第三版：JVM 77 道

java 中会存在内存泄漏吗，请简单描述。

会。自己实现堆载的数据结构时有可能会出现内存泄露。

64 位 JVM 中，int 的长度是多数?

Java 中，int 类型变量的长度是一个固定值，与平台无关，都是 32 位。意思就是说，在 32 位和 64 位的 Java 虚拟机中，int 类型的长度是相同的。

Serial 与 Parallel GC 之间的不同之处?

Serial 与 Parallel 在 GC 执行的时候都会引起 stop-the-world。它们之间主要不同 serial 收集器是默认的复制收集器，执行 GC 的时候只有一个线程，而 parallel 收集器使用多个 GC 线程来执行。

32 位和 64 位的 JVM，int 类型变量的长度是多数?

32 位和 64 位的 JVM 中，int 类型变量的长度是相同的，都是 32 位或者 4 个字节。

Java 中 WeakReference 与 SoftReference 的区别?

虽然 WeakReference 与 SoftReference 都有利于提高 GC 和 内存的效率，但是 WeakReference ，一旦失去最后一个强引用，就会被 GC 回收，而软引用虽然不能阻止被回收，但是可以延迟到 JVM 内存不足的时候。

JVM 选项 -XX:+UseCompressedOops 有什么作用？为什么要使用

当你将你的应用从 32 位的 JVM 迁移到 64 位的 JVM 时，由于对象的指针从 32 位增加到了 64 位，因此堆内存会突然增加，差不多要翻倍。这也会对 CPU 缓存（容量比内存小很多）的数据产生不利的影响。因为，迁移到 64 位的 JVM 主要动机在于可以指定最大堆大小，通过压缩 OOP 可以节省一定的内存。通过 -XX:+UseCompressedOops 选项，JVM 会使用 32 位的 OOP，而不是 64 位的 OOP。

怎样通过 Java 程序来判断 JVM 是 32 位 还是 64 位?

你可以检查某些系统属性如 sun.arch.data.model 或 os.arch 来获取该信息。

32 位 JVM 和 64 位 JVM 的最大堆内存分别是多数?

理论上说上 32 位的 JVM 堆内存可以到达 2^32，即 4GB，但实际上会比这个小很多。不同操作系统之间不同，如 Windows 系统大约 1.5GB，Solaris 大约 3GB。64 位 JVM 允许指定最大的堆内存，理论上可以达到 2^64，这是一个非常大的数字，实际上你可以指定堆内存大小到 100GB。甚至有的 JVM，如 Azul，堆内存到 1000G 都是可能的。

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 程序打开发工具，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。

解释 Java 堆空间及 GC?

当通过 Java 命令启动 Java 进程的时候，会为它分配内存。内存的一部分用于创建堆空间，当程序中创建对象的时候，就从对空间中分配内存。GC 是 JVM 内部的一个进程，回收无效对象的内存用于将来的分配。

JVM 内存区域

JVM 内存区域主要分为线程私有区域【程序计数器、虚拟机栈、本地方法区】、线程共享区域【JAVA 堆、方法区】、直接内存。

线程私有数据区域生命周期与线程相同，依赖用户线程的启动/结束 而 创建/销毁(在 Hotspot VM 内，每个线程都与操作系统的本地线程直接映射，因此这部分内存区域的存/否跟随本地线程的生/死对应)。

线程共享区域随虚拟机的启动/关闭而创建/销毁。

直接内存并不是 JVM 运行时数据区的一部分，但也会被频繁的使用：在 JDK 1.4 引入的 NIO 提供了基于 Channel 与 Buffer 的 IO 方式，它可以使用 Native

函数库直接分配堆外内存，然后使用 DirectByteBuffer 对象作为这块内存的引用进行操作(详见: Java I/O 扩展)，这样就避免了在 Java 堆和 Native 堆中来回复制数据，因此在一些场景中可以显著提高性能。

程序计数器(线程私有)

一块较小的内存空间，是当前线程所执行的字节码的行号指示器，每条线程都要有一个独立的程序计数器，这类内存也称为“线程私有”的内存。

正在执行 java 方法的话，计数器记录的是虚拟机字节码指令的地址（当前指令的地址）。如果还是 Native 方法，则为空。

这个内存区域是唯一一个在虚拟机中没有规定任何 OutOfMemoryError 情况的区域。

虚拟机栈(线程私有)

是描述 java 方法执行的内存模型，每个方法在执行的同时都会创建一个栈帧（Stack Frame）用于存储局部变量表、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，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

栈帧（ Frame）是用来存储数据和部分过程结果的数据结构，同时也被用来处理动态链接(Dynamic Linking)、 方法返回值和异常分派（Dispatch Exception）。栈帧随着方法调用而创建，随着方法结束而销毁——无论方法是正常完成还是异常完成（抛出了在方法内未被捕获的异常）都算作方法结束。

本地方法区(线程私有)

本地方法区和 Java Stack 作用类似，区别是虚拟机栈为执行 Java 方法服务，而本地方法栈则为 Native 方法服务，如果一个 VM 实现使用 C-linkage 模型来支持 Native 调用，那么该栈将会是一个 C 栈，但 HotSpot VM 直接就把本地方法栈和虚拟机栈合二为一 。

你能保证 GC 执行吗?

不能，虽然你可以调用 System.gc() 或者 Runtime.gc()，但是没有办法保证 GC 的执行。

怎么获取 Java 程序使用的内存？堆使用的百分比?

可以通过 java.lang.Runtime 类中与内存相关方法来获取剩余的内存，总内存及最大堆内存。通过这些方法你也可以获取到堆使用的百分比及堆内存的剩余空间。Runtime.freeMemory() 方法返回剩余空间的字节数，Runtime.totalMemory() 方法总内存的字节数，Runtime.maxMemory() 返回最大内存的字节数。

Java 中堆和栈有什么区别?

JVM 中堆和栈属于不同的内存区域，使用目的也不同。栈常用于保存方法帧和局部变量，而对象总是在堆上分配。栈通常都比堆小，也不会在多个线程之间共享，而堆被整个 JVM 的所有线程共享。

描述一下 JVM 加载 class 文件的原理机制

1、 JVM 中类的装载是由类加载器（ClassLoader）和它的子类来实现的，Java 中各类加载器是一个重要的 Java 运行时系统组件，它负责在运行时查找和装入类文件中的类。

2、 由于 Java 的跨平台性，经过编译的 Java 源程序并不是一个可执行程序，而是一个或多个类文件。当 Java 程序需要使用某个类时，JVM 会确保这个类已经被加载、连接（验证、准备和解析）和初始化。类的加载是指把类的.class 文件中的数据读入到内存中，通常是创建一个字节数组读入.class 文件，然后产生与所加载类对应的 Class 对象。

3、 加载完成后，Class 对象还不完整，所以此时的类还不可用。当类被加载后就进入连接阶段，这一阶段包括验证、准备（为静态变量分配内存并设置默认的初始值）和解析（将符号引用替换为直接引用）三个步骤。最后 JVM 对类进行初始化，包括：1)如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类；2)如果类中存在初始化语句，就依次执行这些初始化语句。

4、 类的加载是由类加载器完成的，类加载器包括：根加载器（BootStrap）、扩展加载器（Extension）、系统加载器（System）和用户自定义类加载器（java.lang.ClassLoader 的子类）。

5、 从 Java 2（JDK 1.2）开始，类加载过程采取了父亲委托机制（PDM）。PDM 更好的保证了 Java 平台的安全性，在该机制中，JVM 自带的 Bootstrap 是根加载器，其他的加载器都有且仅有一个父类加载器。类的加载首先请求父类加载器加载，父类加载器无能为力时才由其子类加载器自行加载。JVM 不会向 Java 程序提供对 Bootstrap 的引用。下面是关于几个类

加载器的说明：

1、 Bootstrap：一般用本地代码实现，负责加载 JVM 基础核心类库（rt.jar）；

2、 Extension：从 java.ext.dirs 系统属性所指定的目录中加载类库，它的父加载器是 Bootstrap；

3、 System：又叫应用类加载器，其父类是 Extension。它是应用最广泛的类加载器。它从环境变量 classpath 或者系统属性

java.class.path 所指定的目录中记载类，是用户自定义加载器的默认父加载器。

GC 是什么？为什么要有 GC？

GC 是垃圾收集的意思，内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java 提供的 GC 功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java 语言没有提供释放已分配内存的显示操作方法。Java 程序员不用担心内存管理，因为垃圾收集器会自动进行管理。要请求垃圾收集，可以调用下面的方法之一：System.gc() 或 Runtime.getRuntime().gc()，但 JVM 可以屏蔽掉线示的垃圾回收调用。

垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。垃圾回收器通常是作为一个单独的低优先级的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。在 Java 诞生初期，垃圾回收是 Java 最大的亮点之一，因为服务器端的编程需要有效的防止内存泄露问题，然而时过境迁，如今 Java 的垃圾回收机制已经成为被诟病的东。移动智能终端用户通常觉得 iOS 的系统比 Android 系统有更好的用户体验，其中一个深层次的原因就在于 Android 系统中垃圾回收的不可预知性。

堆（Heap-线程共享） -运行时数据区

是被线程共享的一块内存区域， 创建的对象和数组都保存在 Java 堆内存中，也是垃圾收集器进行垃圾收集的最重要的内存区域。由于现代 VM 采用分代收集算法，因此 Java 堆从 GC 的角度还可以细分为：新生代(Eden 区、 From Survivor 区和 To Survivor 区)和老年代。

方法区/永久代（线程共享）

即我们常说的永久代(Permanent Generation)，用于存储被 JVM 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.HotSpot VM 把 GC 分代收集扩展至方法区，即使用 Java 堆的永久代来实现方法区，这样 HotSpot 的垃圾收集器就可以像管理 Java 堆一样管理这部分内存,而不必为方法区开发专门的内存管理器(永久带的内存回收的主要目标是针对常量池的回收和类型的卸载，因此收益一般很小）。

运行时常量池（Runtime Constant Pool）是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述等信息外，还有一项信息是常量池（Constant Pool Table），用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。Java 虚拟机对 Class 文件的每一部分（自然也包括常量池）的格式都有严格的规定，每一个字节用于存储哪种数据都必须符合规范上的要求，这样才会被虚拟机认可、装载和执行。

JVM 运行时内存

Java 堆从 GC 的角度还可以细分为：新生代(Eden 区、 From Survivor 区和 To Survivor 区)和老年代。

新生代

是用来存放新生的对象。一般占据堆的 1/3 空间。由于频繁创建对象，所以新生代会频繁触发 MinorGC 进行垃圾回收。新生代又分为 Eden 区、ServivorFrom、ServivorTo 三个区。

Eden 区

Java 新对象的出生地（如果新创建的对象占用内存很大，则直接分配到老年代）。当 Eden 区内存不够的时候就会触发 MinorGC，对新生代区进行一次垃圾回收。

ServivorFrom

上一次 GC 的幸存者，作为这一次 GC 的被扫描者。

ServivorTo

保留了一次 MinorGC 过程中的幸存者。

MinorGC 的过程（复制->清空->互换）

MinorGC 采用复制算法。

eden、 servicorFrom 复制到 ServicorTo，年龄+1

首先，把 Eden 和 ServivorFrom 区域中存活的对象复制到 ServicorTo 区域（如果有对象的年龄以及达到了老年的标准，则赋值到老年代区），同时把这些对象的年龄+1（如果 ServicorTo 不够位置了就放到老年区）；

清空 eden、 servicorFrom

然后，清空 Eden 和 ServicorFrom 中的对象；

ServicorTo 和 ServicorFrom 互换

最后，ServicorTo 和 ServicorFrom 互换，原 ServicorTo 成为下一次 GC 时的 ServicorFrom 区。

老年代

1、 主要存放应用程序中生命周期长的内存对象。

2、 老年代的对象比较稳定，所以 MajorGC 不会频繁执行。在进行 MajorGC 前一般都先进行了一次 MinorGC，使得有新生代的对象晋身入老年代，导致空间不够用时才触发。当无法找到足够大的连续空间分配给新创建的较大对象时也会提前触发一次 MajorGC 进行垃圾回收腾出空间。

3、 MajorGC 采用标记清除算法：首先扫描一次所有老年代，标记出存活的对象，然后回收没有标记的对象。ajorGC 的耗时比较长，因为要扫描再回收。MajorGC 会产生内存碎片，为了减少内存损耗，我们一般需要进行合并或者标记出来方便下次直接分配。当老年代也满了装不下的时候，就会抛出 OOM（Out of Memory）异常。

永久代

指内存的永久保存区域，主要存放 Class 和 Meta（元数据）的信息,Class 在被加载的时候被放入永久区域， 它和和存放实例的区域不同,GC 不会在主程序运行期对永久区域进行清理。所以这也导致了永久代的区域会随着加载的 Class 的增多而胀满，最终抛出 OOM 异常。

JAVA8 与元数据

在 Java8 中， 永久代已经被移除，被一个称为“元数据区”（元空间）的区域所取代。元空间的本质和永久代类似，元空间与永久代之间最大的区别在于：元空间并不在虚拟机中，而是使用本地内存。因此，默认情况下，元空间的大小仅受本地内存限制。类的元数据放入 nativememory，字符串池和类的静态变量放入 java 堆中，这样可以加载多少类的元数据就不再由 MaxPermSize 控制，而由系统的实际可用空间来控制。

引用计数法

在 Java 中，引用和对象是有关联的。如果要操作对象则必须用引用进行。因此，很显然一个简单的办法是通过引用计数来判断一个对象是否可以回收。简单说，即一个对象如果没有任何与之关联的引用， 即他们的引用计数都不为 0， 则说明对象不太可能再被用到，那么这个对象就是可回收对象。

可达性分析

为了解决引用计数法的循环引用问题， Java 使用了可达性分析的方法。通过一系列的“GC roots”对象作为起点搜索。如果在“GC roots”和一个对象之间没有可达路径，则称该对象是不可达的。要注意的是，不可达对象不等价于可回收对象， 不可达对象变为可回收对象至少要经过两次标记过程。两次标记后仍然是可回收对象，则将面临回收。

标记清除算法（ Mark-Sweep）

最基础的垃圾回收算法，分为两个阶段，标注和清除。标记阶段标记出所有需要回收的对象，清除阶段回收被标记的对象所占用的空间。

从图中我们就可以发现，该算法最大的问题是内存碎片化严重，后续可能发生大对象不能找到可利用空间的问题。

复制算法（copying）

为了解决 Mark-Sweep 算法内存碎片化的缺陷而被提出的算法。按内存容量将内存划分为等大小的两块。每次只使用其中一块，当这一块内存满后将尚存活的对象复制到另一块上去，把已使用的内存清掉

这种算法虽然实现简单，内存效率高，不易产生碎片，但是最大的问题是可用内存被压缩到了原本的一半。且存活对象增多的话， Copying 算法的效率会大大降低。

标记整理算法(Mark-Compact)

结合了以上两个算法，为了避免缺陷而提出。标记阶段和 Mark-Sweep 算法相同， 标记后不是清理对象，而是将存活对象移向内存的一端。然后清除端边界外的对象。

分代收集算法

分代收集法是目前大部分 JVM 所采用的方法，其核心思想是根据对象存活的不同生命周期将内存划分为不同的域，一般情况下将 GC 堆划分为老生代(Tenured/Old Generation)和新生代(YoungGeneration)。老生代的特点是每次垃圾回收时只有少量对象需要被回收，新生代的特点是每次垃圾回收时都有大量垃圾需要被回收，因此可以根据不同区域选择不同的算法。

新生代与复制算法

目前大部分 JVM 的 GC 对于新生代都采取 Copying 算法，因为新生代中每次垃圾回收都要回收大部分对象，即要复制的操作比较少，但通常并不是按照 1：1 来划分新生代。一般将新生代划分为一块较大的 Eden 空间和两个较小的 Survivor 空间(From Space, To Space)，每次使用 Eden 空间和其中的一块 Survivor 空间，当进行回收时，将该两块空间中还存活的对象复制到另一块 Survivor 空间中。

老年代与标记复制算法

而老年代因为每次只回收少量对象，因而采用 Mark-Compact 算法。

1、 JAVA 虚拟机提到过的处于方法区的永生代(Permanet Generation)， 它用来存储 class 类，常量，方法描述等。对永生代的回收主要包括废弃常量和无用的类。

2、 对象的内存分配主要在新生代的 Eden Space 和 Survivor Space 的 From Space(Survivor 目前存放对象的那一块)，少数情况会直接分配到老生代。

3、 当新生代的 Eden Space 和 From Space 空间不足时就会发生一次 GC，进行 GC 后， EdenSpace 和 From Space 区的存活对象会被挪到 To Space，然后将 Eden Space 和 FromSpace 进行清理。

4、 如果 To Space 无法足够存储某个对象，则将这个对象存储到老生代。

5、 在进行 GC 后，使用的便是 Eden Space 和 To Space 了，如此反复循环。

6、 当对象在 Survivor 去躲过一次 GC 后，其年龄就会+1。默认情况下年龄到达 15 的对象会被移到老生代中。

JAVA 强引用

在 Java 中最常见的就是强引用， 把一个对象赋给一个引用变量，这个引用变量就是一个强引用。当一个对象被强引用变量引用时，它处于可达状态，它是不可能被垃圾回收机制回收的，即使该对象以后永远都不会被用到 JVM 也不会回收。因此强引用是造成 Java 内存泄漏的主要原因之一。

JAVA 软引用

软引用需要用 SoftReference 类来实现，对于只有软引用的对象来说，当系统内存足够时它不会被回收，当系统内存空间不足时它会被回收。软引用通常用在对内存敏感的程序中。

JAVA 弱引用

弱引用需要用 WeakReference 类来实现，它比软引用的生存期更短，对于只有弱引用的对象来说，只要垃圾回收机制一运行，不管 JVM 的内存空间是否足够，总会回收该对象占用的内存。

JAVA 虚引用

虚引用需要 PhantomReference 类来实现，它不能单独使用，必须和引用队列联合使用。虚引用的主要作用是跟踪对象被垃圾回收的状态。

分代收集算法

当前主流 VM 垃圾收集都采用”分代收集“ (Generational Collection)算法，这种算法会根据对象存活周期的不同将内存划分为几块，如 JVM 中的新生代、老年代、永久代， 这样就可以根据各年代特点分别采用最适当的 GC 算法

在新生代-复制算法

每次垃圾收集都能发现大批对象已死，只有少量存活、因此选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集

在老年代-标记整理算法

因为对象存活率高、没有额外空间对它进行分配担保，就必须采用“标记—清理”或“标记—整理” 算法来进行回收，不必进行内存复制，且直接腾出空闲内存。

分区收集算法

分区算法则将整个堆空间划分为连续的不同小区间，每个小区间独立使用，独立回收、这样做的好处是可以控制一次回收多少个小区间 ，根据目标停顿时间，每次合理地回收若干个小区间(而不是整个堆)，从而减少一次 GC 所产生的停顿。

GC 垃圾收集器

Java 堆内存被划分为新生代和年老代两部分，新生代主要使用复制和标记-清除垃圾回收算法；年老代主要使用标记-整理垃圾回收算法，因此 java 虚拟中针对

新生代和年老代分别提供了多种不同的垃圾收集器， JDK1.6 中 Sun HotSpot 虚拟机的垃圾收集器

Serial 垃圾收集器（单线程、 复制算法）

Serial（英文连续） 是最基本垃圾收集器，使用复制算法，曾经是 JDK1.3.1 之前新生代唯一的垃圾收集器。Serial 是一个单线程的收集器，它不但只会使用一个 CPU 或一条线程去完成垃圾收集工作，并且在进行垃圾收集的同时，必须暂停其他所有的工作线程，直到垃圾收集结束。

Serial 垃圾收集器虽然在收集垃圾过程中需要暂停所有其他的工作线程，但是它简单高效，对于限定单个 CPU 环境来说，没有线程交互的开销，可以获得最高的单线程垃圾收集效率，因此 Serial 垃圾收集器依然是 java 虚拟机运行在 Client 模式下默认的新生代垃圾收集器。

ParNew 垃圾收集器（Serial+多线程）

ParNew 垃圾收集器其实是 Serial 收集器的多线程版本，也使用复制算法，除了使用多线程进行垃圾收集之外，其余的行为和 Serial 收集器完全一样， ParNew 垃圾收集器在垃圾收集过程中同样也要暂停所有其他的工作线程。

ParNew 收集器默认开启和 CPU 数目相同的线程数，可以通过 -XX:ParallelGCThreads 参数来限制垃圾收集器的线程数。【Parallel：平行的】

ParNew 虽然是除了多线程外和 Serial 收集器几乎完全一样，但是 ParNew 垃圾收集器是很多 java 虚拟机运行在 Server 模式下新生代的默认垃圾收集器。

Parallel Scavenge 收集器（多线程复制算法、高效）

Parallel Scavenge 收集器也是一个新生代垃圾收集器，同样使用复制算法，也是一个多线程的垃圾收集器， 它重点关注的是程序达到一个可控制的吞吐量（Thoughput， CPU 用于运行用户代码的时间/CPU 总消耗时间，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)），高吞吐量可以最高效率地利用 CPU 时间，尽快地完成程序的运算任务，主要适用于在后台运算而不需要太多交互的任务。自适应调节策略也是 ParallelScavenge 收集器与 ParNew 收集器的一个重要区别。

Serial Old 收集器（单线程标记整理算法 ）

Serial Old 是 Serial 垃圾收集器年老代版本，它同样是个单线程的收集器，使用标记-整理算法，这个收集器也主要是运行在 Client 默认的

java 虚拟机默认的年老代垃圾收集器。在 Server 模式下，主要有两个用途：

1、 在 JDK1.5 之前版本中与新生代的 Parallel Scavenge 收集器搭配使用。

2、 作为年老代中使用 CMS 收集器的后备垃圾收集方案。新生代 Serial 与年老代 Serial Old 搭配垃圾收集

新生代 Parallel Scavenge 收集器与 ParNew 收集器工作原理类似，都是多线程的收集器，都使用的是复制算法，在垃圾收集过程中都需要暂停所有的工作线程。新生代 ParallelScavenge/ParNew 与年老代 Serial Old 搭配垃圾收集过程图：

Parallel Old 收集器（多线程标记整理算法）

Parallel Old 收集器是 Parallel Scavenge 的年老代版本，使用多线程的标记-整理算法，在 JDK1.6 才开始提供。

在 JDK1.6 之前，新生代使用 ParallelScavenge 收集器只能搭配年老代的 Serial Old 收集器，只能保证新生代的吞吐量优先，无法保证整体的吞吐量，Parallel Old 正是为了在年老代同样提供吞吐量优先的垃圾收集器，如果系统对吞吐量要求比较高，可以优先考虑新生代 Parallel Scavenge 和年老代 Parallel Old 收集器的搭配策略。

CMS 收集器（多线程标记清除算法）

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：

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。

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。

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。

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现在一起并发工作， 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。

G1 收集器

Garbage first 垃圾收集器是目前垃圾收集器理论发展的最前沿成果，相比与 CMS 收集器， G1 收集器两个最突出的改进是：

1、 基于标记-整理算法，不产生内存碎片。

2、 可以非常精确控制停顿时间，在不牺牲吞吐量前提下，实现低停顿垃圾回收。G1 收集器避免全区域垃圾收集，它把堆内存划分为大小固定的几个独立区域，并且跟踪这些区域的垃圾收集进度，同时在后台维护一个优先级列表，每次根据所允许的收集时间， 优先回收垃圾最多的区域。区域划分和优先级区域回收机制，确保 G1 收集器可以在有限时间获得最高的垃圾收集效率

JVM 类加载机制

JVM 类加载机制分为五个部分：加载，验证，准备，解析，初始化。

加载

加载是类加载过程中的一个阶段， 这个阶段会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种数据的入口。注意这里不一定非得要从一个 Class 文件获取，这里既可以从 ZIP 包中读取（比如从 jar 包和

war 包中读取），也可以在运行时计算生成（动态代理），也可以由其它文件生成（比如将 JSP 文件转换成对应的 Class 类）。

验证

这一阶段的主要目的是为了确保 Class 文件的字节流中包含的信息是否符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

准备

准备阶段是正式为类变量分配内存并设置类变量的初始值阶段，即在方法区中分配这些变量所使用的内存空间。注意这里所说的初始值概念，比如一个类变量定义为：

实际上变量 v 在准备阶段过后的初始值为 0 而不是 8080，将 v 赋值为 8080 的 put static 指令是程序被编译后，存放于类构造器方法之中。

但是注意如果声明为：

```
public static final int v = 8080;
```

在编译阶段会为 v 生成 ConstantValue 属性，在准备阶段虚拟机会根据 ConstantValue 属性将 v 赋值为 8080。

解析

解析阶段是指虚拟机将常量池中的符号引用替换为直接引用的过程。符号引用就是 class 文件中的

```
public static int v = 8080;
```

实际上变量 v 在准备阶段过后的初始值为 0 而不是 8080， 将 v 赋值为 8080 的 put static 指令是程序被编译后，存放于类构造器方法之中。但是注意如果声明为：

在编译阶段会为 v 生成 ConstantValue 属性，在准备阶段虚拟机会根据 ConstantValue 属性将 v 赋值为 8080。

解析

解析阶段是指虚拟机将常量池中的符号引用替换为直接引用的过程。符号引用就是 class 文件中的

public static final int v = 8080;

在编译阶段会为 v 生成 ConstantValue 属性，在准备阶段虚拟机会根据 ConstantValue 属性将 v 赋值为 8080。

解析

解析阶段是指虚拟机将常量池中的符号引用替换为直接引用的过程。符号引用就是 class 文件中的：

1、 CONSTANT_Class_info

2、 CONSTANT_Field_info

3、 CONSTANT_Method_info

等类型的常量。

符号引用

符号引用与虚拟机实现的布局无关， 引用的目标并不一定要已经加载到内存中。各种虚拟机实现的内存布局可以各不相同，但是它们能接受的符号引用必须是一致的，因为符号引用的字面量形式明确定义在 Java 虚拟机规范的 Class 文件格式中。

直接引用

直接引用可以是指向目标的指针，相对偏移量或是一个能间接定位到目标的句柄。如果有了直接引用，那引用的目标必定已经在内存中存在。

初始化

初始化阶段是类加载最后一个阶段，前面的类加载阶段之后，除了在加载阶段可以自定义类加载器以外，其它操作都由 JVM 主导。到了初始阶段，才开始真正执行类中定义的 Java 程序代码。

类构造器

初始化阶段是执行类构造器方法的过程。方法是由编译器自动收集类中的类变量的赋值操作和静态语句块中的语句合并而成的。虚拟机会保证子方法执行之前，父类的方法已经执行完毕， 如果一个类中没有对静态变量赋值也没有静态语句块，那么编译器可以不为这个类生成()方法。

注意以下几种情况不会执行类初始化：

1、 通过子类引用父类的静态字段，只会触发父类的初始化，而不会触发子类的初始化。

2、 定义对象数组，不会触发该类的初始化。

3、 常量在编译期间会存入调用类的常量池中，本质上并没有直接引用定义常量的类，不会触发定义常量所在的类。

4、 通过类名获取 Class 对象，不会触发类的初始化。

5、 通过 Class.forName 加载指定类时，如果指定参数 initialize 为 false 时，也不会触发类初始化，其实这个参数是告诉虚拟机，是否要对类进行初始化。

6、 通过 ClassLoader 默认的 loadClass 方法，也不会触发初始化动作。

类加载器

虚拟机设计团队把加载动作放到 JVM 外部实现，以便让应用程序决定如何获取所需的类， JVM 提供了 3 种类加载器：

启动类加载器(Bootstrap ClassLoader)

负责加载 JAVA_HOME\lib 目录中的， 或通过-Xbootclasspath 参数指定路径中的， 且被虚拟机认可（按文件名识别， 如 rt.jar） 的类。

扩展类加载器(Extension ClassLoader)

负责加载 JAVA_HOME\lib\ext 目录中的，或通过 java.ext.dirs 系统变量指定路径中的类库。

应用程序类加载器(Application ClassLoader)：

负责加载用户路径（classpath）上的类库。JVM 通过双亲委派模型进行类的加载， 当然我们也可以通过继承 java.lang.ClassLoader 实现自定义的类加载器。

双亲委派

当一个类收到了类加载请求，他首先不会尝试自己去加载这个类，而是把这个请求委派给父类去完成，每一个层次类加载器都是如此，因此所有的加载请求都应该传送到启动类加载其中，只有当父类加载器反馈自己无法完成这个请求的时候（在它的加载路径下没有找到所需加载的 Class），子类加载器才会尝试自己去加载。

采用双亲委派的一个好处是比如加载位于 rt.jar 包中的类 java.lang.Object，不管是哪个加载器加载这个类，最终都是委托给顶层的启动类加载器进行加载，这样就保证了使用不同的类加载器最终得到的都是同样一个 Object 对象

OSGI（动态模型系统）

OSGi(Open Service Gateway Initiative)，是面向 Java 的动态模型系统，是 Java 动态化模块化系统的一系列规范。

动态改变构造

OSGi 服务平台提供在多种网络设备上无需重启的动态改变构造的功能。为了最小化耦合度和促使这些耦合度可管理，OSGi 技术提供一种面向服务的架构，它能使这些组件动态地发现对方。

模块化编程与热插拔

OSGi 旨在为实现 Java 程序的模块化编程提供基础条件，基于 OSGi 的程序很可能可以实现模块级的热插拔功能，当程序升级更新时，可以只停用、重新安装然后启动程序的其中一部分，这对企业级程序开发来说是非常具有诱惑力的特性。

OSGi 描绘了一个很美好的模块化开发目标，而且定义了实现这个目标的所需要服务与架构，同时也有成熟的框架进行实现支持。但并非所有的应用都适合采用 OSGi 作为基础架构，它在提供强大功能同时，也引入了额外的复杂度，因为它不遵守了类加载的双亲委托模型。

JVM 内存模型

线程独占:栈,本地方法栈,程序计数器

线程共享:堆,方法区

栈

又称方法栈,线程私有的,线程执行方法是都会创建一个栈阵,用来存储局部变量表,操作栈,动态链接,方法出口等信息.调用方法时执行入栈,方法返回式执行出栈.

本地方法栈

与栈类似,也是用来保存执行方法的信息.执行 Java 方法是使用栈,执行 Native 方法时使用本地方法栈.

程序计数器

保存着当前线程执行的字节码位置,每个线程工作时都有独立的计数器,只为执行 Java 方法服务,执行 Native 方法时,程序计数器为空.

堆

JVM 内存管理最大的一块,对被线程共享,目的是存放对象的实例,几乎所欲的对象实例都会放在这里,当堆没有可用空间时,会抛出 OOM 异常.根据对象的存活周期不同,JVM 把对象进行分代管理,由垃圾回收器进行垃圾的回收管理

方法区

又称非堆区,用于存储已被虚拟机加载的类信息,常量,静态变量,即时编译器优化后的代码等数据.1.7 的永久代和 1.8 的元空间都是方法区的一种实现。

分代回收

分代回收基于两个事实:大部分对象很快就不使用了,还有一部分不会立即无用,但也不会持续很长时间

年轻代->标记-复制

老年代->标记-清除

堆和栈的区别

栈是运行时单位，代表着逻辑，内含基本数据类型和堆中对象引用，所在区域连续，没有碎片；堆是存储单位，代表着数据，可被多个栈共享（包括成员中基本数据类型、引用和引用对象），所在区域不连续，会有碎片。

功能不同

栈内存用来存储局部变量和方法调用，而堆内存用来存储 Java 中的对象。无论是成员变量，局部变量，还是类变量，它们指向的对象都存储在堆内存中。

共享性不同

栈内存是线程私有的。

堆内存是所有线程共有的。

异常错误不同

如果栈内存或者堆内存不足都会抛出异常。

栈空间不足：java.lang.StackOverFlowError。

堆空间不足：java.lang.OutOfMemoryError。

空间大小

栈的空间大小远远小于堆的

什么时候会触发 FullGC

除直接调用 System.gc 外，触发 Full GC 执行的情况有如下四种。

旧生代空间不足

旧生代空间只有在新生代对象转入及创建为大对象、大数组时才会出现不足的现象，当执行 Full GC 后空间仍然不足，则抛出如下错

误：

java.lang.OutOfMemoryError: Java heap space

为避免以上两种状况引起的 FullGC，调优时应尽量做到让对象在 Minor GC 阶段被回收、让对象在新生代多存活一段时间及不要创建过大的对象及数组。

Permanet Generation 空间满

PermanetGeneration 中存放的为一些 class 的信息等，当系统中要加载的类、反射的类和调用的方法较多时，Permanet Generation 可能会被占满，在未配置为采用 CMS GC 的情况下会执行 Full GC。如果经过 Full GC 仍然回收不了，那么 JVM 会抛出如下错误信息：

java.lang.OutOfMemoryError: PermGen space

为避免 Perm Gen 占满造成 Full GC 现象，可采用的方法为增大 Perm Gen 空间或转为使用 CMS GC。

CMS GC 时出现 promotion failed 和 concurrent mode failure

对于采用 CMS 进行旧生代 GC 的程序而言，尤其要注意 GC 日志中是否有 promotion failed 和 concurrent mode failure 两种状况，当这两种状况出现时可能会触发 Full GC。

promotionfailed 是在进行 Minor GC 时，survivor space 放不下、对象只能放入旧生代，而此时旧生代也放不下造成的；concurrentmode failure 是在执行 CMS GC 的过程中同时有对象要放入旧生代，而此时旧生代空间不足造成的。

应对措施为：增大 survivorspace、旧生代空间或调低触发并发 GC 的比率，但在 JDK 5.0+、6.0+的版本中有可能会由于 JDK 的 bug29 导致 CMS 在 remark 完毕后很久才触发 sweeping 动作。对于这种状况，可通过设置 -XX:CMSMaxAbortablePrecleanTime=5（单位为 ms）来避免。

统计得到的 Minor GC 晋升到旧生代的平均大小大于旧生代的剩余空间

这是一个较为复杂的触发情况，Hotspot 为了避免由于新生代对象晋升到旧生代导致旧生代空间不足的现象，在进行 Minor GC 时，做了一个判断，如果之前统计所得到的 Minor GC 晋升到旧生代的平均大小大于旧生代的剩余空间，那么就直接触发 Full GC。

例如程序第一次触发 MinorGC 后，有 6MB 的对象晋升到旧生代，那么当下一次 Minor GC 发生时，首先检查旧生代的剩余空间是否大于 6MB，如果小于 6MB，则执行 Full GC。

当新生代采用 PSGC 时，方式稍有不同，PS GC 是在 Minor GC 后也会检查，例如上面的例子中第一次 Minor GC 后，PS GC 会检查此时旧生代的剩余空间是否大于 6MB，如小于，则触发对旧生代的回收。除了以上 4 种状况外，对于使用 RMI 来进行 RPC 或管理的 Sun JDK 应用而言，默认情况下会一小时执行一次 Full GC。可通过在启动时通过- java-Dsun.rmi.dgc.client.gcInterval=3600000 来设置 Full GC 执行的间隔时间或通过-XX:+ DisableExplicitGC 来禁止 RMI 调用 System.gc

什么是 Java 虚拟机？为什么 Java 被称作是“平台无关的编程语言”？

Java 虚拟机是一个可以执行 Java 字节码的虚拟机进程。Java 源文件被编译成能被 Java 虚拟机执行的字节码文件。Java 被设计成允许应用程序可以运行在任意的平台，而不需要程序员为每一个平台单独重写或者是重新编译。Java 虚拟机让这个变为可能，因为它知道底层硬件平台的指令长度和其他特性。

对象分配规则

1、 对象优先分配在 Eden 区，如果 Eden 区没有足够的空间时，虚拟机执行一次 Minor GC。

2、 大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。这样做的目的是避免在 Eden 区和两个 Survivor 区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。

3、 长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了 1 次 Minor GC 那么对象会进入 Survivor 区，之后每经过一次 Minor GC 那么对象的年龄加 1，知道达到阀值对象进入老年区。

4、 动态判断对象的年龄。如果 Survivor 区中相同年龄的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可以直接进入老年代。

5、 空间分配担保。每次进行 Minor GC 时，JVM 会计算 Survivor 区移至老年区的对象的平均大小，如果这个值大于老年区的剩余值大小则进行一次 Full GC，如果小于检查 HandlePromotionFailure 设置，如果 true 则只进行 Monitor GC，如果 false 则进行 Full GC

描述一下 JVM 加载 class 文件的原理机制?

JVM 中类的装载是由类加载器（ClassLoader）和它的子类来实现的，Java 中的类加载器是一个重要的 Java 运行时系统组件，它负责在运行时查找和装入类文件中的类。

由于 Java 的跨平台性，经过编译的 Java 源程序并不是一个可执行程序，而是一个或多个类文件。当 Java 程序需要使用某个类时，JVM 会确保这个类已经被加载、连接（验证、准备和解析）和初始化。

类的加载是指把类的.class 文件中的数据读入到内存中，通常是创建一个字节数组读入.class 文件，然后产生与所加载类对应的 Class 对象。加载完成后，Class 对象还不完整，所以此时的类还不可用。

当类被加载后就进入连接阶段，这一阶段包括验证、准备（为静态变量分配内存并设置默认的初始值）和解析（将符号引用替换为直接引用）三个步骤。最后 JVM 对类进行初始化，

包括：

1、 如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类；

2、 如果类中存在初始化语句，就依次执行这些初始化语句。类的加载是由类加载器完成的，类加载器包括：根加载器（BootStrap）、扩展加载器（Extension）、系统加载器（System）和用户自定义类加载器（java.lang.ClassLoader 的子类）。

从 Java 2（JDK 1.2）开始，类加载过程采取了父亲委托机制（PDM）。PDM 更好的保证了 Java 平台的安全性，在该机制中，JVM 自带的 Bootstrap 是根加载器，其他的加载器都有且仅有一个父类加载器。类的加载首先请求父类加载器加载，

父类加载器无能为力时才由其子类加载器自行加载。JVM 不会向 Java 程序提供对 Bootstrap 的引用。下面是关于几个类加载器的说明

Bootstrap：一般用本地代码实现，负责加载 JVM 基础核心类库（rt.jar）；

Extension：从 java.ext.dirs 系统属性所指定的目录中加载类库，它的父加载器是 Bootstrap；

System：又叫应用类加载器，其父类是 Extension。它是应用最广泛的类加载器。它从环境变量 classpath 或者系统属性 java.class.path 所指定的目录中记载类，是用户自定义加载器的默认父加载器。

Java 对象创建过程

1、 JVM 遇到一条新建对象的指令时首先去检查这个指令的参数是否能在常量池中定义到一个类的符号引用。然后加载这个类（类加载过程在后边讲）

2、 为对象分配内存。一种办法“指针碰撞”、一种办法“空闲列表”，最终常用的办法“本地线程缓冲分配(TLAB)”

3、 将除对象头外的对象内存空间初始化为 0

4、 对对象头进行必要设置

简述 Java 的对象结构

Java 对象由三个部分组成：对象头、实例数据、对齐填充。

对象头由两部分组成，第一部分存储对象自身的运行时数据：哈希码、GC 分代年龄、锁标识状态、线程持有的锁、偏向线程 ID（一般占 32/64 bit）。第二部分是指针类型，指向对象的类元数据类型（即对象代表哪个类）。如果是数组对象，则对象头中还有一部分用来记录数组长度。

实例数据用来存储对象真正的有效信息（包括父类继承下来的和自己定义的）

对齐填充：JVM 要求对象起始地址必须是 8 字节的整数倍（8 字节对齐 ）

如何判断对象可以被回收

判断对象是否存活一般有两种方式：

引用计数：每个对象有一个引用计数属性，新增一个引用时计数加 1，引用释放时计数减 1，计数为 0 时可以回收。此方法简单，无法解决对象相互循环引用的问题。

可达性分析（Reachability Analysis）：从 GC Roots 开始向下搜索，搜索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时，则证明此对象是不可用的，不可达对象。

JVM 的永久代中会发生垃圾回收么

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小对避免 Full GC 是非常重要的原因。请参考下 Java8：从永久代到元数据区（注：Java8 中已经移除了永久代，新加了一个叫做元数据区的 native 内存区）

垃圾收集算法

GC 最基础的算法有三种：标记 -清除算法、复制算法、标记-压缩算法，我们常用的垃圾回收器一般都采用分代收集算法。

标记 -清除算法

“标记-清除”（Mark-Sweep）算法，如它的名字一样，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，在标记完成后统一回收掉所有被标记的对象。

复制算法

“复制”（Copying）的收集算法，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。

标记-压缩算法

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

分代收集算法

“分代收集”（Generational Collection）算法，把 Java 堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法

调优命令有哪些?

Sun JDK 监控和故障处理命令有 jps jstat jmap jhat jstack jinfo

1、 jps，JVM Process Status Tool,显示指定系统内所有的 HotSpot 虚拟机进程。

2、 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。

3、 jmap，JVM Memory Map 命令用于生成 heap dump 文件

4、 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

5、 jstack，用于生成 java 虚拟机当前时刻的线程快照。

6、 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

调优工具

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。

1、 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

2、 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。

3、 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

4、 GChisto，一款专业分析 gc 日志的工具

Minor GC 与 Full GC 分别在什么时候发生?

新生代内存不够用时候发生 MGC 也叫 YGC，JVM 内存不够的时候发生 FGC

你知道哪些 JVM 性能调优

设定堆内存大小

1、 -Xmx：堆内存最大限制。设定新生代大小。新生代不宜太小，否则会有大量对象涌入老年代

2、 -XX:NewSize：新生代大小

3、 -XX:NewRatio 新生代和老生代占比

4、 -XX:SurvivorRatio：伊甸园空间和幸存者空间的占比

5、 设定垃圾回收器 年轻代用 -XX:+UseParNewGC 年老代用 -XX:+UseConcMarkSweepGC