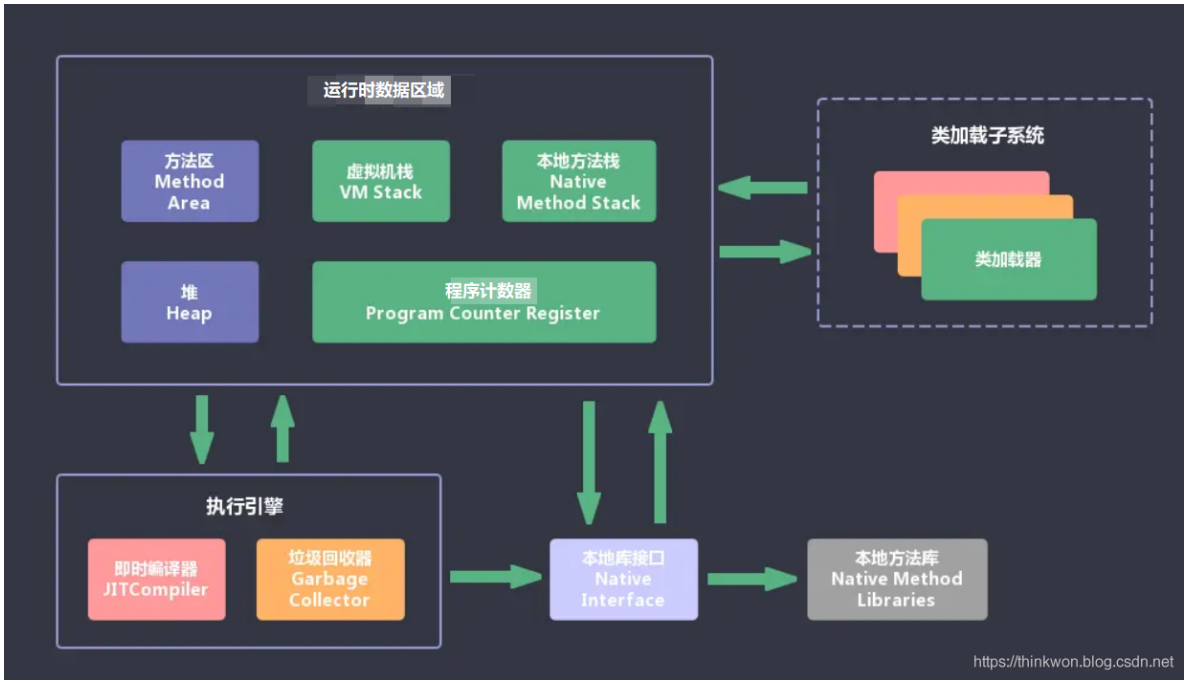
JVM

194.说一下 jvm 的主要组成部分？及其作用?

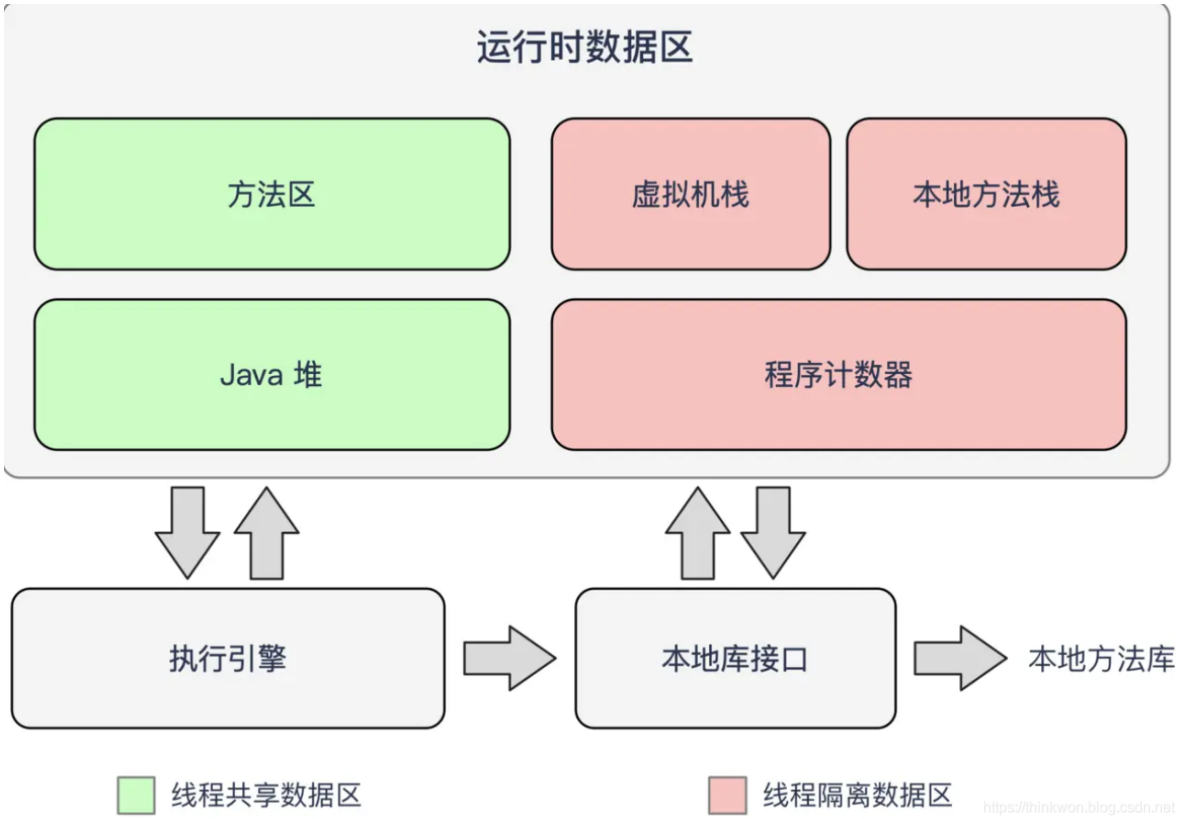
1.类加载器（Class Loader）：加载类文件到内存。Class loader只管加载，只要符合文件结构就加载，至于能否运行，它不负责，那是有Exectution Engine 负责的。
2.执行引擎（Execution Engine）：也叫解释器，负责解释命令，交由操作系统执行。
3.本地库接口（Native Interface）：本地接口的作用是融合不同的语言为java所用
4.运行时数据区（Runtime Data Area）：堆栈方法区那些

作用 ：首先通过编译器把 Java 代码转换成字节码，类加载器（ClassLoader）再把字节码加载到内存中，将其放在运行时数据区（Runtime data area）的方法区内，而字节码文件只是 JVM 的一套指令集规范，并不能直接交给底层操作系统去执行，因此需要特定的命令解析器执行引擎（Execution Engine），将字节码翻译成底层系统指令，再交由 CPU 去执行，而这个过程中需要调用其他语言的本地库接口（Native Interface）来实现整个程序的功能。



195.说一下 jvm 运行时数据区?

（1）堆。堆是java对象的存储区域，任何用new字段分配的java对象实例和数组，都被分配在堆上，java堆可用-Xms和-Xmx进行内存控制，jdk1.7以后，运行时常量池从方法区移到了堆上。
（2）方法区：用于存储已被虚拟机加载的类信息，常量，静态变量，即时编译器编译后的代码等数据。
误区：方法区不等于永生代
很多人原因把方法区称作“永久代”（Permanent Generation），本质上两者并不等价，只是HotSpot虚拟机垃圾回收器团队把GC分代收集扩展到了方法区，或者说是用来永久代来实现方法区而已，这样能省去专门为方法区编写内存管理的代码，但是在Jdk8也移除了“永久代”，使用Native Memory来实现方法区。
（3）虚拟机栈：虚拟机栈中执行每个方法的时候，都会创建一个栈桢用于存储局部变量表，操作数栈，动态链接，方法出口等信息。
（4）本地方法栈：与虚拟机栈发挥的作用相似，相比于虚拟机栈为Java方法服务，本地方法栈为虚拟机使用的Native方法服务，执行每个本地方法的时候，都会创建一个栈帧用于存储局部变量表，操作数栈，动态链接，方法出口等信息。
（5）程序计数器。指示Java虚拟机下一条需要执行的字节码指令。



196.说一下堆栈的区别?

栈内存:栈内存首先是一片内存区域，存储的都是局部变量，凡是定义在方法中的都是局部变量（方法外的是全局变量），for循环内部定义的也是局部变量，是先加载函数才能进行局部变量的定义，所以方法先进栈，然后再定义变量，变量有自己的作用域，一旦离开作用域，变量就会被释放。栈内存的更新速度很快，因为局部变量的生命周期都很短。

堆内存:存储的是数组和对象（其实数组就是对象），凡是new建立的都是在堆中，堆中存放的都是实体（对象），实体用于封装数据，而且是封装多个（实体的多个属性），如果一个数据消失，这个实体也没有消失，还可以用，所以堆是不会随时释放的，但是栈不一样，栈里存放的都是单个变量，变量被释放了，那就没有了。堆里的实体虽然不会被释放，但是会被当成垃圾，Java有垃圾回收机制不定时的收取。

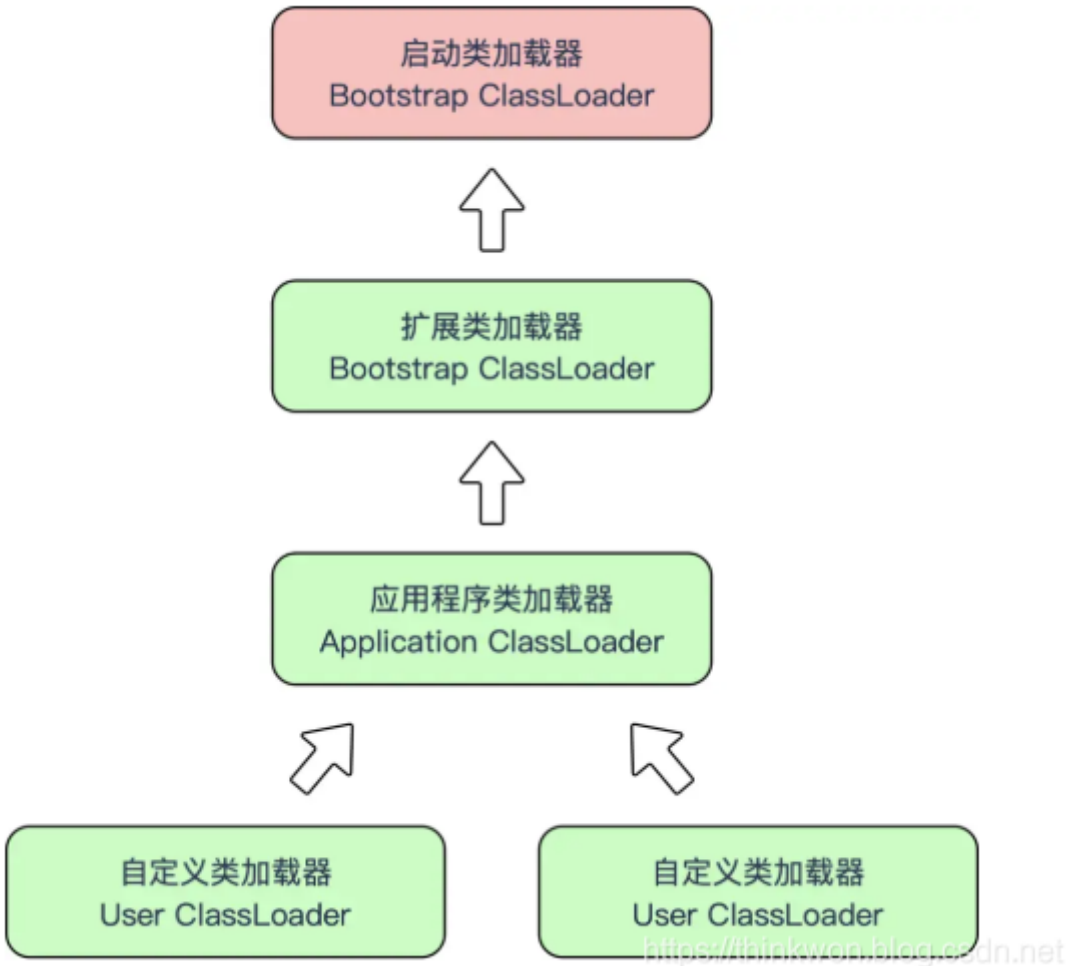
197.队列和栈是什么？有什么区别?

队列和栈都是被用来预存数据的。
（1）操作的名称不同。队列的插入称为入队，队列的删除称为出队。栈的插入称为进栈，栈的删除称为出栈。
（2）可操作的方式不同。队列是在队尾入队，队头出队，即两边都可操作。而栈的进栈和出栈都是在栈顶进行的，无法对栈底直接进行操作。
（3）操作的方法不同。队列是先进先出（FIFO），即队列的修改是依先进先出的原则进行的。新来的成员总是加入队尾（不能从中间插入），每次离开的成员总是队列头上（不允许中途离队）。而栈为后进先出（LIFO）,即每次删除（出栈）的总是当前栈中最新的元素，即最后插入（进栈）的元素，而最先插入的被放在栈的底部，要到最后才能删除。队列允许先进先出检查元素，但也有例外情况。Deque接口允许从两端检索元素。

198.什么是双亲委派模型?

当需要加载一个类的时候，子类加载器并不会马上去加载，而是依次去请求父类加载器加载，一直往上请求到最高类加载器：启动类加载器。当启动类加载器加载不了的时候，依次往下让子类加载器进行加载。当达到最底下的时候，如果还是加载不到该类，就会出现ClassNotFound的情况。

好处：保证了程序的安全性。例子：比如我们重新写了一个String类，加载的时候并不会去加载到我们自己写的String类，因为当请求上到最高层的时候，启动类加载器发现自己能够加载String类，因此就不会加载到我们自己写的String类了。



199.说一下类加载的执行过程?

举个通俗点的例子来说，JVM在执行某段代码时，遇到了class A， 然而此时内存中并没有class A的相关信息，于是JVM就会到相应的class文件中去寻找class A的类信息，并加载进内存中，这就是我们所说的类加载过程。

由此可见，JVM不是一开始就把所有的类都加载进内存中，而是只有第一次遇到某个需要运行的类时才会加载，且只加载一次。

类加载
类加载的过程主要分为三个部分：加载\链接\初始化
而链接又可以细分为三个小部分：验证\准备\解析

https://blog.csdn.net/ln152315/article/details/79223441

200.怎么判断对象是否可以被回收?

一、引用计数算法：
　　判断对象的引用数量：

通过判断对象的引用数量来决定对象是否可以被回收；每个对象实例都有一个引用计数器，被引用则+1，完成引用则-1；任何引用计数为0的对象实例可以被当作垃圾收集；

优缺点：
优点：执行效率高，程序执行受影响较小；
缺点：无法检测出循环引用的情况，导致内存泄漏；

二、可达性分析算法：
通过判断对象的引用链是否可达来决定对象是否可以被回收；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没有任何引用链时，则认为此对象可以被回收。大家可以认为就是一个树的根节点开始计算引用情况。

可以作为GC Root的对象有：
虚拟机栈中引用的对象（栈帧中的本地变量表）；
方法区中的常量引用对象；
方法区中类静态属性引用对象；
本地方法栈中JNI（Native方法）的引用对象；
活跃线程中的引用对象；

201.java 中都有哪些引用类型?

无论是通过引用计数算法判断对象的引用数量，还是通过可达性分析算法判断对象是否引用链可达，判定对象是否存活都和“引用”离不开关系。如今，将引用分为强引用、软引用、弱引用和虚引用4种，强度依次减弱。

1、强引用：发生GC的时候不会被回收。
2、软引用：有用但不是必须的对象，在发生内存溢出之前会被回收。
3、弱引用：有用但不是必须的对象，在下一次GC的时候会被回收。
4、虚引用：无法通过虚引用获得对象，用PhantomReference实现虚引用，虚引用的用途是在GC时返回一个通知。

202.说一下 jvm 有哪些垃圾回收算法?

常用的垃圾回收算法有四种：标记-清除算法、复制算法、标记-整理算法、分代收集算法。

1.标记-清除算法
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，标记完成后统一回收所有被标记的对象，如下图。缺点：标记和清除两个过程效率都不高；标记清除之后会产生大量不连续的内存碎片。
2.复制算法
把内存分为大小相等的两块，每次存储只用其中一块，当这一块用完了，就把存活的对象全部复制到另一块上，同时把使用过的这块内存空间全部清理掉，往复循环，如下图。缺点：实际可使用的内存空间缩小为原来的一半，比较适合。copy算法是在新生代执行的，因为新生代对象大多朝生夕灭，存活时间短，占内存小，所以一次gc后剩下的对象少而且小，直接从from survivor和eden区copy到to survivor就好了，这叫minor gc。
3.标记-整理算法
先对可用的对象进行标记，然后所有被标记的对象向一段移动，最后清除可用对象边界以外的内存，如下图。标记整理是老年代常用的，因为老年代对象大而且生存时间长，不适合用copy算法，这叫major gc。
4.分代收集算法
把堆内存分为新生代和老年代，新生代又分为 Eden 区、From Survivor 和 To Survivor。一般新生代中的对象基本上都是朝生夕灭的，每次只有少量对象存活，因此采用复制算法，只需要复制那些少量存活的对象就可以完成垃圾收集；老年代中的对象存活率较高，就采用标记-清除和标记-整理算法来进行回收。

203.说一下 jvm 有哪些垃圾回收器?

一、串行垃圾回收器
二、并行垃圾回收器
三、CMS收集器
四、G1垃圾收集器

204.详细介绍一下 CMS 垃圾回收器?

CMS（Concurrent Mark Sweep）收集器是一种以获得最短回收停顿时间为目标的收集器。从名字就能知道它是标记-清除算法的。但是它比一般的标记-清除算法要复杂一些，分为以下4个阶段：

- 初始标记：标记一下GC Roots能直接关联到的对象，会"Stop The World"。
- 并发标记：GC Roots Tracing，可以和用户线程并发执行。
- 重新标记：标记期间产生的对象存活的再次判断，修正对这些对象的标记，执行时间相对并发标记短，会“Stop The World”.
- 并发清除：清除对象，可以和用户线程并发执行。

由于垃圾回收线程可以和用户线程同时运行，也就是说它是并发的，那么它会对CPU的资源非常敏感，CMS默认启动的回收线程数是（CPU数量+3）/ 4，当CPU<4个时，并发回收是垃圾收集线程就不会少于25%，而且随着CPU减少而增加，这样会影响用户线程的执行。而且由于它是基于标记-清除算法的，那么就无法避免空间碎片的产生。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。

所谓浮动垃圾，在CMS并发清理阶段用户线程还在运行着，伴随程序运行自然还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只能留待下一次GC时再清理掉。

205.新生代垃圾回收器和老生代垃圾回收器都有哪些？有什么区别?

206.简述分代垃圾回收器是怎么工作的?

分代回收器有两个分区：老生代和新生代，新生代默认的空间占比总空间的 1/3，老生代的默认占比是 2/3。

新生代使用的是复制算法，新生代里有 3 个分区：Eden、To Survivor、From Survivor，它们的默认占比是 8:1:1，它的执行流程如下：

把 Eden + From Survivor 存活的对象放入 To Survivor 区；
清空 Eden 和 From Survivor 分区；
From Survivor 和 To Survivor 分区交换，From Survivor 变 To Survivor，To Survivor 变 From Survivor。

每次在 From Survivor 到 To Survivor 移动时都存活的对象，年龄就 +1，当年龄到达 15（默认配置是 15）时，升级为老生代。大对象也会直接进入老生代。

老生代当空间占用到达某个值之后就会触发全局垃圾收回，一般使用标记整理的执行算法。以上这些循环往复就构成了整个分代垃圾回收的整体执行流程
——————————————
原文链接：https://blog.csdn.net/miachen520/article/details/104908811

207.说一下 jvm 调优的工具?

jvm监控分析工具一般分为两类，一种是jdk自带的工具，一种是第三方的分析工具。jdk自带工具一般在jdk bin目录下面，以exe的形式直接点击就可以使用，其中包含分析工具已经很强大，几乎涉及了方方面面，但是我们最常使用的只有两款：jconsole.exe和jvisualvm.exe；第三方的分析工具有很多，各自的侧重点不同，比较有代表性的：MAT(Memory Analyzer Tool)、GChisto等。

208.常用的 jvm 调优的参数都有哪些?

-Xms2g：初始化推大小为 2g；
-Xmx2g：堆最大内存为 2g；
-XX:NewRatio=4：设置年轻的和老年代的内存比例为 1:4；
-XX:SurvivorRatio=8：设置新生代 Eden 和 Survivor 比例为 8:2；
–XX:+UseParNewGC：指定使用 ParNew + Serial Old 垃圾回收器组合；
-XX:+UseParallelOldGC：指定使用 ParNew + ParNew Old 垃圾回收器组合；
-XX:+UseConcMarkSweepGC：指定使用 CMS + Serial Old 垃圾回收器组合；
-XX:+PrintGC：开启打印 gc 信息；
-XX:+PrintGCDetails：打印 gc 详细信息。

深拷贝和浅拷贝

浅拷贝（shallowCopy）只是增加了一个指针指向已存在的内存地址，
深拷贝（deepCopy）是增加了一个指针并且申请了一个新的内存，使这个增加的指针指向这个新的内存，
使用深拷贝的情况下，释放内存的时候不会因为出现浅拷贝时释放同一个内存的错误。
浅复制：仅仅是指向被复制的内存地址，如果原地址发生改变，那么浅复制出来的对象也会相应的改变。
深复制：在计算机中开辟一块新的内存地址用于存放复制的对象。

对象的创建

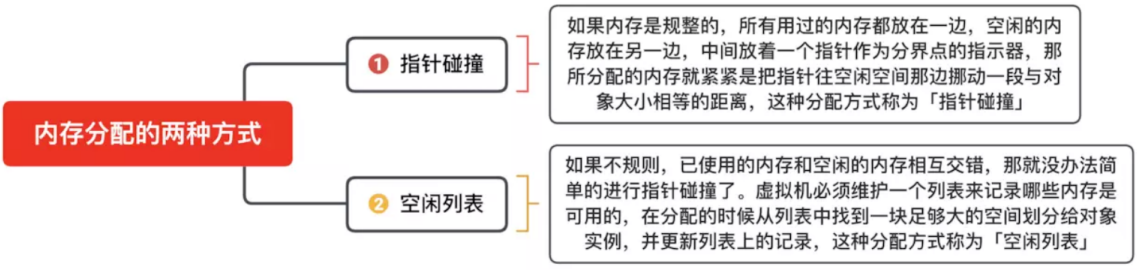
说到对象的创建，首先让我们看看 Java 中提供的几种对象创建方式：

Header	解释
使用new关键字	调用了构造函数
使用Class的newInstance方法	调用了构造函数
使用Constructor类的newInstance方法	调用了构造函数
使用clone方法	没有调用构造函数
使用反序列化	没有调用构造函数

虚拟机遇到一条new指令时，先检查常量池是否已经加载相应的类，如果没有，必须先执行相应的类加载。类加载通过后，接下来分配内存。若Java堆中内存是绝对规整的，使用“指针碰撞“方式分配内存；如果不是规整的，就从空闲列表中分配，叫做”空闲列表“方式。划分内存时还需要考虑一个问题-并发，也有两种方式：CAS同步处理，或者本地线程分配缓冲(Thread Local Allocation Buffer, TLAB)。然后内存空间初始化操作，接着是做一些必要的对象设置(元信息、哈希码…)，最后执行<init>方法。

为对象分配内存

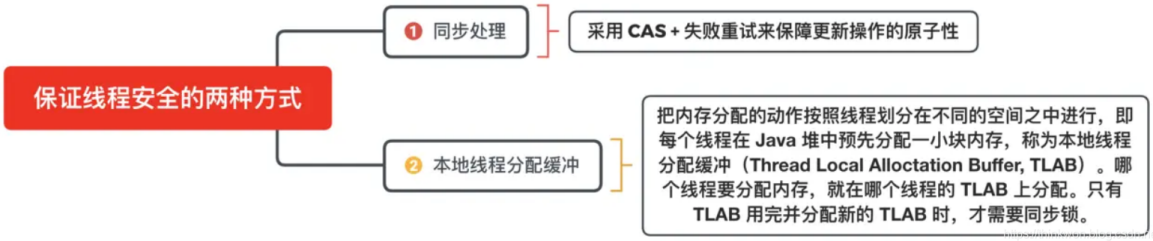
类加载完成后，接着会在Java堆中划分一块内存分配给对象。内存分配根据Java堆是否规整，有两种方式：
指针碰撞：如果Java堆的内存是规整，即所有用过的内存放在一边，而空闲的的放在另一边。分配内存时将位于中间的指针指示器向空闲的内存移动一段与对象大小相等的距离，这样便完成分配内存工作。
空闲列表：如果Java堆的内存不是规整的，则需要由虚拟机维护一个列表来记录那些内存是可用的，这样在分配的时候可以从列表中查询到足够大的内存分配给对象，并在分配后更新列表记录。
选择哪种分配方式是由 Java 堆是否规整来决定的，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。



处理并发安全问题

对象的创建在虚拟机中是一个非常频繁的行为，哪怕只是修改一个指针所指向的位置，在并发情况下也是不安全的，可能出现正在给对象 A 分配内存，指针还没来得及修改，对象 B 又同时使用了原来的指针来分配内存的情况。解决这个问题有两种方案：

1. 对分配内存空间的动作进行同步处理（采用 CAS + 失败重试来保障更新操作的原子性）；
2. 把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在 Java 堆中预先分配一小块内存，称为本地线程分配缓冲（Thread Local Allocation Buffer，TLAB）。哪个线程要分配内存，就在哪个线程的 TLAB 上分配。只有 TLAB 用完并分配新的 TLAB 时，才需要同步锁。通过-XX:+/-UserTLAB参数来设定虚拟机是否使用TLAB。



对象的访问定位

Java程序需要通过 JVM 栈上的引用访问堆中的具体对象。对象的访问方式取决于 JVM 虚拟机的实现。目前主流的访问方式有 句柄 和 直接指针 两种方式。
指针： 指向对象，代表一个对象在内存中的起始地址。
句柄： 可以理解为指向指针的指针，维护着对象的指针。句柄不直接指向对象，而是指向对象的指针（句柄不发生变化，指向固定内存地址），再由对象的指针指向对象的真实内存地址。

内存溢出异常

内存泄漏是指不再被使用的对象或者变量一直被占据在内存中。理论上来说，Java是有GC垃圾回收机制的，也就是说，不再被使用的对象，会被GC自动回收掉，自动从内存中清除。

但是，即使这样，Java也还是存在着内存泄漏的情况，java导致内存泄露的原因很明确：长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄露，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收，这就是java中内存泄露的发生场景。

简述Java垃圾回收机制

在java中，程序员是不需要显示的去释放一个对象的内存的，而是由虚拟机自行执行。在JVM中，有一个垃圾回收线程，它是低优先级的，在正常情况下是不会执行的，只有在虚拟机空闲或者当前堆内存不足时，才会触发执行，扫面那些没有被任何引用的对象，并将它们添加到要回收的集合中，进行回收。

GC是什么？为什么要GC

GC 是垃圾收集的意思（Gabage Collection）,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。

垃圾回收的优点和原理。

java语言最显著的特点就是引入了垃圾回收机制，它使java程序员在编写程序时不再考虑内存管理的问题。
由于有这个垃圾回收机制，java中的对象不再有“作用域”的概念，只有引用的对象才有“作用域”。
垃圾回收机制有效的防止了内存泄露，可以有效的使用可使用的内存。
垃圾回收器通常作为一个单独的低级别的线程运行，在不可预知的情况下对内存堆中已经死亡的或很长时间没有用过的对象进行清除和回收。
程序员不能实时的对某个对象或所有对象调用垃圾回收器进行垃圾回收。

JVM中的永久代中会发生垃圾回收吗

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.

对象优先在 Eden 区分配

多数情况，对象都在新生代 Eden 区分配。当 Eden 区分配没有足够的空间进行分配时，虚拟机将会发起一次 Minor GC。如果本次 GC 后还是没有足够的空间，则将启用分配担保机制在老年代中分配内存。
这里我们提到 Minor GC，如果你仔细观察过 GC 日常，通常我们还能从日志中发现 Major GC/Full GC。
Minor GC 是指发生在新生代的 GC，因为 Java 对象大多都是朝生夕死，所有 Minor GC 非常频繁，一般回收速度也非常快；
Major GC/Full GC 是指发生在老年代的 GC，出现了 Major GC 通常会伴随至少一次 Minor GC。Major GC 的速度通常会比 Minor GC 慢 10 倍以上。

大对象直接进入老年代

所谓大对象是指需要大量连续内存空间的对象，频繁出现大对象是致命的，会导致在内存还有不少空间的情况下提前触发 GC 以获取足够的连续空间来安置新对象。
前面我们介绍过新生代使用的是标记-清除算法来处理垃圾回收的，如果大对象直接在新生代分配就会导致 Eden 区和两个 Survivor 区之间发生大量的内存复制。因此对于大对象都会直接在老年代进行分配。

长期存活对象将进入老年代

虚拟机采用分代收集的思想来管理内存，那么内存回收时就必须判断哪些对象应该放在新生代，哪些对象应该放在老年代。因此虚拟机给每个对象定义了一个对象年龄的计数器，如果对象在 Eden 区出生，并且能够被 Survivor 容纳，将被移动到 Survivor 空间中，这时设置对象年龄为 1。对象在 Survivor 区中每「熬过」一次 Minor GC 年龄就加 1，当年龄达到一定程度（默认 15） 就会被晋升到老年代。