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，最小化的使用内存和并行开销
  Parallel GC：多个 GC 线程并行，暂停所有用户线程，最大化吞吐量
  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，最小化 GC 的中断和停顿

- 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

---选择---
1. 优先调整堆大小，让 JVM 自适应选择
2. 内存小于 100M，使用串行
3. 单核、单机程序、停顿时间不敏感，使用串行
4. 多核、要求高吞吐、运行停顿超过 1s，选择并行或JVM 自选
5. 多核、要求低延迟、快速响应、使用并发
6. 官方推荐 G1，互联网首选

- 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、STW

  - UseParNewGC

    SerialGC 的多线程版本（并行）
    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

    复制，并行，STW，自适应调节策略，吞吐量优先

    -XX:MaxGCPauseMills，设置 GC 最大停顿时间

- 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，STW

  - UseConcMarkSweepGC

    并发标记清除，低停顿，CPU压力大，内存碎片多，清除+空闲列表

    Initial Mark - 初始标记，标记 GC Roots直接可达对象，STW
    Concurrent Mark - 并发标记，从 GC Roots 直接可达对象开始遍历标记整个对象图，与用户线程并发进行
    Remark - 重新标记，修正并发标记期间发生变动的一部分对象的标记记录，STW
    Concurrent-Sweeop - 并发清除，清理标记阶段判断为已死亡的对象，与用户线程并发进行

    要保证用户线程并发进行，需要给用户线程保留一定内存，因此 CMS 不是到老年代空间耗尽时进行 GC，而是达到阈值就 GC
    当 GC 期间用户线程不足时，临时启用 SerialOldGC 重新对老年代 GC

    ---优点---
    并发收集，低延迟

    ---缺点---
    1. 内存碎片，导致大对象内存分配失败，提前 Full GC
    2. CPU 资源敏感，占用用户线程，导致用户程序变慢，总吞吐量降低
    3. 浮动垃圾无法处理，在并发标记阶段新产生的垃圾只能在下次 GC 时被回收

  - UseParallelOldGC

并行，标记整理，STW

- UseG1GC

标记整理，相比 CMS，碎片小，STW 时间可控

未必比 CMS 最好情况下停顿少，但比最差情况好很多
CMS 适用于小内存应用，G1 适用于大内存引用，平衡点：6~8GB

延迟可控情况下尽可能搞的吞吐量：避免整堆回收，在后台维护一个优先列表，每次根据允许的回收时间，优先回收价值最大的 Region

---特点---
并行性：多 GC 线程并行，用户线程有 STW
并发性：与用户线程交替进行，部分阶段可以与应用程序同时执行
分代：不要求整个 Eden、年轻代、老年代连续，不在固定大小和数量
分区：划分区域，分区包含年轻代和老年代
兼容性：同时适用新生代和老年代

可以调用用户线程加速垃圾回收过程

Humongous - 用于存储大对象（超过1.5 Region），一个不够就使用多个，G1 一般将 Humongous 看作老年代的一部分

- 过程

1. 新生代的 Eden 用完时开始 Young GC，并行，STW
2. 堆内存使用大道阈值（默认45%），开始老年代并发标记
3. 混合回收，进行新生代 GC，同时移动老年代存活对象到空闲 Region，一次只扫描小部分老年代 Region，STW

  - Young GC

    1. 扫描 GC Roots + Remembered Set
    2. 更新 Remembered Set，处理 dirty card queue，修正更改
    3. 处理 Remembered Set，这些对象指向的 Eden 中的对象是存活的
    4. 复制存活对象到 Survivor 区
    5. 处理引用，Soft，Weak，Phantom，Final，JNI Weak 等引用

  - 并发标记

    初始标记：标记 GC Roots 直接可达对象，并触发一次 Young GC，STW
    Root Region Scanning：根区域扫描，在 Young GC 之前，扫描 Survivor 区直接可达的老年区对象，标记被引用对象
    Concurrent Marking：整堆并发标记，记录 Region 活性，全垃圾 Region 直接回收，可能被 Young GC 打断
    Remark：再次标记，采用初始快照算法，修正上次标记的结果，STW
    CleanUp：独占清理，对 Region 活性排序，为混合回收做准备，STW
    并发清理：识别并清理完全空闲区域

  - 混合回收

    分段回收，默认分 8 段（-XX:G1MixedGCCountTarget），优先回收活性最低的分段

  - Full GC

    G1 回收失败，退回 Full GC

- Remembered Set

  Remembered Set 作为临时加入到 GC Roots 的对象集合

  写入引用所指向的对象时，若跨 Region，在被引用者所在区的 Remembered Set 记录引用者，对某个 Region 执行 GC 时，只需要从 GC Roots 外加该 Region 的 Remembered Set 开始遍历，避免了全局扫描，也不会遗漏

- ZGC

JDK 11 出现，并发标记压缩算法，试验阶段
保持吞吐量情况下，极低延迟
不设分代，基于 Region，读屏障，染色指针、内存多重映射等
并发标记-并发预备重分配-并发重分配-并发重映射

- Shenandoah

Open JDK 12，低停顿，实验性

- 参数

  - -XX:ParallelGCThreads

    限制年轻代 GC 线程数量，默认是 CPU 核数，CPU 核数大于 8 时等于 3+[5* CPU]/8

    G1 最大为 8

  - -XX:MaxGCPauseMillis

    最大 GC 停顿时间(ms)，软指标

    G1 默认200 ms

  - -XX:GCTimeRadio

    GC 时间占总时间的比例

  - -XX:CMSInitiatingoccupanyFraction

    内存使用率阈值，达到阈值开始 CMS GC，增长缓慢的应当设置大些

    JDK5 - 68%
    JDK6 - 92%

  - -XX:+UseCMSCompactAtFullCollection

启用 Full GC 后的压缩整理

- -XX:CMSFullGCBeforeCompaction

  设置多少次 Full GC 后对内存进行压缩整理

- -XX:ParallelCMSThreads

  CMS 线程数量
  默认（ParallelGCThreads+3)/4

- -XX:ConcGCThreads

  并发标记使用的线程数，建议设置为 ParallelGCThreads 的 1/4 左右

- -XX:G1HeapRegionSize

  G1 区域大小，2的冥，范围 1MB~32MB，默认是堆的 1/2000
  一旦使用该参数，所有 Region 大小相同，且 JVM 生命周期内不会被改变

- -XX:InitiatingHeapOccupancyPercent

  触发 GC 的堆占比阈值，默认 45

- -XX:G1ReservePercent

  作为空闲空间的预留百分比

- -XX:G1MixedGCLiveThresholdPercent

  默认 65%，G1 Mixed 分段垃圾占比达到该值才回收

- -XX:G1HeapWastePercent

  默认 10%，整堆允许被浪费的空间占比，低于该值不进行混合回收

- -XX:+PrintGC

  输出简要堆变化，类似 -verbose:gc

- -XX:+PrintGCDetails

  输出 GC 详细日志

- -XX:+PrintGCTimestamps

  输出 GC 的时间戳（以基准时间的形式）

- -XX:+PrintGCDatestamps

  输出 GC 时间戳（以日期的形式）

- -XX:+PrintHeapAtGC

  在 GC 前后打印堆信息
  -Xloggc:.../logs/gc.log 指定日志文件输出路径