目录

[1. JVM内存管理的机制是什么？(待进一步确定) 3](#_Toc456643467)

[2. JVM实例 3](#_Toc456643468)

[3. JVM运行时结构 3](#_Toc456643469)

[4. JVM运行时结构与异常 4](#_Toc456643470)

[5. 栈和堆的区别 4](#_Toc456643471)

[6. 堆内存的区域有哪些划分? JVM的堆内存是什么样的结构? 5](#_Toc456643472)

[7. 垃圾收集算法有哪些？ 5](#_Toc456643473)

[8. JVM中GC停顿是为什么？ 6](#_Toc456643474)

[9. 垃圾回收是什么? 6](#_Toc456643475)

[10. 垃圾回收的两面性？ 6](#_Toc456643476)

[11. 为什么要有垃圾回收? 6](#_Toc456643477)

[12. 什么时候运行垃圾回收？(待进一步确认) 6](#_Toc456643478)

[13. Java类加载机制是什么 7](#_Toc456643479)

[14. Java内存泄露和内存溢出啥区别？ 8](#_Toc456643480)

[15. 怎样判断对象是否还活着? 9](#_Toc456643481)

[16. GC roots包含哪些对象？ 10](#_Toc456643482)

[17. 引用的类型 10](#_Toc456643483)

[18. 回收方法区 10](#_Toc456643484)

[19. 了解哪些垃圾收集器 10](#_Toc456643485)

[20. 知道哪些JVM的命令行工具 11](#_Toc456643486)

[JVM命令行工具 11](#_Toc456643487)

[JDK的可视化工具 12](#_Toc456643488)

[21. 是否了解Java class文件的格式、阅读方法 12](#_Toc456643489)

[22. Java内存模型 13](#_Toc456643490)

[23. 现在JVM中有一个线程挂起了，如何用工具查出原因？ 14](#_Toc456643491)

[24. 用什么工具调试程序？JConsole，用过吗？ 14](#_Toc456643492)

[25. 通过日志看到，系统的请求数目很少，每个请求资源也不多，但是系统堆内存占用非常高，可能出现了什么问题？怎样定位问题在哪里？ 14](#_Toc456643493)

[26. JVM监控和调优的一般步骤 14](#_Toc456643494)

[27. JVM优化的原则、目的、需要做的事情 15](#_Toc456643495)

[28. 经常遇到的异常有哪些？异常的原因？异常StackOverflowStack和OutOfMemory(OOM)怎样发生？ 15](#_Toc456643496)

[29. JVM参数中的server模式和client模式 16](#_Toc456643497)

字长是CPU的主要技术指标之一，指的是CPU一次能并行处理的二进制位数，字长总是8的整数倍，通常PC机的字长为16位（早期），32位，64位。

Java里天生可以动态扩展的语言特性就是依赖运行期动态加载和动态连接的特点实现的。

1. JVM内存管理的机制是什么？(待进一步确定)

Java技术体系中所提倡的自动内存管理最终可以归结为自动化地解决了给对象分配内存空间以及回收分配给对象的内存。

Java虚拟机在执行java程序的过程中会把它所管理的内存划分为若干个不同的数据区域：程序计数器、Java虚拟机栈、本地方法栈、Java堆和方法区。这些区域都有各自的用途，以及创建和销毁的时间，有的区域随着虚拟机进程的启动而启动，有的区域则是随着进程的启动和结束而建立和销毁。

对象的内存分配，往大方向讲，就是在堆上分配，该对象主要分配在新生代的Eden区上，如果启动了本地线程分配缓冲，将按线程优先在TLAB上分配。少数情况下也可能会直接分配在老年代中，分配的规则不是百分百固定的，其细节取决于当前使用的是哪一种垃圾收集器组合，还有虚拟机与内存相关的参数的设置。

1. JVM实例

每当一个java程序运行时，都会有一个对应的JVM实例，只有当程序运行结束后，这个JVM才会退出。JVM实例通过调用类的main()方法来启动一个Java程序，而这个main()方法必须是公有的、静态的且返回值为void的方法，该方法接受一个字符串数组的参数，只有同时满足这些条件才可以作为程序的入口方法。

1. JVM运行时结构

java 运行时数据区域有程序计数器、Java虚拟机栈、本地方法栈、Java堆和方法区。其中前三个线程私有，后面两个是线程间共享。程序计数器、虚拟机栈、本地方法栈是随线程而生的就生，线程灭而灭。

① 程序计数器

当前线程所执行的字节码的行号指示器。

Pc(Program Counter Register)，正在执行的指令，当正在执行的是native方法是计数器为空。

②Java虚拟机栈

存放基本数据类型和引用变量。为执行java方法服务，描述的是Java方法执行的内存模型。

栈内存的管理是通过压栈和弹栈来实现的，以栈帧(每个方法开始运行时都会创建一个栈帧，栈帧是用于存储局部变量表、操作数栈、动态链接、方法出口等信息)为单位来进行管理方法之间的调用关系，当有方法调用时，会通过压栈方式进行创建新的栈帧，当方法调用结束时会通过弹栈的操作释放栈帧。

③本地方法栈

为虚拟机使用到的native本地方法服务。

④Java堆

几乎所有的对象实例以及数组都在堆上分配。线程之间共享堆内存，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例。

Java堆是垃圾收集器管理的主要区域，因此很多时候也被称为是”GC”堆。从内存分配的角度看，线程共享的Java堆可能划分出多个线程私有的分配缓冲区(Thread Local Allocation Buffer,TLAB)。从内存回收的角度来看，由于现在收集器基本都采用分代收集，所以Java堆还可以细分为：新生代和老年代；再细致一点新生代可以分为Eden空间、From Survivor空间、To Survivor空间等。不过无论如何划分，都与存放的内容无关，无论哪个区域，存放的都仍然是对象实例。

在实现时，堆可以实现成固定大小的，也可以实现成可以扩展的(当前主流虚拟机中都是按照可扩展实现的)。Java堆在物理上不要求存储在连续的内存空间中。

⑤方法区

线程共享，用于存储已被虚拟机加载的类信息、常量、静态变量等。有时方法区也被称作永久代(但是本质上两者并不等价，仅仅因为是HotSpot虚拟机的设计团队选择把GC分代收集扩展至方法区，或者说使用永久代来实现方法区而已)，Java虚拟机规范对方法区的限制非常宽松，除了和Java堆一样不需要连续的内存和可以选择固定大小或者可扩展外，还可以选择不实现垃圾收集。方法区的内存回收目标主要是针对常量池的回收和对类型的卸载。

运行时常量池是方法区的一部分，用于存放编译期生成的各种字面量和符号引用，这些内容在类加载后进入方法区的运行时常量池中存放。当然并非class文件常量池中的内容才能进入运行时常量池，在运行期间也可将新的常量放入运行时常量池中，如String的intern方法。

1. 直接内存

直接内存不属于JVM运行时数据区的一部分，它是在NIO类中引入一种基于通道和缓冲区的IO方式，它可以使用Native函数库直接分配堆外内存 ，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。

直接内存的分配不会受到Java堆大小的限制，但是会受到本机内存大小的限制，所以也有可能抛出OutOfMemoryError异常。

1. JVM运行时结构与异常

① 程序计数器

此区域是唯一一个在Java虚拟机中没有规定任何OutOfMemory情况的区域。

②Java虚拟机栈

如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常。如果虚拟机动态扩展的时候没有申请到足够的内存则会抛出OutOfMemory异常。

③本地方法栈

与Java虚拟机栈一样，本区域也是会抛出StackOverflowStack和OutOfMemory两种异常。

④Java堆

如果堆中的内存不够为实例分配内存并且也扩展失败了，就会抛出OutOfMemory异常。

⑤方法区

当方法区无法满足内存分配需求时，将抛出OutOfMemory异常。

1. 栈和堆的区别

① 栈中存放的是基本数据类型和引用变量，为执行java方法服务。堆中存放的是所有的对象和数组实例。

② 栈是线程私有的，描述的是方法执行的内存模型；堆是线程共享的。

③ 栈主要是用来执行程序的，堆主要用来存放对象的。

④ 栈相对于堆来说，栈的存取速度更快 ，但栈的大小和生存期必须是确定的，因此缺乏一定的灵活性。但堆却可以在运行时动态地分配内存，生存期不需要提前告诉编译器，但这也导致了其存取速度的缓慢。

1. 堆内存的区域有哪些划分? JVM的堆内存是什么样的结构?

其实这两个问题是一个问题。Java堆是Java虚拟机所管理的内存中最大的一块。

Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建，此内存的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。

Java堆是垃圾收集器管理的主要区域，因此很多时候也被称为是”GC”堆。从内存分配的角度看，线程共享的Java堆可能划分出多个线程私有的分配缓冲区(Thread Local Allocation Buffer,TLAB)。从内存回收的角度来看，由于现在收集器基本都采用分代收集，所以Java堆还可以细分为：新生代和老年代；再细致一点新生代可以分为Eden空间、From Survivor空间、To Survivor空间等。不过无论如何划分，都与存放的内容无关，无论哪个区域，存放的都仍然是对象实例。进一步划分的目的是为了更好地回收和分配内存。

新生代采用复制的方式收集内存，将内存分为一块较大的Eden区和两个较小的Survivor区。新建对象总是在Eden区，当Eden区已满，就触发一次young GC，将还存活的对象复制到From Survivor空间。这样Eden区都是未被使用的空间，可供继续创建对象，当Eden区再次被使用满，就会触发一次young GC，当回收时，将Eden区和Survivor区中还存活的对象一次性地复制到另外一个Survivor区上。后面就是每次使用Eden区和一个survivor区，当回收时，将Eden区和Survivor区中还存活的对象一次性地复制到另外一个Survivor区上。如果超过某个阈值对象还未被释放，则将该对象复制到老年代。

Eden区存新分配内存的对象，Survivor区存至少经历了一次垃圾回收，并得以幸存的对象。

老年代则存放年长的对象(存在时间较长，经过垃圾回收次数较多的对象)。

持久代(方法区)。

1. 垃圾收集算法有哪些？
2. 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。

主要的不足：① 效率问题：标记和清除的效率都不高；②空间问题：标记清除后，空间内会有大量不连续的内存碎片，空间碎片太多会导致以后在程序运行过程中需要分配较大对象时，无法找到足够大的内存而不得不提前触发一次垃圾回收操作。

1. 复制算法

将可用的内存按容量大小分为两块大小相等的两块，每次只使用其中的一块。当某一块的内存使用完了，就把还存活的对象移到另一块内存上，然后把当前这块做清理掉。

优点：实现简单，运行高效。

缺点：代价太大，将原来的内存缩小为原来的一半。

1. 标记-整理算法

标记过程与”标记-清除算法“一样，但是后续步骤不是直接对可回收的对象进行清理，而是把所有存活的对象都移到一端，然后直接清理掉端边界以外的内存。

1. 分代收集算法

当前商业虚拟机的垃圾收集都采用”分代收集”(Generational Collection)，根据对象存活周期的不同将内存划分为几块。一般是把堆分为新生代和老年代，在新生代采用复制算法回收内存，老年代采用标记-整理或标记-清除算法。算法在运行的过程中优先收集处于新生代的对象，如果一个对象经过多次收集还存活，那么就可以把这个对象移到高一级的堆里，减少对其扫描的次数。

1. JVM中GC停顿是为什么？

判断对象是否存活的可达性分析对时间的敏感还体现在GC停顿上，因为可达性分析工作必须在一个能确保一致性的快照中进行，这里的“一致性”的意思是指在整个分析期间整个执行系统看起来就像被冻结在某个时间点上，不可以出现分析过程中对象引用关系还在不断变化的情况。不然的话可达性分析的结果就无法得到保证，这是导致GC进行时必须停顿所有Java执行线程的一个重要原因。

1. 垃圾回收是什么?

概念：在Java语言中，垃圾回收(Garbage Collection，GC)是一个非常重要的概念，它的主要作用是回收程序中不再被使用的内存，Java提供的GC功能可以自动监测对象是否已经超过作用域从而达到自动回收内存的目的。即垃圾回收的是无任何引用的对象占据的内存空间而不是对象本身。

任务：Java语言提供了垃圾回收器来自动检测对象的作用域，可自动地把不再被使用的存储空间释放掉。具体而言，垃圾回收器主要负责三个事情①分配内存；②保证不再被使用的内存被释放掉；③保证被引用的对象不会被回收。

方式：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。

1. 垃圾回收的两面性？

好的一面：垃圾回收器的存在一方面把开发人员从释放内存的复杂工作中解脱出来，提高了开发人员的生产效率；另一方面，对开发人员屏蔽了释放内存的方法，可以避免因开发人员错误地操作内存而导致应用程序的崩溃，保证了程序的稳定性。

不好的一面：但是垃圾回收也带来了问题，为了实现垃圾回收，垃圾回收器必须跟踪内存的使用情况，释放没用的对象，在完成内存的释放后还需要处理堆中的碎片，这些操作必定会增加JVM的负担，从而降低程序的执行效率。

1. 为什么要有垃圾回收?

在程序开发过程中如果忘记或者错误的释放内存，往往会导致程序运行不正常甚至是导致程序崩溃。为了减轻程序开发人员的工作和保证程序的安全与稳定，java语言提供了垃圾回收器来自动检测对象的作用域，可自动地回收不可能再被引用的对象所占用的内存。

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可以屏蔽掉显式的垃圾回收调用。

1. 什么时候运行垃圾回收？(待进一步确认)

垃圾回收可能发生的时间是堆可用空间不足或CPU空闲。

对于HotSpot虚拟机，用OopMap数据结构存储程序中哪些地方存放着对象引用的信息。程序执行时并不是在所有地方都能停顿下来开始GC，只有到达安全点时才能暂停。其中安全点指的是记录OopMap数据结构的位置。

安全点的选择基本上是以程序“是否能让程序长时间执行的特征”为标准进行选定的。“长时间执行”的最明显的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，所有具有这些功能的指令才会产生safepoint。对于safepoint，另一个需要考虑的问题就是如何在GC发生时让所有线程都“跑”到最近的安全点上再停顿下来。有两种方案可以选择：抢占式中断和主动式中断。

当线程处于sleep状态或者blocked状态时，这时候线程可能无法响应JVM的中断请求，“走到”安全的地方中断挂起。对于这种情况，就需要安全区域来解决。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，在这个区域的任何地方开始GC都是安全的。因此也可以把安全区域看做被扩展了的安全点。在线程执行到Safe Region中的代码时，首先标识自己已经进入到了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，要检查系统是否已经完成了根结点的枚举(或者整个GC过程)，如果完成了，那线程就继续执行，否则它就继续等待直到收到可以安全离开Safe Region的信号为止。

总之，程序并非是时时刻刻都可以去执行GC操作，只有程序运行到安全区域的时候才可以发起GC的操作。

1. Java类加载机制是什么

### ① 类加载机制的概念

类加载机制就是把class文件变成虚拟机可以直接使用的java类型的过程。具体过程是从class文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型。

### ②类加载的过程

类从被加载到虚拟机内存中开始，到卸载出内存位置，整个生命周期包括：加载、验证、准备、解析、初始化、使用、卸载。

其中类加载过程包括加载、链接、初始化。

|  |  |  |
| --- | --- | --- |
| 加载 | | 由类加载器完成，做的事情有3个：  ①根据权限定名获取此类的二进制字节流；  ②将这个字节流代表的静态存储结构转化为方法区的运行时数据结构；③在内存中生成一个代表这个类的Class对象，作为方法区这个类的各种数据的访问入口。 |
| 链接 | 验证 | 检查待加载的class文件的正确性，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。格式检查（是否是class文件、） 元数据检查（是否继承了final的类，是否实现了接口中的所有方法） 符号引用验证（访问性 private protected） |
| 准备 | 在方法区内为类中的静态变量分配存储空间。  准备阶段是正式为类变量(被static修饰的变量)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。 |
| 解析 | 将虚拟机常量池中的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，只要使用时可以无歧义地定位到目标即可。符号引用和虚拟机实现的内存布局无关。  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。直接引用和虚拟机实现的内存布局有关。 |
| 初始化 | | 如果该类具有超类，则对其初始化，执行静态初始化变量和静态初始化代码块。 |

### ③类加载的分类

类加载的方式分为隐式加载与显示加载两种。隐式加载指的是程序使用new关键字创建对象，则会隐式地调用类的加载器把对应的类加载到JVM中。显示加载指的是直接通过使用class.forName()方法来把所需的类加载到JVM中。

在Java语言中，每个类或接口都对应一个class文件，这些文件可以看成是一个个可以被动态加载的单元，因此当只有类被修改时，只需要重新编译变化的类即可，而不需要重新编译所有文件，因此加快了编译速度。

### ④类加载的特点

在java语言中，类的加载是动态的，当程序启动时只把需要的类加载到JVM中，其他类只有被使用的时候才会被加载到虚拟机中。好处是可以加快加载的速度，而且可以节约运行时对内存的开销。

### ⑤类加载器

虚拟机设计团队把类加载阶段的“通过一个类的权限定名来获取描述此类的二进制字节流”这个动作放到Java虚拟机外部去实现，以便让应用程序自己去决定如何加载所需要的类，实现这个动作的代码模块称为“类加载器”。

类加载器的双亲委派模型在JDK1.2期间被引入并被广泛应用于之后几乎所有的java程序中，但它并不是一个强制性的约束模型，而是java设计者推荐给开发者的一种类加载器实现方式。

对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在Java虚拟机中的唯一性 ，每一个类加载器，都拥有一个独立的类名称空间。通俗地讲就是说：比较两个类是否相等，只有在这两个类是由同一个类加载器加载的前提下才有意义，否则，即使这两个类来源于同一个class文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那这两个类就必定不相等。

### ⑥双亲委派模型

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，它首先不会自己去尝试加载这个类，而是会把这个加载的请求委派给父类加载器去完成，每一个层次的类加载器都是这么做，只有当父类加载器无法完成类的加载工作时，子类加载器才会尝试自己去加载这个类。

使用双亲委派模型来组织类加载器之间的关系的一个好处是就是Java类随着它的加载器一起具备了一种带有优先级的层次关系，保证了java程序的稳定性。



注意：在深入理解Java虚拟机P231中提到，在双亲委派模型中的父子关系一般不会以继承的关系来实现，而是都使用组合关系来复用父加载器的代码。

从Java虚拟机的角度看，只存在两种不同的类加载器：①启动类加载器，C++语言实现，是虚拟机自身的一部分；②所有的其他的类加载器，Java语言实现，独立于虚拟机外部，并且全部都继承自抽象类java.lang.ClassLoader.

从Java开发人员的角度来看，类加载器可以划分为启动类加载器、扩展类加载器和应用程序类加载器。

启动类加载器(Bootstrap ClassLoader)：这个类主要负责将存放在<JAVA\_HOME>\lib目录中的，或者被-Xbootclasspath参数所指定的路径中，并且可以被虚拟机识别的(如rt.jar)类库加载到虚拟机内存中。由于引导类加载器涉及到虚拟机本地实现细节，开发者无法直接获取到启动类加载器的引用，所以不允许直接通过引用进行操作。

标准扩展(Extension)类加载器：是由Sun的ExtClassLoader (sun.misc.Launcher $ExtClassLoader)实现的，它负责将<JAVA\_HOME>\lib\ext或者由系统变量java.ext.dirs指定位置中的类库加载到内存中，开发者可以直接使用标准扩展类加载器。

应用程序类加载器(Application ClassLoader)：这个类加载器由sun.misc.Launcher $ApplicationClassLoader实现。由于这个类加载器是ClassLoader中的getSystemClassLoader()方法的返回值，所以一般也被称作是系统类加载器。它负责加载用户类路径(ClassPath)上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

1. Java内存泄露和内存溢出啥区别？
   1. **Java内存泄露**

内存泄露是指一个不再被程序使用的对象或变量还在内存中占用空间。在java语言中，判断一个内存空间是否符合垃圾回收的标准有两个：第一，给对象赋予了空值null，以后再也没有不会被使用；第二，给对象赋予了新值，重新分配了内存空间。

一般来说，内存泄露主要有两种情况：①在堆中申请的空间没有被释放；②对象已不再使用但是还仍然在内存中保留着。垃圾回收机制的引入可以有效地解决第一种情况，但是对第二种情况却没有办法解决，因此java语言中内存泄露主要指的是第二种情况。

在Java中，内存泄漏就是存在一些被分配的对象，这些对象有下面两个特点，首先，这些对象是可达的，即在有向图中，存在通路可以与其相连；其次，这些对象是无用的，即程序以后不会再使用这些对象。如果对象满足这两个条件，这些对象就可以判定为Java中的内存泄漏，这些对象不会被GC所回收，然而它却占用内存。

在Java语言中，容易引起内存泄露的原因有很多，主要可以分为以下几种。

① 静态集合类。例如HashMap和Vector，如果这些容器是静态的，那么它们的声明周期与程序一样，在程序结束之前这些容器所占的空间将得不到释放，从而造成内存泄露；

② 各种连接。比如Connection、Statement、ResultSet等如果使用之后不显示地关闭，会造成大量的对象无法回收，造成内存泄露；

③ 监听器。在java中，往往一个程序使用多个监听器，但是在释放对象的时候却没有删除相应的监听器对象就会导致内存泄露；

④ 变量不合理的作用域。如果一个变量定义的作用范围大于其使用范围 ，就有可能造成内存泄露，另一个方面如果没有及时地把一个对象设置为null，也有可能导致内存泄露；

⑤ 单例模式可能会造成内存泄露。如果以静态的方式存储单例对象的话，那么它在JVM的整个生命周期中都存在，就会导致内存泄露。

* 1. **Java内存溢出**

内存溢出是指程序要求的内存，超出了系统所能分配的范围，从而发生溢出。

1. 怎样判断对象是否还活着?

主要有引用计数法和可达性分析两种方法来判断对象是否还在存活。《深入理解Java虚拟机》P64

① 引用计数法

每个对象都有一个引用计数器，每当该对象被引用时则计数器加1，当某个引用失效时，则引用计数器减1。任何时刻引用计数器的值为0的对象就是不可能再被引用的。

缺点是无法解决循环引用的问题，现在主流的java虚拟机没有采用引用计数法来管理内存。

② 可达性分析算法

通过一系列称为”GC Roots“的对象作为起始点，从这些节点开始向下搜索与其相连的对象，搜索过程中经过的路径称为引用链(Reference Chain)，当一个对象到GC Roots没有任何引用链时，则证明该对象是不可用的。

在主流的商用程序语言(Java C#)的主流实现中，都是通过可达性分析算法来判定对象是否存活的。

在可达性分析算法中不可达的对象，也并非是”非死不可的“，要真正宣告一个对象是否死亡，需要经历两次被标记过程。第一次标记判断是否有必要执行finalize()方法，将有必要执行的对象放到F-Queue的队列之中；稍后GC会对F-Queue进行第二次标记，finalize()方法是对象逃脱死亡命运的最后一次机会，如果对象在finalize()方法中成功地与引用链上的任何一个对象相关联，那么在第二次标记时会把这个对象彻底移除”即将回收“的集合。

finalize()方法是Object类的一个方法，在垃圾回收器执行时会调用被回收对象的finalize()方法，可以覆盖此方法来实现对其他资源的回收，比如关闭文件等。需要注意的是，一旦垃圾回收器准备好释放对象占用的空间时，将首先调用其finalize()方法，并且在下一次垃圾回收动作发生时，才会真正回收对象占用的内存。

1. GC roots包含哪些对象？

在Java语言中，可作为GC Roots的对象主要包括以下四个方面。

① 虚拟机栈中的引用对象；

② 本地方法栈中本地方法引用的对象；

③ 方法区中类静态属性引用的对象；

④ 方法区内常量引用的对象。

1. 引用的类型

强引用、软引用、弱引用、虚引用

强引用：如果一个对象具有强引用，那垃圾回收器当内存不够时，宁愿抛出outofmemory错误也不会程序终止, 类似于Object obj = new Object().

软引用：如果一个对象具有软引用，如内存空间足够，垃圾回收器就不会回收它，如果内存不足，就会回收这些对象的内存（可以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）

Soft Reference的主要特点是据有较强的引用功能。只有当内存不够的时候，才进行回收这类内存，因此在内存足够的时候，它们通常不被回收。另外，这些引用对象还能保证在Java抛出OutOfMemory 异常之前，被设置为null。它可以用于实现一些常用图片的缓存，实现Cache的功能，保证最大限度的使用内存而不引起OutOfMemory。

弱引用：在系统GC时，只要发现弱引用，不管堆空间是否足够，都会讲对象进行回收。

虚引用：随时都有可能被垃圾回收器回收。就是一旦有gc扫过它就会清除对象空间。作用在于跟踪垃圾回收过程，清理被销毁对象的相关资源。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。

1. 回收方法区

方法区(或者HotSpot虚拟机中的永久代)的垃圾收集主要集中在两部分内容：废弃常量和无用的类。

废弃常量：没有其他地方引用的常量；

无用的类：① 该类所有的对象实例都已经被回收；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。（满足这三个条件的无用类可以被回收，但不一定被回收，最后是否回收还要看其他相关参数的配置）

1. 了解哪些垃圾收集器

① Serial收集器：一个单线程的收集器，但它“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。新生代收集器、采用复制算法。

优点：与其他收集器的单线程比，简单而高效。

Serial收集器对于运行在Client模式下的虚拟机来说是一个很好的选择。

② ParNew收集器：其实就是Serival收集器的多线程版本。新生代收集器、采用复制算法。 适用于运行在server模式下的虚拟机中首选的新生代收集器。

③ Parallel Scavenge收集器：新生代收集器、采用复制算法、并行的多线程收集。

主要适合在后台运算而不需要太多交互的任务，注重的是吞吐量。

如果用户对于收集器运作不是很了解，手工优化困难的时候，使用Parallel Scavenge收集器配合自适应调节策略，把内存管理的调节任务交给虚拟机去完成将是一个不错的选择。

自适应调节策略也是Parallel Scavenge收集器与ParNew收集器的一个不同。

④SerialOld收集器：是Serial的老年代版本，同样是一个单线程收集器，使用“标记-整理”算法。主要给在Client模式下的虚拟机使用。如果是在Server模式下则主要有两大用途，一是在JDK1.5以及之前的版本中与Parallel Scavenge 收集器搭配使用，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：是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。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收集器和Parallel Old收集器。

⑥ CMS(Conourrent Mark Sweep )收集器：以获取最短回收停顿时间为目标的收集器。基于”标记-清除”算法实现。

整个过程分为四步：初始标记、并发标记、重新标记、并发清除。

优点：并发收集、低停顿。

缺点：对CPU资源敏感；无法处理浮动垃圾；收集结束会有大量碎片。

⑦ G1收集器：面向服务端应用的垃圾收集器，标记整理。与其他GC收集器相比：G1具有如下特点：并发与并行；分代收集；空间整合；可预测的停顿。

运作的过程大致分为四个步骤：初始标记；并发标记；最终标记；筛选回收。

备注：并发(Parallel):指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态；

并行(Concurrent):指用户线程与垃圾收集线程同时执行(但不一定是并行的，可能会交替执行 )，用户程序在继续运行，而垃圾收集程序运行于另一个CPU上。

1. 知道哪些JVM的命令行工具

JVM命令行工具

① jps：虚拟机进程状况工具；

只运行jps输出的是当前运行的java进程(Java程序的进程ID，Main函数)；

运行jps –q只输出进程ID，而不输出类的名称；

jps –m输出传递给Java进程(主函数)的短名称；

jps –l输出主函数的完整路径；

jps –v输出传递给JVM的参数。

② jstat：虚拟机统计信息监视工具；可以用于观察Java应用程序运行时信息的工具。

jstat可以实时显示本地或远程JVM进程中类装载、内存、垃圾收集、JIT编译等数据（如果要显示远程JVM信息，需要远程主机开启RMI支持）。如果在服务启动时没有指定启动参数-verbose:gc，则可以用jstat实时查看gc情况。

在没有GUI图形界面，只提供了纯文本控制台环境的服务器上，jstat是运行期定位虚拟性能问题的首选工具。

如jstat –gc 2764 250 20表示查询进程2764垃圾收集状况，每250ms查询一次，一共查询20次。后面两个参数省略时表示只查询一次。

命令格式为jstat option vmid interval count.

其中option主要分为3类：类装载、垃圾收集、运行期编译。

|  |  |
| --- | --- |
| -class | 监视类装载、卸载数量、总空间以及类装载耗费的时间。 |
| -gc | 监视java堆状况，包括Eden区、两个survivor区、老年代、永久代等的容量、已用空间、GC时间合计等信息。 |
| -gccapacity | 监视内容与-gc相同，但输出主要关注Java堆各个区域使用到的最大、最小空间 |
| -gcutil | 监视内容与-gc相同，但输出主要关注已经使用的空间占总空间的百分比 |
| -gnew | 监视新生代GC状态 |
| -gold | 监视老年代GC状态 |
| -gcompiler | 输出JIT编译器编译过的方法、耗时等信息。 |

③ jinfo：Java配置信息工具；用于查询当前运行时的JVM属性和参数的值。  
jinfo可以使用如下选项：

   -flag:显示未被显示指定的参数的系统默认值  
   -flag [+|-]name或-flag name=value: 修改部分参数  
   -sysprops:打印虚拟机进程的System.getProperties()  
 命令格式:jinfo [option] pid

④ jmap：java内存映像工具；

用于显示当前Java堆和永久代的详细信息（如当前使用的收集器，当前的空间使用率等）.

jmap –heap:显示java堆的相信信息，如使用哪种收集器、参数配置、分代状态等。

Jmap-dump:生成java堆转储快照。

⑤ jhat：虚拟机堆转储快照分析工具；

用于分析使用jmap生成的dump文件，是JDK自带的工具，使用方法为： jhat -J -Xmx512m [file]

⑥ jstack：java堆栈跟踪工具；

用于生成当前JVM的所有线程快照，线程快照是虚拟机每一条线程正在执行的方法,目的是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等都是导致线程长时间停顿的原因。线程出现停顿的时候通过jstack来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做些什么事情或者是在等待什么资源。

   -F:当正常输出的请求不被响应时，强制输出线程堆栈  
   -l:除堆栈外，显示关于锁的附加信息

   -m:如果调用到本地方法的话，可以显示C/C++的堆栈  
命令格式:jstack [option] vmid

例如jstack –l 3500会显示3500编号的进程的带有锁信息的堆栈信息。

⑦ HSDIS：JIT生成代码反汇编。

JDK的可视化工具

① JConsole

JConsole工具是JDK自带的图形化性能监控工具。通过JConsole工具，可以查看Java应用程序的运行概况，监控堆信息、永久区使用情况、类加载情况。

在JConsole中可以查看堆的详细信息，包括堆的大小、使用率、eden区大小、survivor区大小、永久区大小。

JConsole可以方便地查看系统内的线程信息，并且可以快速定位死锁问题。

JConsole的类页面可以显示系统已经装载的类数量。

JConsole的VM摘要显示了当前Java应用程序的基本信息，如虚拟机类型、虚拟机版本、系统线程信息、操作系统内存信息、堆信息、垃圾回收器信息以及路径信息等。

② VisualVM

1. 是否了解Java class文件的格式、阅读方法

Class文件是一组以8位字节为基础单位的二进制流，各个数据项目严格按照顺序紧凑地排列在Class文件中，中间没有添加任何分隔符，这使得整个Class文件中存储的内容几乎全部是程序运行的必要数据，没有空隙存在，当遇到需要占用8位字节以上空间的数据项时，则会按照高位在前的方式分割成若干个8位字节进行存储。

每个class文件的头4个字节称为魔数，它的唯一作用就是确定这个文件是否为一个能被虚拟机接收的Class文件。紧着着魔数之后的4个字节存储的class文件的版本号，后面是常量池入口。后面是代表访问标志的两个字节，这个表示用于识别一些类或者接口层次的访问信息。再往后就是类索引、父索引与接口索引集合。字段表集合用户描述接口或者类中声明的变量。方法表集合、属性表集合。

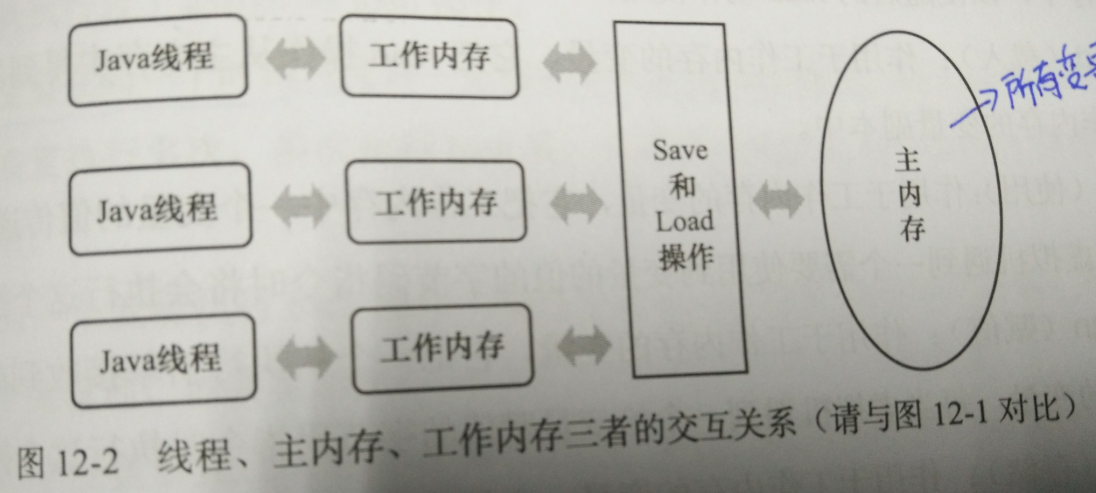
魔数🡪版本号🡪常量池入口🡪访问标志🡪类索引、父索引与接口索引集合

1. Java内存模型

* Java内存模型

Java内存模型的主要目标是定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。

Java内存模型规定了所有的变量都存储在主内存中，但是每个线程还是有自己的工作内存，线程的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都必须在工作内存中进行，而不能直接读写主内存中的变量。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成。



* Volatile变量有两个特性：

① 保证可见性：此变量对于所有线程的可见性，指的是当一个线程修改了这个变量的值，新值对于其他线程来说是立即可见的。

② 禁止指令重排序优化（内存屏障，也会牺牲掉一些性能）。指令重排指的是CPU采用了允许将多条指令不按程序规定的顺序分开发送给各相应电路单元处理，但并不是说指令任意重排，CPU需要能正确处理指令依赖情况以保障程序能得出正确的执行结果。Volatile变量为了保证数据的一致性，不允许使用Volatile变量的程序代码进行指令重排。

* 某个变量定义为Volatile的应用场景

① 运算结果不依赖于变量的当前值或者能够确保只有一个线程修改这个变量的值；

② 变量不需要与其他的状态变量共同参与不变约束。

* Java内存模型对volatile变量定义的特殊规则

① 在工作内存中每次使用变量前都需要从主内存中刷新最新值；

② 每次修改变量的值之后都必须立刻同步到主内存中；

③ 要求volatile修饰的变量不会被指令重新排序。

* 原子性、可见性、有序性

Java的内存模型是围绕着在并发过程中如何处理原子性、可见性和有序性3个特征建立的。Java语言提供了volatile和synchronized两个关键字来保证线程之间操作的有序性。

1. 现在JVM中有一个线程挂起了，如何用工具查出原因？

线程出现停顿的时候通过jstack来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做些什么事情或者是在等待什么资源。用于生成当前JVM的所有线程快照，线程快照是虚拟机内每一条线程正在执行的方法堆栈集合,目的是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等都是导致线程长时间停顿的原因。

1. 用什么工具调试程序？JConsole，用过吗？

JConsole工具是JDK自带的图形化性能监控工具。通过JConsole工具，可以查看Java应用程序的运行概况，监控堆信息、永久区使用情况、类加载情况。

在JConsole中可以查看堆的详细信息，包括堆的大小、使用率、eden区大小、survivor区大小、永久区大小。

JConsole可以方便地查看系统内的线程信息，并且可以快速定位死锁问题。

JConsole的类页面可以显示系统已经装载的类数量。

JConsole的VM摘要显示了当前Java应用程序的基本信息，如虚拟机类型、虚拟机版本、系统线程信息、操作系统内存信息、堆信息、垃圾回收器信息以及路径信息等。

1. 通过日志看到，系统的请求数目很少，每个请求资源也不多，但是系统堆内存占用非常高，可能出现了什么问题？怎样定位问题在哪里？

参见《深入理解Java虚拟机》P52

可能是发生了堆溢出或内存泄露，要解决堆内存异常的情况一般的手段是通过内存映像分析工具(如Eclipse Memory Analyzer)对Dump出来的堆转储快照进行分析，重点是确认内存中的对象是否是有必要的，也就是要先分清楚是内存泄露还是内存溢出。

如果是内存泄露，可进一步通过工具查看泄露对象到GC Roots的引用链。于是就能找到泄露的对象是怎么与GC Roots相连接导致进行垃圾回收时没能回收泄露对象所占的内存，有了泄露对象的信息和GC Roots引用链的信息，就可以准确地定位出泄露代码的位置。

如果没有发出内存泄露，也是说，内存中的对象确实还活着，那就应该去检查虚拟机的堆参数(-Xmx与-Xms)，与物理机器对比看是否还可以进一步扩大，从代码上检查是否存在某些对象声明周期过长、持有状态时间过长等，尝试减少程序运行内存消耗。

1. JVM监控和调优的一般步骤

1，监控GC的状态

使用各种JVM工具，查看当前日志，分析当前JVM参数设置，并且分析当前堆内存快照和gc日志，根据实际的各区域内存划分和GC执行时间，觉得是否进行优化；

2，分析结果，判断虚拟机的配置参数是否需要优化

如果各项参数设置合理，系统没有超时日志出现，GC频率不高，GC耗时不高，那么没有必要进行GC优化；如果GC时间超过1-3秒，或者频繁GC，则必须优化；

注：如果满足下面的指标，则一般不需要进行GC：

Minor GC执行时间不到50ms；

Minor GC执行不频繁，约10秒一次；

Full GC执行时间不到1s；

Full GC执行频率不算频繁，不低于10分钟1次；

3，调整GC类型和内存分配

如果内存分配过大或过小，或者采用的GC收集器比较慢，则应该优先调整这些参数，并且先找1台或几台机器进行beta，然后比较优化过的机器和没有优化的机器的性能对比，并有针对性的做出最后选择；

4，不断的分析和调整

通过不断的试验和试错，分析并找到最合适的参数

5，全面应用参数

如果找到了最合适的参数，则将这些参数应用到所有服务器，并进行后续跟踪。

1. **JVM优化的原则、目的、需要做的事情**

* 优化的原则

一切都是为了这一步，调优，在调优之前，我们需要记住下面的原则：

1. 多数的Java应用不需要在服务器上进行GC优化；
2. 多数导致GC问题的Java应用，都不是因为我们参数设置错误，而是代码问题；
3. 在应用上线之前，先考虑将机器的JVM参数设置到最优（最适合）；
4. 减少创建对象的数量；
5. 减少使用全局变量和大对象；
6. GC优化是到最后不得已才采用的手段；
7. 在实际使用中，分析GC情况优化代码比优化GC参数要多得多；

* GC优化的目的有两个（http://www.360doc.com/content/13/0305/10/15643\_269388816.shtml）：
* 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；
* 需要做的工作
* 减少使用全局变量和大对象；
* 调整新生代的大小到最合适；
* 设置老年代的大小为最合适；
* 选择合适的GC收集器；

1. 经常遇到的异常有哪些？异常的原因？异常StackOverflowStack和OutOfMemory(OOM)怎样发生？

* OutOfMemory

① java.lang.OutOfMemory：Java heap space

首先查看是否是堆溢出或内存泄露。

要解决堆内存异常的情况一般的手段是通过内存映像分析工具(如Eclipse Memory Analyzer)对Dump出来的堆转储快照进行分析，重点是确认内存中的对象是否是有必要的，也就是要先分清楚是内存泄露还是内存溢出。

如果是内存泄露，可进一步通过工具查看泄露对象到GC Roots的引用链。于是就能找到泄露的对象是怎么与GC Roots相连接导致进行垃圾回收时没能回收泄露对象所占的内存，有了泄露对象的信息和GC Roots引用链的信息，就可以准确地定位出泄露代码的位置。

如果没有发出内存泄露，也是说，内存中的对象确实还活着，那就应该去检查虚拟机的堆参数(-Xmx与-Xms)，与物理机器对比看是否还可以进一步扩大，从代码上检查是否存在某些对象声明周期过长、持有状态时间过长等，尝试减少程序运行内存消耗。

② java.lang.OutOfMemory：PermGen space

说明是运行时常量池出现问题。需要扩大方法区来保证动态生成的class可以加载入内存。

③ java.lang.OutOfMemory

直接内存(直接内存并不是虚拟机运行数据区的一部分)可以通过-XX：MaxDirectMemorySize指定 ，如果不指定，则默认与Java堆最大值(-Xmx)一样。

虽然使用DirectByteBuffer分配内存也会抛出内存溢出异常，但是它抛出异常时并没有真正向操作系统申请分配内存，而是通过计算得知内存无法分配，于是手动抛出异常，真正申请分配内存的方法是unsafe.allocateMemory().

由直接内存导致的内存溢出，一个明显的特征是Heap Dump文件中不会看见明显的异常，如果发现OOM之后Dump文件很小，而程序中又直接或间接使用了NIO，那就可以考虑检查一下是不是出现了方法区溢出。

* StackOverflowStack

抛出StackOverflowStack异常是线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常。产生这种内存溢出与栈空间是否足够大并没有任何关系，反而给每个线程的栈分配的内存越大，反而越容易产生内存溢出异常。

出现StackOverflowError异常有时有错误堆栈可以阅读，相对比较容易找到问题的所在。而且，如果使用虚拟机默认参数，在大多数情况下达到1000~2000是完全没有问题的，对于正常的方法调用(包括递归)，这个深度是完全没有问题的。但是如果建立过多线程导致的内存溢出，在不能减少线程数或者更换64位虚拟机的情况下，就只能通过减少最大堆和减少栈容量来换取更多的线程。

以上的信息是总结自《深入理解Java虚拟机》。其他的相关内容可以参考<http://ifeve.com/java-concurrency-9-3-3/>。

1. JVM参数中的server模式和client模式

具体见《java并发编程实战》P31

对于服务器应用程序，无论是开发阶段还是测试阶段，当启动JVM时一定都要指定-server命令行选项。Server模式的JVM比client模式的JVM进行更多的优化，例如将循环中未被修改的变量提升到循环外部，因此在开发模式(client模式的JVM)中能正确运行的代码，可能会在部署环境(server模式的JVM)中运行失败。如下代码：

|  |
| --- |
| volatile boolean asleep;  while(!asleep){  …..  } |

如果在代码中忘记把asleep变量声明为volatile变量，则server模式的JVM会把asleep变量的判断提升到循环体外部(这将导致一个无限循环)，但client模式的JVM不会这么做。在解决开发环境中出现无限循环问题时，解决这个问题的开销远小于解决在应用环境中出现无限循环的开销。