总结的JVM面试题

JVM运行内存的分类

- 程序计数器：当前线程所执行的字节码的行号指示器，用于记录下一条要运行的指令，线程私有
  注：如果正在执行的是Native方法，计数器值则为空
- Java虚拟栈：存放基本数据类型、对象的引用、方法出口等，线程私有
- Native方法栈：和虚拟栈相似，只不过它服务于Native方法，线程私有
- Java堆：java内存最大的一块，所有对象实例、数组都存放在java堆，GC回收的地方，线程共享
- 方法区：存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码数据等。（即永久带），回收目标主要是常量池的回收和类型的卸载，各线程共享

Java内存堆和栈区别

- 栈内存用来存储基本类型的变量和对象的引用变量，堆内存用来存储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
- 栈内存归属于单个线程，每个线程都会有一个栈内存，其存储的变量只能在其所属线程中可见，即栈内存可以理解成线程的私有内存，堆内存中的对象对所有线程可见。堆内存中的对象可以被所有线程访问
- 如果栈内存没有可用的空间存储方法调用和局部变量，JVM会抛出java.lang.StackOverFlowError，如果是堆内存没有可用的空间存储生成的对象，JVM会抛出java.lang.OutOfMemoryError
- 栈的内存要远远小于堆内存，如果你使用递归的话，那么你的栈很快就会充满，-Xss选项设置栈内存的大小。-Xms选项可以设置堆的开始时的大小

Java四引用

- 强引用（StrongReference）强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题
- 软引用（SoftReference）
  如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中
- 弱引用（WeakReference）
  弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。
  弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中
- 虚引用（PhantomReference）
  虚引用在任何时候都可能被垃圾回收器回收，主要用来**跟踪对象被垃圾回收器回收的活动，被回**

收时会收到一个系统通知。虚引用与软引用和弱引用的一个区别在于：虚引用**必须**和引用队列（ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。

GC回收机制

- Java中对象是采用new或者反射的方法创建的，这些对象的创建都是在堆（Heap）中分配的，所有对象的回收都是由Java虚拟机通过垃圾回收机制完成的。GC为了能够正确释放对象，会监控每个对象的运行状况，对他们的申请、引用、被引用、赋值等状况进行监控
- Java程序员不用担心内存管理，因为垃圾收集器会自动进行管理
- 可以调用下面的方法之一：System.gc() 或Runtime.getRuntime().gc() ，但JVM可以屏蔽掉显示的垃圾回收调用

GC 标记对象的死活

- 引用计数法：给对象添加一个引用计数器,没当被引用的时候,计数器的值就加一。引用失效的时候减一,当计数器的值为 0 的时候就表示改对象可以被 GC 回收了，弊端:A->B,B->A,那么 AB 将永远不会被回收了。也就是引用有环的情况
- 根搜索算法(可达性算法) GC Roots Tracing：通过一个叫 GC Roots 的对象作为起点,从这些结点开始向下搜索,搜索所走过的路径称为引用链,当一个对象没有与任何的引用链相连的时候则改对象就可以被。 GC 回收回收了Roots 包括：java 虚拟机栈中引用的对象,本地方法栈中引用的对象,方法区中常量引用的对象,方法区中静态属性引用的对象
    - 在Java语言里，可作为GC Roots的对象包括以下几种：

```
虚拟机栈（栈帧中的本地变量表）中的引用的对象
方法区中的类静态属性引用的对象
方法区中的常量引用的对象。
本地方法栈中JNI(即一般说的Native方法)的引用的对象。
```

GC回收算法

- 标记-清除法：标记出没有用的对象，然后一个一个回收掉
    - 缺点：标记和清除两个过程效率不高，产生内存碎片导致需要分配较大对象时无法找到足够的连续内存而需要触发一次GC操作
- 复制算法: 按照容量划分二个大小相等的内存区域，当一块用完的时候将活着的对象复制到另一块上，然后再把已使用的内存空间一次清理掉
    - 缺点：将内存缩小为了原来的一半
- 标记-整理法：标记出没有用的对象，让所有存活的对象都向一端移动，然后直接清除掉端边界以外的内
    - 优点：解决了标记- 清除算法导致的内存碎片问题和在存活率较高时复制算法效率低的问题。
- 分代回收：根据对象存活周期的不同将内存划分为几块，一般是新生代和老年代，新生代基本采用复制算法，老年代采用标记整理算法

MinorGC&FullGC

- 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，一般采用复制-回收算法
- 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，所采用的是标记-清除算法

内存分配与回收策略

- 结构（堆大小 = 新生代 + 老年代 ）：
  - 新生代(1/3)(初始对象，生命周期短)：Eden 区、survivior 0、survivior 1 （ 8 : 1 : 1）
  - 老年代(2/3)(长时间存在的对象)
- 一般小型的对象都会在 Eden 区上分配，如果Eden区无法分配，那么尝试把活着的对象放到survivor0中去（Minor GC）
  - 如果survivor0可以放入，那么放入之后清除Eden区
  - 如果survivor0不可以放入，那么尝试把Eden和survivor0的存活对象放到survivor1中
    - 如果survivor1可以放入，那么放入survivor1之后清除Eden和survivor0，之后再把survivor1中的对象复制到survivor0中，保持survivor1一直为空。
    - 如果survivor1不可以放入，那么直接把它们放入到老年代中，并清除Eden和survivor0，这个过程也称为分配担保（Full GC）
- 大对象、长期存活的对象则直接进入老年代
- 动态对象年龄判定
- 空间分配担保，Full GC...

GC垃圾收集器

- Serial New收集器是针对新生代的收集器，采用的是复制算法
- Parallel New（并行）收集器，新生代采用复制算法，老年代采用标记整理
- Parallel Scavenge（并行）收集器，针对新生代，采用复制收集算法
- Serial Old（串行）收集器，新生代采用复制，老年代采用标记清理
- Parallel Old（并行）收集器，针对老年代，标记整理
- CMS收集器，基于标记清理
- G1收集器(JDK)：整体上是基于标记清理，局部采用复制
- 综上：新生代基本采用复制算法，老年代采用标记整理算法。cms采用标记清理

Java类加载机制

- 概念：
  - 虚拟机把描述类的数据文件（字节码）加载到内存，并对数据进行验证、准备、解析以及类初始化，最终形成可以被虚拟机直接使用的java类型（java.lang.Class对象）
- 类的生命周期：
  - 加载过程：通过一个类的全限定名来获取定义此类的二进制字节流，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。在内存中(方法区)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访问入口；

    - 验证过程：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，文件格式验证、元数据验证、字节码验证、符号引用验证
    - 准备过程：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进行分配
    - 解析阶段：虚拟机将常量池内的符号引用替换为直接引用的过程
    - 初始化阶段：类初始化阶段是类加载过程的最后一步。初始化阶段就是执行类构造器()方法的过程
    - 使用阶段：
    - 卸载阶段：
- Java类加载器：
    - 类加载器负责加载所有的类，同一个类(一个类用其全限定类名(包名加类名)标志)只会被加载一次
    - Bootstrap ClassLoader:根类加载器，负责加载java的核心类，它不是java.lang.ClassLoader的子类，而是由JVM自身实现
    - Extension ClassLoader:扩展类加载器，扩展类加载器的加载路径是JDK目录下jre/lib/ext,扩展类的getParent()方法返回null,实际上扩展类加载器的父类加载器是根加载器，只是根加载器并不是Java实现的
    - System ClassLoader:系统(应用)类加载器，它负责在JVM启动时加载来自java命令的-classpath选项、java.class.path系统属性或CLASSPATH环境变量所指定的jar包和类路径。程序可以通过getSystemClassLoader()来获取系统类加载器。系统加载器的加载路径是程序运行的当前路径
- 双亲委派模型的工作过程：
    - 首先会先查找当前ClassLoader是否加载过此类，有就返回；
    - 如果没有，查询父ClassLoader是否已经加载过此类，如果已经加载过,就直接返回Parent加载的类；
    - 如果整个类加载器体系上的ClassLoader都没有加载过，才由当前ClassLoader加载(调用findClass)，整个过程类似循环链表一样。
- 双亲委托机制的作用：
    - 共享功能，一些Framework层级的类一旦被顶层的ClassLoader加载过就缓存在内存里面，以后任何地方用到都不需要重新加载。
    - 隔离功能，保证java/Android核心类库的纯净和安全，防止恶意加载。
- 如何打破双亲委派模型?
    - 双亲委派模型的逻辑都在`loadClass()`中，重写`loaderClass()`，一般是重写`findClass()`的
    - 系统自带的三个类加载器都加载特定目录下的类，如果我们自己的类加载器放在一个特殊的目录，那么系统的加载器就无法加载，也就是最终还是由我们自己的加载器加载
- 自定义ClassLoader：
    - `loadClass(String name,boolean resolve)`：根据指定的二进制名称加载类
    - `findClass(String name)`： 根据二进制名称来查找类
    - 直接使用或继承已有的ClassLoader实现：`java.net.URLClassLoader`、`java.security.SecureClassLoader`、`java.rmi.server.RMIClassLoader`
    - 在调用`loadClass()`，会先根据委派模型在父加载器中加载，如果加载失败，则会调用自

己的findClass方法来完成加载

引起类加载操作的五个行为

- 遇到new、getstatic、putstatic或invokestatic这四条字节码指令
- 反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化
- 子类初始化的时候，如果其父类还没初始化，则需先触发其父类的初始化
- 虚拟机执行主类的时候(有 main(string[] args))
- JDK1.7 动态语言支持

Java对象创建时机

- 使用new关键字创建对象
- 使用Class类的newInstance方法(反射机制)
- 使用Constructor类的newInstance方法(反射机制)
- 使用Clone方法创建对象
- 使用(反)序列化机制创建对象