JVM 常见面试题

简单介绍下 JVM 运行时数据区

JVM 在执行 Java 程序的过程中会把它管理的内存分为若干个不同的区域，这些组成部分有些是线程私有的，有些则是线程共享的
线程私有的：程序计数器，虚拟机栈，本地方法栈
线程共享的：方法区，堆

简单介绍下 JVM 常见异常

StackOverFlowError： 当线程请求栈的深度超过当前 Java 虚拟机栈的最大深度的时候，就抛出 StackOverFlowError 异常。
OutOfMemoryError：

1. 当线程请求栈时内存用完了，无法再动态扩展了，此时抛出 OutOfMemoryError 异常。
2. 堆内存或者永久代代/元空间不够，无法在分配对象或存放数据，同时堆空间或者永久代/元空间无法再拓展，，此时抛出 OutOfMemoryError 异常。
3. 垃圾回收器占用 JVM98%的资源，同时回收效率不到 2%，JVM 会抛出 OutOfMemoryError，

默认参数下，如果当前 Eden 区大小为 80M，求堆空间总大小

根据比例可以推算出（Eden:survivor1:survivor2 =8:1:1），两个 survivor 区各 10M，新生代 100M。老年代默认是年轻代的两倍，即 200M。那么堆总大小就是 3000M。

程序计数器

记录当前线程正在执行的字节码的地址或行号。
主要作用是为了确保多线程情况下 JVM 程序的正常执行。

讲一讲方法区

方法区是所有线程共享。主要用于存储类的信息、常量池、方法数据、方法代码等。方法区逻辑上属于堆的一部分，但是为了与堆进行区分，通常又叫“非堆”。

JVM 中对象的创建过程

虚拟机遇到一条 new 指令时：根据 new 的参数是否能在常量池中定位到一个类的符号引用（运行时常量池）,如果没有，那必须先执行相应的类加载过程。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。对象所需内存的大小在类加载完成后便可完全确定，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。真的就这么简单吗？答案并不是，具体的实现是比较复杂，下面将描述完整的过程。

1 检查加载

先执行相应的类加载过程。如果没有，则进行类加载。

2 分配内存

假如 Java 堆中内存是绝对规整的，所有用过的内存都放在一边，空闲的内存放在另一边，中间放着一个指针作为分界点的指示器，那所分配内存就仅仅是把那个指针向空闲空间那边挪动一段与对象大小相等的距离，这种分配方式称为“**指针碰撞**”。

如果 Java 堆中的内存并不是规整的，已使用的内存和空闲的内存相互交错，那就没有办法简单地进行指针碰撞了，虚拟机就必须维护一个列表，记录上哪些内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录，这种分配方式称为“**空闲列表**”。

选择哪种分配方式由 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，比如使用 CMS 这种基于 Mark-Sweep 算法的收集器时，java 堆中的内存并不是规整的，通常采用空闲列表。

3 内存空间初始化

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(如 int 值为 0，boolean 值为 false 等等)。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

接下来，虚拟机要对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的 GC 分代年龄等信息。这些信息存放在对象的对象头之中。

5 对象初始化

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，对象创建才刚刚开始，所有的字段都还为零值。所以，一般来说，执行 new 指令之后会接着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。

对象的访问定位的两种方式

句柄和直接指针两种方式，
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。
HotSpot 中使用直接指针，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。

JVM 中如何判断对象是否死亡（可回收）

可达性分析算法
这个算法的基本思想就是通过一系列的称为 “GC Roots” 的对象作为起点，从这些节点开始向下搜索，节点所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链相连的话，则证明此对象是不可用的。

简单的介绍一下 Java 中的各种引用

1. 强引用
以前我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于必不可少的生活用品，垃圾回收器绝不会回收它。当内存空 间不足，Java 虚拟机宁愿抛出 OutOfMemoryError 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。
2. 软引用（SoftReference）
如果一个对象只具有软引用，那就类似于可有可无的生活用品。如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。
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（WeakReference）
如果一个对象只具有弱引用，那就类似于可有可无的生活用品。弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它 所管辖的内存区域

的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程， 因此不一定会很快发现那些只具有弱引用的对象。
弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。
4. 虚引用（PhantomReference）
"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。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。
虚引用主要用来跟踪对象被垃圾回收的活动。
在程序设计中除了强引用，使用软引用的情况较多，这是因为软引用可以加速 JVM 对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出（OutOfMemory）等问题的产生

垃圾收集有哪些算法，各自的特点？

1. 标记-清除算法
标记-清除算法分为“标记”和“清除”阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。它是最基础的收集算法，效率也很高，但是会带来两个明显的问题：
1）效率问题
2）空间问题（标记清除后会产生大量不连续的碎片）
2. 复制算法
为了解决效率问题，“复制”收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。
3. 标记-整理算法
根据老年代的特点特出的一种标记算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象回收，而是让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存。

HotSpot 为什么要分为新生代和老年代？

将 java 堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。
比如在新生代中，每次收集都会有大量对象死去，所以可以选择复制算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集

哪些情况需要对类进行初始化？

虚拟机规范严格规定了有且只有五种情况必须立即对类进行“初始化”：

1。使用 new 关键字实例化对象的时候、读取或设置一个类的静态字段的时候，已经调用一个类的静态方法的时候。
2.使用 java.lang.reflect 包的方法对类进行反射调用的时候，如果类没有初始化，则需要先触发其初始化。
3.当初始化一个类的时候，如果发现其父类没有被初始化就会先初始化它的父类。
4.当虚拟机启动的时候，用户需要指定一个要执行的主类（就是包含 main()方法的那个类），虚拟机会先初始化这个类；
5.使用动态语言支持的时候，如果一个 java.lang.invoke.MethodHandle 实例最后的解析结果 REF_getstatic,REF_putstatic,REF_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行初始化，则需要先出触发其初始化。

双亲委派模型

双亲委派模型（Pattern Delegation Model）,要求除了顶层的启动类加载器外，其余的类加载器都应该有自己的父类加载器。这里父子关系通常是子类通过组合关系而不是继承关系来复用父加载器的代码。
双亲委派模型的工作过程： 如果一个类加载器收到了类加载的请求，先把这个请求委派给父类加载器去完成（所以所有的加载请求最终都应该传送到顶层的启动类加载器中），只有当父加载器反馈自己无法完成加载请求时，子加载器才会尝试自己去加载。