## 一、垃圾收集器与内存分配策略

### 1、怎样判断对象已经死亡？

**引用计数法？**（Reference Counting）：给对象添加一个引用计数器，每当有地方引用它时，计数器加1，当引用失效时，计数器就减1，任何时刻计数器为0的对象就是不可能再被使用的。

jvm没有选择引用计数算法来管理内存，最主要的原因是它很难解决对象之间的相互循环引用的问题。

**可达性分析**：java，c#,都是通过可达性分析来判定对象是否存活的，这个算法的基本思路就是通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径就称为引用链（Reference Chain）,当一个对象到GC Roots没有任何引用链相连（用图论来说，就是从GC Roots到这个对象不可达）时，证明这个对象是不可用的。

在java中，可作为GC Roots的对象包括几种：

1、虚拟机栈（栈帧中的本地变量表）中引用的对象

2、方法区中类静态属性引用的对象

3、方法区中常量引用的对象

4、本地方法栈中JNI(即一般说的Native方法)，引用的对象。

**引用分类：JDK1.2之后**

**强引用**：只要强引用存在，垃圾收集器永远不会回收掉被引用的对象。

**软引用**：在系统将要发生内存溢出异常之前，将这些对象列进回收范围，如果这次回收还没有足够的内存。

**弱引用**：被弱引用关联的对象只能生存到下一次垃圾收集器发生之前，当垃圾回收器工作时，无论内存是否足够，都会回收掉只被弱引用关联的对象。

**虚引用**：一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例，为一个对象设置虚引用的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。

**判断死亡**

一个对象死亡，至少要经历两次标记

**第一次**：如果对象在进行可达性分析后发现没有与GC Roots相连接的引用链，那么将会进行第一次标记，并且进行一次筛选，**筛选的目的是判断对象是否有必要执行finalize（）方法。**

如果对象没有覆盖finalize（）方法或者finalize（）方法已经被虚拟机调用过，则没必要执行。

如果判断有必要执行，那么这个对象将会放置在F-Queue的队列之中，并在稍后由一个低优先级Finalizer线程去执行它，finalize（）方法是对象逃脱死亡命运的最后一次机会，

如果对象要在finalize()中成功拯救自己，只要重新与引用链上的任何一个对象建立关联即可。

**第二次**：GC将对F-Queue中的对象进行第二次小规模标记，如果对象这个时候还没有逃脱，那么基本上它就真的被回收了。

**注意：任何一个对象的finalize（）方法都只会被系统自动调用一次，如果对象面临下一次回收，它的finalize（）方法不会被再次执行。**

**尽量避免使用这种方式拯救对象。**

**回收方法区：（HotSpot虚拟机的永久代）**

回收内容：废弃常量和无用的类

**判断废弃常量:**加入一个字符串“abc”已经进入了常量池中，但是当前系统中没有任何一个String对象叫做”abc”的，换句话说，就是没有任何String对象引用常量池中的“abc”常量，也没有其他地方引用了这个字面量，那么这个“abc”常量就会被系统清理出常量池。

**判断无用的类：**

1、该类的所有实例都已经被回收，也就是java堆中不存在该类的任何实例。

2、加载该类的ClassLoader已经被回收。

3、该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

### 2、垃圾回收算法

**标记清除算法：**

首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。

**不足：**

1、标记和清除两个过程的效率都不高

2、标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另外一次垃圾收集动作。

**复制算法：**

把内存分为两块，每次只使用其中一块，当这一块内存用完了，就将还存活着的对象复制到另外一块上面，然后将内存空间一次清理掉

**不足：**

太浪费内存空间了

现代商业虚拟机都采用这种复制收集算法来回收新生代，研究表明，新生代中对象的98%都是“朝生夕死”的，所以并不需要按照1：1的比例来划分内存空间，而是将内存分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden和其中一块Survivor，当回收时，将Eden和Survivor中还存活着的对象一次性地复制到另外一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间，HotSpot虚拟机默认的Eden和Survivor大小比例为8：1，也就是每次新生代中可用内存空间为整个新生代容量的90%，只有10%的内存会被“浪费”，当Survivor空间不够用时，需要依赖其他内存（老年代）进行分配担保（Handle Promotion）。

## 二、类加载机制

### 1、什么是类加载器？

负责读取 Java 字节代码，并转换成java.lang.Class类的一个实例

### 2、类加载器与类的”相同“判断？

**类加载器的作用**：

1、加载类

2、确定类在java虚拟机中的唯一性

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。换言之，要判断两个类是否“相同”，前提是这两个类必须被同一个类加载器加载，否则这两个类不相同。

**类相同判断**

1、两个对象的判断可以使用equals，或者==，equals方法在基类Object中定义的，从源码可以看出，内部就是==的实现。

2、判断对象是否是某个类用instanceof,isInstance

3、如果两个类比较，可以使用isAssignForm，但是只能得到是否是自身或者子类的结果，不能完全确定是否相同。

### `3、类加载机制

**启动类加载器，Bootstrap ClassLoader**，加载JACA\_HOME\lib，或者被-Xbootclasspath参数限定的类，用来加载核心类库，该加载器无法直接获取。

**扩展类加载器，Extension ClassLoader**，加载\lib\ext，或者被java.ext.dirs系统变量指定的类

**应用程序类加载器，Application ClassLoader**，加载ClassPath中的类库

**自定义类加载器**，通过继承ClassLoader实现，一般是加载我们的自定义类，

### 4、双亲委派模型

双亲委派模型要求除了顶层启动类加载器外其余类加载器 都应该有自己的父类加载器，类加载器之间通过复用关系来复用父加载器代码。

**模型工作流程：**

1.当Application ClassLoader 收到一个类加载请求时，他首先不会自己去尝试加载这个类，而是将这个请求委派给父类加载器Extension ClassLoader去完成。

2.当Extension ClassLoader收到一个类加载请求时，他首先也不会自己去尝试加载这个类，而是将请求委派给父类加载器Bootstrap ClassLoader去完成。

3.如果Bootstrap ClassLoader加载失败(在<JAVA\_HOME>\lib中未找到所需类)，就会让Extension ClassLoader尝试加载。

4.如果Extension ClassLoader也加载失败，就会使用Application ClassLoader加载。

5.如果Application ClassLoader也加载失败，就会使用自定义加载器去尝试加载。

6.如果均加载失败，就会抛出**ClassNotFoundException**异常。

双亲委派模型的实现过程：

实现双亲委派模型的代码集中在java.lang.ClassLoader的loadClass（）的方法中。

1、检查请求加载的类是否已经被加载过了。

2、若没有被加载过，递归调用父类加载器loadClass()

3、父类加载器为空后就使用启动类加载器加载，如果父类加载器和启动类加载器均无法加载请求，则调用自身的加载功能。

**双亲委派模型的好处？**

Java类伴随其类加载器具备了带有优先级的层次关系，确保了在各种加载环境中的加载顺序。保证了运行的安全性。避免了同一个类被多次加载，每个加载器类都只能加载自己范围的类。

### 5、类加载过程

类加载分为三个步骤：加载，连接，初始化

其中连接又分为：验证、准备、解析

**加载：**根据一个类的全限定名，来读取此类的二进制字节流到JVM内部，将字节流所代表的静态存储结构转换为方法区的运行时数据结构。转换为一个与目标类型对应的 java.lang。Class对象。

**连接：**

验证：文件格式校验、元数据验证、字节码验证、符号引用验证

文件格式验证：验证字节流是否符合Class文件格式的规范；

元数据验证：对字节码描述的信息进行语义分析，以保证其描述的信息符合Java语言规范的要求；

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。

准备：为类中的所有的静态变量分配内存空间，并为其设置一个初始值。

解析：将常量池中所有的符号引用转为直接引用，这个阶段可以在初始化之后再执行。

**初始化：**

1、所有类变量初始化语句和静态代码块都会在编译时被存放到一个特殊的方法<clinit>中，即类/接口初始化方法，该方法只能在类加载的过程中由JVM调用。

2、如果超类还没有被初始化，那么优先对超类进行初始化，由JVM负责保证一个类的<clinit>方法执行之前，他的超类的<clinit>方法已经被执行。

3、JVM必须确保一个类在初始化过程中，如果多线程需要同时初始化它，仅仅只允许其中的一个线程对其执行初始化操作，其余线程必须等待

4、如果一个类没有声明任何的类变量，也没有静态代码块，那么可以没有类<clint>方法。

**何时触发初始化？**

1、为一个类型创建一个新的对象实例时（new、反射、序列化）

2、调用一个类型的静态方法。

3、调用一个类型或接口的静态字段时。

4、调用JavaAPI中的反射方法时。

5、初始化一个类的子类的时候。

6、JVM启动包含main方法的启动类时。

**自定义类加载器**

要创建用户自己的类加载器，只需要继承java.lang.ClassLoader类，然后覆盖它的findClass（String name）方法即可

如果要符合双亲委派规范，则重写findClass（）方法。

如果要破坏的话重写loadClass（）方法。

### 6、 JAVA热部署实现

热部署（hotswap）: 热部署是在不重启 Java 虚拟机的前提下，能自动侦测到 class 文件的变化，更新运行时 class 的行为.JAVA类是通过java虚拟机加载的，某个类的class文件在被classloader加载后，会生成对应的Class对象，之后就可以创建该类的实列。默认的虚拟机行为只会在启动时加载类，如果后期有一个类需要更新的 话，单纯替换编译的class文件，JVM是不会更新正在运行的class

如果要实现热部署，最根本的方法就是修改虚拟机的源代码，改变classloader的加载行为，使虚拟机能监听class文件的更新，重新加载class文件，这样的行为破坏性很大，为后续的JVM升级埋下了一个大坑。

JVM中的Class只有满足以下三个条件，才能被GC回收，也就是该Class被卸载。

1、该类的所有实例都已经被GC，也就是在JVM不存在该类的任何实例

2、加载该类的classloader已经被GC

3、该类的java.lang.Class对象没有在任何地方被引用。

### 7、类加载顺序

静态代码块—>非静态代码块—>构造代码块->构造方法

有父子类交替

### 8、可不可以自己写个String类

不可以，因为根据类加载的双亲委派机制，回去加载父类，父类发现冲突String就不再加载了。

### 9、能否在加载类的时候，对类的字节码进行修改

可以使用java探针技术：基于javaAgent和Java字节码注入技术

JavaAgent 是运行在 main方法之前的拦截器，它内定的方法名叫 premain ，也就是说先执行 premain 方法然后再执行 main 方法。在JVM加载class二进制文件的时候，利用ASM动态的修改加载的class文件。

Premain与main方法运行在同一JVM中。