# Java面试知识点解析——JVM篇

2018年06月07日 00:00:00

阅读数：791

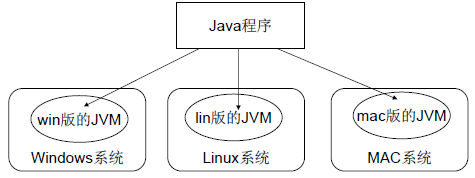
一、JVM 基础知识

#### ****1）Java 是如何实现跨平台的？****

****注意：跨平台的是 Java 程序，而不是 JVM。JVM 是用 C/C++ 开发的，是编译后的机器码，不能跨平台，不同平台下需要安装不同版本的 JVM****

答：我们编写的 Java 源码，编译后会生成一种 .class 文件，称为字节码文件。Java 虚拟机（JVM）就是负责将字节码文件翻译成特定平台下的机器码然后运行，也就是说，只要在不同平台上安装对应的 JVM，就可以运行字节码文件，运行我们编写的 Java 程序。

而这个过程，我们编写的 Java 程序没有做任何改变，仅仅是通过 JVM 这一 “中间层” ，就能在不同平台上运行，真正实现了 “一次编译，到处运行” 的目的。



#### ****2）什么是 JVM ？****

解析：不仅仅是基本概念，还有 JVM 的作用。

答：JVM，即 Java Virtual Machine，Java 虚拟机。它通过模拟一个计算机来达到一个计算机所具有的的计算功能。JVM 能够跨计算机体系结构来执行 Java 字节码，主要是由于 JVM 屏蔽了与各个计算机平台相关的软件或者硬件之间的差异，使得与平台相关的耦合统一由 JVM 提供者来实现。

#### ****3）JVM 由哪些部分组成？****

解析：这是对 JVM 体系结构的考察

答：JVM 的结构基本上由 4 部分组成：

类加载器，在 JVM 启动时或者类运行时将需要的 class 加载到 JVM 中

执行引擎，执行引擎的任务是负责执行 class 文件中包含的字节码指令，相当于实际机器上的 CPU

内存区，将内存划分成若干个区以模拟实际机器上的存储、记录和调度功能模块，如实际机器上的各种功能的寄存器或者 PC 指针的记录器等

本地方法调用，调用 C 或 C++ 实现的本地方法的代码返回结果



#### ****4）类加载器是有了解吗？****

解析：底层原理的考察，其中涉及到类加载器的概念，功能以及一些底层的实现。

答：顾名思义，类加载器（class loader）用来加载 Java 类到 Java 虚拟机中。一般来说，Java 虚拟机使用 Java 类的方式如下：Java 源程序（.java 文件）在经过 Java 编译器编译之后就被转换成 Java 字节代码（.class 文件）。

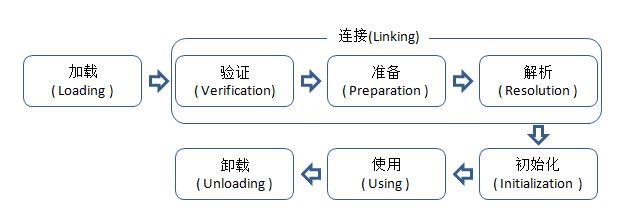
类加载器负责读取 Java 字节代码，并转换成 java.lang.Class类的一个实例。每个这样的实例用来表示一个 Java 类。通过此实例的 newInstance()方法就可以创建出该类的一个对象。实际的情况可能更加复杂，比如 Java 字节代码可能是通过工具动态生成的，也可能是通过网络下载的。

****面试官：Java 虚拟机是如何判定两个 Java 类是相同的？****

答：Java 虚拟机不仅要看类的全名是否相同，还要看加载此类的类加载器是否一样。只有两者都相同的情况，才认为两个类是相同的。即便是同样的字节代码，被不同的类加载器加载之后所得到的类，也是不同的。比如一个 Java 类 com.example.Sample，编译之后生成了字节代码文件 Sample.class。两个不同的类加载器 ClassLoaderA和 ClassLoaderB分别读取了这个 Sample.class文件，并定义出两个 java.lang.Class类的实例来表示这个类。这两个实例是不相同的。对于 Java 虚拟机来说，它们是不同的类。试图对这两个类的对象进行相互赋值，会抛出运行时异常 ClassCastException。

#### ****5）类加载器是如何加载 class 文件的？****

答：下图所示是 ClassLoader 加载一个 class 文件到 JVM 时需要经过的步骤：



第一个阶段是找到 .class 文件并把这个文件包含的字节码加载到内存中

第二阶段又可以分为三个步骤，分别是字节码验证、Class 类数据结构分析及相应的内存分配和最后的符号表的链接

第三个阶段是类中静态属性和初始化赋值，以及静态块的执行等

****面试官：能详细讲讲吗？****

答：

****1.加载****

查找并加载类的二进制数据加载时类加载过程的第一个阶段，在加载阶段，虚拟机需要完成以下三件事情：

通过一个类的全限定名来获取其定义的二进制字节流。

将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。

在Java堆中生成一个代表这个类的 java.lang.Class 对象，作为对方法区中这些数据的访问入口。

相对于类加载的其他阶段而言，加载阶段（准确地说，是加载阶段获取类的二进制字节流的动作）是可控性最强的阶段，因为开发人员既可以使用系统提供的类加载器来完成加载，也可以自定义自己的类加载器来完成加载。

加载阶段完成后，虚拟机外部的二进制字节流就按照虚拟机所需的格式存储在方法区之中，而且在Java堆中也创建一个 java.lang.Class类的对象，这样便可以通过该对象访问方法区中的这些数据。

****2.连接****

****验证：确保被加载的类的正确性****

验证是连接阶段的第一步，这一阶段的目的是为了确保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。验证阶段大致会完成4个阶段的检验动作：

****文件格式验证：****验证字节流是否符合Class文件格式的规范；例如：是否以 0xCAFEBABE开头、主次版本号是否在当前虚拟机的处理范围之内、常量池中的常量是否有不被支持的类型。

****元数据验证：****对字节码描述的信息进行语义分析（注意：对比javac编译阶段的语义分析），以保证其描述的信息符合Java语言规范的要求；例如：这个类是否有父类，除了 java.lang.Object之外。

****字节码验证：****通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。

****符号引用验证：****确保解析动作能正确执行。

验证阶段是非常重要的，但不是必须的，它对程序运行期没有影响，如果所引用的类经过反复验证，那么可以考虑采用 -Xverifynone 参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间。

****准备：为类的**静态变量**分配内存，并将其初始化为默认值****

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中分配。对于该阶段有以下几点需要注意：

① 这时候进行内存分配的仅包括类变量（static），而不包括实例变量，实例变量会在对象实例化时随着对象一块分配在Java堆中。

② 这里所设置的初始值通常情况下是数据类型默认的零值（如0、0L、null、false等），而不是被在Java代码中被显式地赋予的值。

假设一个类变量的定义为： public static int value = 3;

那么变量value在准备阶段过后的初始值为 0，而不是 3，因为这时候尚未开始执行任何 Java 方法，而把 value 赋值为 3 的public static指令是在程序编译后，存放于类构造器 <clinit>（）方法之中的，所以把value赋值为3的动作将在初始化阶段才会执行。

这里还需要注意如下几点：

对基本数据类型来说，对于类变量（static）和全局变量，如果不显式地对其赋值而直接使用，则系统会为其赋予默认的零值，而对于局部变量来说，在使用前必须显式地为其赋值，否则编译时不通过。

对于同时被static和final修饰的常量，必须在声明的时候就为其显式地赋值，否则编译时不通过；而只被final修饰的常量则既可以在声明时显式地为其赋值，也可以在类初始化时显式地为其赋值，总之，在使用前必须为其显式地赋值，系统不会为其赋予默认零值。

对于引用数据类型reference来说，如数组引用、对象引用等，如果没有对其进行显式地赋值而直接使用，系统都会为其赋予默认的零值，即null。

如果在数组初始化时没有对数组中的各元素赋值，那么其中的元素将根据对应的数据类型而被赋予默认的零值。

③ 如果类字段的字段属性表中存在 ConstantValue 属性，即同时被 final 和 static 修饰，那么在准备阶段变量 value 就会被初始化为 ConstValue 属性所指定的值。

假设上面的类变量 value 被定义为： public static final int value = 3;

编译时 Javac 将会为 value 生成 ConstantValue 属性，在准备阶段虚拟机就会根据 ConstantValue 的设置将 value 赋值为 3。我们可以理解为 static final 常量在编译期就将其结果放入了调用它的类的常量池中

****解析：把类中的符号引用转换为直接引用****

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程，解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。符号引用就是一组符号来描述目标，可以是任何字面量。

直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。

****3.初始化****

初始化，为类的静态变量赋予正确的初始值，JVM负责对类进行初始化，主要对类变量进行初始化。在Java中对类变量进行初始值设定有两种方式：

① 声明类变量是指定初始值

② 使用静态代码块为类变量指定初始值

JVM初始化步骤

1、假如这个类还没有被加载和连接，则程序先加载并连接该类

2、假如该类的直接父类还没有被初始化，则先初始化其直接父类

3、假如类中有初始化语句，则系统依次执行这些初始化语句

类初始化时机：只有当对类的主动使用的时候才会导致类的初始化，类的主动使用包括以下六种：

创建类的实例，也就是new的方式

访问某个类或接口的静态变量，或者对该静态变量赋值

调用类的静态方法

反射（如 Class.forName(“com.shengsiyuan.Test”)）

初始化某个类的子类，则其父类也会被初始化

Java虚拟机启动时被标明为启动类的类（ JavaTest），直接使用 java.exe命令来运行某个主类

****结束生命周期****

在如下几种情况下，Java虚拟机将结束生命周期

执行了 System.exit()方法

程序正常执行结束

程序在执行过程中遇到了异常或错误而异常终止

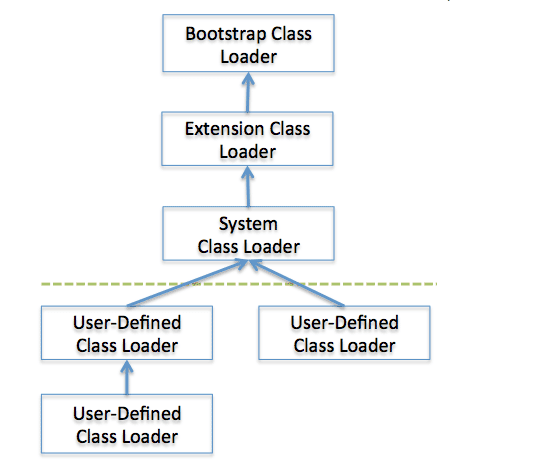
由于操作系统出现错误而导致Java虚拟机进程终止

参考文章：jvm系列(一):java类的加载机制 - 纯洁的微笑

#### ****6）双亲委派模型（Parent Delegation Model）？****

解析：类的加载过程采用双亲委派机制，这种机制能更好的保证 Java 平台的安全性

答：类加载器 ClassLoader 是具有层次结构的，也就是父子关系，其中，Bootstrap 是所有类加载器的父亲，如下图所示：



该模型要求除了顶层的 Bootstrap class loader 启动类加载器外，其余的类加载器都应当有自己的父类加载器。子类加载器和父类加载器不是以继承（Inheritance）的关系来实现，而是通过组合（Composition）关系来复用父加载器的代码。每个类加载器都有自己的命名空间（由该加载器及所有父类加载器所加载的类组成，在同一个命名空间中，不会出现类的完整名字（包括类的包名）相同的两个类；在不同的命名空间中，有可能会出现类的完整名字（包括类的包名）相同的两个类）

****面试官：双亲委派模型的工作过程？****

答：

1.当前 ClassLoader 首先从自己已经加载的类中查询是否此类已经加载，如果已经加载则直接返回原来已经加载的类。

每个类加载器都有自己的加载缓存，当一个类被加载了以后就会放入缓存，  
等下次加载的时候就可以直接返回了。

2.当前 ClassLoader 的缓存中没有找到被加载的类的时候，委托父类加载器去加载，父类加载器采用同样的策略，首先查看自己的缓存，然后委托父类的父类去加载，一直到 bootstrap ClassLoader.

当所有的父类加载器都没有加载的时候，再由当前的类加载器加载，并将其放入它自己的缓存中，以便下次有加载请求的时候直接返回。

****面试官：为什么这样设计呢？****

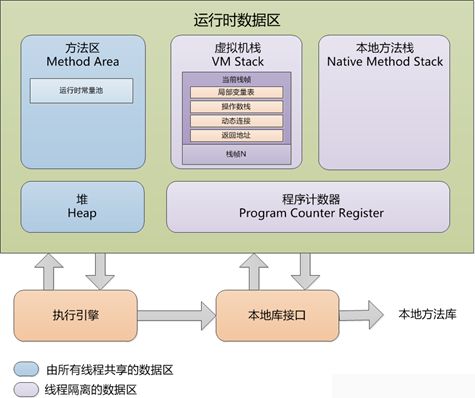
解析：这是对于使用这种模型来组织累加器的好处

答：主要是为了安全性，避免用户自己编写的类动态替换 Java 的一些核心类，比如 String，同时也避免了重复加载，因为 JVM 中区分不同类，不仅仅是根据类名，相同的 class 文件被不同的 ClassLoader 加载就是不同的两个类，如果相互转型的话会抛java.lang.ClassCaseException.

参考文章： JVM 的 工作原理，层次结构 以及 GC工作原理

## **二、JVM 内存管理**

#### ****1）JVM 内存划分：****



答：

方法区（线程共享）：各个线程共享的一个区域，用于存储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。虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却又一个别名叫做 Non-Heap（非堆），目的应该是与 Java 堆区分开来。

运行时常量池：是方法区的一部分，用于存放编译器生成的各种字面量和符号引用。

堆内存（线程共享）：所有线程共享的一块区域，垃圾收集器管理的主要区域。目前主要的垃圾回收算法都是分代收集算法，所以 Java 堆中还可以细分为：新生代和老年代；再细致一点的有 Eden 空间、From Survivor 空间、To Survivor 空间等，默认情况下新生代按照8:1:1的比例来分配。根据 Java 虚拟机规范的规定，Java 堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，就像我们的磁盘一样。

程序计数器： Java 线程私有，类似于操作系统里的 PC 计数器，它可以看做是当前线程所执行的字节码的行号指示器。如果线程正在执行的是一个 Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是 Native 方法，这个计数器值则为空（Undefined）。此内存区域是唯一一个在 Java 虚拟机规范中没有规定任何 OutOfMemoryError 情况的区域。

虚拟机栈（栈内存）：Java线程私有，虚拟机展描述的是Java方法执行的内存模型：每个方法在执行的时候，都会创建一个栈帧用于存储局部变量、操作数、动态链接、方法出口等信息；每个方法调用都意味着一个栈帧在虚拟机栈中入栈到出栈的过程；

本地方法栈 ：和Java虚拟机栈的作用类似，区别是该区域为 JVM 提供使用 native 方法的服务

#### ****2）对象分配规则？****

答：

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。

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）。

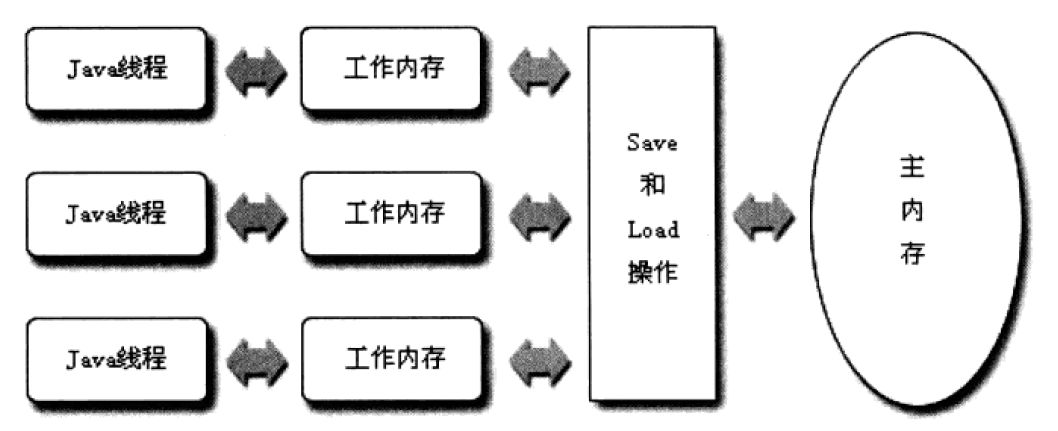
长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了1次Minor GC那么对象会进入Survivor区，之后每经过一次Minor GC那么对象的年龄加1，知道达到阀值对象进入老年区。

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。

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。

#### ****3）Java 的内存模型：****

答：

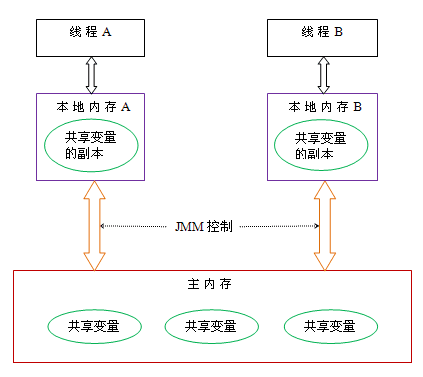


Java 虚拟机规范中试图定义一种 Java 内存模型（Java Memory Model, JMM）来屏蔽掉各层硬件和操作系统的内存访问差异，以实现让 Java 程序在各种平台下都能达到一致的内存访问效果。

Java 内存模型规定了所有的变量都存储在主内存（Main Memory）中。每条线程还有自己的工作内存（Working Memory），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（读取、赋值等）都必须在主内存中进行，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，线程间的变量值的传递均需要通过主内存来完成，线程、主内存、工作内存三者的关系如上图。

****面试官：两个线程之间是如何通信的呢？****

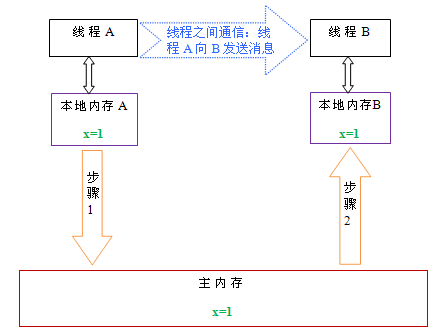
答：在****共享内存****的并发模型里，线程之间共享程序的公共状态，线程之间通过写-读内存中的公共状态来隐式进行通信，典型的共享内存通信方式就是通过共享对象进行通信。



例如上图线程 A 与 线程 B 之间如果要通信的话，那么就必须经历下面两个步骤：

1.首先，线程 A 把本地内存 A 更新过得共享变量刷新到主内存中去

2.然后，线程 B 到主内存中去读取线程 A 之前更新过的共享变量



在****消息传递****的并发模型里，线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信，在 Java 中典型的消息传递方式就是 wait() 和 notify()。

#### ****4）内存屏障？****

解析：在这之前应该对重排序的问题有所了解，这里我找到一篇很好的文章分享一下：Java内存访问重排序的研究：https://tech.meituan.com/java-memory-reordering.html

答：内存屏障，又称内存栅栏，是一组处理器指令，用于实现对内存操作的顺序限制。

****面试官：内存屏障为何重要？****

答：对主存的一次访问一般花费硬件的数百次时钟周期。处理器通过缓存（caching）能够从数量级上降低内存延迟的成本这些缓存为了性能重新排列待定内存操 作的顺序。也就是说，程序的读写操作不一定会按照它要求处理器的顺序执行。当数据是不可变的，同时/或者数据限制在线程范围内，这些优化是无害的。如果把 这些优化与对称多处理（symmetric multi-processing）和共享可变状态（shared mutable state）结合，那么就是一场噩梦。当基于共享可变状态的内存操作被重新排序时，程序可能行为不定。一个线程写入的数据可能被其他线程可见，原因是数据 写入的顺序不一致。适当的放置内存屏障通过强制处理器顺序执行待定的内存操作来避免这个问题。

#### ****5）类似-Xms、-Xmn这些参数的含义：****

答：

堆内存分配：

JVM初始分配的内存由-Xms指定，默认是物理内存的1/64

JVM最大分配的内存由-Xmx指定，默认是物理内存的1/4

默认空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制；空余堆内存大于70%时，JVM会减少堆直到 -Xms的最小限制。

因此服务器一般设置-Xms、-Xmx相等以避免在每次GC 后调整堆的大小。对象的堆内存由称为垃圾回收器的自动内存管理系统回收。

非堆内存分配：

JVM使用-XX:PermSize设置非堆内存初始值，默认是物理内存的1/64；

由XX:MaxPermSize设置最大非堆内存的大小，默认是物理内存的1/4。

-Xmn2G：设置年轻代大小为2G。

-XX:SurvivorRatio，设置年轻代中Eden区与Survivor区的比值。

#### ****6）内存泄漏和内存溢出****

答：

概念：

内存溢出指的是内存不够用了。

内存泄漏是指对象可达，但是没用了。即本该被GC回收的对象并没有被回收

内存泄露是导致内存溢出的原因之一；内存泄露积累起来将导致内存溢出。

内存泄漏的原因分析：

长生命周期的对象引用短生命周期的对象

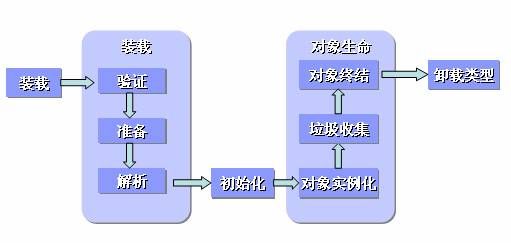
没有将无用对象置为null

小结：本小节涉及到 JVM 虚拟机，包括对内存的管理等知识，相对较深。除了以上问题，面试官会继续问你一些比较深的问题，可能也是为了看看你的极限在哪里吧。比如：内存调优、内存管理，是否遇到过内存泄露的实际案例、是否真正关心过内存等。

#### ****7）简述一下 Java 中创建一个对象的过程？****

解析：回答这个问题首先就要清楚类的生命周期

答：下图展示的是类的生命周期流向：



Java中对象的创建就是在堆上分配内存空间的过程，此处说的对象创建仅限于new关键字创建的普通Java对象，不包括数组对象的创建。

大致过程如下：

****1.检测类是否被加载：****

当虚拟机执行到new时，会先去常量池中查找这个类的符号引用。如果能找到符号引用，说明此类已经被加载到方法区（方法区存储虚拟机已经加载的类的信息），可以继续执行；如果找不到符号引用，就会使用类加载器执行类的加载过程，类加载完成后继续执行。

****2.为对象分配内存：****

类加载完成以后，虚拟机就开始为对象分配内存，此时所需内存的大小就已经确定了。只需要在堆上分配所需要的内存即可。

具体的分配内存有两种情况：第一种情况是内存空间绝对规整，第二种情况是内存空间是不连续的。

对于内存绝对规整的情况相对简单一些，虚拟机只需要在被占用的内存和可用空间之间移动指针即可，这种方式被称为指针碰撞。

对于内存不规整的情况稍微复杂一点，这时候虚拟机需要维护一个列表，来记录哪些内存是可用的。分配内存的时候需要找到一个可用的内存空间，然后在列表上记录下已被分配，这种方式成为空闲列表。

分配内存的时候也需要考虑线程安全问题，有两种解决方案：

第一种是采用同步的办法，使用CAS来保证操作的原子性。

另一种是每个线程分配内存都在自己的空间内进行，即是每个线程都在堆中预先分配一小块内存，称为本地线程分配缓冲（TLAB），分配内存的时候再TLAB上分配，互不干扰。

****3.为分配的内存空间初始化零值：****

对象的内存分配完成后，还需要将对象的内存空间都初始化为零值，这样能保证对象即使没有赋初值，也可以直接使用。

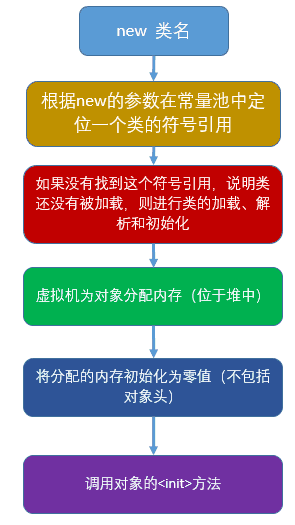
****4.对对象进行其他设置：****

分配完内存空间，初始化零值之后，虚拟机还需要对对象进行其他必要的设置，设置的地方都在对象头中，包括这个对象所属的类，类的元数据信息，对象的hashcode，GC分代年龄等信息。

****5.执行 init 方法：****

执行完上面的步骤之后，在虚拟机里这个对象就算创建成功了，但是对于Java程序来说还需要执行init方法才算真正的创建完成，因为这个时候对象只是被初始化零值了，还没有真正的去根据程序中的代码分配初始值，调用了init方法之后，这个对象才真正能使用。

到此为止一个对象就产生了，这就是new关键字创建对象的过程。过程如下：



参考文章：Java创建对象的过程简介

****面试官：对象的内存布局是怎样的？****

答：对象的内存布局包括三个部分：对象头，实例数据和对齐填充。

对象头：对象头包括两部分信息，第一部分是存储对象自身的运行时数据，如哈希码，GC分代年龄，锁状态标志，线程持有的锁等等。第二部分是类型指针，即对象指向类元数据的指针。

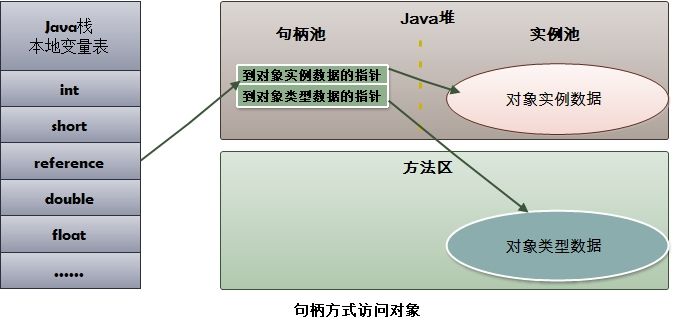
实例数据：就是数据啦

对齐填充：不是必然的存在，就是为了对齐的嘛

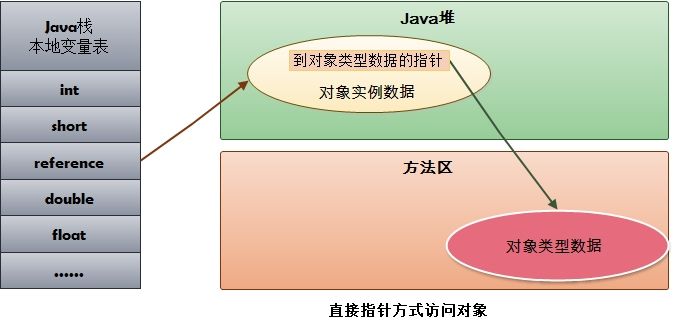
****面试官：对象是如何定位访问的？****

答：对象的访问定位有两种：句柄定位和直接指针

句柄定位：Java 堆会画出一块内存来作为句柄池，reference中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息



直接指针访问：java堆对象的不居中就必须考虑如何放置访问类型数据的相关信息，而reference中存储的直接就是对象地址



****比较：使用直接指针就是速度快，使用句柄reference指向稳定的句柄，对象被移动改变的也只是句柄中实例数据的指针，而reference本身并不需要修改。****

参考文章：JAVA对象创建的过程

## **三、GC 相关**

#### ****1）如何判断一个对象是否已经死去？****

答：

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。

#### ****2）垃圾回收算法有哪些？****

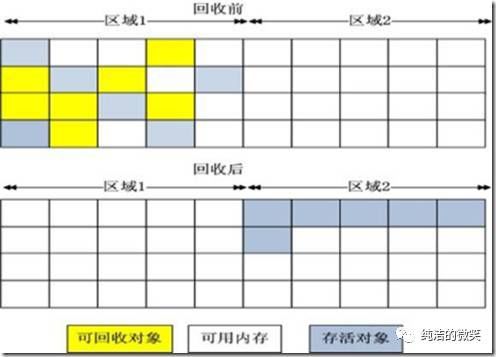
答：

引用计数：  
原理是此对象有一个引用，即增加一个计数，删除一个引用则减少一个计数。垃圾回收时，只用收集计数为0的对象。此算法最致命的是无法处理循环引用的问题。

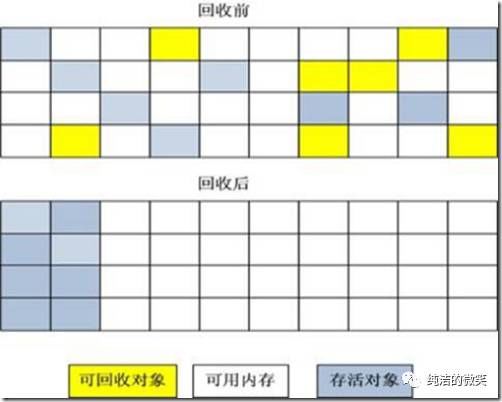
标记-清除：  
此算法执行分两阶段。第一阶段从引用根节点开始标记所有被引用的对象，第二阶段遍历整个堆，把未标记的对象清除。此算法需要暂停整个应用，同时，会产生内存碎片。



复制算法：  
此算法把内存空间划为两个相等的区域，每次只使用其中一个区域。垃圾回收时，遍历当前使用区域，把正在使用中的对象复制到另外一个区域中。此算法每次只处理正在使用中的对象，因此复制成本比较小，同时复制过去以后还能进行相应的内存整理，不会出现“碎片”问题。当然，此算法的缺点也是很明显的，就是需要两倍内存空间。



标记-整理：  
此算法结合了 “标记-清除” 和 “复制” 两个算法的优点。也是分两阶段，第一阶段从根节点开始标记所有被引用对象，第二阶段遍历整个堆，把清除未标记对象并且把存活对象“压缩”到堆的其中一块，按顺序排放。此算法避免了 “标记-清除” 的碎片问题，同时也避免了 “复制” 算法的空间问题。



分代收集算法：

分代收集算法并没有提出新的思想，只是根据对象存活周期的不同将内存划为几块。一般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适当的收集算法。

在新生袋中每次垃圾手机时都会由大批对象死去，只有少量存活，那就用复制算法，只需要付出少量存活对象的复制成本就可以。老年代中对象存活率高、没有额外担保，所以必须使用“标记-清理”或者“标记整理算法。

参考文章：jvm系列(三):GC算法 垃圾收集器——纯洁的微笑

#### ****3）GC什么时候开始？****

答：GC经常发生的区域是堆区，堆区还可以细分为新生代、老年代，新生代还分为一个Eden区和两个Survivor区。

对象优先在Eden中分配，当Eden中没有足够空间时，虚拟机将发生一次Minor GC，因为Java大多数对象都是朝生夕灭，所以Minor GC非常频繁，而且速度也很快；

Full GC，发生在老年代的GC，当老年代没有足够的空间时即发生Full GC，发生Full GC一般都会有一次Minor GC。大对象直接进入老年代，如很长的字符串数组，虚拟机提供一个-XX:PretenureSizeThreadhold参数，令大于这个参数值的对象直接在老年代中分配，避免在Eden区和两个Survivor区发生大量的内存拷贝；

发生Minor GC时，虚拟机会检测之前每次晋升到老年代的平均大小是否大于老年代的剩余空间大小，如果大于，则进行一次Full GC，如果小于，则查看HandlePromotionFailure设置是否允许担保失败，如果允许，那只会进行一次Minor GC，如果不允许，则改为进行一次Full GC。

#### ****4）引用的分类？****

答：

强引用：通过new出来的引用，只要强引用还存在，则不会回收。

软引用：通过SoftReference类来实现，用来描述一些有用但非必须的对象。在系统将要发生内存溢出异常之前，会把这些对象回收了，如果这次回收还是内存不够的话，才抛出内存溢出异常。

弱引用：非必须对象，通过WeakReference类来实现，被弱引用引用的对象，只要已发生GC就会把它干掉。

虚引用：通过PhantomReference类来实现，无法通过徐引用获得对象的实例，唯一作用就是在这个对象被GC时会收到一个系统通知。

扩展阅读：重新认识java（一） ---- 万物皆对象 ，文章中有对这四个引用有详细的描述，还有一些典型的应用，这里就不摘过来啦...

#### ****5）垃圾收集器？****

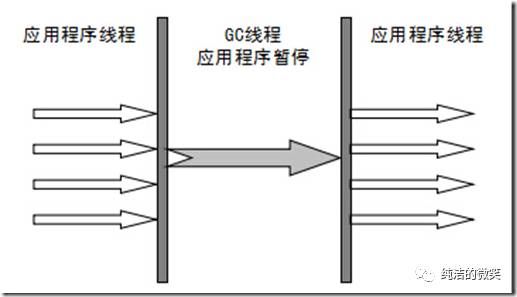
解析：如果说收集算法是内存回收的方法论，垃圾收集器就是内存回收的具体实现

答：

****1. Serial 收集器****

串行收集器是最古老，最稳定以及效率高的收集器，可能会产生较长的停顿，只使用一个线程去回收。新生代、老年代使用串行回收；新生代复制算法、老年代标记-压缩；垃圾收集的过程中会 Stop The World（服务暂停）

参数控制： -XX:+UseSerialGC 串行收集器

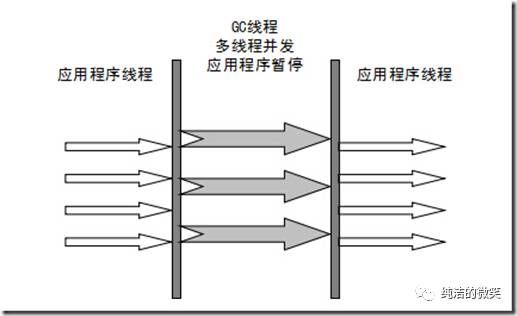


****2.ParNew 收集器****

ParNew收集器 ParNew收集器其实就是Serial收集器的多线程版本。新生代并行，老年代串行；新生代复制算法、老年代标记-压缩

参数控制：

-XX:+UseParNewGC ParNew收集器  
-XX:ParallelGCThreads 限制线程数量



****3.Parallel Scavenge收集器****

Parallel Scavenge收集器类似ParNew收集器，Parallel收集器更关注系统的吞吐量。可以通过参数来打开自适应调节策略，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或最大的吞吐量；也可以通过参数控制GC的时间不大于多少毫秒或者比例；新生代复制算法、老年代标记-压缩

参数控制： -XX:+UseParallelGC 使用Parallel收集器+ 老年代串行

****4.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

参数控制： -XX:+UseParallelOldGC 使用Parallel收集器+ 老年代并行

****5.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应用都集中在互联网站或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。

从名字（包含“Mark Sweep”）上就可以看出CMS收集器是基于“标记-清除”算法实现的，它的运作过程相对于前面几种收集器来说要更复杂一些，整个过程分为4个步骤，包括：

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）

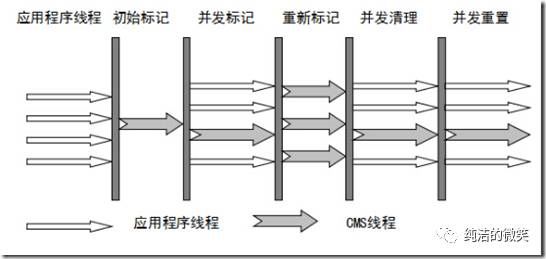
其中初始标记、重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，并发标记阶段就是进行GC Roots Tracing的过程，而重新标记阶段则是为了修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。

由于整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，所以总体上来说，CMS收集器的内存回收过程是与用户线程一起并发地执行。老年代收集器（新生代使用ParNew）

****优点:**** 并发收集、低停顿  
****缺点:**** 产生大量空间碎片、并发阶段会降低吞吐量

参数控制：

-XX:+UseConcMarkSweepGC 使用CMS收集器  
-XX:+ UseCMSCompactAtFullCollection Full GC后，进行一次碎片整理；整理过程是独占的，会引起停顿时间变长  
-XX:+CMSFullGCsBeforeCompaction 设置进行几次Full GC后，进行一次碎片整理  
-XX:ParallelCMSThreads 设定CMS的线程数量（一般情况约等于可用CPU数量）



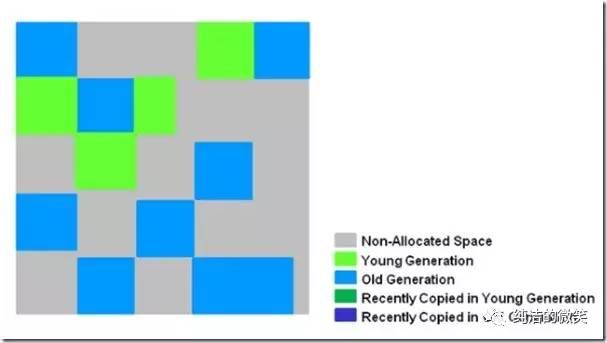
****6.G1收集器****

G1是目前技术发展的最前沿成果之一，HotSpot开发团队赋予它的使命是未来可以替换掉JDK1.5中发布的CMS收集器。与CMS收集器相比G1收集器有以下特点：

****空间整合****，G1收集器采用标记整理算法，不会产生内存空间碎片。分配大对象时不会因为无法找到连续空间而提前触发下一次GC。

****可预测停顿****，这是G1的另一大优势，降低停顿时间是G1和CMS的共同关注点，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N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。

上面提到的垃圾收集器，收集的范围都是整个新生代或者老年代，而G1不再是这样。使用G1收集器时，Java堆的内存布局与其他收集器有很大差别，它将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔阂了，它们都是一部分（可以不连续）Region的集合。



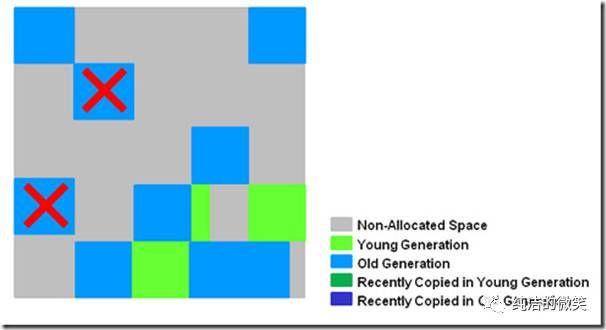
G1的新生代收集跟ParNew类似，当新生代占用达到一定比例的时候，开始出发收集。和CMS类似，G1收集器收集老年代对象会有短暂停顿。

收集步骤：

1、标记阶段，首先初始标记(Initial-Mark),这个阶段是停顿的(Stop the World Event)，并且会触发一次普通Mintor GC。对应GC log:GC pause (young) (inital-mark)

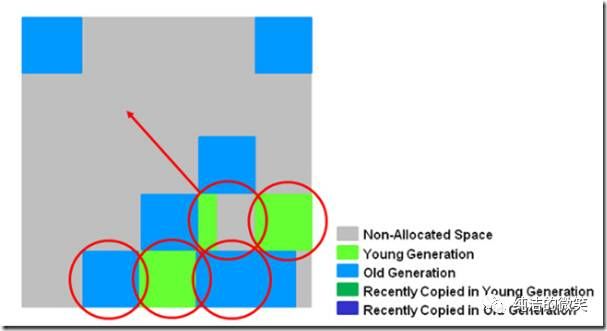
2、Root Region Scanning，程序运行过程中会回收survivor区(存活到老年代)，这一过程必须在young GC之前完成。

3、Concurrent Marking，在整个堆中进行并发标记(和应用程序并发执行)，此过程可能被young GC中断。在并发标记阶段，若发现区域对象中的所有对象都是垃圾，那个这个区域会被立即回收(图中打X)。同时，并发标记过程中，会计算每个区域的对象活性(区域中存活对象的比例)。

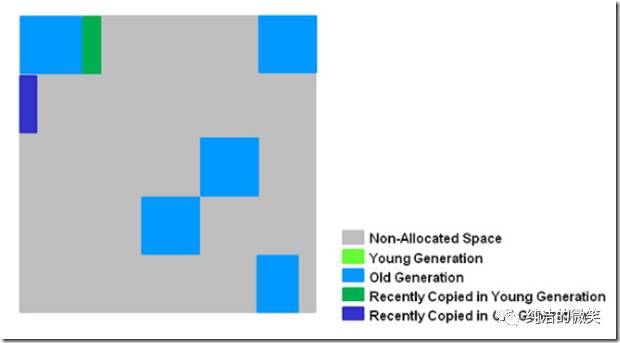


4、Remark, 再标记，会有短暂停顿(STW)。再标记阶段是用来收集 并发标记阶段 产生新的垃圾(并发阶段和应用程序一同运行)；G1中采用了比CMS更快的初始快照算法:snapshot-at-the-beginning (SATB)。

5、Copy/Clean up，多线程清除失活对象，会有STW。G1将回收区域的存活对象拷贝到新区域，清除Remember Sets，并发清空回收区域并把它返回到空闲区域链表中。



6、复制/清除过程后。回收区域的活性对象已经被集中回收到深蓝色和深绿色区域。



参考文章：jvm系列(三):GC算法 垃圾收集器——纯洁的微笑

## **四、其他 JVM 相关面试题整理**

#### ****1）64 位 JVM 中，int 的长度是多数？****

答：Java 中，int 类型变量的长度是一个固定值，与平台无关，都是 32 位或者 4 个字节。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。

#### ****2）怎样通过 Java 程序来判断 JVM 是 32 位 还是 64 位？****

答：Sun有一个Java System属性来确定JVM的位数：32或64：

sun.arch.data.model=32 // 32 bit JVM  
sun.arch.data.model=64 // 64 bit JVM

我可以使用以下语句来确定 JVM 是 32 位还是 64 位：

System.getProperty("sun.arch.data.model")

#### ****3）32 位 JVM 和 64 位 JVM 的最大堆内存分别是多数？****

答：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。

#### ****4）你能保证 GC 执行吗？****

答：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。

#### ****5）怎么获取 Java 程序使用的内存？堆使用的百分比？****

答：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。

#### ****6）Java 中堆和栈有什么区别？****

答：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。

