理论知识学习：垃圾收集器与内存分配策略，《深入理解 JAVA 虚拟机》中的第三章(在 TB 的技术书籍文件夹中可以找到此电子书)。

1. Java 程序员为什么需要了解 GC 和 JVM 的自动内存分配回收机制？

当需要排查各种内存溢出、内存泄漏问题时，当垃圾收集成为系统达到更高并发量的瓶颈时，我们就需要对这些“自动化”的技术实施必要的监控和调节。

2. 为什么 JVM 内存回收的重点关注区域是堆和方法区而不是 JVM 栈或本地方法栈？

每一个栈帧中分配多少内存基本上是在类结构确定下来时就已知的（尽管在运行期会由 JIT 编译器进行一些优化，但在本章基于概念模型的讨论中，大体上可以认为是编译期可知的），因此这几个区域的内存分配和回收都具备确定性，在这几个区域内就不需要过多考虑回收的问题，因为方法结束或者线程结束时，内存自然就跟随着回收了。而 Java 堆和方法区则不一样，一个接口中的多个实现类需要的内存可能不一样，一个方法中的多个分支需要的内存也可能不一样，我们只有在程序处于运行期间时才能知道会创建哪些对象，这部分内存的分配和回收都是动态的。

3. 哪几种算法可以得知哪些对象是可回收的？各种算法的优缺点是什么？

1）引用计数算法：引用计数算法（Reference Counting）的实现简单，判定效率也很高。如果对象之间相互循环引用 GC 回收器就无法回收。
2）可达性分析算法：这个算法的基本思路就是通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到 GC Roots 没有任何引用链相连（用图论的话来说，就是从 GC Roots 到这个对象不可达）时，则证明此对象是不可用的。缺点是速度相对于引用计数算法比较慢，可以解决相互引用的问题。

4. 可作为 GC ROOT 的对象分为哪几种？为什么是这几种？

虚拟机栈（栈帧中的本地变量表）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。

5. Java 中定义了几种引用？请举例说明他们的应用场景。

1）强引用（StrongReference）：强引用就是指在程序代码之中普遍存在的，类似“Object obj=new Object（）”这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象。
2）软引用（Soft Reference）：软引用是用来描述一些还有用但并非必需的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中进

行第二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。在 JDK 1.2 之后，提供了 SoftReference 类来实现软引用。

3）弱引用（Weak Reference）：弱引用也是用来描述非必需对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。在 JDK 1.2 之后，提供了 WeakReference 类来实现弱引用。

4）虚引用（PhantomReference）：虚引用也称为幽灵引用或者幻影引用，它是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。在 JDK 1.2 之后，提供了 PhantomReference 类来实现虚引用。

6. 如何让一个 GC ROOT 无法触达的对象也就是即将被回收的对象复活？能复活几次？

如果对象要在 finalize（）中成功拯救自己——只要重新与引用链上的任何一个对象建立关联即可，譬如把自己（this 关键字）赋值给某个类变量或者对象的成员变量，那在第二次标记时它将被移除出“即将回收”的集合；如果对象这时候还没有逃脱，那基本上它就真的被回收了。一次。

7. 为什么不推荐使用 finalize()方法？

它的运行代价高昂，不确定性大，无法保证各个对象的调用顺序。finalize（）能做的所有工作，使用 try-finally 或者其他方式都可以做得更好、更及时，所以笔者建议大家完全可以忘掉 Java 语言中有这个方法的存在。

8. 什么叫做类卸载？

1）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。
2）加载该类的 ClassLoader 已经被回收。
3）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。

9. 说说标记-清除算法的优缺点。

它的主要不足有两个：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次垃圾收集动作。

10. 说说复制算法的优缺点和特点。

它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。

这样使得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。只是这种算法的代价是将内存缩小为了原来的一半，未免太高了一点。复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低。

11. 说说标记-整理算法的优缺点。

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，然后直接清理掉端边界以外的内存。

12. 以上几种算法分别在什么情况下使用比较恰当？

复制算法来回收新生代。标记-整理算法回收老年代。
在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记—清理”或者“标记—整理”算法来进行回收。

13. 说说寄存器和存储器的区别。

寄存器读取快，容量小。
存储器读取慢，容量大。

14. HotSpot 虚拟机是如何得知所有的 GCRoot 对象的内存地址的？

在 HotSpot 的实现中，是使用一组称为 OopMap 的数据结构来达到这个目的的，在类加载完成的时候，HotSpot 就把对象内什么偏移量上是什么类型的数据计算出来，在 JIT 编译过程中，也会在特定的位置记录下栈和寄存器中哪些位置是引用。这样，GC 在扫描时就可以直接得知这些信息了。

15.除了内存不够时会触发 GC，JVM 中什么条件也会触发 GC？

没有。

16. 说说 JVM GC 回收过程中安全点的概念，什么代码能达到 safepoint 的条件？

程序执行时并非在所有地方都能停顿下来开始 GC，只有在到达安全点时才能暂停。Safepoint 的选定既不能太少以致于让 GC 等待时间太长，也不能过于频繁以致于过分增大运行时的负荷。所以，安全点的选定基本上是以程序“是否具有让程序长时间执行的特征”为标准进行选定的。
对于 Sefepoint，另一个需要考虑的问题是如何在 GC 发生时让所有线程（这里不包括执行 JNI 调用的线程）都“跑”到最近的安全点上再停顿下来。
条件：“长时间执行”的最明显特征就是指令序列复用，例如方法调用、循环跳转、异常跳转等，所以具有这些功能的指令才会产生 Safepoint。

17. 简述 safe region，什么代码能达到 safe region 的条件？

安全区域是指在一段代码片段之中，引用关系不会发生变化。在这个区域中的任意地方开始 GC 都是安全的。我们也可以把 Safe Region 看做是被扩展了的 Safepoint。在线程执行到 Safe Region 中的代码时，首先标识自己已经进入了 Safe Region，那样，当在这段时间里 JVM 要发起 GC 时，就不用管标识自己为 Safe Region 状态的线程了。在线程要离开 Safe Region 时，它要检查系统是否已经完成了根节点枚举（或者是整个 GC 过程），如果完成了，那线程就继续执行，否则它就必须等待直到收到可以安全离开 Safe Region 的信号为止。
条件：Sleep 状态或者 Blocked 状态。

18. JVM 触发 GC 时需要中断其他线程吗？如何实现中断？Android 虚拟机触发 GC 时需要中断其他线程吗？

要，主动式中断。需要。

19. 简述各种 GC 收集器的实现与其优缺点。

1）Serial 收集器：Serial 收集器是最基本、发展历史最悠久的收集器，曾经（在 JDK 1.3.1 之前）是虚拟机新生代收集的唯一选择。大家看名字就会知道，这个收集器是一个单线程的收集器，但它的“单线程”的意义并不仅仅说明它只会使用一个 CPU 或一条收集线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。“Stop The World”这个名字也许听起来很酷，但这项工作实际上是由虚拟机在后台自动发起和自动完成的，在用户不可见的情况下把用户正常工作的线程全部停掉，这对很多应用来说都是难以接受的。简单而高效（与其他收集器的单线程比），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可以获得最高的单线程收集效率。在用户的桌面应用场景中，分配给虚拟机管理的内存一般来说不会很大，收集几十兆甚至一两百兆的新生代（仅仅是新生代使用的内存，桌面应用基本上不会再大了），停顿时间完全可以控制在几十毫秒最多一百多毫秒以内，只要不是频繁发生，这点停顿是可以接受的。
2）ParNew 收集器：ParNew 收集器其实就是 Serial 收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为包括 Serial 收集器可用的所有控制参数（例如：-XX：SurvivorRatio、-XX：PretenureSizeThreshold、-XX：HandlePromotionFailure 等）、收集算法、Stop The World、对象分配规则、回收策略等都与 Serial 收集器完全一样，在实现上，这两种收集器也共用了相当多的代码。ParNew 收集器除了多线程收集之外，其他与 Serial 收集器相比并没有太多创新之处，但它却是许多运行在 Server 模式下的虚拟机中首选的新生代收集器，其中有一个与性能无关但很重要的原因是，除了 Serial 收集器外，目前只有它能与 CMS 收集器配合工作。ParNew 收集器在单 CPU 的环境中绝对不会有比 Serial 收集器更好的效果，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 Serial 收集器。当然，随着可以使用的 CPU 的数量的增加，它对于 GC 时系统资源的有效利用还是很有好处的。它默认开启的收集线程数与 CPU 的数量相同，在 CPU 非常多（譬如 32 个，现在 CPU 动辄就 4 核加超线程，服务器超过 32 个逻辑 CPU 的情况越来越多了）的环境下，可以使用-XX：ParallelGCThreads 参数来限制垃圾收集的线程数。

3）Parallel Scavenge 收集器：Parallel Scavenge 收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器……看上去和 ParNew 都一样，那它有什么特别之处呢？Parallel Scavenge 收集器的特点是它的关注点与其他收集器不同，CMS 等收集器的关注点是尽可能地缩短垃圾收集时用户线程的停顿时间，而 Parallel Scavenge 收集器的目标则是达到一个可控制的吞吐量（Throughput）。

4）Serial Old 收集器：Serial Old 是 Serial 收集器的老年代版本，它同样是一个单线程收集器，使用“标记-整理”算法。这个收集器的主要意义也是在于给 Client 模式下的虚拟机使用。如果在 Server 模式下，那么它主要还有两大用途：一种用途是在 JDK 1.5 以及之前的版本中与 Parallel Scavenge 收集器搭配使用[1]，另一种用途就是作为 CMS 收集器的后备预案，在并发收集发生 Concurrent Mode Failure 时使用。

5）Parallel Old 收集器：Parallel Old 是 Parallel Scavenge 收集器的老年代版本，使用多线程和“标记-整理”算法。这个收集器是在 JDK 1.6 中才开始提供的，在此之前，新生代的 Parallel Scavenge 收集器一直处于比较尴尬的状态。原因是，如果新生代选择了 Parallel Scavenge 收集器，老年代除了 Serial Old（PS MarkSweep）收集器外别无选择（还记得上面说过 Parallel Scavenge 收集器无法与 CMS 收集器配合工作吗？）。由于老年代 Serial Old 收集器在服务端应用性能上的“拖累”，使用了 Parallel Scavenge 收集器也未必能在整体应用上获得吞吐量最大化的效果，由于单线程的老年代收集中无法充分利用服务器多 CPU 的处理能力，在老年代很大而且硬件比较高级的环境中，这种组合的吞吐量甚至还不一定有 ParNew 加 CMS 的组合“给力”。直到 Parallel Old 收集器出现后，“吞吐量优先”收集器终于有了比较名副其实的应用组合，在注重吞吐量以及 CPU 资源敏感的场合，都可以优先考虑 Parallel Scavenge 加 Parallel Old 收集器。

6）CMS 收集器：CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的 Java 应用集中在互联网站或者 B/S 系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。CMS 收集器就非常符合这类应用的需求。

初始标记（CMS initial mark）

并发标记（CMS concurrent mark）

重新标记（CMS remark）

并发清除（CMS concurrent sweep）

初始标记、重新标记这两个步骤仍然需要“Stop The World”。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，并发标记阶段就是进行 GC RootsTracing 的过程，而重新标记阶段则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，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。

7）G1 收集器：

20. GC 回收中吞吐量的概念是？吞吐量的高低受什么因素影响（不是问什么参数哦）？

吞吐量就是 CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值，即吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间），虚拟机总共运行了 100 分钟，其中垃圾收集花掉 1 分钟，那吞吐量就是 99%。

GC 停顿时间缩短是以牺牲吞吐量和新生代空间来换取的：系统把新生代调小一些，收集

300MB 新生代肯定比收集 500MB 快吧，这也直接导致垃圾收集发生得更频繁一些，原来 10 秒收集一次、每次停顿 100 毫秒，现在变成 5 秒收集一次、每次停顿 70 毫秒。停顿时间的确在下降，但吞吐量也降下来了。

21. GC 收集器的各种使用组合分别有哪些请一一列举，以及什么情况下应该使用什么组合(每种组合的应用场景)并说明为什么。

Serial 和 Serial Old：客户端
ParNew 和 Serial Old：多核服务器端，CMS 的后备预案
Parallel Scavenge 和 Serial Old：追求吞吐量
Parallel Scavenge 和 Parallel Old：吞吐量优先
CMS 和 Serial：客户端，GC 停顿时间短
CMS 和 ParNew：多核服务器端，GC 停顿时间短
G1：单独使用

22. CMS 收集器什么时候会 CM Failure，会造成什么后果？

CMS 收集器无法处理浮动垃圾（Floating Garbage），可能出现“Concurrent Mode Failure”失败而导致另一次 Full GC 的产生。在 JDK 1.6 中，CMS 收集器的启动阈值已经提升至 92%。要是 CMS 运行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这时虚拟机将启动后备预案：临时启用 Serial Old 收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。

23. G1 收集器比 CMS 收集器的先进之处？

并行与并发： G1 能充分利用多 CPU、多核环境下的硬件优势，使用多个 CPU（CPU 或者 CPU 核心）来缩短 Stop-The-World 停顿的时间，部分其他收集器原本需要停顿 Java 线程执行的 GC 动作，G1 收集器仍然可以通过并发的方式让 Java 程序继续执行。
分代收集：与其他收集器一样，分代概念在 G1 中依然得以保留。虽然 G1 可以不需要其他收集器配合就能独立管理整个 GC 堆，但它能够采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过多次 GC 的旧对象以获取更好的收集效果。
空间整合：与 CMS 的“标记—清理”算法不同，G1 从整体来看是基于“标记—整理”算法实现的收集器，从局部（两个 Region 之间）上来看是基于“复制”算法实现的，但无论如何，这两种算法都意味着 G1 运作期间不会产生内存空间碎片，收集后能提供规整的可用内存。这种特性有利于程序长时间运行，分配大对象时不会因为无法找到连续内存空间而提前触发下一次 GC。
可预测的停顿：这是 G1 相对于 CMS 的另一大优势，降低停顿时间是 G1 和 CMS 共同的关注点，但 G1 除了追求低停顿外，还能建立可预测的停顿时间模型，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（RTSJ）的垃圾收集器的特征了。

24. 为什么大对象要直接分配到老年代中？

大对象容易导致内存还有不少空间时就提前触发垃圾收集以获取足够的连续空间来“安置”它们。这样做的目的是避免大对象在 Eden 区及两个 Survivor 区之间发生大量的内存复制。

25. 对象的“岁数”是如何增长的？

虚拟机给每个对象定义了一个对象年龄（Age）计数器。如果对象在 Eden 出生并经过第一次 Minor GC 后仍然存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，并且对象年龄设为 1。对象在 Survivor 区中每“熬过”一次 Minor GC，年龄就增加 1 岁，当它的年龄增加到一定程度（默认为 15 岁），就将会被晋升到老年代中。对象晋升老年代的年龄阈值，可以通过参数-XX：MaxTenuringThreshold 设置。

26. 新生代对象何时会进入老年代？

1）虚拟机给每个对象定义了一个对象年龄（Age）计数器。如果对象在 Eden 出生并经过第一次 Minor GC 后仍然存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，并且对象年龄设为 1。对象在 Survivor 区中每“熬过”一次 Minor GC，年龄就增加 1 岁，当它的年龄增加到一定程度（默认为 15 岁），就将会被晋升到老年代中。
2）如果在 Survivor 空间中相同年龄所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等到 MaxTenuringThreshold 中要求的年龄。

27. 简述空间分配担保。

在发生 Minor GC 之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那么 Minor GC 可以确保是安全的。如果不成立，则虚拟机会查看 HandlePromotionFailure 设置值是否允许担保失败。如果允许，那么会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次 Minor GC，尽管这次 Minor GC 是有风险的；如果小于，或者 HandlePromotionFailure 设置不允许冒险，那这时也要改为进行一次 Full GC。

新生代使用复制收集算法，但为了内存利用率，只使用其中一个 Survivor 空间来作为轮换备份，因此当出现大量对象在 MinorGC 后仍然存活的情况（最极端的情况就是内存回收后新生代中所有对象都存活），就需要老年代进行分配担保，把 Survivor 无法容纳的对象直接进入老年代。与生活中的贷款担保类似，老年代要进行这样的担保，前提是老年代本身还有容纳这些对象的剩余空间，一共有多少对象会活下来在实际完成内存回收之前是无法明确知道的，所以只好取之前每一次回收晋升到老年代对象容量的平均大小值作为经验值，与老年代的剩余空间进行比较，决定是否进行 Full GC 来让老年代腾出更多空间。