JVM 虚拟机

1. **Java虚拟机**

   Java源文件被编译成能被Java虚拟机执行的字节码文件。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，因为它知道底层硬件平台的指令长度和其他特性。

2. **JRE,JDK,JVM,JIT**

   1. JRE 代表 Java 运行时（Java run-time），是运行 Java 引用所必须的。
   2. JDK 代表 Java 开发工具（Java development kit），是 Java 程序的开 发工具，如 Java编译器，它也包含 JRE。
   3. JVM 代表 Java 虚拟机（Java virtual machine），它的责任是运行 Java 应用。
   4. 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。

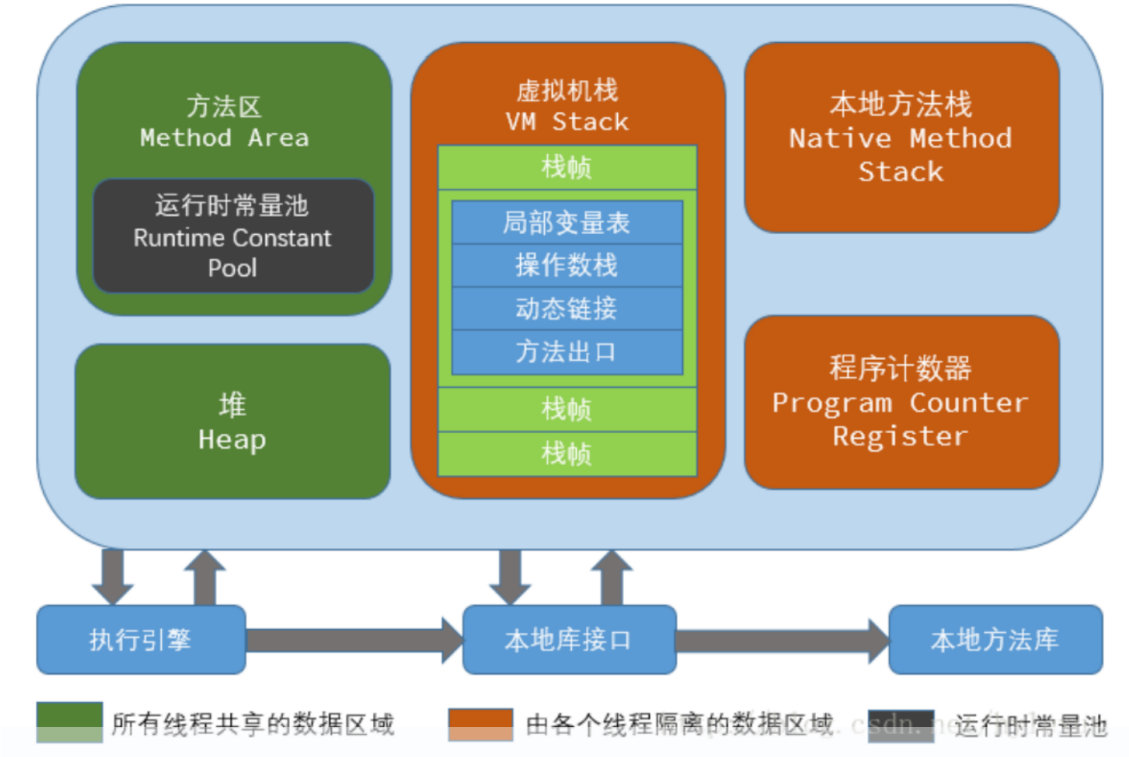
3. **如何判断JVM是32位还是64位？最大堆内存数是多少？int 类型变量的长度是多少?**

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

   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 1.5 GB，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。

   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。

4. **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

1. **栈**：又称方法栈，线程私有的。线程执行方法都会创建一个栈阵，用来存储局部变量表，操作栈，动态链接，方法出口等信息。调用时入栈，方法返回时出栈。

   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。

2. **本地方法栈**：执行Java方法时使用栈，执行Native方法时使用本地方法栈。
3. **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。
4. **堆**：JVM内存管理最大的一块，被线程共享。用来存放对象的实例。当堆没有可用空间时，会抛出OOM异常
5. **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。

5. 栈和堆的区别

栈：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：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。

1. **功能不同**：栈内存：存储局部变量和方法调用 堆：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
2. **共享性不同**：栈：线程私有的 堆：所有线程共享的
3. **异常错误不同**：栈：StackOverFlowError 堆：OutOfMemoryError
4. **空间大小**：栈的空间远远小于堆的

6. Java对象的创建过程

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。
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

7. Java对象的结构

Java对象由三个部分组成：对象头、实例数据、对齐填充。

1. 对象头由两部分组成：
   1. 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）。
   2. 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。
2. 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）
3. 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）

8. 如何判断对象可以被回收

1. 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。
2. 可达性分析（Reachability Analysis）：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。

9. ## JAVA中会存在内存泄漏吗?

   会。自己实现堆载的数据结构时有可能会出现内存泄露，可参看effective java.

10. ## GC 是什么？为什么要有GC?

    GC 是垃圾收集。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 没有提供释放已分配内存的显示操作方法 ，垃圾收集器会自动管理 。要请求垃圾收集，可以调用System.gc() 或 Runtime.getRuntime().gc() ，但 JVM 可以屏蔽掉显示的垃圾回收调用 。

    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。

11. ## 运行时数据 - 堆（Heap)

    是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，也是垃圾回收器进行垃圾收集的最重要的内存区域。

    1. **新生代**（1/3）：是用来存放新生的对象。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。Eden：8/10， From Survivor：1/10， To Survivor：1/10
    2. **老年代**（2/3）：主要存放应用程序中生命周期长的内存对象。 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. **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。

       *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。

12. ## 垃圾回收算法

    1. **标记清除算法（Mark-Sweep）**
    2. **复制算法（Coping）**

       为了解决算法内存碎片化的缺陷而提出的算法。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。

    3. **标记整理算法 （Mark - Compact）**
    4. **分代收集算法**

       根据对象存活的不同生命周期将内存划分为不同的域，老年代和新生代。老年代：每次垃圾回收时只有少量对象需要被回收；新生代：垃圾回收时都有大量垃圾需要被回收。

13. ## Minor GC

    Minor GC 采用复制算法（复制 -> 清空 -> 互换）

    新生代因为每次回收都能发现大批对象已死，只有少量存活，因此选用复制算法。只要付出少量存活对象的复制成本就可以完成收集。

    1. Eden, survivorFrom 复制到survivorTo， 年龄加1

       Eden全部都复制到To区， From中只有年龄加1后到达阈值的直接进入老年区，其余复制进入To区

    2. 清空Eden, SurvivorFrom

    3. SurvivorTo 和 SurvivorFrom 互换

       相当于清空SurvivorTo 以进行下一轮循环

    Minor GC 触发条件：Eden区内存不足

14. ## Full GC

    Full GC 采用标记清除算法

    1. 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。

    Full GC 触发条件：

    1. 调用System.gc时，系统建议执行Full GC, 但是不是必然执行

       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

    2. Permanent Generation方法区空间不足

       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，Permanent 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。

    3. 老年代空间不足

       抛出java.lang.OutOfMemory: Java heap space。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。

    4. 调用Minor GC后进入老年代的平均大小小于老年代的可用内存

       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。

    5. 由Eden区，From 区向To区复制时，对象大小大于To区可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小。

       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

15. **对象分配规则**

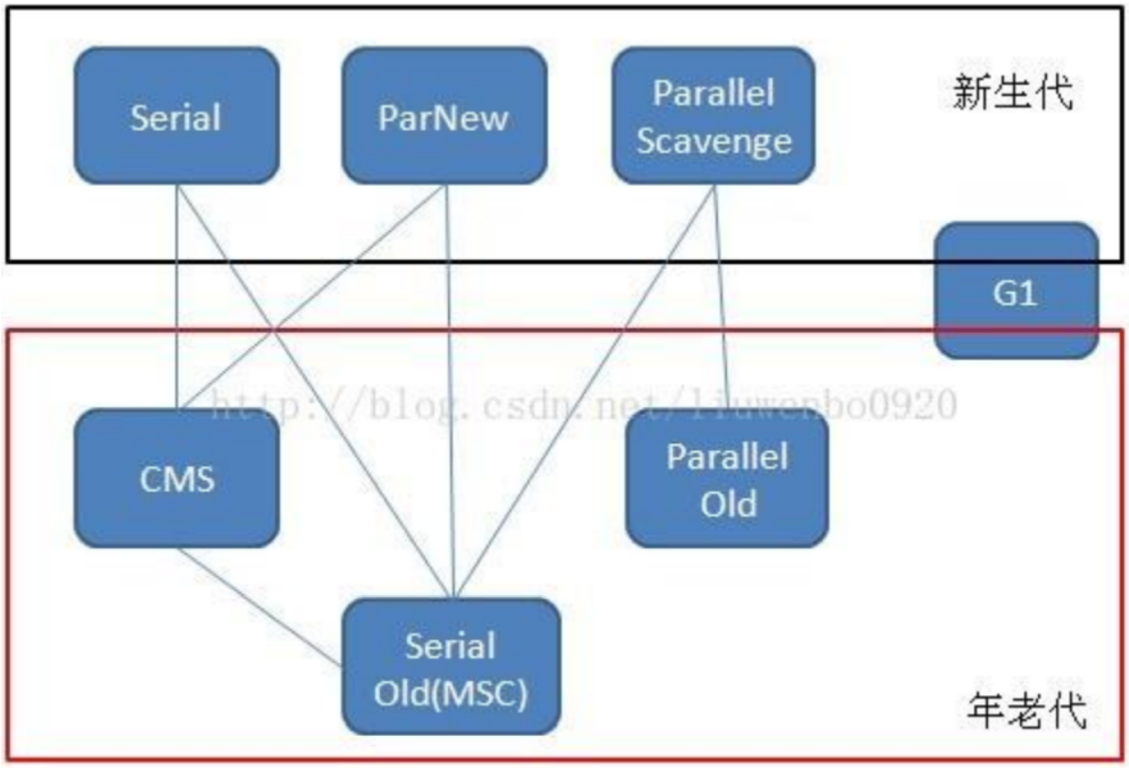
    1. 对象优先分配在Eden区，如果Eden区没有足够的空间时，虚拟机执行一次Minor GC。
    2. 大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。这样做的目的是避免在Eden区和两个Survivor区之间发生大量 的内存拷贝（新生代采用复制算法收集内存）。
    3. 长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了1次Minor GC那么对象会进入Survivor区，之 后每经过一次Minor GC那么对象的年龄加1，知道达到阀值对象进入老年区。
    4. 动态判断对象的年龄。如果Survivor区中相同年龄的所有对象大小的总和大于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Minor GC, 如果false则进行Full GC

16. **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HandlerPromotionFailure这个参数设置的值(true或flase)是否允许担保失败(如果这个值为true，代表着JVM说，我允许在这种条件下尝试执行Minor GC，出了事我负责)。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，将尝试进行一次Minor GC，尽管这次Minor GC是有风险的；如果小于，或者HandlerPromotionFailure为false，那么这次Minor GC将升级为Full GC。

    -XX:+Handle PromotinoFailure 如果允许担保失败，则只会触发Minor GC， 不然则会同时触发Full GC 。

17. **GC 垃圾收集器**



    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

    1. **Serial**：复制 + 单线程。新生代
    2. **ParNew**：Serial + 多线程。新生代
    3. **Parallel Scavenge** :

复制 + 多线程。新生代。 它重点关注的是程序达到一个 可控制的吞吐量（Throughput，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。

4. **Serial Old**: 标记整理 + 单线程。Serial 老年代
5. **Parallel Old**: 标记整理 + 多线程，Parallel Scavenge的老年代版本。如果系统对吞吐量要求比较高，可以优先考虑新生代Parallel Scavenge 和老年代Parallel Old收集器搭配使用。
6. **Concurrent mark sweep(CMS)**:

   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。整个过程分为以下 4 个阶段：

   1. 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。
   2. 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。
   3. 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。
   4. 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
7. **G1（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

18. ## 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。

19. ## 永久代会发生垃圾回收吗?

永久代的垃圾回收主要包括类型的卸载和废弃常量池的回收。

1. 常量回收：当没有对象引用一个常量的时候，该常量即可以被回收。
2. 类型卸载：
   1. 该类型的所有实例都被回收。
   2. 该类型的ClassLoader被回收。
   3. 该类型对应的java.lang.Class没有在任何地方被引用，在任何地方都无法通过反射来实例化一个对象。

20. ## JVM 类加载机制

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。 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。

1. **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）。

2. **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

3. **准备**

   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。

4. **解析**

   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

5. **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。

   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。以下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。

6. **使用**

7. **卸载**

21. 类加载器（ClassLoader）

1. **启动类加载器（Bootstrap）**

   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

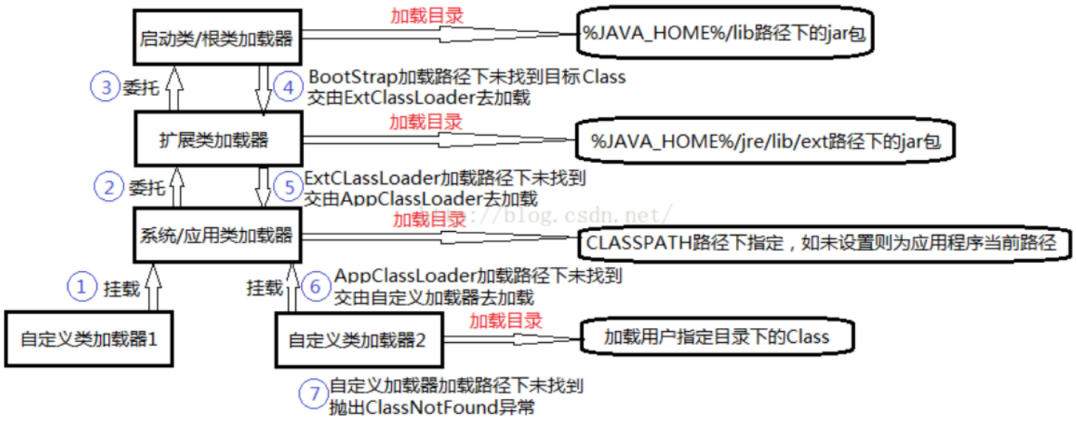
2. **扩展类加载器（Extension）**

   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。

3. **应用程序类加载器 （Application）**

   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实现自定义类加载器（User ClassLoader）

22. 双亲委派



一个类收到了类加载请求，不会尝试自己加载，而是把这个请求委派给父类去完成。父类不能完成，子类才会尝试自己加载。启动类是根类。

好处：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。

23. Java 引用

1. **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。（永远不会被回收）
2. **软引用**：需要用 SoftReference 类来实现，系统内存足够时它不会被回收，不足时才会被回收。通常用在内存敏感的程序。（内存不够时才会被回收）
3. **弱引用**：需要用 WeakReference 类来实现，它比软引用的生存期更短。只要垃圾回收一运行，不管JVM的内存空间足够，总会回收。（垃圾回收机制一运行就会被回收）
4. **虚引用**：需要用 PhantomRefrence来实现，不能单独使用，必须和引用队列联合使用。主要作用：跟踪对象被垃圾回收的状态。

24. 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偶尔度可管理，OSGI技术提供一种面向服务的架构，它能使这些组件动态的发现对方。

模块化编程与热插拔：OSGI旨在为实现Java程序的模块化变成提供基础条件，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和架构，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。它在提供强大功能的同时，也引用了额外的复杂度，因为它不遵守了类加载的双亲委托模型。

25. JVM调优命令

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。

26. 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：MAT(Memory Analyzer Tool)、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。

27. JVM性能调优

1. 设定堆内存大小

   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。Runtime.freeMemory() 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。

2. -Xmx: 堆内存最大限制

3. -XX:NewSize 新生代大小。设定新生代大小：新生代太小会有大量对象涌入老年代。

4. -XX:NewRatio 新生代和老年代占比

5. --XX:Survivor Ratio 伊甸园空间和幸存者空间的占比

6. 设定垃圾回收器：新生代：-XX:+UserParNewGC 老生代：-XX:+UseConcMarkSweepGC

28. 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。