第5章 Java虚拟机原理面试题汇总

5.5 Java中对象如何晋升到老年代？
详细讲解
这道题想考察什么？
考察的知识点
考生如何回答
空间分配担保
为什么要进行空间担保？
总结
5.6 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）
详细讲解
这道题想考察什么？
考察的知识点
考生如何回答
5.7 Class会不会回收？用不到的Class怎么回收？(东方头条)
详细讲解
这道题想考察什么？
考察的知识点
考生应该如何回答
5.8 Java中有几种引用关系，它们的区别是什么？
详细讲解
这道题想考察什么？
考察的知识点
考生应该如何回答
5.9 描述JVM内存模型
详细讲解
这道题想考察什么？
考察的知识点
考生如何回答
内存模型
程序计数器
栈
本地方法栈
堆
方法区
运行时常量池
5.10 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）
这道题想考察什么？
考察的知识点
考生应该如何回答
5.11 StringBuffer与StringBuilder在进行字符串操作时的效率（字节跳动）
这道题想考察什么？
考察的知识点
考生应该如何回答
5.12 JVM DVM ART的区别
详细讲解
这道题想考察什么？
考察的知识点
考生应该如何回答
JVM
Dalvik
与JVM区别
基于的架构不同
执行的字节码不同
ART
与DVM的区别

5.1 描述JVM类加载过程

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《轻松通过大厂面试的Android JVM原理》

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

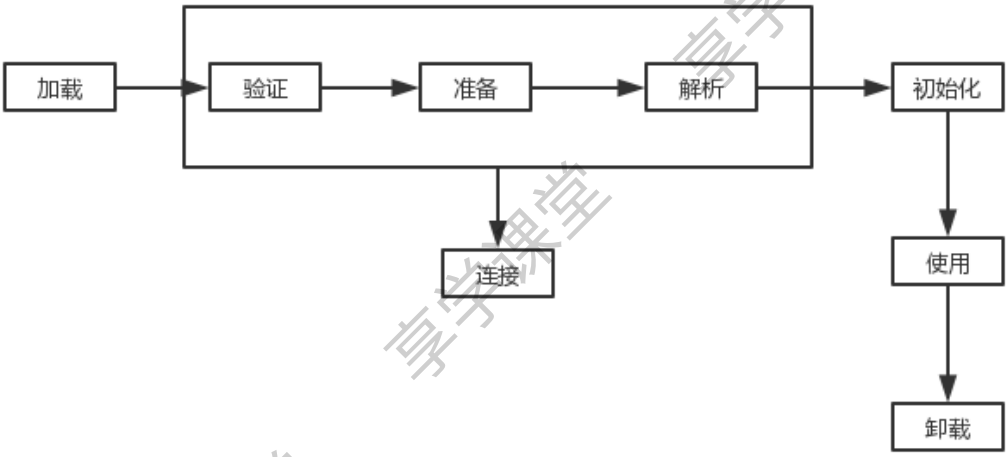
一般情况下，类的数据都是在`Class`文件中。将描述类的数据 从`Class`文件加载到内存 同时 对数据进行校验、转换解析 和 初始化，最终形成可被虚拟机直接使用的`Java`使用类型。

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。

5.2 请描述new一个对象的流程

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《轻松通过大厂面试的Android JVM原理》

这道题想考察什么？

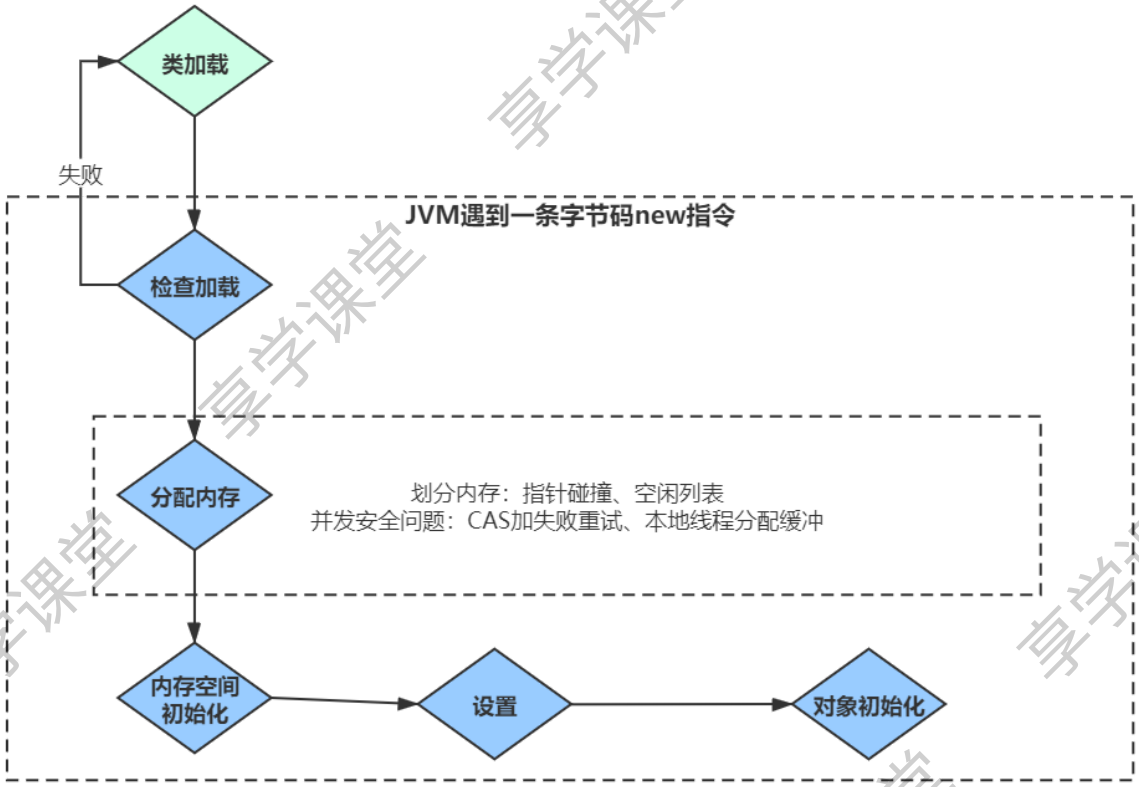
对JVM的理解

考察的知识点

JVM 对象分配、并发安全

考生应该如何回答

JVM创建对象的过程如下图：



虚拟机遇到一条new指令时，首先检查是否被类加载器加载，如果没有，那必须先执行相应的类加载过程。类加载就是把class加载到JVM的运行时数据区的过程。

检查加载

首先检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查类是否已经被加载、解析和初始化过。

> 符号引用：以一组符号来描述所引用的目标。符号引用可以是任何形式的字面量，JAVA在编译的时候一个每个java类都会被编译成一个class文件，但在编译的时候虚拟机并不知道所引用类的地址(实际地址)，就用符号引用来代替，而在类的解析阶段就是为了把这个符号引用转化成为真正的地址的阶段。
>
> 假设People类被编译成一个class文件时，如果People类引用了Tool类，但是在编译时People类并不知道引用类的实际内存地址，因此只能使用符号引用（org.simple.Tool）来代替。而在类装载器装载People类时，此时可以通过虚拟机获取Tool类的实际内存地址，因此便可以既将符号org.simple.Tool替换为Tool类的实际内存地址。

分配内存

完成类的加载检查后，虚拟机将为新生对象分配内存。为对象分配空间的任务等同于把一块确定大小的内存从Java堆中划分出来。

指针碰撞

如果Java堆中内存是绝对规整的，所有用过的内存都放在一边，空闲的内存放在另一边，中间放着一个指针作为分界点的指示器，那所分配内存就仅仅是把那个指针向空闲空间那边挪动一段与对象大小相等的距离，这种分配方式称为一指针碰撞。

空闲列表

如果Java堆中的内存并不是规整的，已使用的内存和空闲的内存相互交错，那就没有办法简单地进行指针碰撞了，虚拟机就必须维护一个列表，记录上哪些内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录，这种分配方式称为一空闲列表。

选择哪种分配方式由Java堆是否规整决定，而Java堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

- 如果是Serial、ParNew等带有压缩的整理的垃圾回收器的话，系统采用的是指针碰撞，既简单又高效。
- 如果是使用CMS这种不带压缩（整理）的垃圾回收器的话，理论上只能采用较复杂的空闲列表。

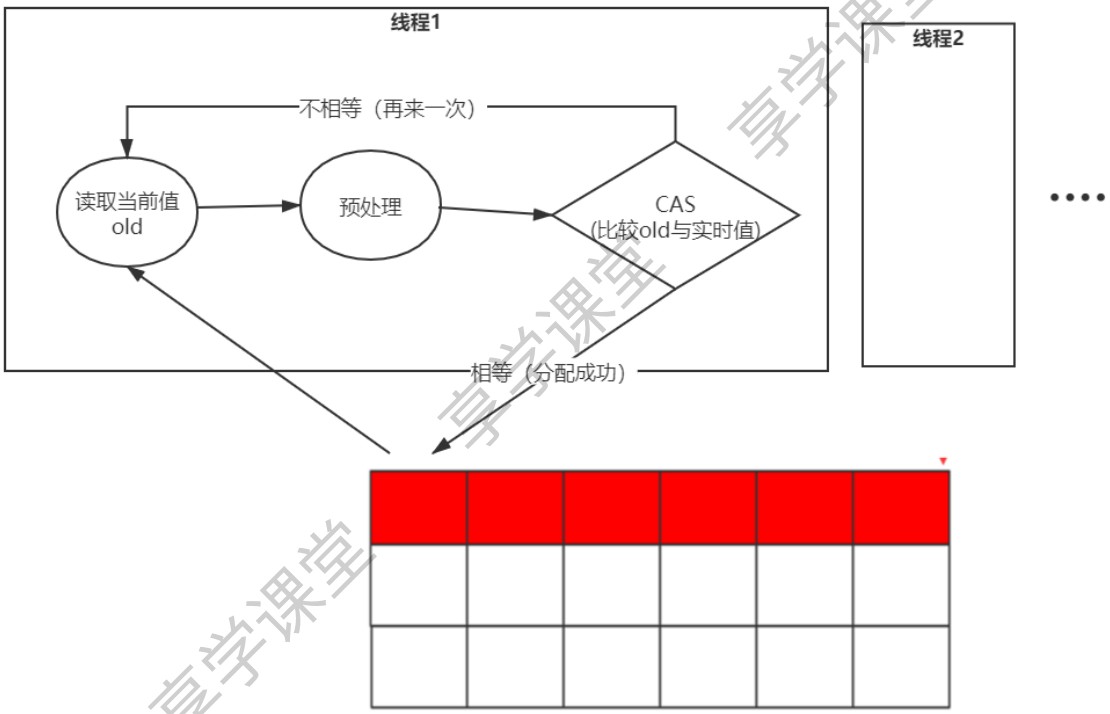
并发安全

除如何划分可用空间之外，还有另外一个需要考虑的问题是对象创建在虚拟机中是非常频繁的行为，即使是仅仅修改一个指针所指向的位置，在并发情况下也并不是线程安全的，可能出现正在给对象A分配内存，指针还没来得及修改，对象B又同时使用了原来的指针来分配内存的情况。

解决这个问题有两种方案：

CAS

对分配内存空间的动作进行同步处理一实际上虚拟机采用CAS配上失败重试的方式保证更新操作的原子性。



分配缓冲

把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在Java堆中预先分配一小块私有内存，也就是本地线程分配缓冲（Thread Local Allocation Buffer,TLAB）。

JVM在线程初始化时，同时也会申请一块指定大小的内存，只给当前线程使用，这样每个线程都单独拥有一个Buffer，如果需要分配内存，就在自己的Buffer上分配，这样就不存在竞争的情况，可以大大提升分配效率，当Buffer容量不够的时候，再重新从Eden区域申请一块继续使用。

TLAB的目的是在为新对象分配内存空间时，让每个Java应用线程能在使用自己专属的分配指针来分配空间，减少同步开销。

TLAB只是让每个线程有私有的分配指针，但底下存对象的内存空间还是给所有线程访问的，只是其它线程无法在这个区域分配而已。当一个TLAB用满（分配指针top撞上分配极限end了），就新申请一个TLAB。

默认情况下启用允许在年轻代空间中使用线程本地分配块（TLAB）。要禁用TLAB，需要指定`-XX:-UseTLAB`。

内存空间初始化

内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值。这一步操作保证了对象的实例字段在Java代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。(如int值为0，boolean值为false等等)。

设置

完成空间初始化后，虚拟机对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息（Java classes在Java hotspot VM内部表示为类元数据）、对象的哈希码、对象的GC分代年龄等信息。这些信息存放在对象的对象头之中。

对象初始化

在以上工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了。但从Java程序的视角来看，对象创建才刚刚开始，所有的字段都还为零值。所以，一般来说，执行new指令之后会接着把对象按照程序员的意愿进行初始化(构造方法)，这样一个真正可用的对象才算完全产生出来。

5.3 Java对象会不会分配到栈中？

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《轻松通过大厂面试的Android JVM原理》

这道题想考察什么？

创建的对象是否都在堆中，如果不是，对照JVM运行时数据区堆栈相关内容，能够把控对象不在堆中对程序的影响

考察的知识点

逃逸分析

考生应该如何回答

Java对象可能会分配到栈中。

逃逸分析

逃逸分析指的是分析对象动态作用域，当一个对象在方法中定义后，它可能被外部方法所引用。比如：调用参数传递到其他方法中，这种称之为方法逃逸。甚至还有可能被外部线程访问到，例如：赋值给其他线程中访问的变量，这个称之为线程逃逸。从不逃逸到方法逃逸到线程逃逸，称之为对象由低到高的不同逃逸程度。

如果确定一个对象不会逃逸出线程之外，那么让对象在栈上分配内存可以提高JVM的效率。如果是逃逸分析出来的对象可以在栈上分配的话，那么该对象的生命周期就跟随线程了，就不需要垃圾回收，如果是频繁的调用此方法则可以得到很大的性能提高。

> 逃逸分析的触发前提条件必须触发JIT执行

```
public class EscapeAnalysisTest{
    public static void main(String[] args){
        long start = System.currentTimeMillis();
        for (int i = 0; i < 50000000; i++){
            allocate();
        }
        System.out.println((System.currentTimeMillis() - start) + " ms");
    }
    static void allocate(){
        Object obj = new Object();
    }
}
```

在上述代码中，Object对象属于不可逃逸，JVM可以做栈上分配。在启动JVM时候，通过 `-XX:-DoEscapeAnalysis` 参数可以关闭逃逸分析（JVM默认开启）。

开启逃逸分析：

```
EscapeAnalysisTest
"C:\Program Files\Java\jdk1.8.0_101\bin\java.exe" ...
5 ms
```

关闭逃逸分析：

```
EscapeAnalysisTest
"C:\Program Files\Java\jdk1.8.0_101\bin\java.exe" ...
290 ms
```

测试结果可见，开启逃逸分析对代码的执行性能有很大的影响！

5.4 GC的流程是怎么样的？介绍下GC回收机制与分代回收策略

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《轻松通过大厂面试的Android JVM原理》

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

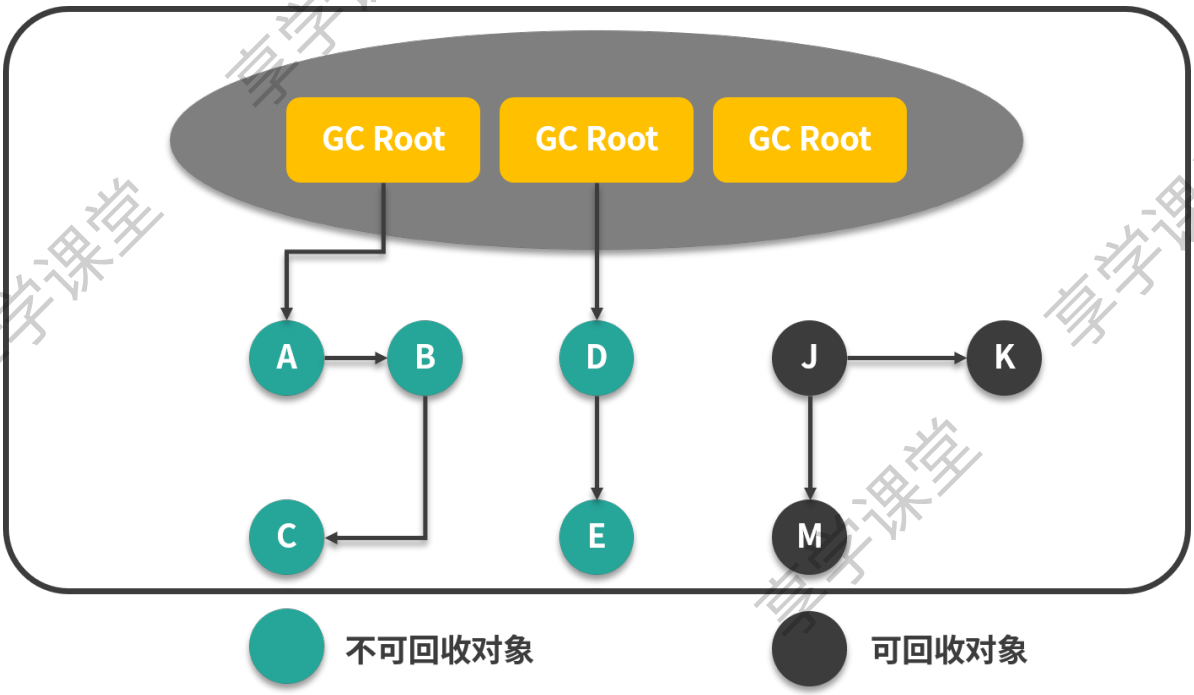
GC机制

考生如何回答

说到GC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

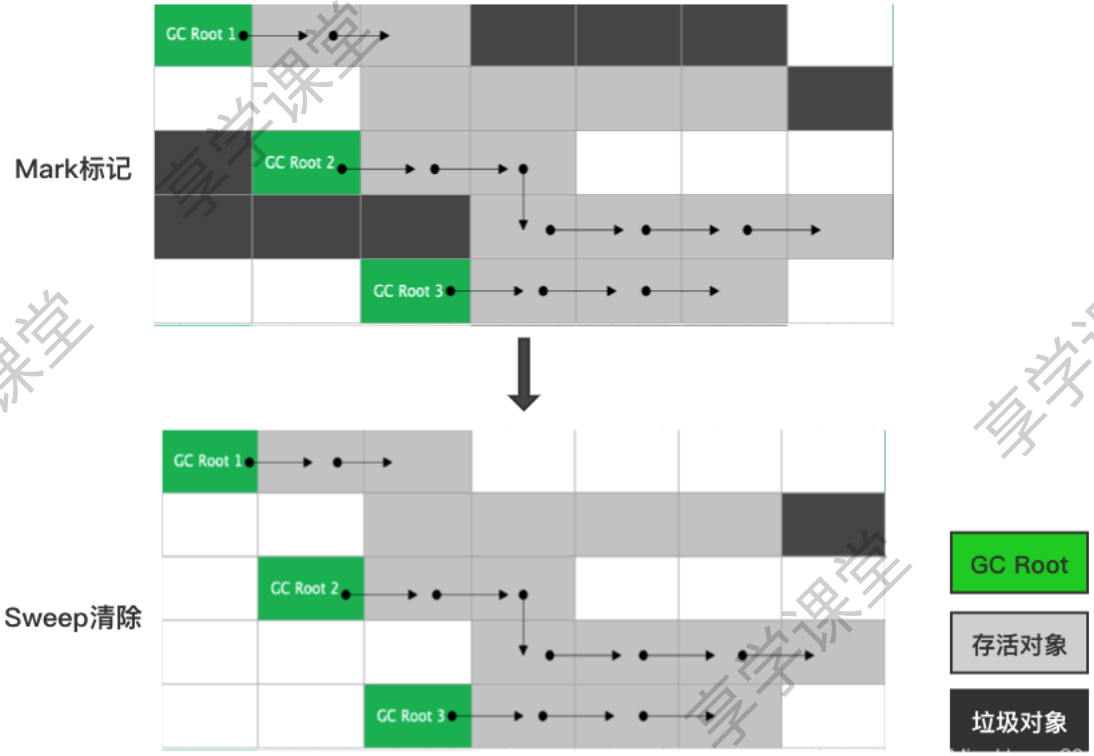
在多线程环境下，其他线程可能会更新已经访问过的对象中的引用，从而造成误报(将引用设置为null)或者漏报(将引用设置为未被访问过的对象)。误报并没有什么伤害，Java虚拟机至多损失了部分垃圾回收的机会。漏报则比较麻烦，因为垃圾回收器可能回收事实上仍被引用的对象内存。一旦从原引用访问已经被回收了的对象，则很有可能会直接导致Java虚拟机崩溃。

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。

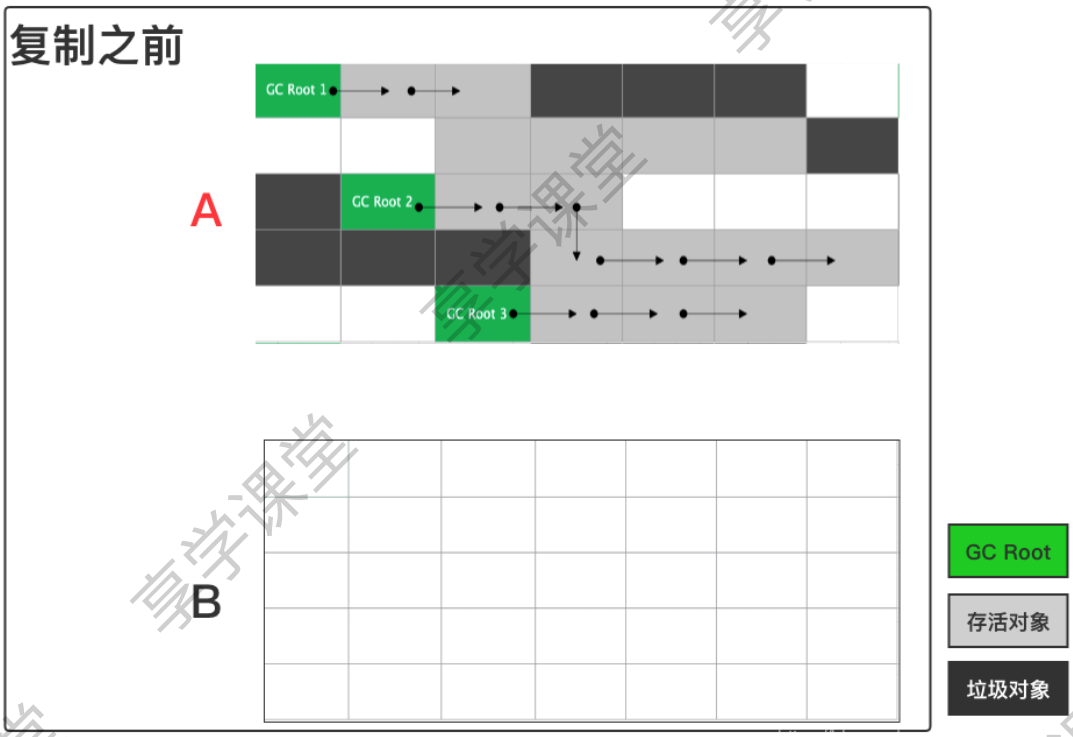
标记整理算法

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

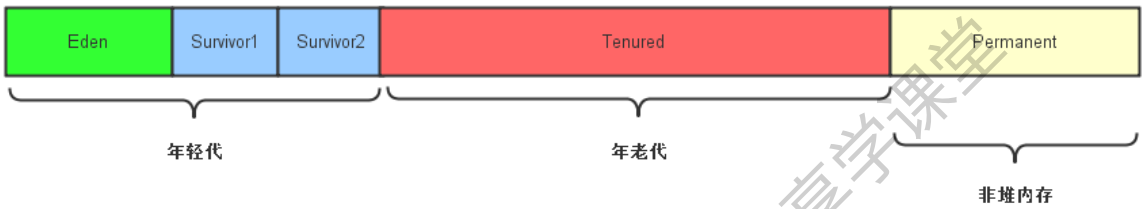
不同的垃圾收集器实现采用不同的算法进行垃圾回收，除此之外现代虚拟机还会采用分代机制来进行垃圾回收，根据对象存活的周期不同，把堆内存划分为不同区域，不同区域采用不同算法进行垃圾回收。

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



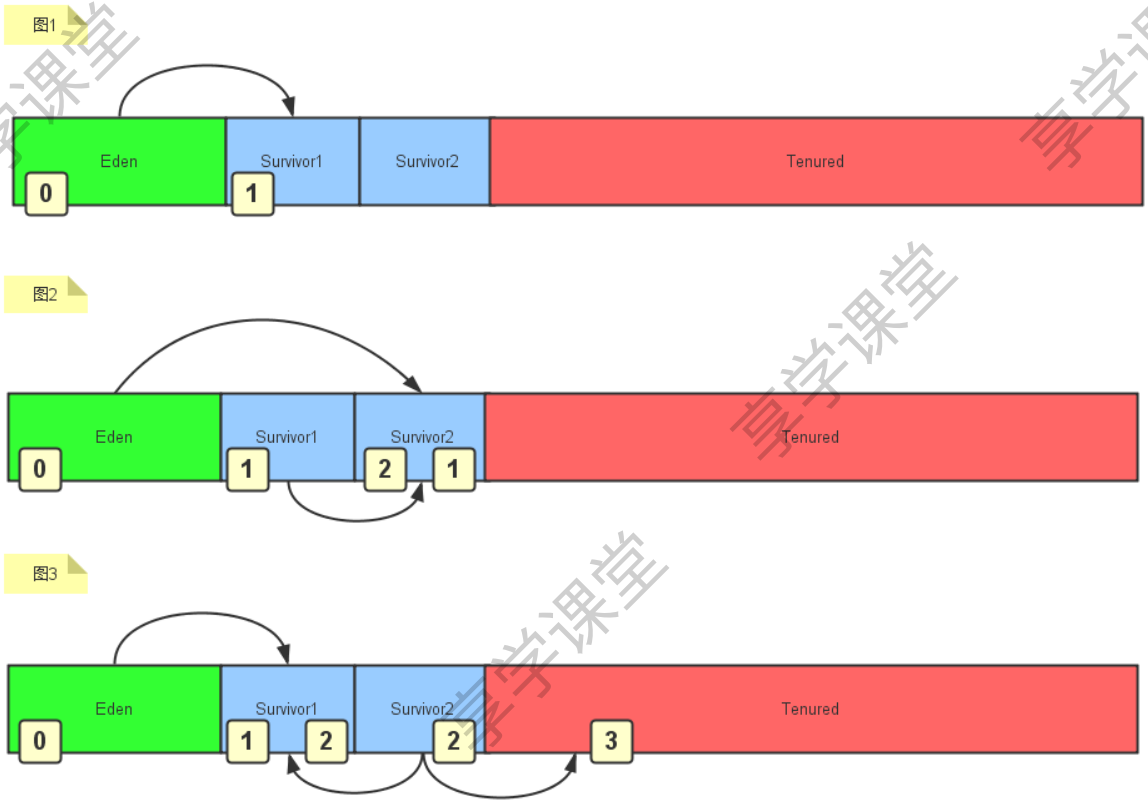
堆内存分为年轻代（Young Generation）和老年代（Old Generation）。而持久代使用非堆内存，主要用于存储一些类的元数据，常量池，java类，静态文件等信息。

垃圾回收

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，这是因为根据统计95%的对象朝生夕死，存活时间极短。

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

当新对象生成，并且在Eden申请空间失败时，就会触发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。所以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。



minor gc

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，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（年龄可以通过JVM参数进行指定，默认为15岁，CMS收集器默认6岁，不同的垃圾收集器会略微有点不同 ），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。

major gc

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。

5.5 Java中对象如何晋升到老年代？

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《轻松通过大厂面试的Android JVM原理》

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

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。

空间分配担保

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

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

5.6 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）

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《轻松通过大厂面试的Android JVM原理》

这道题想考察什么？

是否掌握可达性分析法了解如何确定对象是否可被回收，从而避免程序内存泄露问题

考察的知识点

GC机制、可达性分析发、引用计数

考生如何回答

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。

首先GC会帮助我们自动回收内存，那么GC是如何确定对象是否可被回收的？在**《5.4 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;
```

因此实际现在Java虚拟机都是采用的**可达性分析法**。

5.7 Class会不会回收？用不到的Class怎么回收？(东方头条)

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《轻松通过大厂面试的Android JVM原理》

这道题想考察什么？

JVM的内部机制

考察的知识点

GC机制、类加载机制

考生应该如何回答

Java 虚拟机理论上会回收Class，Class要被回收，条件比较"苛刻"，必须同时满足以下的条件：

1、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

2、加载该类的类加载器已经被回收

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

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` 参数禁用类的垃圾收集。

5.8 Java中有几种引用关系，它们的区别是什么？

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《轻松通过大厂面试的Android JVM原理》

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，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
}
```

5.9 描述JVM内存模型

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《轻松通过大厂面试的Android JVM原理》

这道题想考察什么？

JVM如何分配资源执行程序

考察的知识点

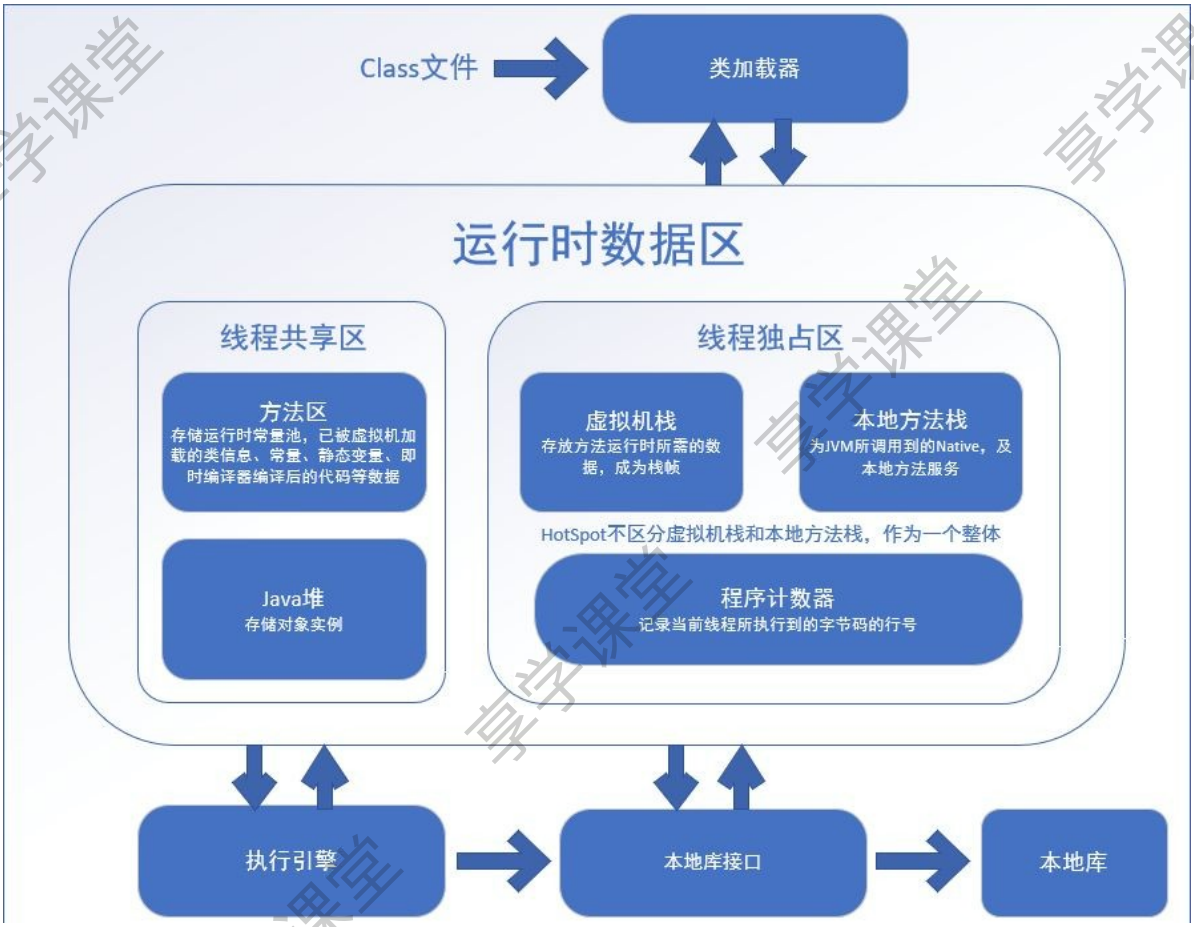
JVM运行时数据区

考生如何回答

Java的口号是：“Write once, run anywhere”，即一次编写，到处运行。为什么可以做到这样呢，其实就是依赖于JVM。在不同的操作系统上，只要安装了对应的虚拟机，那么同样的一份代码，就可以随意移植。
当编写完Java代码时，即产生 .Java文件，会通过Java编译器编译为.class 文件，然后通过Class Loader把类信息加载到JVM中，最后JVM再去调用操作系统。这样，只要JVM正确执行.class文件，就可以实现跨平台了。

内存模型

JVM和计算机一样有操作栈和程序计数器，运行的方式也基本一致。JVM同样以线程为最小单位运行，其中防御去与堆属于线程共享区，而栈与程序计数器属于线程独占区。



程序计数器

程序计数器是一块较小的内存，可以看作是当前线程所执行的字节码的行号指示器，即记录当前线程所执行到的字节码的行号。当字节码解释器工作时，就是通过改变计数器的值来选取下一条需要执行的字节码指令。由此来完成分支、循环、跳转、线程恢复、异常处理等功能。
程序计数器是线程私有的（即每个线程拥有一个程序计数器），各个线程之间的程序计数器互不干扰。
程序计数器的生命周期跟随线程的生命周期，若线程消亡，则程序计数器也会消亡。
如果一个线程正在执行的是Java方法，则程序计数器记录的是正在执行的字节码指令的地址；如果正在执行的是 native 本地方法，则程序计数器记录的是 Undefined .

栈

指的是Java虚拟机栈，它也是线程私有的，因此生命周期和线程相同。每当线程创建的时候，都会创建一个私有的Java虚拟机栈。Java栈中保存了局部变量和方法参数等，同时和Java方法的调用、返回密切相关。
每个方法在执行的同时都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用到执行完成的过程，就对应一个栈帧在虚拟机栈中从入栈到出栈的过程。

本地方法栈

本地方法栈和Java虚拟机栈非常类似，它们最大的不同在于，Java虚拟机栈用于Java方法的调用，而本地方法栈用于Native本地方法的调用。

堆

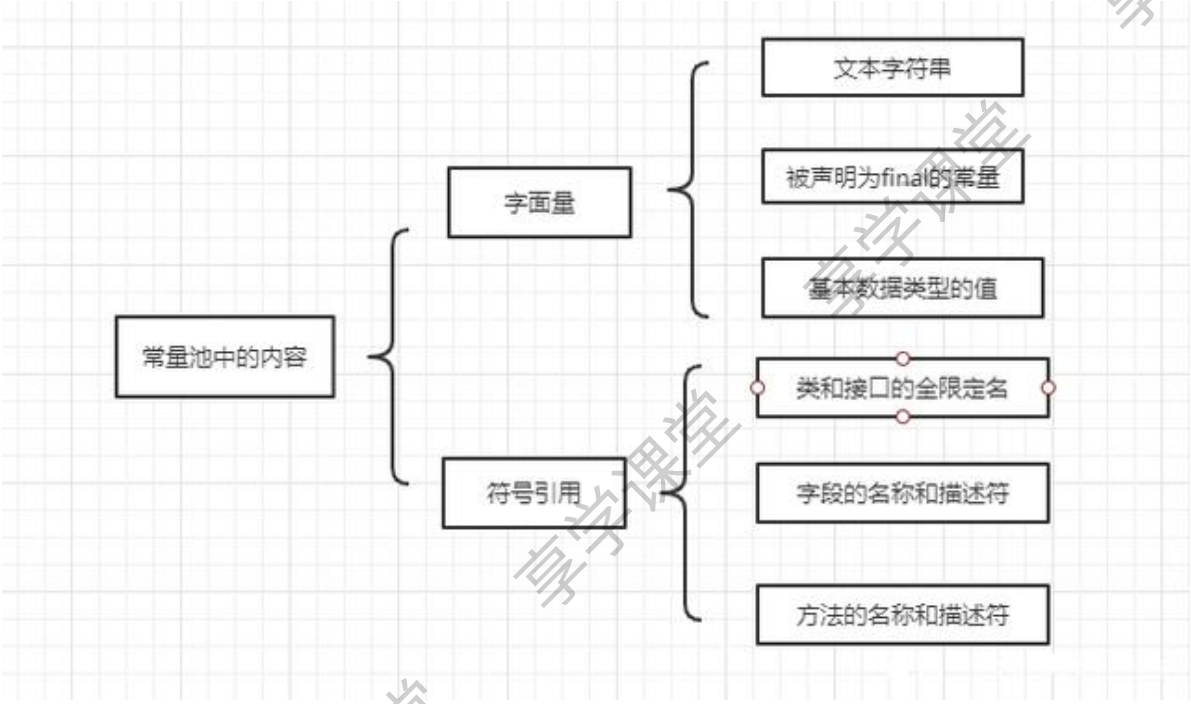
Java堆是所有线程共享的一块内存区域，在虚拟机启动时创建。对于绝大多数应用来说，Java堆是JVM所管理的内存中最大的一块，几乎所有的对象实例和数组都存放在这里。
Java堆也是垃圾收集器管理的主要区域。堆中分为新生代、老年代和永久代，新生代还可细分为Eden区、From、To 区。当堆中没有内存可分配时，就会抛出OOM异常。

方法区

方法区同Java堆一样，也是所有线程共享的内存区域。用于存储已被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。在JDK8以前，HotSpot是用“永久代”来实现方法区的，其他虚拟机（如JRockit、J9VM）不存在永久代这个概念。这样的话，方法区可以和Java堆一样被 HotSpot的垃圾收集器所管理，不需要单独处理。
由于我们可以通过 -XX:MaxPermSize 来设置永久代大小，因此若使用永久代来实现方法区，则会有内存溢出的风险。因此，在JDk8中，取消了永久代，用元空间代替之。也就是说，用元空间来实现方法区。
元空间的本质和永久代类似，都是对JVM规范中方法区的实现。元空间与永久代之间最大的区别在于：永久代是堆的一部分，和新生代，老年代地址是连续的。元空间并不在虚拟机中，而属于 Native Memeory（本地内存）。因此，默认情况下，元空间的大小仅受本地内存限制。

运行时常量池

首先需要知道常量池和运行时常量池的区别。
常量池，即指class文件常量池，是class文件的一部分。java文件被编译成class文件之后，除了包含了类的版本、字段、方法、接口等描述信息，还有一项信息叫做class文件常量池。其用于存放编译期生成的各种字面量和符号引用。



运行时常量池是方法区的一部分。当类加载到内存中，JVM就会将class文件常量池中的内容（字面量和符号引用）存放到运行时常量池中。
Java并不要求常量一定只有在编译期才可以产生，在运行期间也可以产生新的常量并放入池中。

5.10 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）

这道题想考察什么？

JVM内存

考察的知识点

内存泄露，JVM运行时数据区

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主要是堆区申请的内存空间不够用时出现，比如单次申请大对象超出了堆中连续的可用空间。

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。

5.11 StringBuffer与StringBuilder在进行字符串操作时的效率（字节跳动）

这道题想考察什么？

在开发中需要进行字符串操作时如何合理利用StringBuffer与StringBuilder

考察的知识点

字符串工具类

考生应该如何回答

当对字符串进行修改的时候，使用 StringBuffer 和 StringBuilder 能够多次的修改，并且不产生新的未使用字符串对象。

String在进行字符串操作时，每次操作都会new StringBuilder，如进行字符串拼接:

```
String result = "Hello ";
for(int i = 0; i < 100; i++){
    result += "Hello ";
}
```

上述代码等价于:

```
String result = "Hello ";
for(int i = 0; i < 100; i++){
   result = new StringBuilder().append(result).append("Hello ");
}
```

可以看到每次循环都需要创建一个StringBuilder 使用其`append`方法完成字符串拼接。频繁创建与回收对象会导致 "内存抖动" 问题，引发程序卡顿甚至OOM。

因此相对直接使用String进行字符串操作，应将代码优化为：

```
StringBuilder result = new StringBuilder("Hello ");
for(int i = 0; i < 100; i++){
   result.append("Hello ");
}
```

而StringBuffer与StringBuilder的功能与实现原理一致，两者的区别在于，StringBuffer中的方法都被关键字`synchronized`声明，这意味着其是线程安全的，但是因为每次执行方法都需要获取锁，因此效率比StringBuilder稍低。

在涉及到多线程环境下操作字符串时，应该使用StringBuffer；否则应该选择StringBuilder效率更高。

5.12 JVM DVM ART的区别

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《轻松通过大厂面试的Android JVM原理》

这道题想考察什么？

Android虚拟机与Java虚拟机的差异

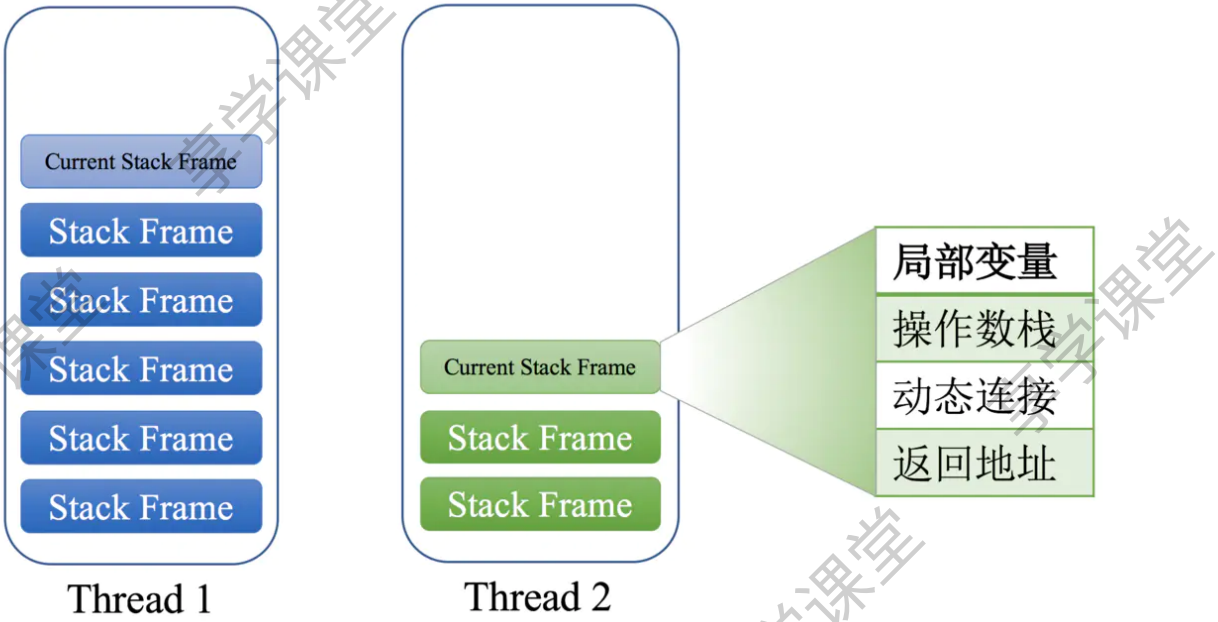
考察的知识点

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

JVM

JVM是基于栈的虚拟机，对于基于栈的虚拟机来说，每一个运行时的线程，都有一个独立的栈。栈中记录了方法调用的历史，每有一次方法调用，栈中便会多一个栈桢。最顶部的栈桢称作当前栈桢，其代表着当前执行的方法。基于栈的虚拟机通过操作数栈进行所有操作。



在JVM中执行字节码，将

```
int a = 1;
int b = 2;
int c = a + b;
```

编译为字节码，得到的指令为：

```
ICONST_1  #将int类型常量1压入操作数栈
ISTORE 0  #将栈顶int类型值存入局部变量0
ICONST_2
ISTORE 1
ILOAD 0  #从局部变量表加载0：int类型数据
ILOAD 1
IADD     #执行int类型加法
ISTORE 2
```

数据会不断在操作数栈与局部变量表之间移动。

Dalvik

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。

与JVM区别

基于的架构不同

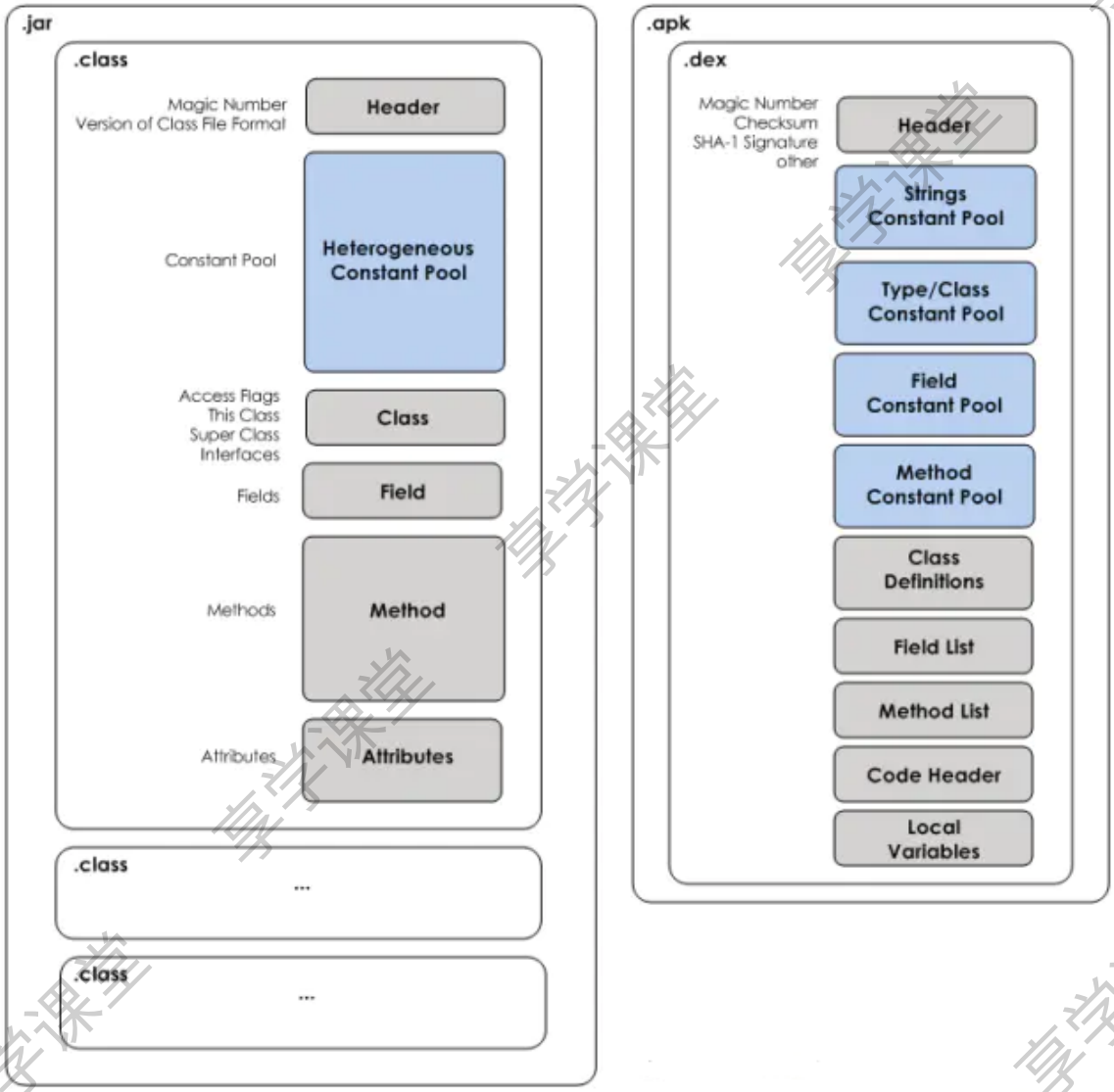
DVM是基于寄存器的虚拟机，并没有JVM栈帧中的操作数栈，但是有很多虚拟寄存器。其实和操作数栈相同，这些寄存器也存放在运行时栈中，本质上就是一个数组。与JVM相似，在Dalvik VM中每个线程都有自己的PC和调用栈，方法调用的活动记录以帧为单位保存在调用栈上。

与JVM版相比，Dalvik无需从栈中读写数据，所需的指令更少，数据也不再需要频繁的移动。

```
#int a = 1
const/4 v0, #int 1 // #1
#int b = 2
const/4 v1, #int 2 // #2
#int c = a + b
add-int v2, v0, v1
```

执行的字节码不同

- 在Java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。
- DVM会用dx工具将所有的.class文件或者jar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。



.jar文件在中包含多个.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。

而.dex文件中也包含很多类信息，但是dex将多个类信息整合在一起了，多个类复用常量池等区域。

ART

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。

与DVM的区别

Dalvik下应用在安装的过程，会执行一次优化，将dex字节码进行优化生成odex文件。

ART能够兼容Dalvik中的字节码执行。但是ART 引入了预先编译机制（Ahead Of Time），在Android4.4到Android 7.0以下的设备中安装应用时，ART 会使用 dex2oat 程序编译应用，将应用中dex编译成本地机器码。但是此过程会导致应用安装变慢。

因此，从Android N(Android 7.0)开始，ART 混合使用AOT、JIT与解释执行：

1、最初安装应用时不进行任何 AOT 编译（避免安装过慢），运行过程中解释执行，对经常执行的方法进行JIT，经过 JIT 编译的方法将会记录到Profile配置文件中。

2、当设备闲置和充电时，编译守护进程会运行，根据Profile文件对常用代码进行 AOT 编译。待下次运行时直接使用。