第4章 Java虚拟机原理面试题汇总

摘要：本章内容，享学课堂在全面分析JVM原理中有系统化-全面完整的直播讲解，详情加微信：xxgfwx03

4.1 描述JVM类加载过程 ~lance

这道题想考察什么?

了解JVM是如何加载类的，并且通过JVM类加载过程能更直观了解掌握如APT注解处理器执行、热修复等技术的本质

考察的知识点

JVM类加载过程

考生如何回答

类加载的本质

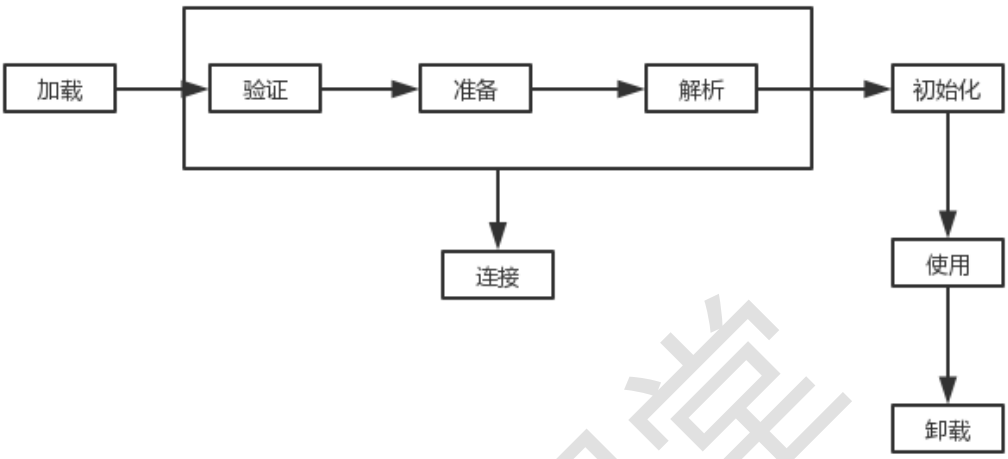
一般情况下，类的数据都是在`Class`文件中。将描述类的数据 从`Class`文件加载到内存 同时 对数据进行校验、转换解析 和 初始化，最终形成可被虚拟机直接使用的`Java`使用类型。

类加载过程

java类加载过程：**加载-->验证-->准备-->解析-->初始化**，之后类就可以被使用了。绝大部分情况下是按这

样的顺序来完成类的加载全过程的。但是是有例外的地方，解析也是可以在初始化之后进行的，这是为了支持

java的运行时绑定，并且在一个阶段进行过程中也可能会激活后一个阶段，而不是等待一个阶段结束再进行后一个阶段。



1.加载

加载时jvm做了这三件事：

1）通过一个类的全限定名来获取该类的二进制字节流

2）将这个字节流的静态存储结构转化为方法区运行时数据结构

3）在内存堆中生成一个代表该类的java.lang.Class对象，作为该类数据的访问入口

2.验证

验证、准备、解析这三步可以看做是一个连接的过程，将类的字节码连接到JVM的运行状态之中

验证是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，不会威胁到jvm的安全

验证主要包括以下几个方面的验证：

1）文件格式的验证，验证字节流是否符合Class文件的规范，是否能被当前版本的虚拟机处理

2）元数据验证，对字节码描述的信息进行语义分析，确保符合java语言规范

3）字节码验证 通过数据流和控制流分析，确定语义是合法的，符合逻辑的

4）符号引用验证 这个校验在解析阶段发生

3.准备

为类的静态变量分配内存，初始化为系统的初始值。对于final static修饰的变量，直接赋值为用户的定义值。如下面的例子：这里在准备阶段过后的初始值为0，而不是7

```
public static int a=7
```

4.解析

解析是将常量池内的符号引用转为直接引用（如物理内存地址指针）

5.初始化

到了初始化阶段，jvm才真正开始执行类中定义的java代码

1）初始化阶段是执行类构造器<clinit>()方法的过程。类构造器<clinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块(static块)中的语句合并产生的。

2）当初始化一个类的时候，如果发现其父类还没有进行过初始化、则需要先触发其父类的初始化。

3）虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确加锁和同步。

4.2 JVM中一次完整的GC流程是怎样的，对象如何晋升到老年代? ~lance

这道题想考察什么?

Java基础掌握情况，掌握对象回收过程以避免开发时出现内存问题

考察的知识点

GC机制

考生如何回答

完整的GC流程见《介绍下GC回收机制与分代回收策略》

通过《介绍下GC回收机制与分代回收策略》，我们了解到：

> 新对象的内存分配都是先在Eden区域中进行的，当Eden区域的空间不足于分配新对象时，就会触发年轻代上的垃圾回收，我们称之为"minor gc"。同时，每个对象都有一个“年龄”，这个年龄实际上指的就是该对象经历过的minor gc的次数。当对象的年龄足够大，当minor gc再次发生时，它会从Survivor内存区域中升级到年老代中。

因此对象晋升老年代的条件之一为：若年龄超过一定限制（如15），则被晋升到老年态。**即长期存活的对象进入老年代。**除此之外，以下情况都会导致对象晋升老年代：

- 大对象直接进入老年代

  多大由JVM参数`-XX:PretenureSizeThreshold=x`决定；

- 动态对象年龄判定

  当 Survivor 空间中相同年龄所有对象的大小总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，而不需要达到默认的分代年龄。

除了以上提到的几种情况外，其实还有一种可能导致对象晋升老年代：分配担保机制。

分配担保机制

什么是分配担保?

在发生Minor GC之前，虚拟机会检查老年代最大可用的连续空间是否大于新生代所有对象的总空间，

如果大于，则此次Minor GC是安全的

如果小于，则虚拟机会查看HandlePromotionFailure设置值是否允许担保失败。如果HandlePromotionFailure=true，那么会继续检查老年代最大可用连续空间是否大于历次晋升到老年代的对象的平均大小，如果大于，则尝试进行一次Minor GC，但这次Minor GC依然是有风险的；如果小于或者HandlePromotionFailure=false，则改为进行一次Full GC。

为什么要进行空间担保?

新生代一般采用复制收集算法，假如大量对象在Minor GC后仍然存活（最极端情况为内存回收后新生代中所有对象均存活），而Survivor空间是比较小的，这时就需要老年代进行分配担保，把Survivor无法容纳的对象放到老年代。老年代要进行空间分配担保，前提是老年代得有足够空间来容纳这些对象，但一共有多少对象在内存回收后存活下来是不可预知的，因此只好取之前每次垃圾回收后晋升到老年代的对象大小的平均值作为参考。使用这个平均值与老年代剩余空间进行比较，来决定是否进行Full GC来让老年代腾出更多空间。

Minor Gc后的对象太多无法放入Survivor区怎么办?

假如在发生gc的时候，eden区里有150MB对象存活，而Survivor区只有100MB，无法全部放入，这时就必须把这些**对象直接转移到老年代**里。

如果Minor gc后新生代的对象都存活下来，然后需要全部转移到老年代，但是老年代空间不够，怎么办?

这时如果设置了 "-XX:-HandlePromotionFailure"的参数，就会尝试判断，看老年代内存大小是否大于之前每一次Minor gc后进入老年代的对象的平均大小。比如说，之前Minor gc 平均10M左右的对象进入老年代，此时老年代可用内存大于10MB,那么大概率老年代空间是足够的。

1、如果判断老年代空间不够，或者是根本没设置这个参数，那就直接触发"Full GC（对整个堆回收，包含：年轻代、老年代）"，对老年代进行垃圾回收，腾出空间。

2、如果判断老年代空间足够，就冒风险尝试Minor gc。这时有以下几种可能。

- Minor Gc 后，剩余的存活对象大小，小于Survivor区，那就直接进入Survivor区。
- MInor Gc 后，剩余的存活对象大小，大于Survivor区，小于老年代可用内存，那就直接去老年代。
- Minor Gc后，大于Survivor，老年代，很不幸，就会发生"Handle Promotion Failure"的情况，触发"Full GC"。

如果 Full gc后老年代还是没有足够的空间存放剩余的存活对象，那么就会导致**“OOM”** 内存溢出。

总结

根据上文得知，实际上有四种情况可能会导致对象晋升老年代：

- 大对象直接进入老年代
- 年龄超过阈值
- 动态对象年龄判定
- 年轻代空间不足

4.3 介绍下GC回收机制与分代回收策略 ~lance

这道题想考察什么?

Java基础掌握情况，掌握对象回收过程以避免开发时出现内存问题

考察的知识点

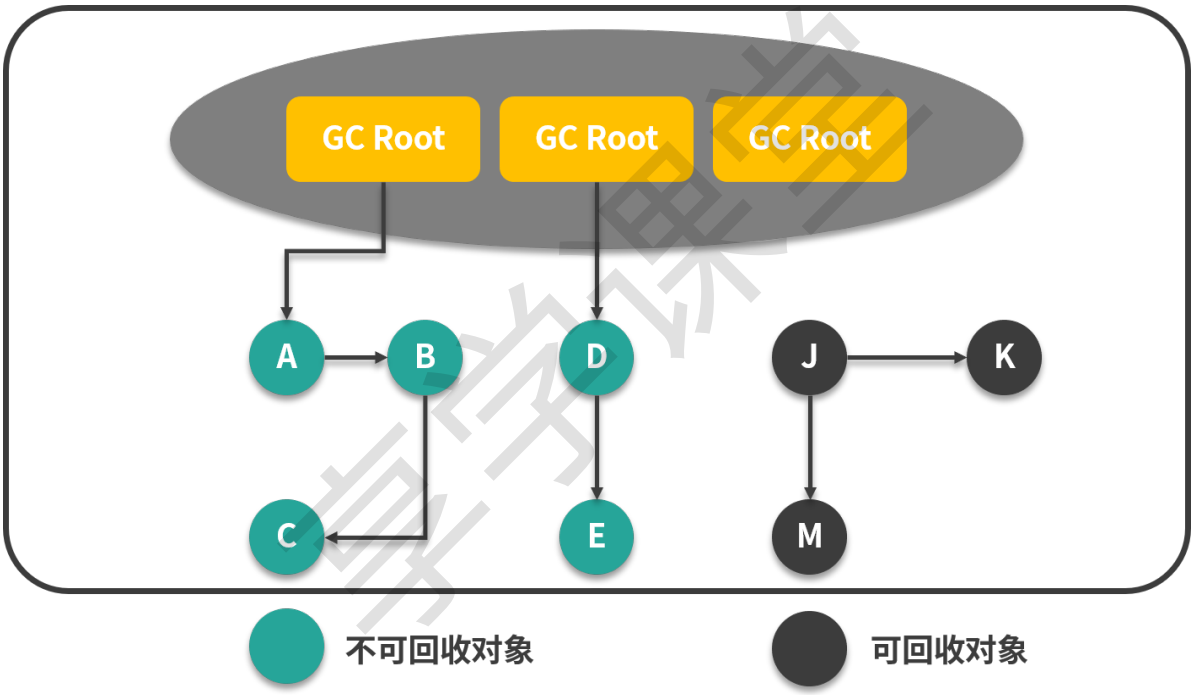
GC机制

考生如何回答

说到垃圾回收，首先要知道什么是“垃圾”，垃圾就是没有用的对象，那么怎样判定一个对象是不是垃圾（能不能被回收）？Java 虚拟机中使用一种叫作**可达性分析**的算法来决定对象是否可以被回收。

可达性分析

可达性分析就通过一组名为”GC Root"的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，最后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。



GC Root指的是：

- Java 虚拟机栈（局部变量表）中的引用的对象。也就是正在运行的方法中的局部变量所引用的对象
- 方法区中静态引用指向的对象。也就是类中的static修饰的变量所引用的对象
- 方法区中常量引用的对象。
- 仍处于存活状态中的线程对象。
- Native 方法中 JNI 引用的对象。

优点:

可达性分析可以解决引用计数器所不能解决的循环引用问题。即便对象a和b相互引用，只要从GC Roots出发无法到达a或者b，那么可达性分析便不会将它们加入存活对象合集之中。

缺点：

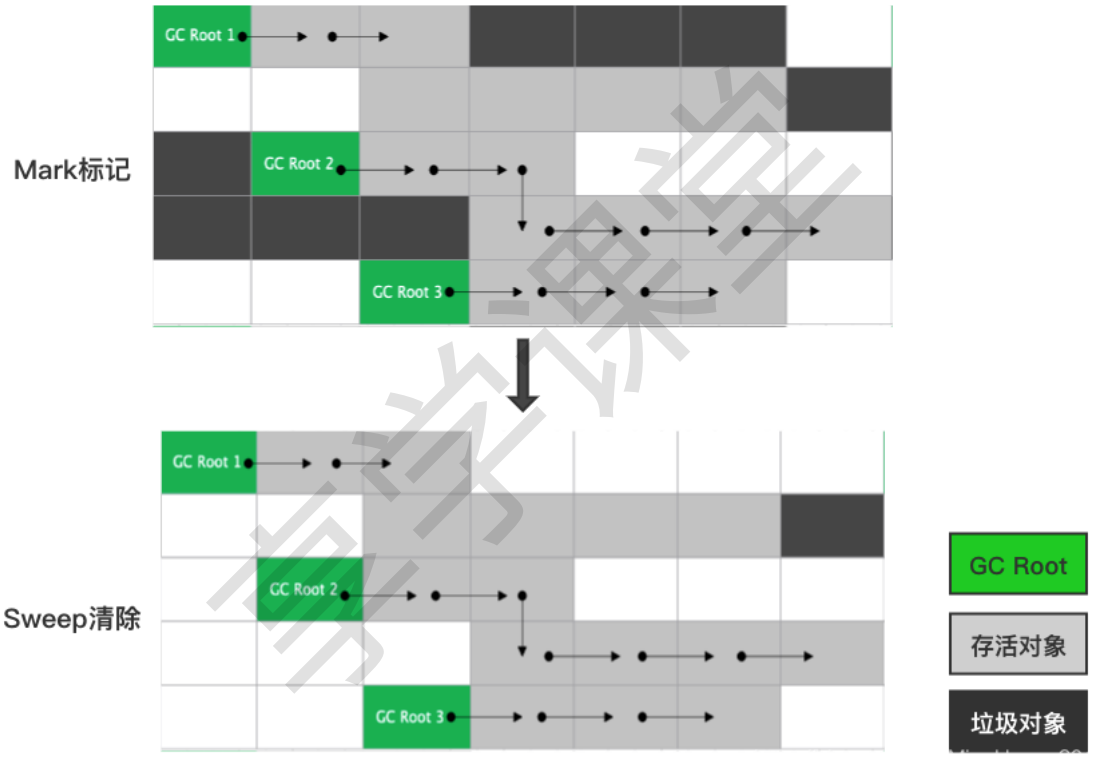
在多线程环境下，其他线程可能会更新已经访问过的对象中的引用，从而造成误报(将引用设置为null)或者漏报(将引用设置为未被访问过的对象)。误报并没有什么伤害，Java虚拟机至多损失了部分垃圾回收的机会。漏报则比较麻烦，因为垃圾回收器可能回收事实上仍被引用的对象内存。 一旦从原引用访问已经被回收了的对象，则很有可能会直接导致Java虚拟机崩溃。

垃圾回收算法

在标记出对象是否可被回收后，接下来就需要对可回收对象进行回收。基本的回收算法有：标记-清理、标记-整理与复制算法。

标记清除算法

从"GC Roots"集合开始，将内存整个遍历一次，保留所有可以被 GC Roots 直接或间接引用到的对象，而剩下的对象都当作垃圾对待并回收，过程分为 **标记** 和 **清除** 两个步骤。



- 优点：实现简单，不需要将对象进行移动。
- 缺点：这个算法需要中断进程内其他组件的执行（stop the world），并且可能产生内存碎片，提高了垃圾回收的频率。

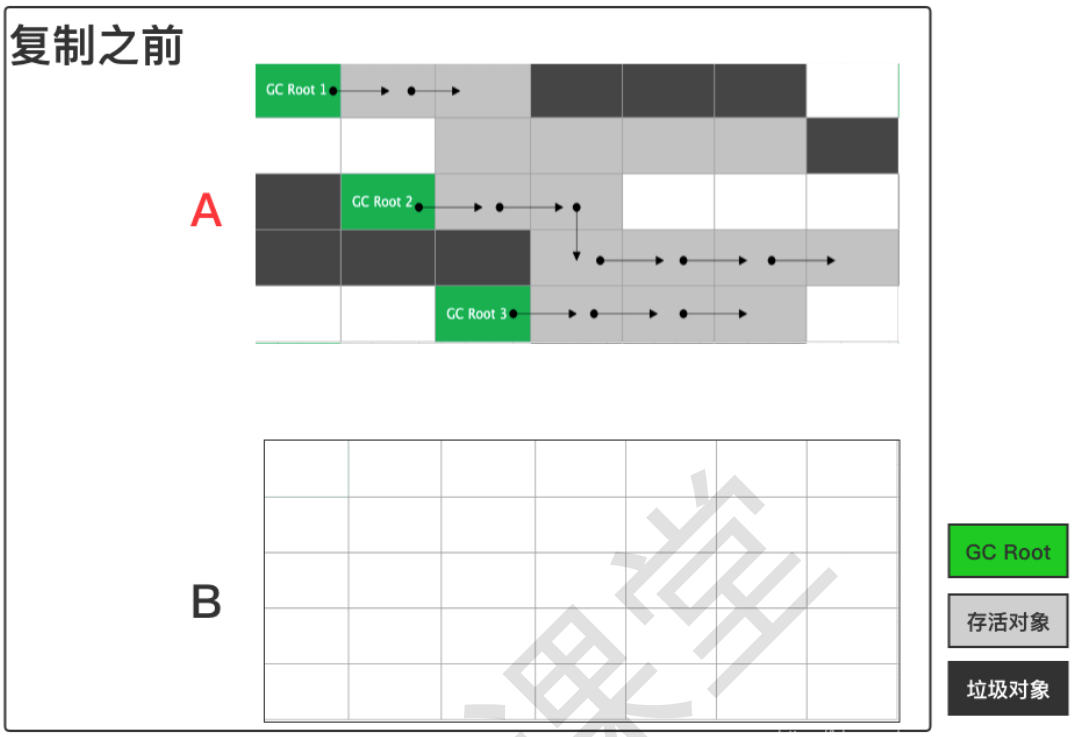
标记整理算法

与标记-清除不同的是它并不简单地清理未标记的对象，而是将所有的存活对象压缩到内存的一端。最后，清理边界外所有的空间。

- 优点：这种方法既避免了碎片的产生，又不需要两块相同的内存空间，因此，其性价比比较高。
- 缺点：所谓压缩操作，仍需要进行局部对象移动，所以一定程度上还是降低了效率。

复制算法

将现有的内存空间分为两快，每次只使用其中一块，在垃圾回收时将正在使用的内存中的存活对象复制到未被使用的内存块中。之后，清除正在使用的内存块中的所有对象，交换两个内存的角色，完成垃圾回收。



- 优点：按顺序分配内存即可，实现简单、运行高效，不用考虑内存碎片。
- 缺点：可用的内存大小缩小为原来的一半，对象存活率高时会频繁进行复制。

分代回收策略

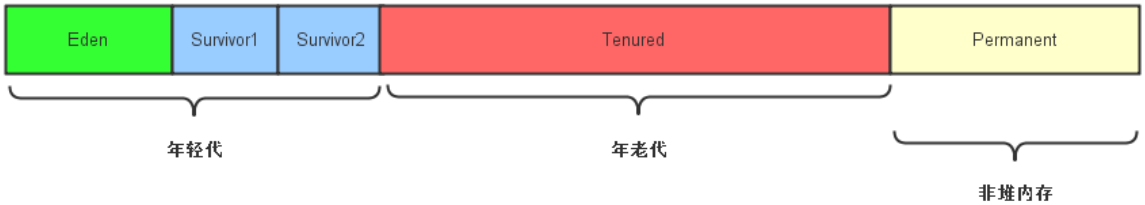
不同的垃圾收集器实现采用不同的算法进行垃圾回收，除此之外现代虚拟机还会采用分代机制来进行垃圾回收，根据对象存活的周期不同，把堆内存划分为不同区域，不同区域采用不同算法进行垃圾回收。

分代的垃圾回收策略，是基于这样一个事实：不同的对象的生命周期是不一样的。因此，不同生命周期的对象可以采取不同的收集方式，以便提高回收效率。

在Java程序运行的过程中，会产生大量的对象，其中有些对象是与业务信息相关，比如Http请求中的Session对象、线程、Socket连接，这类对象跟业务直接挂钩，因此生命周期比较长。但是还有一些对象，主要是程序运行过程中生成的临时变量，这些对象生命周期会比较短，比如：String对象，由于其不变类的特性，系统会产生大量的这些对象，有些对象甚至只用一次即可回收。

试想，在不进行对象存活时间区分的情况下，每次垃圾回收都是对整个堆空间进行回收，花费时间相对会长，同时，因为每次回收都需要遍历所有存活对象，但实际上，对于生命周期长的对象而言，这种遍历是没有效果的，因为可能进行了很多次遍历，但是他们依旧存在。因此，分代垃圾回收采用分治的思想，进行代的划分，把不同生命周期的对象放在不同代上，不同代上采用最适合它的垃圾回收方式进行回收。

代际划分



堆内存分为年轻代（Young Generation）和老年代（Old Generation）。而持久代使用非堆内存，主要用于存储一些类的元数据，常量池，java类，静态文件等信息。

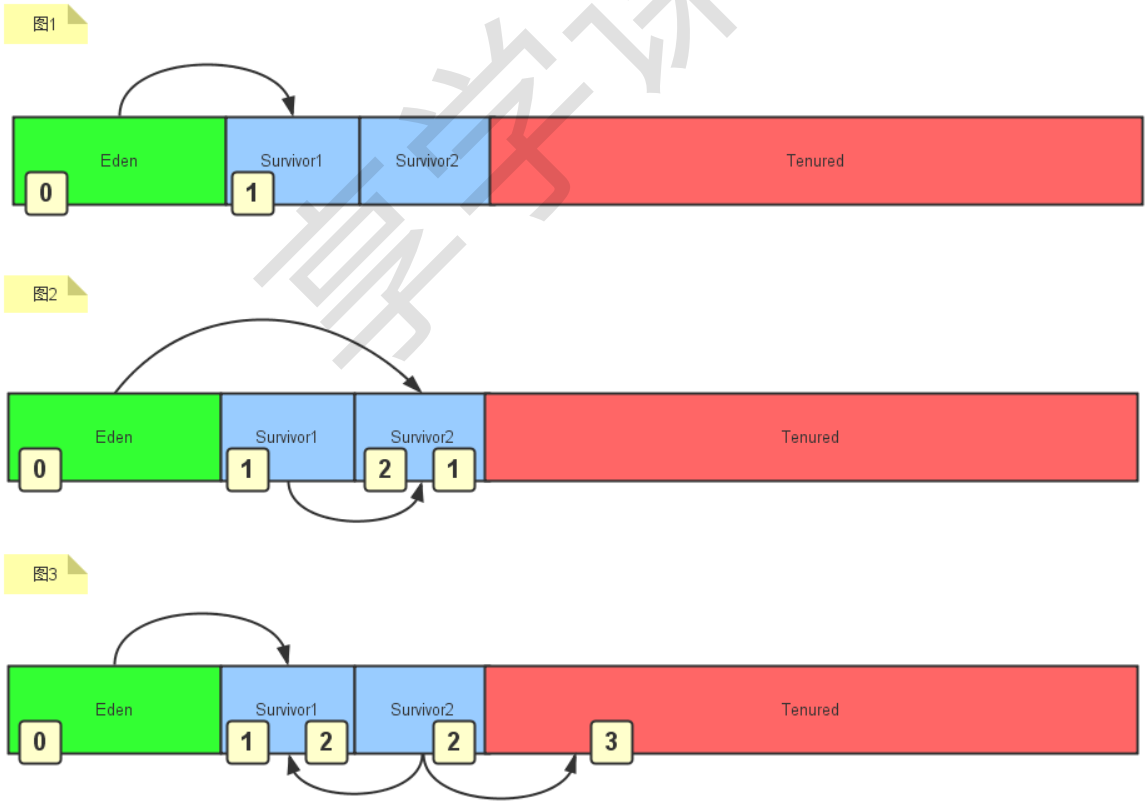
年轻代

年轻代会划分出Eden区域与两个大小对等的Survivor区域。 其比例一般为8：1：1。

1. 新生成的对象优先存放在新生代中
2. 存活率很低，回收效率很高
3. 一般采用的 GC 回收算法是复制算法

垃圾回收

当新对象生成，并且在Eden申请空间失败时，就会触发Scavenge GC，对Eden区域进行GC，清除非存活对象，并且把尚且存活的对象移动到Survivor区，然后整理Survivor的两个区。这种方式的GC是对年轻代的Eden区进行，不会影响到年老代。因为大部分对象都是从Eden区开始的，同时Eden区不会分配的很大，所以Eden区的GC会频繁进行。因而，一般在这里需要使用速度快、效率高的算法，使Eden区能尽快空闲出来。



新对象的内存分配都是先在Eden区域中进行的，当Eden区域的空间不足于分配新对象时，就会触发年轻代上的垃圾回收，我们称之为"minor gc"。同时，每个对象都有一个“年龄”，这个年龄实际上指的就是该对象经历过的minor gc的次数。如图1所示，当对象刚分配到Eden区域时，对象的年龄为“0”，当minor gc被触发后，所有存活的对象（仍然可达对象）会被拷贝到其中一个Survivor区域，同时年龄增长为“1”。并清除整个Eden内存区域中的非可达对象。

当第二次minor gc被触发时（如图2所示），JVM会通过Mark算法找出所有在Eden内存区域和Survivor1内存区域存活的对象，并将他们拷贝到新的Survivor2内存区域(这也就是为什么需要两个大小一样的Survivor区域的原因)，同时对象的年龄加1. 最后，清除所有在Eden内存区域和Survivor1内存区域的非可达对象。

当对象的年龄足够大（这个年龄可以通过JVM参数进行指定，这里假定是2），当minor gc再次发生时，它会从Survivor内存区域中升级到年老代中，如图3所示。

老年代

当minor gc发生时，又有对象从Survivor区域升级到Tenured区域，但是Tenured区域已经没有空间容纳新的对象了，那么这个时候就会触发年老代上的垃圾回收，我们称之为"major gc"。而在年老代上选择的垃圾回收算法则取决于JVM上采用的是什么垃圾回收器。

总结

在JVM中一般采用可达性分析法进行是否可回收的判定，确定对象需要被回收后，对象在哪个代际将会采用不同的垃圾回收算法进行回收，这些算法包括：标记-清除，标记-整理与复制算法。

而之所以采用分代策略的原因是：不同的对象的生命周期是不一样的。因此，不同生命周期的对象可以采取不同的收集方式，以便提高回收效率。 如果每次垃圾回收都是对整个堆空间进行回收，花费时间相对会长，而对于生命周期长的对象而言，这种遍历是没有效果的，因为可能进行了很多次遍历，但是他们依旧存在。

4.4 Java中有几种引用关系，它们的区别是什么？~jett

这道题想考察什么?

java中四种引用的基本语法与其在开发中的用处

考察的知识点

java基础知识

考生应该如何回答

Java中一共有四种引用关系，分别是强引用、软引用、弱引用以及虚引用。

1.强引用：为我们平时直接使用的引用方式，其引用关系没有被断开，会一直存在于引用链上。

比如：Object obj=new Object();

2.软引用：会在系统内存不足时被GC回收，一般可用于缓存的设计，我们通过以下代码来测试。

```
public void testSoftReference(){
    User user=new User(1,"Jett");//定义一个对象
    SoftReference<User> userSoft=new SoftReference<User>(user);//将user放入软引用中
    user=null;
    System.out.println(userSoft.get());//user为空，软引用也不会回收
    System.gc();
    System.out.println("After gc");
    System.out.println(userSoft.get());//GC之后，如果内存够用，还是不会回收
```

```
    //向堆中填充数据,导致OOM
    List<byte[]> list=new LinkedList<>();
    try{
        for (int i = 0; i < 100; i++) {
            System.out.println("for==========="+userSoft.get());
            list.add(new byte[1024*1024*1]);
        }
    }catch(Throwable e){
        System.out.println("Exception======"+userSoft.get());//当内存不足时，软引用
就得到回收
    }
}
```

3.弱引用：GC到来时回收，多数情况下可以很方便的帮助我们在项目中解决内存泄漏问题。

```
public void testWeakReference(){
    User user=new User(1,"Jett");//定义一个对象
    WeakReference<User> userWeakReference=new WeakReference<>(user);//将其放入弱引
用
    user=null;
    System.out.println(userWeakReference.get());//user为空，弱引用也不会回收
    System.gc();
    System.out.println("After gc");
    System.out.println(userWeakReference.get());//GC之后,弱引用得到GC回收
}
```

4.虚引用：GC回收时可得到一个通知，该引用不能直接使用，但可在引用队列中观察到GC回收过的对象，可以用于监听GC回收通知。

```
public void testPhantomReference() throws InterruptedException {
    //虚引用：功能，不会影响到对象的生命周期的，
    // 但是能让程序员知道该对象什么时候被 回收了
    ReferenceQueue<Object> referenceQueuee=new ReferenceQueue<>();
    Object phantomObject=new Object();
    PhantomReference phantomReference   //虚引用需要配合引用队列才能看到效果
                    =new PhantomReference(phantomObject,referenceQueuee);
    phantomObject=null;
    System.out.println("phantomObject:"+phantomObject);//输出null
    System.out.println("phantomReference"+referenceQueuee.poll());//输出null
    System.gc();
    Thread.sleep(2000);
    System.out.println("referenceQueuee:"+referenceQueuee.poll());//输出GC回收的对
象
}
```

4.5 判断对象是否被回收，有哪些GC算法，虚拟机使用最多的是什么算法？（美团）~lance

这道题想考察什么？

Java利用GC机制让开发者不必再像C/C++手动回收内存，但是并不是有了GC机制就万事皆可，在不了解GC机制算法的情况下，很容易出现代码问题导致该回收的对象无法被回收（内存泄漏），一直占用内存，导致程序可用内存越来越少，最终出现OOM。

考察的知识点

GC机制、可达性分析发、引用技术

考生如何回答

如何判断对象是否可被回收的算法，在《介绍下GC回收机制与分代回收策略》中介绍了**可达性分析法。**而除了可达性分析法之外，还有被淘汰的**引用计数算法**：

给对象中添加一个引用计数器,每当有一个地方引用它时,计数器值就加1;当引用失效时,计数器值就减1;任何时刻计数器为0的对象就是不可能再被使用的。

目前主流的java虚拟机都摒弃掉了这种算法，最主要的原因是它很难解决对象之间相互循环引用的问题。尽管该算法执行效率很高。比如：

```
class A{
    B b;
}
class B{
    A a;
}

A a = new A();
B b = new B();
//循环引用 导致无法释放
a.b = b;
b.a = a;
```

而实际现在Java虚拟机都是采用的**可达性分析法。**

4.6 描述JVM内存模型 ~jett

4.7 JVM DVM ART的区别 ~colin

这道题想考察什么?

1. 是否了解虚拟机的相关知识?

考察的知识点

1. JVM虚拟的基本知识
2. DVM虚拟机的基本知识
3. ART虚拟机的基本知识

考生应该如何回答

1. JVM

   JVM在前面的章节中详解介绍，这里不再讲解。

2. Dalvik虚拟机

Dalvik虚拟机（Dalvik Virtual Machine），简称Dalvik VM或者DVM。DVM是Google专门为Android平台开发的虚拟机，它运行在Android运行时库中。需要注意的是DVM并不是一个Java虚拟机（以下简称JVM）

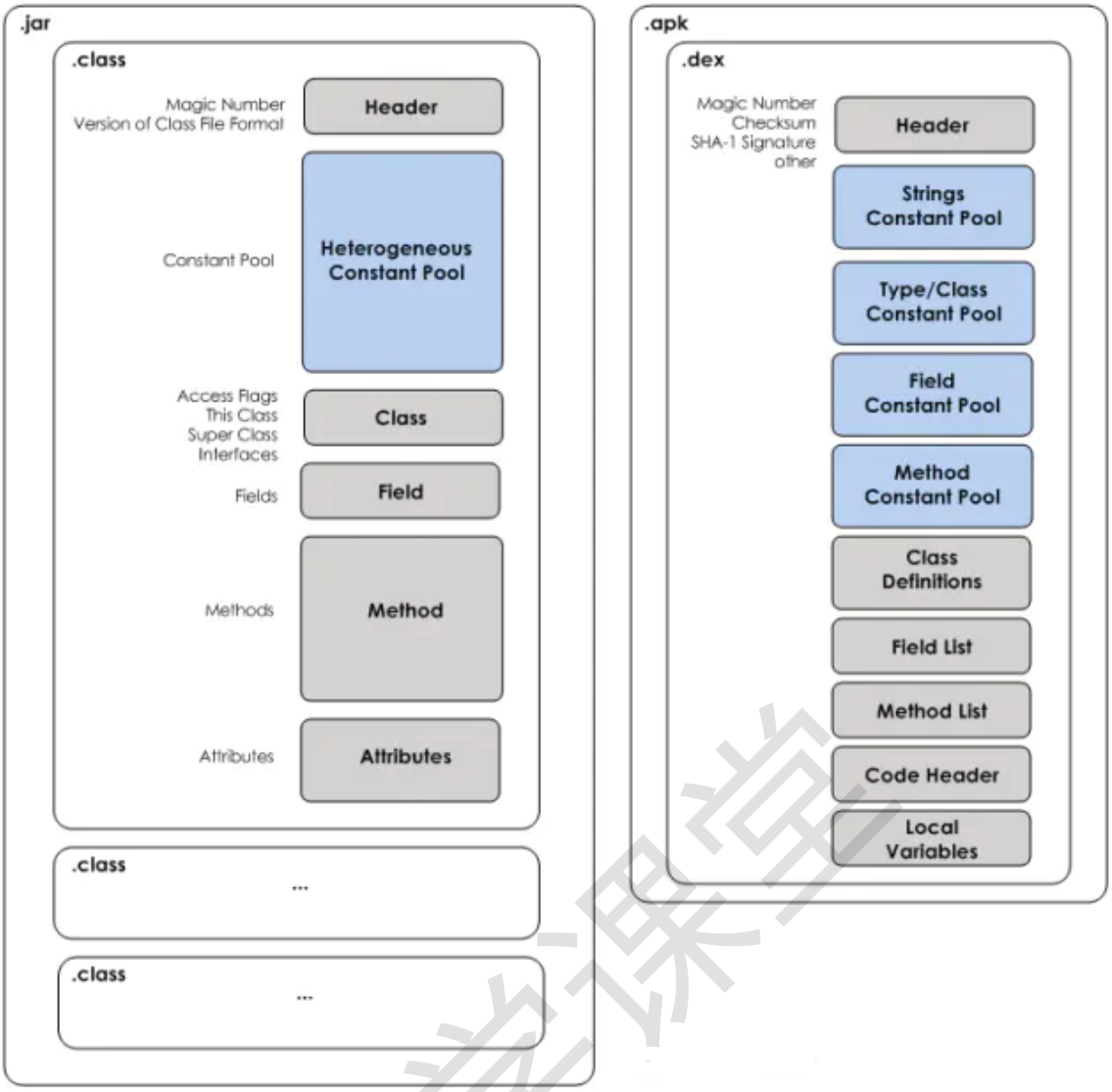
主要原因是DVM并没有遵循JVM规范来实现。DVM与JVM主要有以下区别。

基于的架构不同

- JVM基于栈则意味着需要去栈中读写数据，所需的指令会更多，这样会导致速度慢，对于性能有限的移动设备，显然不是很适合。
- DVM是基于寄存器的，它没有基于栈的虚拟机在拷贝数据而使用的大量的出入栈指令，同时指令更紧凑更简洁。但是由于显示指定了操作数，所以基于寄存器的指令会比基于栈的指令要大，但是由于指令数量的减少，总的代码数不会增加多少。

执行的字节码不同

- 在Java SE程序中，Java类会被编译成一个或多个.class文件，打包成jar文件，而后JVM会通过相应的.class文件和jar文件获取相应的字节码。执行顺序为： .java文件 -> .class文件 -> .jar文件
- DVM会用dx工具将所有的.class文件转换为一个.dex文件，然后DVM会从该.dex文件读取指令和数据。执行顺序为:
  .java文件 –>.class文件-> .dex文件



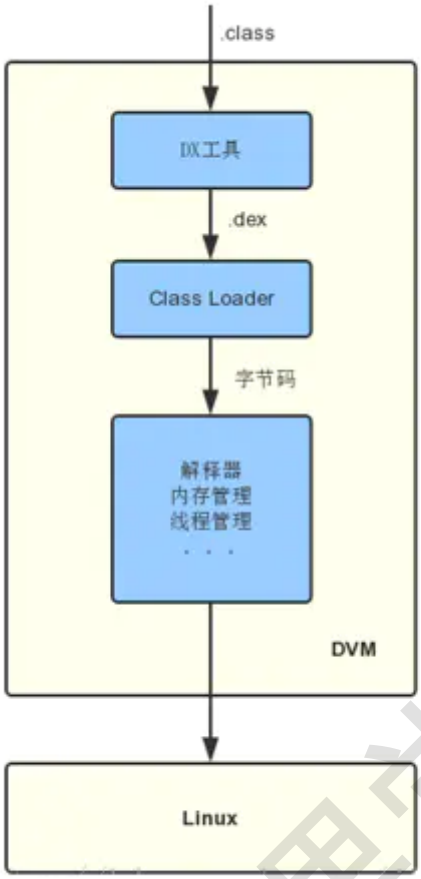
- 如上图所示，.jar文件里面包含多个.class文件，每个.class文件里面包含了该类的常量池、类信息、属性等等。当JVM加载该.jar文件的时候，会加载里面的所有的.class文件，JVM的这种加载方式很慢，对于内存有限的移动设备并不合适。
- 在.apk文件中只包含了一个.dex文件，这个.dex文件里面将所有的.class里面所包含的信息全部整合在一起了，这样再加载就提高了速度。.class文件存在很多的冗余信息，dex工具会去除冗余信息，并把所有的.class文件整合到.dex文件中，减少了I/O操作，提高了类的查找速度。
- DVM允许在有限的内存中同时运行多个进程，DVM经过优化，允许在有限的内存中同时运行多个进程。在Android中的每一个应用都运行在一个DVM实例中，每一个DVM实例都运行在一个独立的进程空间。独立的进程可以防止在虚拟机崩溃的时候所有程序都被关闭。

DVM由Zygote创建和初始化

在Android系统启动流程（二）解析Zygote进程启动过程这篇文章中我介绍过 Zygote，可以称它为孵化器，它是一个DVM进程，同时它也用来创建和初始化DVM实例。每当系统需要创建一个应用程序时，Zygote就会fock自身，快速的创建和初始化一个DVM实例，用于应用程序的运行。

DVM架构

DVM的源码位于dalvik/目录下，其中dalvik/vm目录下的内容是DVM的具体实现部分，它会被编译成libdvm.so；dalvik/libdex会被编译成libdex.a静态库，作为dex工具使用；dalvik/dexdump是.dex文件的反编译工具；DVM的可执行程序位于dalvik/dalvikvm中，将会被编译成dalvikvm可执行程序。DVM架构如下图所示。



- 从上图可以看出，首先Java编译器编译的.class文件经过DX工具转换为.dex文件，.dex文件由类加载器处理，接着解释器根据指令集对Dalvik字节码进行解释、执行，最后交与Linux处理。

DVM的运行时堆：
DVM的运行时堆主要由两个Space以及多个辅助数据结构组成，两个Space分别是Zygote Space（Zygote Heap）和Allocation Space（Active Heap）。Zygote Space用来管理Zygote进程在启动过程中预加载和创建的各种对象，Zygote Space中不会触发GC，所有进程都共享该区域，比如系统资源。Allocation Space是在Zygote进程fork第一个子进程之前创建的，它是一种私有进程，Zygote进程和fock的子进程在Allocation Space上进行对象分配和释放。

除了这两个Space，还包含以下数据结构：
Card Table： 用于DVM Concurrent GC，当第一次进行垃圾标记后，记录垃圾信息。
Heap Bitmap： 有两个Heap Bitmap，一个用来记录上次GC存活的对象，另一个用来记录这次GC存活的对象。
Mark Stack： DVM的运行时堆使用标记-清除（Mark-Sweep）算法进行GC，不了解标记-清除算法的同学查看Java虚拟机（四）垃圾收集算法这篇文章。Mark Stack就是在GC的标记阶段使用的，它用来遍历存活的对象。

1. ART虚拟机

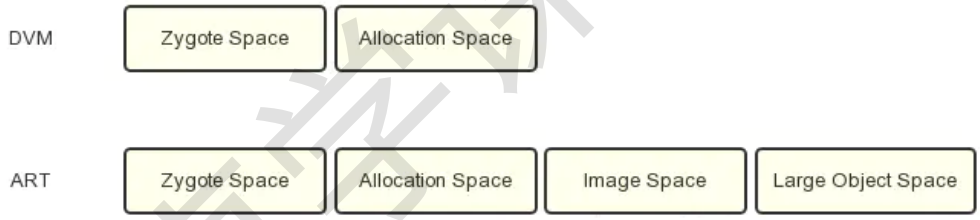
ART(Android Runtime)是Android 4.4发布的，用来替换Dalvik虚拟，Android 4.4默认采用的还是DVM，系统会提供一个选项来开启ART。在Android 5.0时，默认采用ART，DVM从此退出历史舞台。

ART与DVM的区别
DVM中的应用每次运行时，字节码都需要通过即时编译器（JIT，just in time）转换为机器码，这会使得应用的运行效率降低。而在ART中，系统在安装应用时会进行一次预编译（AOT，ahead of time）,将字节码预先编译成机器码并存储在本地，这样应用每次运行时就不需要执行编译了，运行效率也大大提升。

ART占用空间比Dalvik大（字节码变为机器码之后，可能会增加10%-20%），这就是"时间换空间大法"。
预编译也可以明显改善电池续航，因为应用程序每次运行时不用重复编译了，从而减少了 CPU 的使用频率，降低了能耗。

ART的运行时堆
与DVM的GC不同的是，ART的GC类型有多种，主要分为Mark-Sweep GC和Compacting GC。ART的运行时堆的空间根据不同的GC类型也有着不同的划分，如果采用的是Mark-Sweep GC，运行时堆主要是由四个Space和多个辅助数据结构组成，四个Space分别是Zygote Space、Allocation Space、Image Space和Large Object Space。Zygote Space、Allocation Space和DVM中的作用是一样的。Image Space用来存放一些预加载类，Large Object Space用来分配一些大对象（默认大小为12k）。其中Zygote Space和Image Space是进程间共享的。

采用Mark-Sweep GC的运行时堆空间划分如下图所示。



除了这四个Space，ART的Java堆中还包括两个Mod Union Table，一个Card Table，两个Heap Bitmap，两个Object Map，以及三个Object Stack。

4.8 描述GC机制。Class会不会回收？用不到的Class怎么回收？(东方头条)

这道题想考察什么？

1. 是否了解GC机制与真实场景使用，是否熟悉GC机制

考察的知识点

1. GC机制的概念在项目中使用与基本知识

考生应该如何回答

1.你工作怎么多年，GC机制。Class会不会回收？用不到的Class怎么回收？

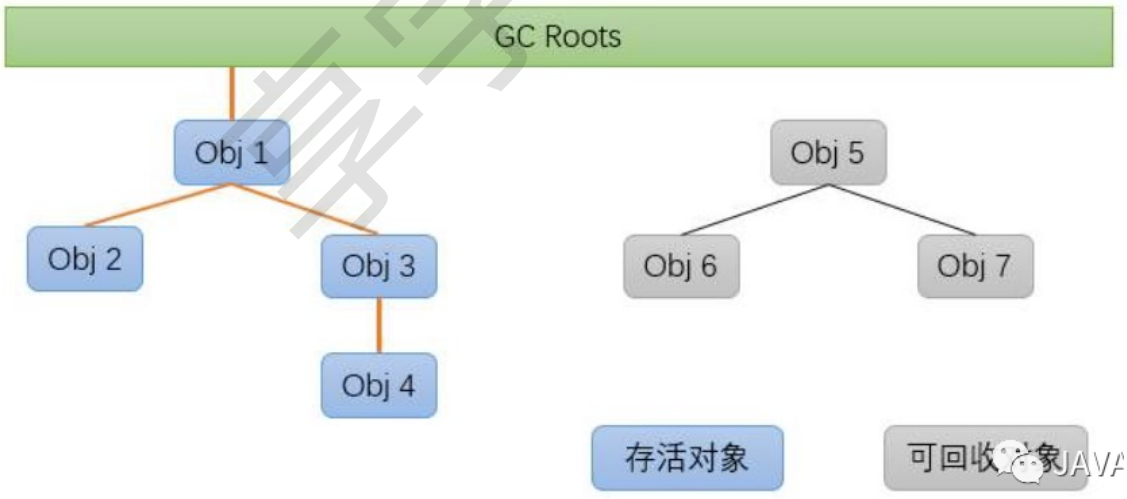
答：

1、垃圾收集发生的区域
之前我们介绍过 Java 内存运行时区域的各个部分，其中程序计数器、虚拟机栈、本地方法栈三个区域随线程共存亡。栈中的每一个栈帧分配多少内存基本上在类结构确定下来时就已知，因此这几个区域的内存分配和回收都具有确定性，不需要考虑如何回收的问题，当方法结束或线程结束，内存自然也跟着回收了

而 Java 堆和方法区这两个区域则有显著的不确定性，只有在程序运行时我们才能知道程序究竟创建了哪些对象，创建了多少对象，所以这部分内存的分配和回收是动态的，垃圾收集器所关注的正是这部分内存该如何管理

2、如何判定需要被回收的对象？
如果一个对象没有被其他对象引用，则证明这个对象可以被回收，因为它已经没有实际用途了。那我们怎么去判断一个对象是否可回收呢？业界主要有两种判断方式：

1. 引用计数法
   在对象中添加一个引用计数器，每当有一个地方引用它时，计数器值加一；当引用失效，计数器值减一；任何时刻计数器值都为零的对象就是不可能再被使用了。这种方法虽然会占用额外的内存空间用于计数，但它的原理简单，判定效率也高，大多数情况下它都是一个不错的算法。然而，这个看似简单的算法却需要考虑很多额外情况，否则将无法保证其正确工作，例如单纯的引用计数法就很难解决对象之间相互循环引用的问题
2. 可达性分析算法
   该算法的基本思路是通过一系列称为 GC Roots 的根对象作为起始节点集，从这些节点开始，根据引用关系向下搜索，搜索过程走过的路径称为引用链。如果某个对象到 GC Roots 间没有任何引用链相连，则证明此对象是不可能再被使用，可以回收



在 Java 技术体系中，可以作为 GC Roots 的对象包括：

在虚拟机栈(栈帧中的本地变量表)中引用的对象

方法区中类静态属性引用的对象

方法区中常量引用的对象

本地方法栈中 JNI(即通常所说的 Native 方法)引用的对象

Java 虚拟机内部的引用，如基本数据类型对应的 Class 对象，一些常驻的异常对象(NullPointException、OutOfMemoryError)

所有被同步锁持有的对象

反映 Java 虚拟机内部情况的 JMXBean、JVMTI 中注册的回调、本地代码缓存等

除了这些固定的 GC Roots 集合外，根据用户所选用的垃圾收集器以及当前回收的内存区域的不同，还可以有其他对象临时加入，共同构成完整的 GC Roots 集合

3、四种引用类型
无论是通过引用计数法还是可达性分析算法，判断对象是否存活都和引用离不开关系。在 JDK1.2 以前，Java 里的引用是很传统的定义：如果 reference 类型的数据中存储的数值代表的是另外一块内存的起始地址，就称该 reference 数据是代表某块内存、某个对象的引用。这种定义当然没有什么不对，但现在看来显得太狭隘了，比如我们希望描述一类对象：当内存空间足够时，能保留在内存中，如果内存空间在进行了垃圾收集后仍然紧张，则可以抛弃这些对象，很多系统的缓存功能都符合这样的应用场景

JDK1.2 对引用的概念作了补充，将引用分为强引用(Strongly Reference)、软引用(SoftReference)、弱引用(Weak Reference)和虚引用(Phantom Reference)，强度依次减弱

强引用

形如 Object obj = new Object() 这种引用关系就是我们常说的强引用。无论什么情况，只要强引用关系存在，对象就永远不会被回收

软引用

用来描述一些有用但非必须的对象。此类对象只有在进行一次垃圾收集仍然没有足够内存时，才会在第二次垃圾收集时被回收。JDK1.2 之后提供了 SoftReference 类来实现软引用

弱引用

也是用来描述那些非必须对象，但它的强度比软引用更弱一些。被软引用关联的对象只能生存到下一次垃圾收集发生为止，当垃圾收集器开始工作，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。JDK1.2 之后提供了 WeakReference 类来实现软引用

虚引用

最弱的一种引用关系，一个对象是否存在虚引用，丝毫不会对其生存时间造成任何影响，也无法通过虚引用来取得一个对象实例。设置虚引用关联的唯一目的就是让这个对象被回收时能收到一个系统通知。JDK1.2 之后提供了 PhantomReference 类来实现软引用

4、finalize() 方法
在可达性分析中被判定为不可达的对象，并不是立即赴死，至少要经历两次标记过程：如果对象在进行可达性分析后发现没有与 GC Root 相连接的引用链，那么它将被第一次标记，随后再进行一次筛选，筛选条件是对象是否有必要执行 finalize() 方法，如果对象没有覆盖 finalize() 方法或是 finalize() 方法已经被调用过，则都视为“没有必要执行”

如果对象被判定为有必要执行 finalize() 方法，那么该对象将会被放置在一个名为 F-Queue 的队列之中，并在稍后由一条由虚拟机自动创建的、低调度优先级的 Finalizer 线程去执行它们的 finalize() 方法。注意这里所说的执行是指虚拟机会触发这个方法开始运行，但并不承诺一定会等待它运行结束。这样做的原因是防止某个对象的 finalize() 方法执行缓慢，或者发生死循环，导致 F-Queue 队列中的其他对象永久处于等待状态

finalize() 方法是对象逃脱死亡命运的最后一次机会，稍后收集器将对 F-Queue 中的对象进行第二次小规模标记，如果对象希望在 finalize() 方法中成功拯救自己，只要重新与引用链上的任何一个对象建立关联即可，那么在第二次标记时它将被移出“即将回收”的集合；如果对象这时候还没有逃脱，那基本上就真的要被回收了

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() 方法将不会再执行。finalize() 方法运行代价高，不确定性大，无法保证各个对象的调用顺序，因此已被官方明确声明为不推荐使用的语法

5、回收方法区
方法区的垃圾收集主要回收两部分：废弃的常量和不再使用的类型。判定一个常量是否废弃相对简单，与对象类似，只要某个常量不再被引用，就会被清理。而判定一个类型是否属于“不再被使用的类”的条件就比较苛刻了，需要同时满足下面三个条件：

该类的所有实例都已经被回收，即 Java 堆中不存在该类及其任何派生子类的实例

加载该类的类加载器已经被回收

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

Java 虚拟机允许对满足上述三个条件的无用类进行回收，但并不是说必然被回收，仅仅是允许而已。关于是否要对类型进行回收，HotSpot 虚拟机提供了 -Xnoclassgc 参数进行控制

6、分代收集理论
当前商业虚拟机的垃圾收集器大多数都遵循了“分代收集”的设计理论，分代收集理论其实是一套符合大多数程序运行实际情况的经验法则，主要建立在两个分代假说之上：

弱分代假说：绝大多数对象都是朝生夕灭的

强分代假说：熬过越多次垃圾收集过程的对象就越难以消亡

这两个分代假说共同奠定了多款常用垃圾收集器的一致设计原则：收集器应该将 Java 堆划分出不同的区域，将回收对象依据年龄(即对象熬过垃圾收集过程的次数)分配到不同的区域之中存储，把存活时间短的对象集中在一起，每次回收只关注如何保留少量存活的对象，即新生代(Young Generation)；把难以消亡的对象集中在一起，虚拟机就可以使用较低的频率来回收这个区域，即老年代(Old Generation)

正因为划出了不同的区域，垃圾收集器才可以每次只回收其中一个或多个区域，因此才有了“Minor GC”、“Major GC”、“Full GC”这样的回收类型划分，也才能够针对不同的区域采用不同的垃圾收集算法，因而有了“标记-复制”算法、“标记-清除”算法、“标记-整理”算法

分代收集并非只是简单划分一下内存区域，它至少存在一个明显的困难：对象之间不是孤立的，对象之间会存在跨代引用。假如现在要进行只局限于新生代的垃圾收集，根据前面可达性分析的知识，与 GC Roots 之间不存在引用链即为可回收，但新生代的对象很有可能会被老年代所引用，那么老年代对象将临时加入 GC Roots 集合中，我们不得不再额外遍历整个老年代中的所有对象来确保可达性分析结果的正确性，这无疑为内存回收带来很大的性能负担。为了解决这个问题，就需要对分代收集理论添加第三条经验法则：

跨代引用假说：跨代引用相对于同代引用仅占少数

存在互相引用的两个对象，应该是倾向于同时生存或同时消亡的，举个例子，如果某个新生代对象存在跨代引用，由于老年代对象难以消亡，会使得新生代对象同样在收集时得以存活，进而年龄增长后晋升到老年代，那么跨代引用也随之消除了。既然跨代引用只是少数，那么就没必要去扫描整个老年代，也不必专门记录每一个对象是否存在哪些跨代引用，只需在新生代上建立一个全局的数据结构，称为记忆集(Remembered Set)，这个结构把老年代划分为若干个小块，标识出老年代的哪一块内存会存在跨代引用。此后当发生 Minor GC 时，只有包含了跨代引用的小块内存里的对象才会被加入 GC Roots 进行扫描

7、标记 - 清除算法
如其名，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，在标记完成之后，统一回收所有被标记的对象，也可以反过来，标记存活的对象，统一回收所有未被标记的对象。标记过程就是对象是否属于垃圾的判定过程。标记 - 清除算法执行过程如图所示：

内存整理前

内存整理后

image-20211027194953177

标记 - 清除算法是最基础的算法，后续的收集算法都是以标记 - 清除算法为基础，对其缺点进行改进，它的主要缺点有两个：

执行效率不稳定

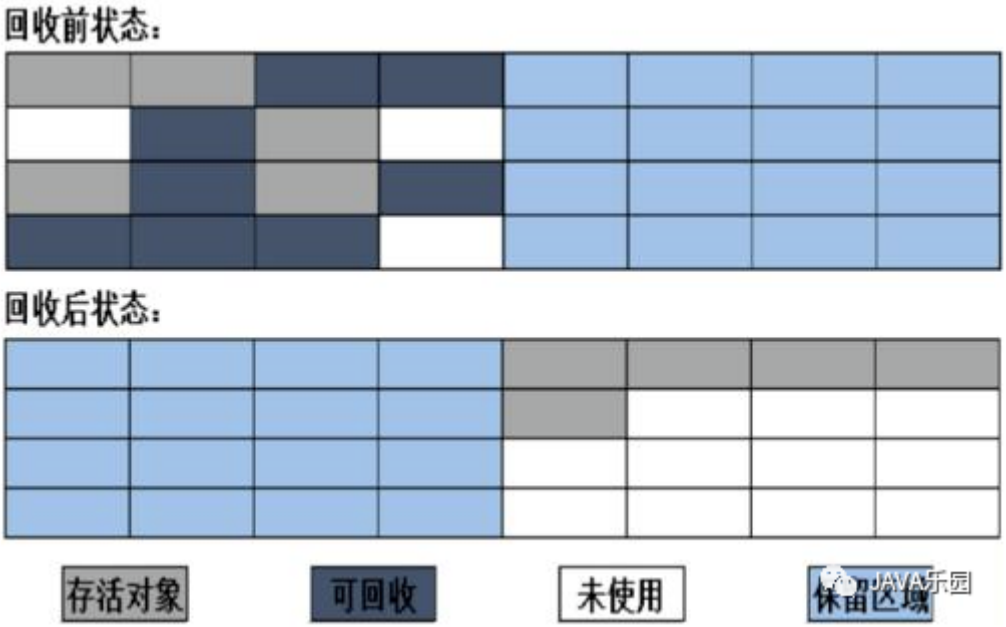
如果 Java 堆中包含大量对象且大部分需要回收，则必须进行大量标记和清除的动作'

内存空间碎片化问题

标记、清除之后会产生大量不连续的内存碎片，内存碎片太多会导致下次分配较大对象时无法找到足够的连续内存，从而不得不提前触发一次垃圾收集动作

8、标记 - 复制算法
为了解决标记 - 清除算法面对大量可回收对象时执行效率低的问题，复制算法将可用内存按容量划分为大小相等的两块，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内存上，再把已使用过的内存空间一次清理掉

如果内存中多数对象都是存活的，这种算法无疑会产生大量内存间复制的开销，但对于多数对象都是可回收的情况，算法需要复制的就是占少数的存活对象，而且每次都是针对整个半区进行内存回收，分配内存时也不用考虑空间碎片的问题，只要移动堆顶指针，按顺序分配即可。不过这种算法的缺陷也显而易见，可用内存被缩小为原来的一半

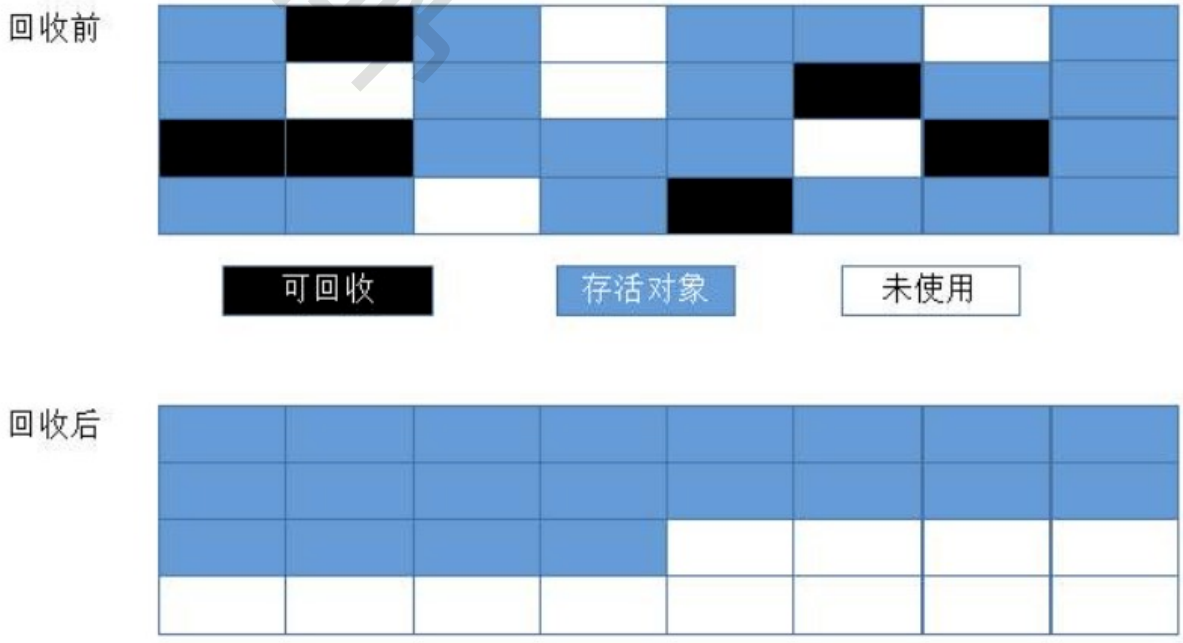


标记 - 复制算法大多用于新生代。实际上，新生代中的对象大多数都熬不过第一轮收集，因此不需要按 1:1 的比例来划分新生代的内存空间。具体做法是将新生代划分为一块较大的 Eden 区和两块较小的 Survivor 区，每次分配只使用 Eden 区和其中一块 Survivor 区。发生垃圾收集时，将 Eden 区和 Survivor 区中仍然存活的对象一次性复制到另一个 Survivor 区，然后直接清理掉 Eden 区和已经用过的 Survivor 区。HotSpot 虚拟机默认 Eden 和 Survivor 的大小比例是 8:1:1

当 Survivor 空间不足以容纳一次 Minor GC 之后存活的对象时，就需要依赖其他内存区域(大多是老年代)进行分配担保，上一次新生代存活下来的对象直接进入老年代

9、标记 - 整理算法

标记 - 复制算法不适合用在对象存活率高的区域，而且会浪费一半的空间，因此老年代一般不采用这种算法，取而代之的是有针对性的标记 - 整理算法。标记 - 整理算法的标记过程与标记 - 清除算法一样，但后续步骤不是直接清理可回收对象，而是让所有存活对象都向内存空间的一侧移动，然后直接清理掉边界以外的内存



是否移动回收后的存活对象是一项优缺点并存的风险决策，尤其是在老年代这种每次回收都有大量对象存活的区域，移动存活对象并更新其引用将会是一个极为繁重的操作，必须暂停用户应用程序线程才能进行，像这样的停顿行为被称为“Stop the World”。但如果不考虑移动存活对象，又会影响内存分配和访问的效率，为此使用者必须小心权衡其中的得失。一种和稀泥式的解决方案就是让虚拟机平时采用标记 - 清除算法，直到内存空间碎片化程度大到影响对象分配时，再采用标记 - 整理算法收集一次，已获得规整的内存空间

4.9 StackOverFlow与OOM的区别？分别发生在什么时候，JVM栈中存储的是什么，堆存储的是什么？（美团）~jett

这道题想考察什么?

对JVM的理解

考察的知识点

JVM

考生应该如何回答

StackOverFlow是栈空间不足出现的，主要是单个线程运行过程中调用方法过多或是方法递归操作时申请的栈帧使用存储空间超出了单个栈申请的存储空间。

OOM是主要是堆区申请的内存空间不够用时出现，主要是两种情况，一是单次申请大对象超出了堆中连续的可用空间，二是大量申请小对象超出了堆中的总可用空间。

栈中主要是用于存放程序运行过程中需要使用的变量，引用和临时数据，堆中主要存储程序中申请的对象空间。

4.10 Java虚拟机和Dalvik虚拟机的区别？~jett

这道题想考察什么?

JVM

考察的知识点

JVM

考生应该如何回答

1.java虚拟机运行的是Java字节码，Dalvik虚拟机运行的是Dalvik字节码；传统的Java程序经过编译，生成Java字节码保存在class文件中，java虚拟机通过解码class文件中的内容来运行程序。而Dalvik虚拟机运行的是Dalvik字节码，所有的Dalvik字节码由Java字节码转换而来，并被打包到一个DEX(Dalvik Executable)可执行文件中Dalvik虚拟机通过解释Dex文件来执行这些字节码。

2.Dalvik可执行文件体积更小。SDK中有一个叫dx的工具负责将java字节码转换为Dalvik字节码。

3.java虚拟机与Dalvik虚拟机架构不同。java虚拟机基于栈架构。程序在运行时虚拟机需要频繁的从栈上读取或写入数据。这过程需要更多的指令分派与内存访问次数，会耗费不少CPU时间，对于像手机设备资源有限的设备来说，这是相当大的一笔开销。Dalvik虚拟机基于寄存器架构，数据的访问通过寄存器间直接传递，这样的访问方式比基于栈方式快的多.

4.11 请描述new一个对象的流程 ~jett

这道题想考察什么?

对JVM的理解

考察的知识点

对JVM的理解

考生应该如何回答

java在new一个对象的时候，会先查看对象所属的类有没有被加载到内存，如果没有的话，就会先通过类的全限定名来加载。加载并初始化类完成后，再进行对象的创建工作。

我们先假设是第一次使用该类，这样的话new一个对象就可以分为两个过程：加载并初始化类和创建对象。

一、类加载过程（第一次使用该类）

java是使用双亲委派模型来进行类的加载的，所以在描述类加载过程前，我们先看一下它的工作过程：

双亲委托模型的工作过程是：如果一个类加载器（ClassLoader）收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委托给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父类加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需要加载的类）时，子加载器才会尝试自己去加载。
使用双亲委托机制的好处是：能够有效确保一个类的全局唯一性，当程序中出现多个限定名相同的类时，类加载器在执行加载时，始终只会加载其中的某一个类。

1、加载

由类加载器负责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，并存储在运行时内存区的方法区，然后将其转换为一个与目标类型对应的java.lang.Class对象实例

2、验证

格式验证：验证是否符合class文件规范
语义验证：检查一个被标记为final的类型是否包含子类；检查一个类中的final方法是否被子类进行重写；确保父类和子类之间没有不兼容的一些方法声明（比如方法签名相同，但方法的返回值不同）
操作验证：在操作数栈中的数据必须进行正确的操作，对常量池中的各种符号引用执行验证（通常在解析阶段执行，检查是否可以通过符号引用中描述的全限定名定位到指定类型上，以及类成员信息的访问修饰符是否允许访问等）

3、准备

为类中的所有静态变量分配内存空间，并为其设置一个初始值（由于还没有产生对象，实例变量不在此操作范围内）
被final修饰的static变量（常量），会直接赋值；

4、解析

将常量池中的符号引用转为直接引用（得到类或者字段、方法在内存中的指针或者偏移量，以便直接调用该方法），这个可以在初始化之后再执行。
解析需要静态绑定的内容。 // 所有不会被重写的方法和域都会被静态绑定

以上2、3、4三个阶段又合称为链接阶段，链接阶段要做的是将加载到JVM中的二进制字节流的类数据信息合并到JVM的运行时状态中。

5、初始化（先父后子）

4.1 为静态变量赋值

4.2 执行static代码块

注意：static代码块只有jvm能够调用
　　如果是多线程需要同时初始化一个类，仅仅只能允许其中一个线程对其执行初始化操作，其余线程必须等待，只有在活动线程执行完对类的初始化操作之后，才会通知正在等待的其他线程。

因为子类存在对父类的依赖，所以类的加载顺序是先加载父类后加载子类，初始化也一样。不过，父类初始化时，子类静态变量的值也有有的，是默认值。

最终，方法区会存储当前类类信息，包括类的静态变量、类初始化代码（定义静态变量时的赋值语句 和静态初始化代码块）、实例变量定义、实例初始化代码（定义实例变量时的赋值语句实例代码块和构造方法）和实例方法，还有父类的类信息引用。

二、创建对象

1、在堆区分配对象需要的内存

分配的内存包括本类和父类的所有实例变量，但不包括任何静态变量

2、对所有实例变量赋默认值

将方法区内对实例变量的定义拷贝一份到堆区，然后赋默认值

3、执行实例初始化代码

初始化顺序是先初始化父类再初始化子类，初始化时先执行实例代码块然后是构造方法

4、如果有类似于Child c = new Child()形式的c引用的话，在栈区定义Child类型引用变量c，然后将堆区对象的地址赋值给它

需要注意的是，每个子类对象持有父类对象的引用，可在内部通过super关键字来调用父类对象，但在外部不可访问

补充：

通过实例引用调用实例方法的时候，先从方法区中对象的实际类型信息找，找不到的话再去父类类型信息中找。

如果继承的层次比较深，要调用的方法位于比较上层的父类，则调用的效率是比较低的，因为每次调用都要经过很多次查找。这时候大多系统会采用一种称为虚方法表的方法来优化调用的效率。

所谓虚方法表，就是在类加载的时候，为每个类创建一个表，这个表包括该类的对象所有动态绑定的方法及其地址，包括父类的方法，但一个方法只有一条记录，子类重写了父类方法后只会保留子类的。当通过对象动态绑定方法的时候，只需要查找这个表就可以了，而不需要挨个查找每个父类。

4.12 Java对象会不会分配到栈中？~jett

这道题想考察什么?

考察程序员对JVM相关知识的理解

考察的知识点

JVM基础知识

考生应该如何回答

Java对象会分配到栈中，但是是有一定的条件的：

条件如下：

1.申请的对象在某个方法中使用，该对象不会被该方法以外的引用使用，则可以进行JVM的逃逸分析

2. 打开虚拟机优化技术--逃逸分析相关的JVM参数，-XX:-DoEscapeAnalysis

达成上面两个条件，Java对象能分配到栈中，测试代码如下：

```
public static class User{
    public int id=0;
    public String name="";
}
public static void alloc(){
    User u=new User();
    u.id=1;
    u.name="jett";
}
/**
 * 虚拟机优化技术--逃逸分析
 */
@Test
public void test(){
    long start=System.currentTimeMillis();
    //申请一亿个User
    for (int i = 0; i < 100000000; i++) {
        alloc();
    }
    long end=System.currentTimeMillis();
    System.out.println(end-start);
}
```

通过上面的测试，-XX:-DoEscapeAnalysis参数(虚拟机优化技术--逃逸分析)在打开和关闭下的性能相差上百倍。

4.13 StringBuffer StringBuilder在进行字符串操作时的效率；这里主要考察String在内存中是如何创建的（字节跳动）~jett

这道题想考察什么?

考察程序员对字符串的操作

考察的知识点

String在内存中是如何创建的

考生应该如何回答

以上三种字符串操作过程中，效率从低到高分别是String，StringBuilder，StringBuffer

String在进行字符串操作时，每次操作都会new StringBuilder，再进行append操作，效率很低

StringBuffer在进行字符串操作时，相关的API上都有synchronizd上锁，效率一般

StringBuilder在进行字符串操作时，性能是最好的