# 2020 面试题总结——JVM 篇

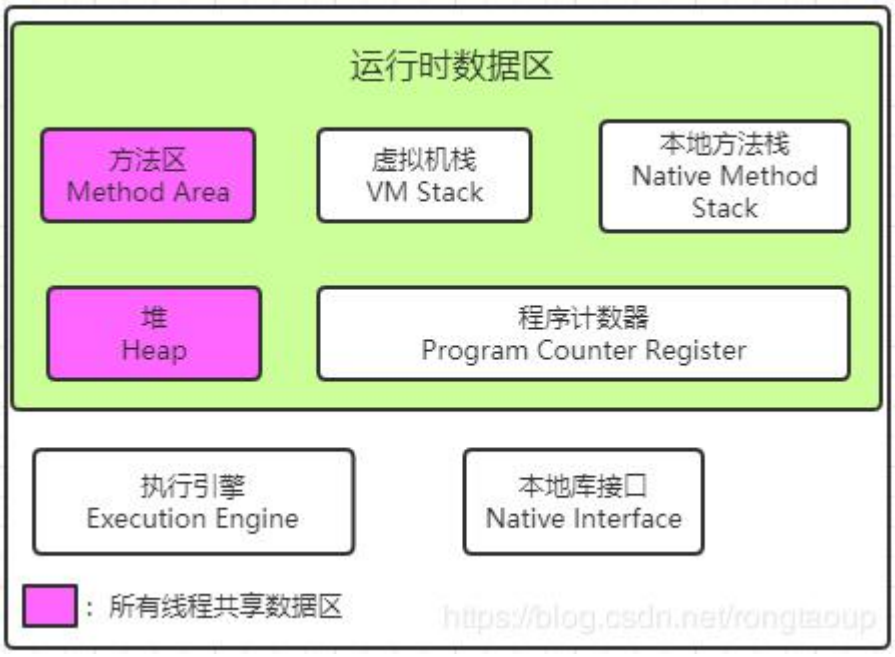
**1、什么情况下会发生栈内存溢出。**

在 HotSpot 虚拟机中是不区分虚拟机栈和本地方法栈，栈是线程私有的，它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。每一个方法从调用直至执行完成的过程，就对应着一个栈桢在虚拟机栈中入栈到出栈的过程。本地方法栈与虚拟机栈相似，区别不过是虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。

栈内存溢出是指线程请求的栈深度大于虚拟机所允许的最大深度，则将抛出 StackOverflowError 异常（StackOverflowError 不属于 OOM 异常）。最有可能的原因就是方法递归产生的这种结果。

另一个可能是引用了大的变量，在拓展栈时无法申请到足够的内存空间，则抛出 OutOfMemoryError 异常（这个属于内存溢出）。

**2、JVM 的内存结构，Eden 和 Survivor 比例。**




Java 虚拟机在执行 Java 程序的过程中把它所管理的内存划分为若干个不同的数据区域，这些区域都有各自的用途。

- 程序计数器。当前线程执行的字节码的行号指示器，是线程私有的。也是唯一一个不会发生内存溢出的区域。
- Java 虚拟机栈。也是线程私有的，描述的是 Java 方法执行的内存模型，线程请求的栈深度大于虚拟机所允许的最大深度，则将抛出 StackOverflowError 异常。

- 本地方法栈。与虚拟机栈相似，区别不过是虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。
- Java 堆。是 Java 虚拟机中管理的内存中最大的一块，所有线程共享区域，唯一目的就是存放对象实例。所有的对象实例以及**数组**都要在堆上分配内存。Java 堆也是垃圾回收器管理的主要区域，也被称为 gc 堆，收集器基本都采用分代收集算法，Java 堆中还可以细分为：新生代和老年代。
- 方法区。所有线程共享区域，用于存储已经被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。很多人也愿意称之为“永久代”。
- 运行时常量池。是方法区的一部分，用于存放编译器生成的各种字面量和符号引用。
- 直接内存。并不是虚拟机运行时数据区的一部分。例如 NIO，它可以使用 Native 函数直接分配堆外内存，然后通过一个存储在 Java 堆中的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样避免了在 Java 堆和 Native 堆中来回复制数据，提高了性能。

JVM 中要对堆进行分代，分代的理由是优化 GC 性能，很多对象都是朝生夕死的，如果分代的话，我们把新创建的对象放到某一地方，当 GC 的时候先把这块存“朝生夕死”对象的区域进行回收，这样就会腾出很大的空间出来。

HotSpot JVM 把新生代分为了三部分：1 个 Eden 区和 2 个 Survivor 区（分别叫 from 和 to）。默认比例为 8：1。一般情况下，新创建的对象都会被分配到 Eden 区(一些大对象特殊处理),这些对象经过第一次 Minor GC 后，如果仍然存活，将会被移到 Survivor 区。对象在 Survivor 区中每熬过一次 Minor GC，年龄就会增加 1 岁，当它的年龄增加到一定程度时，就会被移动到年老代中。

因为年轻代中的对象基本都是朝生夕死的(80%以上)，所以在年轻代的垃圾回收算法使用的是复制算法，复制算法的基本思想就是将内存分为两块，每次只用其中一块，当这一块内存用完，就将还活着的对象复制到另外一块上面。复制算法不会产生内存碎片。

**3、JVM 内存为什么要分成新生代、老年代和持久代。新生代中为什么要分 Eden 和 Survivor。**

堆内存是虚拟机管理的内存中最大的一块，也是垃圾回收最频繁的一块区域，我们程序所有的对象实例都存放在堆内存中。给堆内存分代是为了提高对象内存分配和垃圾回收的效率。试想一下，如果堆内存没有区域划分，所有的新创建的对象和生命周期很长的对象放在一起，随着程序的执行，堆内存需要频繁进行垃圾收集，而每次回收都要遍历所有的对象，遍历这些对象所花费的时间代价是巨大的，会严重影响我们的 GC 效率，这简直太可怕了。

有了内存分代，情况就不同了，新创建的对象会在新生代中分配内存，经过多次回收仍然存活下来的对象存放在老年代中，静态属性、类信息等存放在永久

代中，新生代中的对象存活时间短，只需要在新生代区域中频繁进行 GC，老年代中对象生命周期长，内存回收的频率相对较低，不需要频繁进行回收，永久代中回收效果太差，一般不进行垃圾回收，还可以根据不同年代的特点采用合适的垃圾收集算法。分代收集大大提升了收集效率，这些都是内存分代带来的好处。

HotSpot 将新生代划分为三块，一块较大的 Eden 空间和两块较小的 Survivor 空间，默认比例为 8：1：1。划分的目的是因为 HotSpot 采用复制算法来回收新生代，设置这个比例是为了充分利用内存空间，减少浪费。新生成的对象在 Eden 区分配（大对象除外，大对象直接进入老年代），当 Eden 区没有足够的空间进行分配时，虚拟机将发起一次 Minor GC。

**4、JVM 中一次完整的 GC 流程是什么样子的，对象如何晋升到老年代，说说你知道的几种主要的 JVM 参数。**

GC 开始时，对象只会存在于 Eden 区和 From Survivor 区，To Survivor 区是空的（作为保留区域）。GC 进行时，Eden 区中所有存活的对象都会被复制到 To Survivor 区，而在 From Survivor 区中，仍存活的对象会根据它们的年龄值决定去向，年龄值达到年龄阀值（默认为 15，新生代中的对象每熬过一轮垃圾回收，年龄值就加 1，GC 分代年龄存储在对象的 header 中）的对象会被移到老年代中，没有达到阀值的对象会被复制到 To Survivor 区。接着清空 Eden 区和 From Survivor 区，新生代中存活的对象都在 To Survivor 区。接着， From Survivor 区和 To Survivor 区会交换它们的角色，也就是新的 To Survivor 区就是上次 GC 清空的 From Survivor 区，新的 From Survivor 区就是上次 GC 的 To Survivor 区，总之，不管怎样都会保证 To Survivor 区在一轮 GC 后是空的。GC 时当 To Survivor 区没有足够的空间存放上一次新生代收集下来的存活对象时，需要依赖老年代进行分配担保，将这些对象存放在老年代中。

对象晋升老年代有三种可能：

- 当对象达到成年，经历过 15 次 GC（默认是 15，可配置），对象就晋升到老年代了。
- 大的对象会直接在老年代创建。
- 新生代的 Survivor 空间内存不足时，对象可能直接晋升到老年代。

jvm 参数：

- -Xms：初始堆大小
- -Xmx：堆最大内存
- -Xss：栈内存
- -XX:PermSize 初始永久代内存
- -XX:MaxPermSize 最大永久带内存

**5、你知道哪几种垃圾收集器，各自的优缺点，重点讲下 cms 和 G1，包括原理，流程，优缺点。**

常见的垃圾收集器主要有以下四种：

- 串行收集器（Serial、ParNew 收集器）：简单高效，但它在进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束，中间停顿时间长。
- 并行收集器（Parallel Scavenge 收集器）：吞吐量优先，主要关注点在于精确控制吞吐量，即减少 GC 停顿时间，但收集次数变多。
- CMS：以获取最短回收停顿时间为目标的收集器，并发**标记-清除**，主要步骤有，初始标记，并发标记，重新标记和并发清除。其中，整个过程耗时最长的并发标记和并发清除过程收集器线程都可以和用户线程一起工作，CMS 收集器的内存回收过程始于用户线程一起并发执行的。重新标记是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。但缺点有，CMS 收集器对 CPU 资源非常敏感，并且无法处理浮动垃圾。
- G1：可预测停顿的收集器，并发**标记-整理**，主要步骤分为，初始标记，并发标记，最终标记和筛选回收。G1 把内存“化整为零”，并且可以分代收集。注意：CMS 是清除，所以会存在很多的内存碎片。G1 是整理，所以碎片空间较小。

**6、垃圾回收算法的实现原理。**

垃圾收集算法主要分为以下三种：

- 标记-清除算法：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。
- 复制算法：将可用内存按容量分为两块（Eden 和 Survivor 空间），每次只使用一块，当这一块内存用完后，就将还活着的对象复制到另外一块上面，然后再把已使用过内存空间一次清理掉。
- 标记-整理算法：标记过程与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉边界外的内存。

需要注意的是，“标记-清除”算法存在**两个不足**：

- 一个是**效率问题**，标记和清除两个过程的效率都不高；
- 另一个是**空间问题**，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一个垃圾收集动作。作为对比，复制算法每次都是对整个半区进行内存回收，内存分配时也就不用考虑

内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。

**7、当出现了内存溢出，怎么排错。**

- 首先控制台查看错误日志。
- 然后使用 jdk 自带的 VisualVM 来查看系统的堆栈日志（也可以用 jmap 查看堆转储快照）。
- 定位出内存溢出的空间：堆，栈还是永久代（jdk8 后没有永久代的溢出了）。
- 如果是堆内存溢出，看是否创建了超大的对象。
- 如果是栈内存溢出，看是否创建了超大的对象，或者产生了死循环，或者递归调用。

**8、JVM 内存模型的相关知识了解多少，比如重排序，内存屏障，happen-before，主内存，工作内存等。**

重排序是指编译器和处理器为了优化程序性能而对指令序列进行重新排序的一种手段。

内存屏障是一种 CPU 指令，用于控制特定条件下的重排序和内存可见性问题。Java 编译器也会根据内存屏障的规则禁止重排序。

happen-before 用来阐述操作之间的内存可见性。在 JMM 中，如果一个操作执行的结果需要对另一个操作可见，那么这两个操作之间必须要存在 happens-before 关系。具体原则如下：

- 程序次序法则：线程中的每个动作 A 都 happens-before 于该线程中的每一个动作 B，其中，在程序中，所有的动作 B 都能出现在 A 之后。
- 监视器锁法则：对一个监视器锁的解锁 happens-before 于每一个后续对同一监视器锁的加锁。
- volatile 变量法则：对 volatile 域的写入操作 happens-before 于每一个后续对同一个域的读写操作。
- 线程启动法则：在一个线程里，对 Thread.start 的调用会 happens-before 于每个启动线程的动作。
- 线程终结法则：线程中的任何动作都 happens-before 于其他线程检测到这个线程已经终结、或者从 Thread.join 调用中成功返回，或 Thread.isAlive 返回 false。
- 中断法则：一个线程调用另一个线程的 interrupt happens-before 于被中断的线程发现中断。
- 终结法则：一个对象的构造函数的结束 happens-before 于这个对象 finalizer 的开始。

- 传递性：如果 A happens-before 于 B，且 B happens-before 于 C，则 A happens-before 于 C。

主内存是指所有线程共享的内存空间。

工作内存是指每个线程特有的内存空间。工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作（读取、赋值等）都必须在工作内存中进行，而不能直接读写在主内存中的变量。

参考链接：JVM 内存模型、指令重排、内存屏障概念解析

**9、讲讲 JAVA 的反射机制。**

Java 反射说的是在运行状态中，对于任何一个类，我们都能够知道这个类有哪些方法和属性。对于任何一个对象，我们都能够对它的方法和属性进行调用。我们把这种动态获取对象信息和调用对象方法的功能称之为反射机制。

**10、你们线上应用的 JVM 参数有哪些。**

- -Xms512m //初始堆大小
- -Xmx1024m //最大堆大小
- -XX:PermSize=640m //设置持久代初始值
- -XX:MaxPermSize=1280m //设置持久代最大值
- -XX:NewSize=64m //设置年轻代初始值
- -XX:MaxNewSize=256m //设置年轻代最大值
- -verbose:gc //表示输出虚拟机中 GC 的详细情况
- -XX:+PrintGCDetails //日志输出形式
- -XX:+PrintGCTimeStamps //日志输出形式

在默认情况下，JVM 初始分配的堆内存大小是物理内存的 1/64，最大分配的堆内存大小是物理内存的 1/4。

默认空余堆内存小于 40%时，JVM 就会增大堆直到-Xmx 的最大限制；空余堆内存大于 70%时，JVM 会减少堆直到-Xms 的最小限制。

因此服务器一般设置-Xms、-Xmx 相等，来避免每次 GC 后调整堆的大小。

**11、g1 和 cms 区别，吞吐量优先和响应优先的垃圾收集器选择。**

CMS 是基于“标记-清除”实现的，主要步骤是初始标记，并发标记，重新标记和并发清除。

G1 是基于“标记-整理”实现的，主要步骤是初始标记，并发标记，最终标记和筛选回收。

CMS 的缺点是对 CPU 的要求比较高。

G1 的缺点是将内存划分了多块，所以对内存段的大小有很大的要求。

CMS 是清除，所有会有很多的内存碎片。

G1 是整理，所有碎片空间较小。

G1 和 CMS 都是响应优先，他们的目的都是尽量控制 stop the world 的时间。

G1 和 CMS 的 Full GC 都是单线程 mark sweep compact 算法，直到 JDK10 才优化成并行的。

CMS 目前只用于老年代，而 G1 是将整个 Java 堆划分为多个大小不等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔的了，他们都是一部分 Region（不需要连续）的集合。

吞吐量优先的话可以选择并行垃圾收集器，Parallel Scavenge 收集器。吞吐量是指 CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值。

**12、怎么打印线程栈信息。**

```
StackTraceElement[] elements = (new Throwable()).getStackTrace();
StringBuffer buf = new StringBuffer();
for(int i=0; i<elements.length; i++) {
    buf.append("\n"
            + elements[i].getClassName()//打印线程当前执行的详细类名
            + "."
            + elements[i].getMethodName()//打印线程当前方法名
            + "("
            + elements[i].getFileName()//打印线程当前执行类的文件名
            + ":"
            + elements[i].getLineNumber()//打印线程当前执行的行数
            + ")");
}
```

**13、请解释如下 jvm 参数的含义。**

- -server：服务器模式
- -Xms512m：初始堆空间
- -Xmx512m：最大堆空间
- -Xss1024K：栈空间
- -XX:PermSize=256m：初始永久代空间
- -XX:MaxPermSize=512m ：最大永久代空间
- -XX:MaxTenuringThreshold=20：在新生代对象存活次数（经历 Minor GC 的次数）后依旧存活，就会晋升到老年代。

- -XX:CMSInitiatingOccupancyFraction=80 ：设定 CMS 在对老年代内存占用率达到 80%的时候开始 GC(因为 CMS 会有浮动垃圾,所以一般都较早启动 GC)。
- -XX:+UseCMSInitiatingOccupancyOnly：只是用设定的回收阈值(上面指定的 80%),如果不指定,JVM 仅在第一次使用设定值,后续则自动调整.

**14、Java 中对象创建的几种方式。**

创建对象的方式有四种：

- 用 new 关键字创建。

```
User user = new User();
```

- 调用对象的 clone 方法。
- 利用反射，调用 Class 类的或者是 Constructor 类的 newInstance（）方法。

```
User user = User.class.newInstance();
```

或者是

```
Constructor<User> constructor =User.class.getConstructor();
User user= constructor.newInstance();
```

- 用反序列化，调用 ObjectInputStream 类的 readObject（）方法。

**15、Java 中的对象一定在堆上分配内存吗？**

前面我们说过，Java 堆中主要保存了对象实例，但是，随着 JIT 编译期的发展与逃逸分析技术逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化，所有的对象都分配到堆上也渐渐变得不那么“绝对”了。

其实，在编译期间，JIT 会对代码做很多优化。其中有一部分优化的目的就是减少内存堆分配压力，其中一种重要的技术叫做逃逸分析。

如果 JIT 经过逃逸分析，发现有些对象没有逃逸出方法，那么有可能堆内存分配会被优化成栈内存分配。

参考文章：深入理解 Java 中的逃逸分析 和 对象并不一定都是在堆上分配内存的

**16、运行时数据中哪些区域是线程共享的，哪些是独享的。**

在 JVM 运行时内存区域中，程序计数器、虚拟机栈和本地方法栈是线程独享的。而 Java 堆、方法区是线程共享的。但是值得注意的是，Java 堆其实还为每一个

线程单独分配了一块 TLAB 空间（本地线程分配缓冲），这部分空间在分配时是线程独享的，在使用时是线程共享的。（TLAB 介绍）

创建对象时，内存分配过程如何保证线程安全性？有两种解决方案：

- 对分配内存空间的动作做同步处理，采用 CAS 机制，配合失败重试的方式保证更新操作的线程安全性。
- 每个线程在 Java 堆中预先分配一小块内存，然后再给对象分配内存的时候，直接在自己这块"私有"内存中分配，当这部分区域用完之后，再分配新的"私有"内存。**这个本地线程分配缓冲就叫做 TLAB。**

**17、Java 中数组是存储在堆上还是栈上。**

在 Java 中，数组同样是一个对象，所以对象在内存中如何存放同样适用于数组；

所以，数组的实例是保存在堆中，而数组的引用是保存在栈上的。

**18、Java 对象创建的过程是怎么样的。**

对于一个普通的 Java 对象的创建，大致过程如下：

1. 虚拟机遇到 new 指令，到常量池定位到这个类的符号引用。
2. 检查符号引用代表的类是否被加载、解析、初始化过 ，如果没有的话，则执行相应的类加载过程。
3. 虚拟机为对象分配内存。 根据 Java 内存是否规整，分别通过“指针碰撞”或“空闲列表”来分配。
4. 虚拟机将分配到的内存空间都初始化为零值。
5. 虚拟机对对象进行必要的设置。
6. 执行方法，成员变量进行初始化。

**19、怎么获取堆和栈的 dump 文件。**

Java Dump，Java 虚拟机的运行时快照。将 Java 虚拟机运行时的状态和信息保存到文件。

可以使用在服务器上使用 jmap 命令来获取堆 dump，使用 jstack 命令来获取线程的调用栈 dump。

参考文章：Java 命令学习系列（二）——Jstack 和 Java 命令学习系列（三）——Jmap

**20、Minor GC 和 Full GC 的触发条件。**

Minor GC 触发条件：当 Eden 区满时，触发 Minor GC。

Full GC 触发条件：

- 调用 System.gc 时，系统建议执行 Full GC，但是不必然执行。
- 老年代空间不足。
- 方法区空间不足。
- concurrent mode failure，当执行 CMS GC 过程时（“标记-清除”，存在内存碎片），同时有对象要放入老年代，而此时老年代空间不足造成的（有时候“空间不足”是 CMS GC 时当前的浮动垃圾过多导致暂时性的空间不足触发 Full GC）。

**21、在 Java 语言中，可以作为 GC Roots 的对象有什么。**

可作为 GC Roots 的对象包括以下几种：

- 虚拟机栈（栈桢中的本地变量表）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。
- 本地方法栈中 JNI（即一般说的 Native 方法）引用的对象。

获取 GC Roots 最主要的部分在解决如果快速找到 JVM 栈的栈桢的局部变量表中的局部变量所引用的对象。大致思路是 JVM 采用了 **OopMap** 这个数据结构记录了 GC Roots，GC 的标记开始的时候，直接用 OopMap 就可以获得 GC Roots。OopMap 记录了特定时刻栈上（内存）和寄存器（CPU）的哪些位置是引用，通过这些引用就可以找到堆中的对象，这些对象就是 GC Roots，而不需要一个一个的去判断某个内存位置的值是不是引用。

**22、类加载过程。**

Java 虚拟机中类加载的全过程包括：加载、验证、准备、解析和初始化这 5 个阶段。

**加载**

在加载阶段，虚拟机主要完成以下 3 个事情。

- 通过一个类的全限定名来获取定义此类的二进制字节流。（这一步骤就是通过类加载器来实现的）
- 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
- 在内存中生成一个代表这个类的 java.lang.Class 对象，作为方法区这个类的各类数据的访问入口。

**验证**

验证时连接阶段的第一步，这一阶段的目的是为了**确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求**，并且不会危害虚拟机自身的安全。

- 文件格式验证：验证字节流是否符合 Class 文件格式的规范，包括文件头部的魔数因子、class 文件主次版本号、class 文件的 MD5 指纹等。
- 元数据验证：对字节码描述的信息进行语义分析，以保证其描述的信息符合 Java 语言规范。简单来说就是验证 Java 语法的正确性。
- 字节码验证：主要验证程序的控制流程，如循环、分支等。

**准备**

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存都将在**方法区**中分配。需要注意的是，这时候进行内存分配的仅包括**类变量（被 static 修饰的变量）**，而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中。

**解析**

解析阶段是虚拟机在常量池内寻找类、接口、字段和方法的符号引用，并且将这些**符号引用替换为直接引用**的过程。符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。

**初始化**

初始化阶段是类的加载过程的最后一个阶段，该阶段主要做一件事情就是执行()，该方法**会为所有的静态变量赋予正确的值**。

参考文章：万万没想到，JVM 内存结构的面试题可以问的这么难？

**23、Java 内存泄漏的场景。**

内存泄漏是指，一个不再被程序使用的对象或变量还在内存中占有存储空间。虽然 Java 拥有 GC，但还是会出现内存泄漏。举个例子。

```
//首先，要明白，GC 它回收的是不可到达的对象，但是，在 static 的集合类中，引用可以到达，但是却有可能对象已经不用了
//首先定义一个静态变量
public static ArrayList<Object> list = new ArrayList<Object>();
public void stackOverExam(Object object){
    //当非静态变量被 static 变量持有引用的时候，容易发生内存泄露，因为 object 是一直被 list 引用着的
    list.add(object);
    object = null;//这里设置为 null 并没有达到释放 object 引用对应对象的效果，毕竟 list 还是持有引用
}
```

通过上面的代码可以看到，由于 static 指向的对象是不能被垃圾回收器回收的，所以，间接的 object 也是无法被回收的，当业务对象很大而且很多的时候，便有了内存泄漏的风险。所以，可以总结如下规则：

当全局的静态变量持有局部变量（或者说，大范围的变量持有小范围变量而且小范围变量消耗内存表达、数目变多时），程序便有内存泄漏的风险。一般来说，类似的例子还有，单例模式中的对象，模块之间的调用（后面这个例子提到）等。

先举一个单例对象的例子。由于单例的静态特征使得其生命周期和应用的生命周期一样长，如果一个对象已经不再被使用，而单例对象还会持有该对象的引用，就会使得该对象不能被正常回收，从而导致了内存泄漏。

```
public class Singleton {
    private static Singleton singleton;
    private List<Integer> list;
    private Singleton(List<Integer> list) {
        this.list = list;
    }
    public Singleton getInstance(List<Integer> list) {
        if (singleton == null) {
            singleton = new Singleton(list);
        }
        return singleton;
    }
}
```

另外再举一个模块调用的例子，现在有两个类 A 和 B，其中 B 的默认构造函数上是需要一个 A 的实例作为参数的，这就让 A 和 B 产生了依赖。

```
A a=new A();
B b=new B(a);
a=null;
```

a 是对象 A 的引用，b 是对象 B 的引用，对象 B 同时还依赖对象 A，那么这个时候就可以认为对象 B 是可以到达对象 A 的。当 A 对象的引用 a 置为 null 后，a 不再指向对象 A 的引用了，按理说对象 A 可以 GC 了。但是因为 B 依赖着 A，所以这个时候，A 对象是不可能被回收了，造成了内存泄漏。这个时候可以用弱引用 WeakReference 来代替对象 B，就可以解决了这个问题。如下所示。

```
A a=new A();
WeakReference wr=new WeakReference(a);
a=null;
```

### 24、jdk1.8 的虚拟机中内存模型变化。

在 jdk1.8 中变化最大的是取消了永久区 Perm，而是用元数据空间 Metaspace 来进行替换。需要注意的是，元空间占用的内存不是虚拟机内部的，而是本地内存空间，当然也不是堆内存。这个变化的理由如下：

- 在 jdk1.8 之前的 HotSpot 实现中，类的元数据如方法数据、方法信息（字节码、栈和变量的大小）、运行时常量池等保存在永久代。32 位默认永久代为 64M，64 位默认 85M，可以通过参数-XX：MaxPermSize 进行设置，一旦类的元数据超过了永久代的大小，就会抛出 OOM 异常了。
- 对永久代的调优过程很困难，因为永久代的大小很难确定，其中涉及到很多因素，如类的总数、常量池大小和方法数量等，而且永久代的护具可能会随着每一次 Full GC 而发生移动。
- 在 jdk1.8 中，类的元数据保存在本地内存中，元空间的最大可分配空间就是系统可用内存空间。

**25、频繁 GC 的原因。**

- 人为因素，在代码中调用了 System.gc()方法。
- 内存原因，设置的堆大小比较小，可以提高堆的空间，比如说提高最小堆空间-Xms 和最大堆空间-Xmx 的大小，当然，最好是针对内存的 DUMP 文件进行分析。
- 框架问题，有些框架内部会调用 gc 方法。
- 其他原因，构建的对象实例化十分频繁并且释放对象较为频繁时，也会引起频繁 gc。

如果线上系统突然产生的运行缓慢问题，如果该问题导致线上系统不可用，那么首先需要做的就是，导出 jstack 和内存信息，然后重启系统，尽快保证系统的可用性。这种情况可能的原因主要有两种：

- 代码中某个位置读取数据量较大，导致系统内存耗尽，从而导致 Full GC 次数过多，系统缓慢；
- 代码中有比较耗 CPU 的操作，导致 CPU 过高，系统运行缓慢。

**26、高并发时，JVM 调优。**

优化虚拟机堆的空间大小，根据实际物理内存的大小进行比例分配，并根据程序调整好新生代和老年代的比例。并且，堆不进行自动扩展。然后使用 ParNew（并发）+CMS 进行垃圾回收，在多线程高并发的情况下，表现很好。

调优的目标是：

- 将转移到老年代的对象数量降低到最小 。
- 减少 Full GC 的执行时间。

**27、System.gc()和 Runtime.gc()的区别。**

- java.lang.System.gc()只是 java.lang.Runtime.getRuntime().gc()的简写，两者的行为没有任何不同。

- System.gc()和 runtime.gc()用于建议 jvm 进行垃圾回收，但是否立即回收还是延迟回收由 Java 虚拟机决定。

另外，当我们调用 System.gc()的时候，其实并不会马上进行垃圾回收，甚至不一定会执行垃圾回收。

以上主要参考来源为：《深入理解 Java 虚拟机：JVM 高级特征与最佳实践》