JVM 面试题

一.Java 类加载过程？

Java 类加载需要经历一下 7 个过程：

1. 加载

加载是类加载的第一个过程，在这个阶段，将完成一下三件事情：

- 通过一个类的全限定名获取该类的二进制流。
- 将该二进制流中的静态存储结构转化为方法去运行时数据结构。
- 在内存中生成该类的 Class 对象，作为该类的数据访问入口。

2. 验证

验证的目的是为了确保 Class 文件的字节流中的信息不回危害到虚拟机.在该阶段主要完成以下四钟验证:

- 文件格式验证：验证字节流是否符合 Class 文件的规范，如主次版本号是否在当前虚拟机范围内，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，如这个类是否有父类，是否集成了不被继承的类等。

- 字节码验证：是整个验证过程中最复杂的一个阶段，通过验证数据流和控制流的分析，确定程序语义是否正确，主要针对方法体的验证。如：方法中的类型转换是否正确，跳转指令是否正确等。
- 符号引用验证：这个动作在后面的解析过程中发生，主要是为了确保解析动作能正确执行。

3. 准备

准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中。

```
public static int value=123;//在准备阶段 value 初始值为 0 。在初始化阶段才会变为 123 。
```

4. 解析

该阶段主要完成符号引用到直接引用的转换动作。解析动作并不一定在初始化动作完成之前，也有可能在初始化之后。

5. 初始化

初始化时类加载的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由

虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的 Java 程序代码。

6. 使用

7. 卸载

二.描述一下 JVM 加载 Class 文件的原理机制?

Java 语言是一种具有动态性的解释型语言，类（Class）只有被加载到 JVM 后才能运行。当运行指定程序时，JVM 会将编译生成的 .class 文件按照需求和一定的规则加载到内存中，并组织成为一个完整的 Java 应用程序。这个加载过程是由类加载器完成，具体来说，就是由 ClassLoader 和它的子类来实现的。类加载器本身也是一个类，其实质是把类文件从硬盘读取到内存中。

类的加载方式分为隐式加载和显示加载。隐式加载指的是程序在使用 new 等方式创建对象时，会隐式地调用类的加载器把对应的类加载到 JVM 中。显示加载指的是通过直接调用 class.forName() 方法来把所需的类加载到 JVM 中。

任何一个工程项目都是由许多类组成的，当程序启动时，只把需要的类加载到 JVM 中，其他类只有被使用到的时候才会被加载，采用这种方法一方面可以加快加载速度，另一方面可以节约程序运行时对内存的开销。此外，在 Java 语言中，每个类或接口都对应一个 .class 文件，这些文件可以被看成是一个个可以被动态加载的

单元，因此当只有部分类被修改时，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。

在 Java 语言中，类的加载是动态的，它并不会一次性将所有类全部加载后再运行，而是保证程序运行的基础类（例如基类）完全加载到 JVM 中，至于其他类，则在需要的时候才加载。

类加载的主要步骤：

- 装载。根据查找路径找到相应的 class 文件，然后导入。
- 链接。链接又可分为 3 个小步：
- 检查，检查待加载的 class 文件的正确性。
- 准备，给类中的静态变量分配存储空间。
- 解析，将符号引用转换为直接引用（这一步可选）
- 初始化。对静态变量和静态代码块执行初始化工作。

三 Java 内存分配。

- **寄存器：**我们无法控制。
- **静态域**：static 定义的静态成员。
- **常量池：**编译时被确定并保存在 .class 文件中的（final）常量值和一些文本修饰的符号引用（类和接口的全限定名，字段的名称和描述符，方法和名称和描述符）。
- **非 RAM 存储：**硬盘等永久存储空间。

- **堆内存**：new 创建的对象和数组，由 Java 虚拟机自动垃圾回收器管理,存取速度慢。
- **栈内存：**基本类型的变量和对象的引用变量（堆内存空间的访问地址），速度快，可以共享，但是大小与生存期必须确定，缺乏灵活性。

1. Java 堆的结构是什么样子的？什么是堆中的永久代（Perm Gen space）？

JVM 的堆是运行时数据区，所有类的实例和数组都是在堆上分配内存。它在 JVM 启动的时候被创建。对象所占的堆内存是由自动内存管理系统也就是垃圾收集器回收。

堆内存是由存活和死亡的对象组成的。存活的对象是应用可以访问的，不会被垃圾回收。死亡的对象是应用不可访问尚且还没有被垃圾收集器回收掉的对象。一直到垃圾收集器把这些 对象回收掉之前，他们会一直占据堆内存空间。

四.GC 是什么？为什么要有 GC？

GC 是垃圾收集的意思（GabageCollection），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。

五. 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。

六. 如何判断一个对象是否存活？（或者 GC 对象的判定方法）

判断一个对象是否存活有两种方法：

1. 引用计数法

所谓引用计数法就是给每一个对象设置一个引用计数器，每当有一个地方引用这个对象时，就将计数器加一，引用失效时，计数器就减一。当一个对象的引用计数器为零时，说明此对象没有被引用，也就是“死对象”，将会被垃圾回收.

引用计数法有一个缺陷就是无法解决循环引用问题，也就是说当对象 A 引用对象 B，对象 B 又引用者对象 A，那么此时 A、B 对象的引用计数器都不为零，也就造成无法完成垃圾回收，所以主流的虚拟机都没有采用这种算法。

2. 可达性算法（引用链法）

该算法的思想是：从一个被称为 GC Roots 的对象开始向下搜索，如果一个对象到 GC Roots 没有任何引用链相连时，则说明此对象不可用。

在 Java 中可以作为 GC Roots 的对象有以下几种：

- 虚拟机栈中引用的对象
- 方法区类静态属性引用的对象
- 方法区常量池引用的对象
- 本地方法栈 JNI 引用的对象

虽然这些算法可以判定一个对象是否能被回收，但是当满足上述条件时，一个对象比不一定会被回收。当一个对象不可达 GC Root 时，这个对象并不会立马被回收，而是出于一个死缓的阶段，若要被真正的回收需要经历两次标记.

如果对象在可达性分析中没有与 GC Root 的引用链，那么此时就会被第一次标记并且进行一次筛选，筛选的条件是是否有必要执行 finalize() 方法。当对象没有覆盖 finalize() 方法或者已被虚拟机调用过，那么就认为是没必要的。 如果该对象有必要执行 finalize() 方法，那么这个对象将会放在一个称为 F-Queue 的对队列中，虚拟机会触发一个 Finalize() 线程去执行，此线程是低优先级的，并且虚拟机不会承诺一直等待它运行完，这是因为如果 finalize() 执行缓慢或者发生了死锁，那么就会造成 F-Queue 队

列一直等待，造成了内存回收系统的崩溃。GC 对处于 F-Queue 中的对象进行第二次被标记，这时，该对象将被移除” 即将回收” 集合，等待回收。

七. 垃圾回收的优点和原理。并考虑 2 种回收机制。

Java 语言中一个显著的特点就是引入了垃圾回收机制，使 C++ 程序员最头疼的内存管理的问题迎刃而解，它使得 Java 程序员在编写程序的时候不再需要考虑内存管理。由于有个垃圾回收机制，Java 中的对象不再有 “作用域” 的概念，只有对象的引用才有"作用域"。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级别的线程运行，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楚和回收，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。

回收机制有分代复制垃圾回收和标记垃圾回收，增量垃圾回收。

八. 垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收？

对于 GC 来说，当程序员创建对象时，GC 就开始监控这个对象的地址、大小以及使用情况。通常，GC 采用有向图的方式记录和管理堆（heap）中的所有对象。通过这种方式确定哪些对象是” 可达的” ，哪些对象是” 不可达的” 。当 GC 确定一些对象为 “不可达” 时，GC 就有责任回收这些内存空间。可以。程序员可以手

动执行 System.gc()，通知 GC 运行，但是 Java 语言规范并不保证 GC 一定会执行。

九. Java 中会存在内存泄漏吗，请简单描述。

所谓内存泄露就是指一个不再被程序使用的对象或变量一直被占据在内存中。Java 中有垃圾回收机制，它可以保证一对象不再被引用的时候，即对象变成了孤儿的时候，对象将自动被垃圾回收器从内存中清除掉。由于 Java 使用有向图的方式进行垃圾回收管理，可以消除引用循环的问题，例如有两个对象，相互引用，只要它们和根进程不可达的，那么 GC 也是可以回收它们的，例如下面的代码可以看到这种情况的内存回收：

```
import java.io.IOException;
public class GarbageTest {
    /**
     * @param args
     * @throws IOException
     */
    public static void main(String[] args) throws IOException
{
        // TODO Auto-generated method stub
        try {
            gcTest();
        } catch (IOException e) {
            // TODO Auto-generated catch block
```

```
            e.printStackTrace();
        }
        System.out.println("has exited gcTest!");
        System.in.read();
        System.in.read();
        System.out.println("out begin gc!");
        for(int i=0;i<100;i++)
        {
            System.gc();
            System.in.read();
            System.in.read();
        }
    }
    private static void gcTest() throws IOException {
        System.in.read();
        System.in.read();
        Person p1 = new Person();
        System.in.read();
        System.in.read();
        Person p2 = new Person();
        p1.setMate(p2);
        p2.setMate(p1);
        System.out.println("before exit gctest!");
        System.in.read();
```

```
        System.in.read();
        System.gc();
        System.out.println("exit gctest!");
    }
    private static class Person
    {
        byte[] data = new byte[20000000];
        Person mate = null;
        public void setMate(Person other)
        {
            mate = other;
        }
    }
}
```

Java 中的内存泄露的情况：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，通俗地说，就是程序员可能创建了一个对象，以后一直不再使用这个对象，这个对象却一直被引用，即这个对象无用但是却无法被垃圾回收器回收的，这就是 java 中可能出现内存泄露的情况，例如，缓存系统，我们加载了一个对象放在缓存中（例如放在一个全局 map 对象中），然后一直不再使用它，这个对象一直被缓存引用，但却不再被使用。

检查 Java 中的内存泄露，一定要让程序将各种分支情况都完整执行到程序结束，然后看某个对象是否被使用过，如果没有，则才能判定这个对象属于内存泄露。

如果一个外部类的实例对象的方法返回了一个内部类的实例对象，这个内部类对象被长期引用了，即使那个外部类实例对象不再被使用，但由于内部类持久外部类的实例对象，这个外部类对象将不会被垃圾回收，这也会造成内存泄露。

下面内容来自于网上（主要特点就是清空堆栈中的某个元素，并不是彻底把它从数组中拿掉，而是把存储的总数减少，本人写得可以比这个好，在拿掉某个元素时，顺便也让它从数组中消失，将那个元素所在的位置的值设置为 null 即可）：

我实在想不到比那个堆栈更经典的例子了，以致于我还要引用别人的例子，下面的例子不是我想到的，是书上看到的，当然如果没有在书上看到，可能过一段时间我自己也想的到，可是那时我说是我自己想到的也没有人相信的。

```
public class Stack {
    private Object[] elements=new Object[10];
    private int size = 0;
    public void push(Object e){
        ensureCapacity();
        elements[size++] = e;
```

```
    }

    public Object pop(){
        if( size == 0) throw new EmptyStackException();
        return elements[--size];
    }

    private void ensureCapacity(){
        if(elements.length == size){
            Object[] oldElements = elements;
            elements = new Object[2 * elements.length+1];
            System.arraycopy(oldElements,0, elements, 0,
size);
            }
    }
}
```

上面的原理应该很简单，假如堆栈加了 10 个元素，然后全部弹出来，虽然堆栈是空的，没有我们要的东西，但是这是个对象是无法回收的，这个才符合了内存泄露的两个条件：无用，无法回收。但是就是存在这样的东西也不一定会导致什么样的后果，如果这个堆栈用的比较少，也就浪费了几个 K 内存而已，反正我们的内存都上 G 了，哪里会有什么影响，再说这个东西很快就会被回收的，有什么关系。下面看两个例子。

```
public class Bad{
```

```
    public static Stack s=Stack();
    static{
        s.push(new Object());
        s.pop(); //这里有一个对象发生内存泄露
        s.push(new Object()); //上面的对象可以被回收了，等于是自
愈了
    }
}
```

因为是 static，就一直存在到程序退出，但是我们也可以看到它有自愈功能，就是说如果你的 Stack 最多有 100 个对象，那么最多也就只有 100 个对象无法被回收其实这个应该很容易理解，Stack 内部持有 100 个引用，最坏的情况就是他们都是无用的，因为我们一旦放新的进取，以前的引用自然消失！

内存泄露的另外一种情况：当一个对象被存储进 HashSet 集合中以后，就不能修改这个对象中的那些参与计算哈希值的字段了，否则，对象修改后的哈希值与最初存储进 HashSet 集合中时的哈希值就不同了，在这种情况下，即使在 contains 方法使用该对象的当前引用作为的参数去 HashSet 集合中检索对象，也将返回找不到对象的结果，这也会导致无法从 HashSet 集合中单独删除当前对象，造成内存泄露。

十. 深拷贝和浅拷贝。

简单来讲就是复制、克隆。

```
Person p=new Person("张三");
```

浅拷贝就是对对象中的数据成员进行简单赋值，如果存在动态成员或者指针就会报错。

深拷贝就是对对象中存在的动态成员或指针重新开辟内存空间。

十一. System.gc() 和 Runtime.gc() 会做什么事情？

这两个方法用来提示 JVM 要进行垃圾回收。但是，立即开始还是延迟进行垃圾回收是取决于 JVM 的。

十二. finalize() 方法什么时候被调用？析构函数（finalization）的目的是什么？

垃圾回收器（garbage colector）决定回收某对象时，就会运行该对象的 finalize() 方法 但是在 Java 中很不幸，如果内存总是充足的，那么垃圾回收可能永远不会进行，也就是说 filalize() 可能永远不被执行，显然指望它做收尾工作是靠不住的。 那么 finalize() 究竟是做什么的呢？ 它最主要的用途是回收特殊渠道申请的内存。Java 程序有垃圾回收器，所以一般情况下内存问题不用程序员操心。但有一种 JNI（Java Native Interface）调用 non-Java 程序（C 或 C++），finalize() 的工作就是回收这部分的内存。

十三. 如果对象的引用被置为 null ,垃圾收集器是否会立即释放对象占用的内存？

不会，在下一个垃圾回收周期中，这个对象将是可被回收的。

十四. 什么是分布式垃圾回收（DGC）？它是如何工作的？

DGC 叫做分布式垃圾回收。RMI 使用 DGC 来做自动垃圾回收。因为 RMI 包含了跨虚拟机的远程对象的引用 ,垃圾回收是很困难的。DGC 使用引用计数算法来给远程对象提供自动内存管理。

十五. 串行（serial）收集器和吞吐量（throughput）收集器的区别是什么？

吞吐量收集器使用并行版本的新生代垃圾收集器 ,它用于中等规模和大规模数据的应用程序。 而串行收集器对大多数的小应用（在现代处理器上需要大概 100M 左右的内存）就足够了。

十六. 在 Java 中，对象什么时候可以被垃圾回收？

当对象对当前使用这个对象的应用程序变得不可触及的时候 ,这个对象就可以被回收了。

十七. 简述 Java 内存分配与回收策率以及 Minor GC 和 Major GC。

- 对象优先在堆的 Eden 区分配
- 大对象直接进入老年代
- 长期存活的对象将直接进入老年代

当 Eden 区没有足够的空间进行分配时，虚拟机会执行一次 Minor GC。Minor GC 通常发生在新生代的 Eden 区，在这个区的对象生存期短，往往发生 Gc 的频率较高，回收速度比较快；Full GC/Major GC 发生在老年代，一般情况下，触发老年代 GC 的时候不会触发 Minor GC，但是通过配置，可以在 Full GC 之前进行一次 Minor GC 这样可以加快老年代的回收速度。

十八. 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）。

注：Java 8 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。

十九. Java 中垃圾收集的方法有哪些？

标记 - 清除：这是垃圾收集算法中最基础的，根据名字就可以知道，它的思想就是标记哪些要被回收的对象，然后统一回收。这种方法很简单，但是会有两个主要问题：

1. 效率不高，标记和清除的效率都很低；
2. 会产生大量不连续的内存碎片，导致以后程序在分配较大的对象时，由于没有充足的连续内存而提前触发一次 GC 动作。

复制算法：为了解决效率问题，复制算法将可用内存按容量划分为相等的两部分，然后每次只使用其中的一块，当一块内存用完时，

就将还存活的对象复制到第二块内存上，然后一次性清楚完第一块内存，再将第二块上的对象复制到第一块。但是这种方式，内存的代价太高，每次基本上都要浪费一般的内存。

于是将该算法进行了改进，内存区域不再是按照 1:1 去划分，而是将内存划分为 8:1:1 三部分，较大那份内存交 Eden 区，其余是两块较小的内存区叫 Survior 区。每次都会优先使用 Eden 区，若 Eden 区满，就将对象复制到第二块内存区上，然后清除 Eden 区，如果此时存活的对象太多，以至于 Survivor 不够时，会将这些对象通过分配担保机制复制到老年代中。（java 堆又分为新生代和老年代）

标记 - 整理：该算法主要是为了解决标记 - 清除，产生大量内存碎片的问题；当对象存活率较高时，也解决了复制算法的效率问题。它的不同之处就是在清除对象的时候现将可回收对象移动到一端，然后清除掉端边界以外的对象，这样就不会产生内存碎片了。

分代收集：现在的虚拟机垃圾收集大多采用这种方式，它根据对象的生存周期，将堆分为新生代和老年代。在新生代中，由于对象生存期短，每次回收都会有大量对象死去，那么这时就采用复制算法。老年代里的对象存活率较高，没有额外的空间进行分配担保。

二十. 什么是类加载器，类加载器有哪些？

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。

主要有一下四种类加载器：

- 启动类加载器(Bootstrap ClassLoader)用来加载 Java 核心类库，无法被 Java 程序直接引用。
- 扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。
- 系统类加载器（ system class loader ）：它根据 Java 应用的类路径(CLASSPATH)来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader() 来获取它。
- 用户自定义类加载器，通过继承 java.lang.ClassLoader 类的方式实现。

二十一. 类加载器双亲委派模型机制？

当一个类收到了类加载请求时，不会自己先去加载这个类，而是将其委派给父类，由父类去加载，如果此时父类不能加载，反馈给子类，由子类去完成类的加载。