第三版：JVM 45 道

栈帧里面包含哪些东西?

局部变量表、操作数栈、动态连接、返回地址等

程序计数器有什么作用?

程序计数器是一块较小的内存空间，它的作用可以看作是当前线程所执行的字节码的行号指示器。这里面存的，就是当前线程执行的进度。程序计数器还存储了当前正在运行的流程，包括正在执行的指令、跳转、分支、循环、异常处理等。

字符串常量存放在哪个区域?

1、 字符串常量池，已经移动到堆上（jdk8 之前是 perm 区），也就是执行 intern 方法后存的地方。

2、 类文件常量池，constant_pool，是每个类每个接口所拥有的，这部分数据在方法区，也就是元数据区。而运行时常量池是在类加载后的一个内存区域，它们都在元空间。

你熟悉哪些垃圾收集算法?

标记清除（缺点是碎片化） 复制算法（缺点是浪费空间） 标记整理算法（效率比前两者差） 分代收集算法（老年代一般使用“标记-清除”、“标记-整理”算法，年轻代一般用复制算法）

Java 里有哪些引用类型?

1、 强引用 这种引用属于最普通最强硬的一种存在，只有在和 GC Roots 断绝关系时，才会被消灭掉。

2、 软引用 软引用用于维护一些可有可无的对象。在内存足够的时候，软引用对象不会被回收，只有在内存不足时，系统则会回收软引用对象，如果回收了软引用对象之后仍然没有足够的内存，才会抛出内存溢出异常。可以看到，这种特性非常适合用在缓存技术上。比如网页缓存、图片缓存等。软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，Java 虚拟机就会把这个软引用加入到与之关联的引用队列中。

3、 弱引用 弱引用对象相比较软引用，要更加无用一些，它拥有更短的生命周期。当 JVM 进行垃圾回收时，无论内存是否充足，都会回收被弱引用关联的对象。弱引用拥有更短的生命周期，在 Java 中，用 java.lang.ref.WeakReference 类来表示。它的应用场景和软引用类似，可以在一些对内存更加敏感的系统里采用。

4、 虚引用 这是一种形同虚设的引用，在现实场景中用的不是很多。虚引用必须和引用队列（ReferenceQueue）联合使用。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。实际上，虚引用的 get，总是返回 null。

JVM 怎么判断一个对象是不是要回收?

引用计数法（缺点是对于相互引用的对象，无法进行清除） 可达性分析

GC Roots 有哪些?

1、 GC Roots 是一组必须活跃的引用。用通俗的话来说，就是程序接下来通过直接引用或者间接引用，能够访问到的潜在被使用的对象。

2、 GC Roots 包括：Java 线程中，当前所有正在被调用的方法的引用类型参数、局部变量、临时值等。也就是与我们栈帧相关的各种引用。所有当前被加载的 Java 类。Java 类的引用类型静态变量。运行时常量池里的引用类型常量（String 或 Class 类型）。JVM 内部数据结构的一些引用，比如 sun.jvm.hotspot.memory.Universe 类。用于同步的监控对象，比如调用了对象的 wait() 方法。JNI handles，包括 global handles 和 local handles。

3、 这些 GC Roots 大体可以分为三大类，下面这种说法更加好记一些：活动线程相关的各种引用。类的静态变量的引用。JNI 引用。

4、 有两个注意点：我们这里说的是活跃的引用，而不是对象，对象是不能作为 GC Roots 的。GC 过程是找出所有活对象，并把其余空间认定为“无用”；而不是找出所有死掉的对象，并回收它们占用的空间。所以，哪怕 JVM 的堆非常的大，基于 tracing 的 GC 方式，回收速度也会非常快。

你知道哪些 GC 类型?

Minor GC：发生在年轻代的 GC。Major GC：发生在老年代的 GC。Full GC：全堆垃圾回收。比如 Metaspace 区引起年轻代和老年代的回收。

对象都是优先分配在年轻代上的吗?

不是。当新生代内存不够时，老年代分配担保。而大对象则是直接在老年代分配。

你了解过哪些垃圾收集器?

年轻代 Serial 垃圾收集器（单线程，通常用在客户端应用上。因为客户端应用不会频繁创建很多对象，用户也不会感觉出明显的卡顿。相反，它使用的资源更少，也更轻量级。） ParNew 垃圾收集器（多线程，追求降低用户停顿时间，适合交互式应用。） Parallel Scavenge 垃圾收集器（追求 CPU 吞吐量，能够在较短时间内完成指定任务，适合没有交互的后台计算。）

老年代 Serial Old 垃圾收集器 Parallel Old 垃圾收集器 CMS 垃圾收集器（以获取最短 GC 停顿时间为目标的收集器，它在垃圾收集时使得用户线程和 GC 线程能够并发执行，因此在垃圾收集过程中用户也不会感到明显的卡顿。）

说说 CMS 垃圾收集器的工作原理

Concurrent mark sweep(CMS)收集器是一种年老代垃圾收集器，其最主要目标是获取最短垃圾回收停顿时间， 和其他年老代使用标记-整理算法不同，它使用多线程的标记-清除算法。最短的垃圾收集停顿时间可以为交互比较高的程序提高用户体验。CMS 工作机制相比其他的垃圾收集器来说更复杂

整个过程分为以下 4 个阶段：

1、 初始标记 只是标记一下 GC Roots 能直接关联的对象，速度很快，仍然需要暂停所有的工作线程。

2、 并发标记 进行 GC Roots 跟踪的过程，和用户线程一起工作，不需要暂停工作线程。

3、 重新标记 为了修正在并发标记期间，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，仍然需要暂停所有的工作线程。

4、 并发清除 清除 GC Roots 不可达对象，和用户线程一起工作，不需要暂停工作线程。由于耗时最长的并发标记和并发清除过程中，垃圾收集线程可以和用户线程一起并发工作， 所以总体上来看 CMS 收集器的内存回收和用户线程是一起并发地执行。

说说 G1 垃圾收集器的工作原理

优点：指定最大停顿时间、分 Region 的内存布局、按收益动态确定回收集

1、 G1 开创的基于 Region 的堆内存布局是它能够实现这个目标的关键。虽然 G1 也仍是遵循分代收集理论设计的，但其堆内存的布局与其他收集器有非常明显的差异：G1 不再坚持固定大小以及固定数量的分代区域划分，而是把连续的 Java 堆划分为多个大小相等的独立区域（Region），每一个 Region 都可以根据需要，扮演新生代的 Eden 空间、Survivor 空间，或者老年代空间。收集器能够对扮演不同角色的 Region 采用不同的策略去处理，这样无论是新创建的对象还是已经存活了一段时间、熬过多次收集的旧对象都能获取很好的收集效果。

2、 虽然 G1 仍然保留新生代和老年代的概念，但新生代和老年代不再是固定的了，它们都是一系列区域（不需要连续）的动态集合。G1 收集器之所以能建立可预测的停顿时间模型，是因为它将 Region 作为单次回收的最小单元，即每次收集到的内存空间都是 Region 大小的整数倍，这样可以有计划地避免在整个 Java 堆中进行全区域的垃圾收集。更具体的处理思路是让 G1 收集器去跟踪各个 Region 里面的垃圾堆积的“价值”大小，价值即回收所获得的空间大小以及回收所需时间的经验值，然后在后台维护一个优先级列表，每次根据用户设定允许的收集停顿时间（使用参数-XX：MaxGCPauseMillis 指定，默认值是 200 毫秒），优先处理回收价值收益最大的那些 Region，这也就是“Garbage First”名字的由来。这种使用 Region 划分内存空间，以及具有优先级的区域回收方式，保证了 G1 收集器在有限的时间内获取尽可能高的收集效率。

3、 G1 收集器的运作过程大致可划分为以下四个步骤：·初始标记 （Initial Marking）：仅仅只是标记一下 GC Roots 能直接关联到的对象，并且修改 TAMS 指针的值，让下一阶段用户线程并发运行时，能正确地在可用的 Region 中分配新对象。这个阶段需要停顿线程，但耗时很短，而且是借用进行 Minor GC 的时候同步完成的，所以 G1 收集器在这个阶段实际并没有额外的停顿。·并发标记 （Concurrent Marking）：从 GC Root 开始对堆中对象进行可达性分析，递归扫描整个堆里的对象图，找出要回收的对象，这阶段耗时较长，但可与用户程序并发执行。当对象图扫描完成以后，还要重新处理 SATB 记录下的在并发时有引用变动的对象。·最终标记 （Final Marking）：对用户线程做另一个短暂的暂停，用于处理并发阶段结束后仍遗留下来的最后那少量的 SATB 记录。筛选回收（Live Data Counting and Evacuation）：负责更新 Region 的统计数据，对各个 Region 的回收价值和成本进行排序，根据用户所期望的停顿时间来制定回收计划，可以自由选择任意多个 Region 构成回收集，然后把决定回收的那一部分 Region 的存活对象复制到空的 Region 中，再清理掉整个旧 Region 的全部空间。这里的操作涉及存活对象的移动，是必须暂停用户线程，由多条收集器线程并行完成的。从上述阶段的描述可以看出，G1 收集器除了并发标记外，其余阶段也是要完全暂停用户线程的 。

说说 ZGC 垃圾收集器的工作原理

内存布局 ·小型 Region（Small Region）：容量固定为 2MB，用于放置小于 256KB 的小对象。·中型 Region （Medium Region）：容量固定为 32MB，用于放置大于等于 256KB 但小于 4MB 的对象。·大型 Region （Large Region）：容量不固定，可以动态变化，但必须为 2MB 的整数倍，用于放置 4MB 或以上的大对象。每个大型 Region 中只会存放一个大对象，这也预示着虽然名字叫作“大型 Region”，但它的实际容量完全有可能小于中型 Region，最小容量可低至 4MB。大型 Region 在 ZGC 的实现中是不会被重分配（重分配是 ZGC 的一种处理动作，用于复制对象的收集器阶段，稍后会介绍到）的，因为复制一个大对象的代价非常高昂。

染色指针 染色指针是一种直接将少量额外的信息存储在指针上的技术，可是为什么指针本身也可以存储额外信息呢？在 64 位系统中，理论可以访问的内存高达 16EB（2 的 64 次幂）字节 [3] 。实际上，基于需求（用不到那么多内存）、性能（地址越宽在做地址转换时需要的页表级数越多）和成本（消耗更多晶体管）的考虑，在 AMD64 架构 [4] 中只支持到 52 位（4PB）的地址总线和 48 位（256TB）的虚拟地址空间，所以目前 64 位的硬件实际能够支持的最大内存只有 256TB。此外，操作系统一侧也还会施加自己的约束，64 位的 Linux 则分别支持 47 位（128TB）的进程虚拟地址空间和 46 位（64TB）的物理地址空间，64 位的 Windows 系统甚至只支持 44 位（16TB）的物理地址空间。尽管 Linux 下 64 位指针的高 18 位不能用来寻址，但剩余的 46 位指针所能支持的 64TB 内存在今天仍然能够充分满足大型服务器的需要。鉴于此，ZGC 的染色指针技术继续盯上了这剩下的 46 位指针宽度，将其高 4 位提取出来存储四个标志信息。通过这些标志位，虚拟机可以直接从指针中看到其引用对象的三色标记状态、是否进入了重分配集（即被移动过）、是否只能通过 finalize()方法才能被访问到。当然，由于这些标志位进一步压缩了原本就只有 46 位的地址空间，也直接导致 ZGC 能够管理的内存不可以超过 4TB（2 的 42 次幂） 。

收集过程 ·并发标记 （Concurrent Mark）：与 G1、Shenandoah 一样，并发标记是遍历对象图做可达性分析的阶段，前后也要经过类似于 G1、Shenandoah 的初始标记、最终标记（尽管 ZGC 中的名字不叫这些）的短暂停顿，而且这些停顿阶段所做的事情在目标上也是相类似的。与 G1、Shenandoah 不同的是，ZGC 的标记是在指针上而不是在对象上进行的，标记阶段会更新染色指针中的 Marked 0、Marked 1 标志位。·并发预备重分配 （Concurrent Prepare for Relocate）：这个阶段需要根据特定的查询条件统计得出本次收集过程要清理哪些 Region，将这些 Region 组成重分配集（Relocation Set）。重分配集与 G1 收集器的回收集（Collection Set）还是有区别的，ZGC 划分 Region 的目的并非为了像 G1 那样做收益优先的增量回收。相反，ZGC 每次回收都会扫描所有的 Region，用范围更大的扫描成本换取省去 G1 中记忆集的维护成本。因此，ZGC 的重分配集只是决定了里面的存活对象会被重新复制到其他的 Region 中，里面的 Region 会被释放，而并不能说回收行为就只是针对这个集合里面的 Region 进行，因为标记过程是针对全堆的。此外，在 JDK 12 的 ZGC 中开始支持的类卸载以及弱引用的处理，也

是在这个阶段中完成的。·并发重分配 （Concurrent Relocate）：重分配是 ZGC 执行过程中的核心阶段，这个过程要把重分配集中的存活对象复制到新的 Region 上，并为重分配集中的每个 Region 维护一个转发表（Forward Table），记录从旧对象到新对象的转向关系。得益于染色指针的支持，ZGC 收集器能仅从引用上就明确得知一个对象是否处于重分配集之中，如果用户线程此时并发访问了位于重分配集中的对象，这次访问将会被预置的内存屏障所截获，然后立即根据 Region 上的转发表记录将访问转发到新复制的对象上，并同时修正更新该引用的值，使其直接指向新对象，ZGC 将这种行为称为指针的“自愈”（Self-Healing）能力。这样做的好处是只有第一次访问旧对象会陷入转发，也就是只慢一次，对比 Shenandoah 的 Brooks 转发指针，那是每次对象访问都必须付出的固定开销，简单地说就是每次都慢，因此 ZGC 对用户程序的运行时负载要比 Shenandoah 来得更低一些。还有另外一个直接的好处是由于染色指针的存在，一旦重分配集中某个 Region 的存活对象都复制完毕后，这个 Region 就可以立即释放用于新对象的分配（但是转发表还得留着不能释放掉），哪怕堆中还有很多指向这个对象的未更新指针也没有关系，这些旧指针一旦被使用，它们都是可以自愈的。·并发重映射 （Concurrent Remap）：重映射所做的就是修正整个堆中指向重分配集中旧对象的所有引用，这一点从目标角度看是与 Shenandoah 并发引用更新阶段一样的，但是 ZGC 的并发重映射并不是一个必须要“迫切”去完成的任务，因为前面说过，即使是旧引用，它也是可以自愈的，最多只是第一次 使用时多一次转发和修正操作。重映射清理这些旧引用的主要目的是为了不变慢（还有清理结束后可以释放转发表这样的附带收益），所以说这并不是很“迫切”。因此，ZGC 很巧妙地把并发重映射阶段要做的工作，合并到了下一次垃圾收集循环中的并发标记阶段里去完成，反正它们都是要遍历所有对象的，这样合并就节省了一次遍历对象图 [9] 的开销。一旦所有指针都被修正之后，原来记录新旧对象关系的转发表就可以释放掉了。

ZGC 收集器中的染色指针有什么用?

染色指针是一种直接将少量额外的信息存储在指针上的技术，可是为什么指针本身也可以存储额外信息呢？在 64 位系统中，理论可以访问的内存高达 16EB（2

的 64 次幂）字节 [3] 。实际上，基于需求（用不到那么多内存）、性能（地址越宽在做地址转换时需要的页表级数越多）和成本（消耗更多晶体管）的考虑，在 AMD64 架构 [4] 中只支持到 52 位（4PB）的地址总线和 48 位（256TB）的虚拟地址空间，所以目前 64 位的硬件实际能够支持的最大内存只有 256TB。此外，操作系统一侧也还会施加自己的约束，64 位的 Linux 则分别支持 47 位（128TB）的进程虚拟地址空间和 46 位（64TB）的物理地址空间，64 位的 Windows 系统甚至只支持 44 位（16TB）的物理地址空间。尽管 Linux 下 64 位指针的高 18 位不能用来寻址，但剩余的 46 位指针所能支持的 64TB 内存在今天仍然能够充分满足大型服务器的需要。鉴于此，ZGC 的染色指针技术继续盯上了这剩下的 46 位指针宽度，将其高 4 位提取出来存储四个标志信息。通过这些标志位，虚拟机可以直接从指针中看到其引用对象的三色标记状态、是否进入了重分配集（即被移动过）、是否只能通过 finalize()方法才能被访问到。当然，由于这些标志位进一步压缩了原本就只有 46 位的地址空间，也直接导致 ZGC 能够管理的内存不可以超过 4TB（2 的 42 次幂） 。

说说类加载的过程

加载 验证 准备（为一些类变量分配内存，并将其初始化为默认值） 解析（将符号引用替换为直接引用。类和接口、类方法、接口方法、字段等解析） 初始化

说下有哪些类加载器?

Bootstrap ClassLoader（启动类加载器） Extention ClassLoader（扩展类加载器） App ClassLoader（应用类加载器）

什么是双亲委派机制?

双亲委派机制的意思是除了顶层的启动类加载器以外，其余的类加载器，在加载之前，都会委派给它的父加载器进行加载。这样一层层向上传递，直到祖先们都无法胜任，它才会真正的加载。

双亲委派机制可以被违背吗？请举例说明。

可以被违背。打破双亲委派的例子：Tomcat

对于一些需要加载的非基础类，会由一个叫作 WebAppClassLoader 的类加载器优先加载。等它加载不到的时候，再交给上层的 ClassLoader 进行加载。这个加载器用来隔绝不同应用的 .class 文件，比如你的两个应用，可能会依赖同一个第三方的不同版本，它们是相互没有影响的。

Tomcat 是怎么打破双亲委派机制的呢?

是通过重写 ClassLoader#loadClass 和 ClassLoader#findClass 实现的。可以看图中的 WebAppClassLoader，它加载自己目录下的.class 文件，并不会传递给父类的加载器。但是，它却可以使用 SharedClassLoader 所加载的类，实现了共享和分离的功能。

Java 对象的布局了解过吗?

对象头区域此处存储的信息包括两部分：1、对象自身的运行时数据（MarkWord），占 8 字节 存储 hashCode、GC 分代年龄、锁类型标记、偏向锁线程 ID 、CAS 锁指向线程 LockRecord 的指针等， synconized 锁的机制与这个部分（markwork）密切相关，用 markword 中最低的三位代表锁的状态，其中一位是偏向锁位，另外两位是普通锁位。2、对象类型指针（Class Pointer），占 4 字节 对象指向它的类元数据的指针、 JVM 就是通过它来确定是哪个 Class 的实例。

实例数据区域 此处存储的是对象真正有效的信息，比如对象中所有字段的内容

对齐填充区域 JVM 的实现 HostSpot 规定对象的起始地址必须是 8 字节的整数倍，换句话来说，现在 64 位的 OS 往外读取数据的时候一次性读取 64bit 整数倍的数据，也就是 8 个字节，所以 HotSpot 为了高效读取对象，就做了"对齐"，如果一个对象实际占的内存大小不是 8byte 的整数倍时，就"补位"到 8byte 的整数倍。所以对齐填充区域的大小不是固定的。

什么情况下会发生栈内存溢出?

栈是线程私有的，他的生命周期与线程相同，每个方法在执行的时候都会创建一个栈帧，用来存储局部变量表，操作数栈，动态链接，方法出口等信息。局部变量表又包含基本数据类型，对象引用类型。如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出 StackOverflowError 异常，方法递归调用产生这种结果。如果 Java 虚拟机栈可以动态扩展，并且扩展的动作已经尝试过，但是无法申请到足够的内存去完成扩展，或者在新建立线程的时候没有足够的内存去创建对应的虚拟机栈，那么 Java 虚拟机将抛出一个 OutOfMemory 异常。(线程启动过多)。

JVM 新生代中为什么要分为 Eden 和 Survivor?

如果没有 Survivor，Eden 区每进行一次 Minor GC，存活的对象就会被送到老年代。老年代很快被填满，触发 Major GC.老年代的内存空间远大于新生代，进行一次 Full GC 消耗的时间比 Minor GC 长得多,所以需要分为 Eden 和 Survivor。Survivor 的存在意义，就是减少被送到老年代的对象，进而减少 Full GC 的发生，Survivor 的预筛选保证，只有经历 16 次 Minor GC 还能在新生代中存活的对象，才会被送到老年代。设置两个 Survivor 区最大的好处就是解决了碎片化，刚刚新建的对象在 Eden 中，经历一次 Minor GC，Eden 中的存活对象就会被移动到第一块 survivor space S0，Eden 被清空；等 Eden 区再满了，就再触发一次 Minor

GC，Eden 和 S0 中的存活对象又会被复制送入第二块 survivor space S1（这个过程非常重要，因为这种复制算法保证了 S1 中来自 S0 和 Eden 两部分的存活对象占用连续的内存空间，避免了碎片化的发生）

JVM中一次完整的GC流程是怎样的，对象如何晋升到老年代？

当 Eden 区的空间满了， Java 虚拟机会触发一次 Minor GC，以收集新生代的垃圾，存活下来的对象，则会转移到 Survivor 区。大对象（需要大量连续内存空间的 Java 对象，如那种很长的字符串）直接进入老年态；如果对象在 Eden 出生，并经过第一次 Minor GC 后仍然存活，并且被 Survivor 容纳的话，年龄设为 1，每熬过一次 Minor GC，年龄+1，若年龄超过一定限制（15），则被晋升到老年态。即长期存活的对象进入老年态。老年代满了而无法容纳更多的对象，Minor GC 之后通常就会进行 Full GC，Full GC 清理整个内存堆 – 包括年轻代和年老代。Major GC 发生在老年代的 GC，清理老年区，经常会伴随至少一次 Minor GC，比 Minor GC 慢 10 倍以上。

什么是指令重排序？

在实际运行时，代码指令可能并不是严格按照代码语句顺序执行的。大多数现代微处理器都会采用将指令乱序执行(out-of-order execution，简称 OoOE 或 OOE)的方法，在条件允许的情况下，直接运行当前有能力立即执行的后续指令，避开获取下一条指令所需数据时造成的等待。通过乱序执行的技术，处理器可以大大提高执行效率。而这就是指令重排。

什么是内存屏障？

内存屏障，也叫内存栅栏，是一种 CPU 指令，用于控制特定条件下的重排序和内存可见性问题。LoadLoad 屏障：对于这样的语句 Load1; LoadLoad; Load2，在

Load2 及后续读取操作要读取的数据被访问前，保证 Load1 要读取的数据被读取完毕。StoreStore 屏障：对于这样的语句 Store1; StoreStore; Store2，在 Store2 及后续写入操作执行前，保证 Store1 的写入操作对其它处理器可见。LoadStore 屏障：对于这样的语句 Load1; LoadStore; Store2，在 Store2 及后续写入操作被刷出前，保证 Load1 要读取的数据被读取完毕。StoreLoad 屏障：对于这样的语句 Store1; StoreLoad; Load2，在 Load2 及后续所有读取操作执行前，保证 Store1 的写入对所有处理器可见。它的开销是四种屏障中最大的。在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能。

什么是 happen-before 原则?

单线程 happen-before 原则：在同一个线程中，书写在前面的操作 happen-before 后面的操作。锁的 happen-before 原则：同一个锁的 unlock 操作 happen-before 此锁的 lock 操作。volatile 的 happen-before 原则：对一个 volatile 变量的写操作 happen-before 对此变量的任意操作(当然也包括写操作了)。happen-before 的传递性原则：如果 A 操作 happen-before B 操作，B 操作 happen-before C 操作，那么 A 操作 happen-before C 操作。线程启动的 happen-before 原则：同一个线程的 start 方法 happen-before 此线程的其它方法。线程中断的 happen-before 原则 ：对线程 interrupt 方法的调用 happen-before 被中断线程的检测到中断发送的代码。线程终结的 happen-before 原则：线程中的所有操作都 happen-before 线程的终止检测。对象创建的 happen-before 原则：一个对象的初始化完成先于他的 finalize 方法调用。

说说你知道的几种主要的 JVM 参数

1、 堆栈配置相关 -Xmx3550m：最大堆大小为 3550m。-Xms3550m：设置初始堆大小为 3550m。-Xmn2g：设置年轻代大小为 2g。-Xss128k：每个线程的堆栈大小为 128k。-XX:MaxPermSize：设置持久代大小为 16m -XX:NewRatio=4:

设置年轻代（包括 Eden 和两个 Survivor 区）与年老代的比值（除去持久代）。-XX:SurvivorRatio=4：设置年轻代中 Eden 区与 Survivor 区的大小比值。设置为 4，则两个 Survivor 区与一个 Eden 区的比值为 2:4，一个 Survivor 区占整个年轻代的 1/6 -XX:MaxTenuringThreshold=0：设置垃圾最大年龄。如果设置为 0 的话，则年轻代对象不经过 Survivor 区，直接进入年老代。

2、 垃圾收集器相关 -XX:+UseParallelGC：选择垃圾收集器为并行收集器。-XX:ParallelGCThreads=20：配置并行收集器的线程数

-XX:+UseConcMarkSweepGC：设置年老代为并发收集。

-XX:CMSFullGCsBeforeCompaction：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次 GC 以后对内存空间进行压缩、整理。

-XX:+UseCMSCompactAtFullCollection：打开对年老代的压缩。可能会影响性能，但是可以消除碎片

3、 辅助信息相关 -XX:+PrintGC 输出形式: [GC 118250K->113543K(130112K), 0.0094143 secs] [Full GC 121376K->10414K(130112K), 0.0650971 secs]

4、 -XX:+PrintGCDetails 输出形式: [GC [DefNew: 8614K->781K(9088K), 0.0123035 secs] 118250K->113543K(130112K), 0.0124633 secs] [GC [DefNew: 8614K->8614K(9088K), 0.0000665 secs][Tenured: 112761K->10414K(121024K), 0.0433488 secs] 121376K->10414K(130112K), 0.0436268 secs

怎么打出线程栈信息?

输入 jps，获得进程号。top -Hp pid 获取本进程中所有线程的 CPU 耗时性能 jstack pid 命令查看当前 java 进程的堆栈状态 或者 jstack -l > /tmp/output.txt 把堆栈信息打到一个 txt 文件。可以使用 fastthread 堆栈定位（fastthread.io）

为什么需要双亲委派模式?

在这里，先想一下，如果没有双亲委派，那么用户是不是可以自己定义一个 java.lang.Object 的同名类，java.lang.String 的同名类，并把它放到 ClassPath 中,那么类之间的比较结果及类的唯一性将无法保证，因此，为什么需要双亲委派模型？防止内存中出现多份同样的字节码。

怎么打破双亲委派模型?

打破双亲委派机制则不仅要继承 ClassLoader 类，还要重写 loadClass 和 findClass 方法。

说一下堆和栈的区别

1、 物理地址 堆的物理地址分配对对象是不连续的。因此性能慢些。在 GC 的时候也要考虑到不连续的分配，所以有各种算法。比如，标记-消除，复制，标记-压缩，分代（即新生代使用复制算法，老年代使用标记——压缩） 栈使用的是数据结构中的栈，先进后出的原则，物理地址分配是连续的。所以性能快。

2、 内存分别 堆因为是不连续的，所以分配的内存是在运行期确认的，因此大小不固定。一般堆大小远远大于栈。栈是连续的，所以分配的内存大小要在编译期就确认，大小是固定的。

3、 存放的内容 堆存放的是对象的实例和数组。因此该区更关注的是数据的存储 栈存放：局部变量，操作数栈，返回结果。该区更关注的是程序方法的执行。

4、 程序的可见度 堆对于整个应用程序都是共享、可见的。栈只对于线程是可见的。所以也是线程私有。他的生命周期和线程相同。

Java 8 为什么要将永久代(PermGen)替换为元空间(MetaSpace)呢?

整个永久代有一个 JVM 本身设置固定大小上线，无法进行调整，而元空间使用的是直接内存，受本机可用内存的限制，并且永远不会出现 java.lang.OutOfMemoryError。你可以使用 -XX：MaxMetaspaceSize 标志设置最大元空间大小，默认值为 unlimited，这意味着它只受系统内存的限制。-XX：MetaspaceSize 调整标志定义元空间的初始大小如果未指定此标志，则 Metaspace 将根据运行时的应用程序需求动态地重新调整大小。

说一下 Java 对象的创建过程

1、 类加载检查：虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到这个类的符号引用，并且检查这个符号引用代表的类是否已被加载过、解析和初始化过。如果没有，那必须先执行相应的类加载过程。2）分配内存：在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需的内存大小在类加载完成后便可确定，为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。分配方式有 “指针碰撞” 和 “空闲列表” 两种，选择那种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。选择以上 2 种方式中的哪一种，取决于 Java 堆内存是否规整。而 Java 堆内存是否规整，取决于 GC 收集器的算法是"标记-清除"，还是"标记-整理"（也称作"标记-压缩"），值得注意的是，复制算法内存也是规整的。

2、 在创建对象的时候有一个很重要的问题，就是线程安全，因为在实际开发过程中，创建对象是很频繁的事情，作为虚拟机来说，必须要保证线程是安全的，通常来讲，虚拟机采用两种方式来保证线程安全：CAS+失败重试：CAS 是乐观锁的一种实现方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。虚拟机采用 CAS 配上失败重试的方式保证更新操作的原子性。TLAB：为每一个线程预先在 Eden 区分配一块儿内存，JVM 在给线程中的对象分配内存时，首先在 TLAB 分配，当对象大于 TLAB 中的剩余内存或 TLAB 的内存已用尽时，再采用上述的 CAS 进行内存分配

3、 初始化零值：内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头），这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

4、 设置对象头：初始化零值完成之后，虚拟机要对对象进行必要的设置，例如这个对象是那个类的实例、如何才能找到类的元数据信息、对象的哈希吗、对象的 GC 分代年龄等信息。这些信息存放在对象头中。另外，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。

5、 执行 init 方法：在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从 Java 程序的视角来看，对象创建才刚开始， 方法还没有执行，所有的字段都还为零。所以一般来说，执行 new 指令之后会接着执行 方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。

对象的访问定位有哪几种方式?

建立对象就是为了使用对象，我们的 Java 程序通过栈上的 reference 数据来操作堆上的具体对象。对象的访问方式有虚拟机实现而定，目前主流的访问方式有使用句柄和直接指针 2 种：

句柄：如果使用句柄的话，那么 Java 堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息。

直接指针：如果使用直接指针访问，那么 Java 堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，而 reference 中存储的直接就是对象的地址。

这两种对象访问方式各有优势。使用句柄来访问的最大好处是 reference 中存储的是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而 reference 本身不需要修改。使用直接指针访问方式最大的好处就是速度快，它节省了一次指针定位的时间开销。

说一下堆内存中对象的分配的基本策略

eden 区、s0 区、s1 区都属于新生代，tentired 区属于老年代。大部分情况，对象都会首先在 Eden 区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入 s0 或者 s1，并且对象的年龄还会加 1(Eden 区->Survivor 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 -XX:MaxTenuringThreshold 来设置。另外，大对象和长期存活的对象会直接进入老年代。

Minor Gc 和 Full GC 有什么不同呢?

大多数情况下，对象在新生代中 eden 区分配。当 eden 区没有足够空间进行分配时，虚拟机将发起一次 Minor GC。新生代 GC（Minor GC）:指发生新生代的的垃圾收集动作，Minor GC 非常频繁，回收速度一般也比较快。老年代 GC（Major GC/Full GC）:指发生在老年代的 GC，出现了 Major GC 经常会伴随至少一次的 Minor GC（并非绝对），Major GC 的速度一般会比 Minor GC 的慢 10 倍以上。

Java 会存在内存泄漏吗？请简单描述。

内存泄漏是指不再被使用的对象或者变量一直被占据在内存中。理论上来说，Java 是有 GC 垃圾回收机制的，也就是说，不再被使用的对象，会被 GC 自动回收掉，自动从内存中清除。

但是，即使这样，Java 也还是存在着内存泄漏的情况，java 导致内存泄露的原因很明确：长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄露，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收，这就是 java 中内存泄露的发生场景。

如何判断一个类是无用的类?

方法区主要回收的是无用的类，判定一个常量是否是“废弃常量”比较简单，而要判定一个类是否是“无用的类”的条件则相对苛刻许多。类需要同时满足下面 3 个条件才能算是 “无用的类” ：该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。加载该类的 ClassLoader 已经被回收。该类对应的 java.lang.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。

虚拟机可以对满足上述 3 个条件的无用类进行回收，这里说的仅仅是“可以”，而并不是和对象一样不使用了就会必然被回收。

介绍一下类文件结构吧!

魔数: 确定这个文件是否为一个能被虚拟机接收的 Class 文件。Class 文件版本 ： Class 文件的版本号，保证编译正常执行。常量池 ： 常量池主要存放两大

常量：字面量和符号引用。访问标志 ：标志用于识别一些类或者接口层次的访问信息，包括：这个 Class 是类还是接口，是否为 public 或者 abstract 类型，如果是类的话是否声明为 final 等等。当前类索引,父类索引 ：类索引用于确定这个类的全限定名，父类索引用于确定这个类的父类的全限定名，由于 Java 语言的单继承，所以父类索引只有一个，除了 java.lang.Object 之外，所有的 java 类都有父类，因此除了 java.lang.Object 外，所有 Java 类的父类索引都不为 0。接口索引集合 ：接口索引集合用来描述这个类实现了那些接口，这些被实现的接口将按 implents(如果这个类本身是接口的话则是 extends) 后的接口顺序从左到右排列在接口索引集合中。字段表集合 ：描述接口或类中声明的变量。字段包括类级变量以及实例变量，但不包括在方法内部声明的局部变量。方法表集合 ：类中的方法。属性表集合 ：在 Class 文件，字段表，方法表中都可以携带自己的属性表集合。

说一下 JVM 调优的工具?

常用调优工具分为两类,jdk 自带监控工具：jconsole 和 jvisualvm，第三方有：MAT(Memory AnalyzerTool)、GChisto。

jconsole，Java Monitoring and Management Console 是从 java5 开始，在 JDK 中自带的 java 监控和管理控制台，用于对 JVM 中内存， 线程和类等的监控。jvisualvm，jdk 自带全能工具，可以分析内存快照、线程快照；监控内存变化、GC 变化等。MAT，Memory Analyzer Tool，一个基于 Eclipse 的内存分析工具，是一个快速、功能丰富的 Javaheap 分析工具，它可以帮助我们查找内存泄漏和减少内存消耗。GChisto，一款专业分析 gc 日志的工具。

JVM 调优命令有哪些?

jps,JVM Process Status Tool,显示指定系统内所有的 HotSpot 虚拟机进程。jstat，JVM statistics Monitoring 是用于监视虚拟机运行时状态信息的命令，它可以显

示出虚拟机进程中的类装载、内存、垃圾收集、JIT 编译等运行数据。jmap，JVM Memory Map 命令用于生成 heap dump 文件 jhat，JVM Heap Analysis Tool 命令是与 jmap 搭配使用，用来分析 jmap 生成的 dump，jhat 内置了一个微型的 HTTP/HTML 服务器，生成 dump 的分析结果后，可以在浏览器中查看 jstack，用于生成 java 虚拟机当前时刻的线程快照。jinfo，JVM Configuration info 这个命令作用是实时查看和调整虚拟机运行参数。

JRE、JDK、JVM 及 JIT 之间有什么不同?

JRE 代表 Java 运行时（Java run-time），是运行 Java 引用所必须的。JDK 代表 Java 开发工具（Java development kit），是 Java 程序的开发工具，如 Java 编译器，它也包含 JRE。JVM 代表 Java 虚拟机（Java virtual machine），它的责任是运行 Java 应用。JIT 代表即时编译（Just In Time compilation），当代码执行的次数超过一定的阈值时，会将 Java 字节码转换为本地代码，如，主要的热点代码会被准换为本地代码，这样有利大幅度提高 Java 应用的性能。

程序计数器为什么是私有的?

程序计数器主要有下面两个作用：

字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。需要注意的是，如果执行的是 native 方法，那么程序计数器记录的是 undefined 地址，只有执行的是 Java 代码时程序计数器记录的才是下一条指令的地址。

所以，程序计数器私有主要是为了线程切换后能恢复到正确的执行位置。

如何判断一个常量是废弃常量 ？

运行时常量池主要回收的是废弃的常量。假如在常量池中存在字符串 "abc"，如果当前没有任何 String 对象引用该字符串常量的话，就说明常量 "abc" 就是废弃常量，如果这时发生内存回收的话而且有必要的话，"abc" 就会被系统清理出常量池。