- JVM 内存区域
  - 线程私有
    - 程序计数器
      - 当前线程所执行的字节码的行号指示器
      - 对于 Java 方法，记录正在执行的虚拟机字节码指令的地址；对于 native 方法，记录值为空（Undefined）
      - 唯一一个Java 虚拟机规范中没有规定任何 OutOfMemoryError 的内存区域
    - Java 虚拟机栈
      - 每个线程都有独自的虚拟机栈，并且在栈中的结构是栈帧，当调用一个方法时，会为这个方法创建一个栈帧，每个方法对应的是入栈、出栈的过程
      - 栈帧中包括局部变量表、操作数栈、动态链接、方法出口等信息
      - 线程请求的栈深度大于虚拟机规定的栈深度，会抛出 StackOverflowError，当栈空间动态扩展，但无法申请足够的内存，将抛出 OutOfMemoryError
    - 本地方法栈
      - 与虚拟机栈类似，区别在于虚拟机栈执行Java方法，本地方法栈执行 Native 方法
  - 线程共享
    - 堆
      - 堆是垃圾回收的主要区域，很多时候被称为 GC堆
      - 堆被分为新生代、老年代：Eden Space（伊甸园）、Survivor Space（幸存者区）、Tenured Gen（老年代）
      - 几乎所有的对象实例及数组都在堆上分配
      - 当内存空间不足时，无法完成实例分配，将抛出 OutOfMemoryError
    - 方法区
      - 方法区在 HotSpot 中又被称为永久代
      - 存储虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据
      - 该区域的回收目标主要是对常量池的回收以及类的卸载
      - 当内存空间不足时，无法为方法区开辟新空间时，将抛出 OutOfMemoryError
      - 运行时常量池，存储类加载后的常量池信息
- 垃圾回收
  - why - 为什么要了解垃圾回收
    - 提高系统性能，突破应用性能瓶颈
    - 排查各种内存溢出，内存泄漏的问题
  - what1 - 哪些内存区域需要回收
    - Java 堆和方法区
    - Java 堆是对无用的对象的回收，方法区是对废弃的常量以及无用的类进行回收
  - what2 - 哪些对象需要回收（对象存活判定方法）
    - 引用计数法
      - 原理：给对象添加引用计数器，每当有地方引用它，引用计数器加1；引用失效，计数器减1；当计数器的值为 0，对象可被回收
      - 优点：实现简单，判定效率高
      - 缺点：存在循环引用的问题
    - 可达性分析法（GC Roots）
      - 原理：通过一系列的称为「GC Roots」的对象作为起始点，从起始点搜索的路径称为引用链

，当一个对象没有与任何引用链相连时，则证明此对象是不可用的
        - GC Roots 对象种类
            - Java 虚拟机栈中引用的对象
            - 本地方法栈 (Native方法) 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

- when - 什么时候进行回收
    - 达到 Minor GC 或者 Full GC 的触发条件时
    - Minor GC 触发条件
        - 当新生代（Eden区）空间不足时，发起一次 Minor GC
    - Full GC 触发条件
        - 调用 System.gc()，系统建议执行 Full GC，但是不必然执行
        - 当老年代空间不足时
        - 方法区空间不足时
        - 历次通过 Minor GC 后进入老年代的平均大小大于老年代的可用内存时
        - 由Eden区、From Survior区向To Survior区复制时，对象大小大于ToSurvior区可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小时
- how - 如何回收（垃圾回收算法）
    - 标记 - 清除算法
        - 原理：该算法分为「标记」和「清除」两个阶段，首先标记出所有需要回收的对象，在标记完成后统一进行回收
        - 特点
            - 效率：标记和清除两个阶段的效率都不高
            - 空间：标记清除之后未进行碎片整理，产生大量不连续的内存碎片，分配大对象时空间不够会经常触发垃圾回收，最终影响的仍然是效率
    - 标记 - 整理算法
        - 原理：该算法分为「标记」和「整理」两个阶段，标记如同标记清除中的标记，整理是指不直接对标记的对象进行清理，而是让所有存活的对象移动到一边，清除掉端边界以外的内存
        - 特点
            - 效率：标记和整理两个阶段的效率不高
            - 空间：优点是解决了内存碎片的问题
    - 复制算法（新生代采用的算法）
        - 原理：新生代分为 Eden、Survivor1（from）、Survivor2（to）三部分，每次使用Eden和其中一块Survivor，回收时，将存活对象复制到另一块Survivor，然后清理到刚使用的Eden和Survivor
        - 特点
            - 效率：每次对整个半区进行回收，内存分配是不需要考虑内存碎片的情况；内存分配时只需要移动堆顶指针，效率高
            - 空间：未进行垃圾回收时，会有一部分空间未用上
    - 分代收集算法
        - 原理： 将Java 堆分为新生代和老年代，根据各个年代的特点采用适当的收集算法
        - 特点：充分利用不同各个年代的特点采用最适当的收集算法
        - 新生代：次垃圾收集时都发现只有少量存活，选择使用复制算法，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：因为对象存活率高、没有额外空间进行分配担保，使用「标记-清理」或者「标记-整理」算法进行回收

- 垃圾收集器
    - Serial 收集器
        - 算法：堆内存年轻代采用“复制算法”；堆内存老年代采用“标记-整理算法”
        - 单线程收集器，只会使用一个 CPU 或一条收集线程去完成垃圾收集工作
        - 垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束（又称为「Stop The World」）
        - 优点是简单而高效(与其他收集器的单线程相比)，对于限定单个 CPU 的环境来说，Serial 收集器由于没有线程交互的开销，专心做垃圾收集自然就可以获得最高的单线程收集效率
    - ParNew 收集器
        - 算法：堆内存年轻代采用“复制算法”
        - ParNew 收集器就是 Serial 收集器的多线程版本
        - 只能用于新生代
        - 多线程收集，并行
        - 在多 CPU 环境下，随着 CPU 的数量增加，它对于 GC 时系统资源的有效利用是有益的。它默认开启的收集线程数与 CPU 的数量相同
        - ParNew 收集器在单 CPU 的环境中绝对不会有比 Serial 收集器有更好的效果，甚至由于存在线程交互的开销，该收集器在通过超线程技术实现的两个 CPU 的环境中都不能百分之百地保证可以超越
    - Parallel Scavenge 收集器
        - 算法：堆内存年轻代采用“复制算法”；配合收集器：ParallelOldGC，堆内存老年代采用“标记-整理算法”
        - 多线程收集器
        - 只适用于新生代
        - 自适应调节策略
        - Parallel Scavenge收集器的目标是达到一个可控制的吞吐量
        - Parallel Scavenge 收集器无法与 CMS 收集器配合使用
    - CMS 收集器
        - 运作过程
            - 初始标记
                - 仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，需要“Stop The World”
            - 并发标记
                - 进行 GC Roots 追溯所有对象的过程，在整个过程中耗时最长
            - 重新标记
                - 为了修正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对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。此阶段也需要“Stop The World”
            - 并发清除
        - 特点
            - Concurrent Mark Sweep，基于“标记-清除”算法实现
            - 各阶段耗时：并发标记/并发清除 > 重新标记 > 初始标记
            - 对 CPU 资源非常敏感
            - 标记-清除算法导致的空间碎片

        - 并发收集、低停顿，因此 CMS 收集器也被称为并发低停顿收集器
        - 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
        - 由于整个过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作；所以，从总体上来说，CMS 收集器的内存回收过程是与用户线程一起并发执行的。
    - G1 收集器
        - 运作过程
            - 初始标记
                - 仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，需要“Stop The World”
            - 并发标记
                - 进行 GC Roots 追溯所有对象的过程，可与用户程序并发执行
            - 最终标记
                - 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
            - 筛选回收
                - 对各个Region的回收价值和成本进行排序，根据用户所期望的 GC 停顿时间来指定回收计划
        - 特点
            - 空间整合：整体上看来是基于“标记-整理”算法实现的，从局部（两个Region）上看来是基于“复制”算法实现的
            - 面向服务端应用的垃圾收集器
            - 并行与并发
            - 分代收集
            - 可预测的停顿：G1收集器可以非常精确地控制停顿，既能让使用者明确指定在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒，这几乎已经是实时Java的垃圾收集器的特征
            - G1将整个Java堆（包括新生代、老年代）划分为多个大小相等的内存块（Region），每个Region 是逻辑连续的一段内存，在后台维护一个优先列表，每次根据允许的收集时间，优先回收垃圾最多的区域

- 问题总结
    - 局部变量表与操作数栈有什么区别？
        - 局部变量表是存放编译器可知的各种基本数据类型、对象引用类型和方法返回地址；通过索引访问；局部变量表所需内存空间在编译期已确定大小
        - 操作数栈是工作区，在这里进行根据指令对数据进行压栈、出栈进行数据运算；通过出栈入栈方式访问；
    - StackOverflowError 与 OutOfMemoryError 的区别？
        - StackOverflowError 是指栈请求的深度大于虚拟机规定的栈深度，此时内存空间可能还足够
        - OutOfMemoryError 是指内存空间不足，无法分配内存
    - 内存泄漏和内存溢出的区别？
        - 申请内存后，无法释放已申请的内存空间，内存泄漏堆积的结果是内存溢出
        - 内存溢出是指申请内存时，没有足够的内存空间供申请者使用
    - 符号引用和直接引用的区别？
        - 符号引用是字面量，用符号来描述引用目标，只包含语义信息，与具体实现无关
        - 直接引用是与具体实现息息相关，是直接指向目标的指针

    - 常量池和运行时常量池的区别？
    - 常量池存在于静态的存储文件中，Java 中表现为 .class 文件，主要包含字面量和符号引用
    - 运行时常量池存在于内存中，是常量池被加载到内存之后的版本，并且字面量可以动态添加，比如 String.intern()
- Minor GC 与 Full GC 的区别
    - Minor GC 称为新生代GC，是发生在新生代的垃圾回收动作，因为 Java 对象绝大多数具备朝生夕灭的特性，Minor GC 触发非常频繁，并且回收速度很快。
    - Full GC 称为老年代GC，发生在老年代，出现了 Full GC，通常会伴随一次 Minor GC。Full GC 的速度一般比 Minor GC 慢 10 倍以上。

收集器	运行	区域	算法	目标	适用场景
Serial	串行	新生代	复制算法	响应速度优先	单CPU环境下的Client模式
Serial Old	串行	老年代	标记-整理	响应速度优先	单CPU环境下的Client模式、CMS的后备预案
ParNew	并行	新生代	复制算法	响应速度优先	多CPU环境时在Server模式下与CMS配合
Parallel Scavenge	并行	新生代	复制算法	吞吐量优先	在后台运算而不需要太多交互的任务
Parallel Old	并行	老年代	标记-整理	吞吐量优先	在后台运算而不需要太多交互的任务
CMS	并发	老年代	标记-清除	响应速度优先	集中在互联网站或B/S系统服务端上的Java应用
G1	并发	both	标记-整理+复制算法	响应速度优先	面向服务端应用，将来替换CMS