，值回收新生代，那么老年代可能也需要作为 GC Roots，才能保证可达性分析的准确性

      ---注意---
      可达性分析必须在一个能保证一致性的快照中进行，否则准确性无法保证，这也是 GC 必须 STW 的原因

      - finalization 机制

        GC 之前，GC 对垃圾对象调用 finalize() 方法，用来允许开发人员对垃圾对象自定义处理逻辑，通常用于做释放资源的工作，如文件流，套接字，数据库连接等

        ---不可主动调用 finalize---
        1. finalize 时可能导致对象复活
        2. finalize 有 GC 线程调用，不发生 GC 时不会执行，优先级较低
        3. 糟糕的 finalize 会影响 GC 性能
        - 对象的 3 个状态

          可触及的：GC Roots 可达
          可复活的：GC Roots 不可达，但 finalize 未执行，还可能在 finalize() 中复活
          不可触及的：GC Roots 不可达，且 finalize() 已经执行而没有复活，将再无可能复活（finalize 只会被调用一次）
        - GC Roots两次标记

          1. 第一次标记：对象对一次 GC Roots 不可达时，若对象没有重写 finalize() 或已经调用过 finalize()，直接不可触及；否则进入可复活的队列；
          2. 第二次标记：Finalizer 线程会触发可复活队列对象的 finalize()，若执行后对象没有复活，则变为不可触及；否则从队列移除，变为可触及的
  - 清除

    释放掉无用对象占用的空间
    - 标记清除

      标记：从引用根节点开始遍历，在对象头中标记所有可达对象
      清除：GC 对堆区从头到尾遍历，回收对象头中没有标记可达的对象；清除是将垃圾对象地址放入空闲列表

      ---缺点---
      1. 清理产生内存碎片，需要维护一个空闲列表
      2. GC 时需要停止整个进程，效率不算高，延迟体验差
    - 复制

将正在使用的对象复制到另一块内存中，之后清除当前内存块中所有对象

---优点---
1. 没有标记、清除过程，高效简单
2. 空间连续，无内存碎片，无需维护空闲列表

---缺点---
1. 需要双倍空间
2. G1 需要维护大量 region 之间对象的引用关系，时间空间消耗都不小

适用于回收率高，对象数量不大的区域

- 标记整理

  标记：从根节点开始在对象头标记所有可达对象
  整理：将所有可达对象压缩到内存一段，之后清除边界外所有对象，比标记清除少维护空闲列表

  ---优点---
  1. 相比标记清除：无碎片，不需要维护空闲列表，只需要持有一个边界地址
  2. 相比复制算法：无需双倍内存

  ---缺点---
  1. 效率低于复制算法
  2. 移动对象，需要同事调整其他对象的引用地址
  3. 移动过程需要 STW

- 复合算法
  - 分代收集

    新生代：区域小，对象生命周期短，存活率低，回收频繁 -> 复制算法
    老年代：区域大，对象生命周期长，存活率高，回收频率低 -> 标记清除 + 整理

    Mark ： 存活对象数量
    Sweep ： 管理区域大小
    Compact ： 存活对象数量

  - 增量收集

    允许 GC 线程分阶段完成标记、清理或复制，能减少系统挺短时间，但线程切换和上下文转换使 GC 总体成本上升，吞吐量下降

  - 分区收集

    将堆区根据目标停顿时间，对象生命周期长短等，合理划分成多个不同分区，独立回收，可以控制一次回收的区域大小和停顿时间

Conception

- System.gc() vs. Runtime.getRuntime().gc()

  都能显示调用 FullGC，System.gc() 无法保证立即生效

- 内存溢出

  没有空闲内存，且 GC 无法提供更多内存

- 内存泄漏

  对象不会被用到，而又无法被 GC 回收

  单例模式，一些需要 close 的资源未关闭等

- Stop The World

  JVM 在 GC Roots 分析时，把用户工作线程全部暂停，所有 GC 都不能避免 STW，只能尽可能缩短暂停时间
  GC Roots 分析的准确性需要基于一个能保证内存一致性的快照进行

- Concurrent vs. Parallel

  并发：多个事情在同一段时间内同时发生
  并行：多个事情在同一时间点上同事发生

  Serial GC：单个 GC 线程，暂停所有用户线程
  Parallel GC：多个 GC 线程并行，暂停所有用户线程
  Concurrent GC：多个 GC 线程并行，与用户线程（并行，或交替），尽可能减少用户线程停顿时间

- 安全点

  只有在特定位置（SafePoint）才能停顿下来 GC

  抢先式中断：中断所有线程，存在不在安全点的线程则恢复线程，让其跑到安全点
  主动式中断：设置中断标志，各线程运行到安全点时主动轮询该标志，标志为真则挂起

  SafeRegion：正对 sleep、blocked 等对象

Reference

GC Roots 可达性

- StrongReference

强引用，普通对象引用，被强引用变量引用的对象，处于可达状态，即使 OOM 也不回收

---特点---
1. 可以直接访问引用对象
2. 情愿 OOM 也不回收
3. 可能导致内存泄漏

- SoftReference

  软引用，内存充足时不回收，内存不足回收
  OOM 之前对其尝试回收，回收还是失败则 OOM

  ---特点---

  场景：图片缓存

- WeakReference

  弱引用，有 GC 操作即回收，对象只能存活到下次 GC

  场景：WeekHashMap，GC 时回收 key-Node

- PhantomReference

  虚引用，没有任何引用，get 总是返回 null，和引用队列 ReferenceQueue 联合使用，GC 时引用的对象被送至引用队列，用于实现对象被回收的通知

  场景：在回收时通知相关操作，Spring AOP 后置通知

  - ReferenceQueue

    可以传入软引用、弱引用、虚引用，引用的对象被回收前，会被添加到引用队列

- FinalReference

  终结器引用，用于实现对象的 finalize()，无需手动编写，JVM 自动让重写了 finalize() 且未调用过 finalize() 的对象在 GC Roots 不可达时入队，第二次标记时不可达才会被回收

- **GC**

---分类---
1. 串行（Serail）与并行（Parallel）
2. 并发（CMS）与独占
3. 压缩与碎片

---指标---
1. 吞吐量 - 运行用户代码时间：总运行时间 <- 重点
2. GC 开销 - GC 时间：总运行时间，吞吐量的补数
3. 暂停时间 - GC 期间用户线程被暂停的时间 <- 重点
4. 收集频率
5. 内存占用 - 在 JVM 堆区的占比 <- 提升硬件
6. 生命周期 - 对象从诞生到回收的时长

在最大吞吐量优先情况下降低停顿时间

  - Young

    复制算法，避免碎片

    - UseSerialGC

      JDK1.3前新生代唯一，Client 模式默认新生代 GC
      复制算法、串行

    - UseParNewGC

      SerialGC 的多线程版本
      除 Serial 外唯一与 CMS 配合的年轻代 GC

      -XX：ParallelGCThreads，限制 GC 线程数量，默认是 CPU 核数

    - UseParallelGC

      - -XX:MaxGCPauseMills

        自适应调节策略，吞吐量优先

  - Old

    标记清除、整理

    - UseSerialOldGC

      Client 模式默认老年代 GC
      1. 与新生代 Paralle Scavenge 配合使用
      2. CMS 后背垃圾

      标记整理，串行

    - UseConcMarkSweepGC

      并发标记清除，低停顿，CPU压力大，内存碎片多

      - Initial Mark

        标记 GC Roots 能直接关联的对象

      - Concurrent Mark

GC Roots 跟踪，标记全部对象

- Remark

  修正并发标记期间发生变动的一部分对象的标记记录

- Concurrent Sweep

  清除 GC Roots 不可达对象

- UseParallelOldGC

  并行标记整理

- UseG1GC

  相比 CMS，碎片小，STW时间可控

  - 过程
    - 初始标记
    - 并发标记
    - 最终标记
    - 筛选回收
      - 根据时间进行价值最大化回收
  - 参数
    - -XX:+UseG1GC
    - -XX:G1HeapRegionSize

      G1 区域大小，2的冥，范围 1MB~32MB

    - -XX:MaxGCPauseMillis

      最大 GC 停顿时间(ms)，软指标

    - -XX:InitiatingHeapOccupancyPercent

      触发 GC 的堆占比阈值

    - -XX:ConcGCThreads

      并发GC使用的线程数

    - -XX:G1ReservePercent

      作为空闲空间的预留百分比