

# 2019Android 高级面试题总结

## 1. 说下你所知道的设计模式与使用场景

### a.建造者模式:

将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

使用场景比如最常见的 `AlertDialog`,拿我们开发过程中举例，比如 `Camera` 开发过程中，可能需要设置一个初始化的相机配置，设置摄像头方向，闪光灯开闭，成像质量等等，这种场景下就可以使用建造者模式

装饰者模式：动态的给一个对象添加一些额外的职责，就增加功能来说，装饰模式比生成子类更为灵活。装饰者模式可以在不改变原有类结构的情况下增强类的功能，比如 `Java` 中的 `BufferedInputStream` 包装 `FileInputStream`，举个开发中的例子，比如在我们现有网络框架上需要增加新的功能，那么再包装一层即可，装饰者模式解决了继承存在的一些问题，比如多层继承代码的臃肿，使代码逻辑更清晰

观察者模式：

代理模式：

门面模式：

单例模式：

生产者消费者模式：

## 2. java 语言的特点与 OOP 思想

这个通过对比来描述，比如面向对象和面向过程的对比，针对这两种思想的对比，还可以举个开发中的例子，比如播放器的实现，面向过程的实现方式就是将播放视频的这个功能分解成多个过程，比如，加载视频地址，获取视频信息，初始化解码器，选择合适的解码器进行解码，读取解码后的帧进行视频格式转换和音频重采样，然后读取帧进行播放，这是一个完整的过程，这个过程中不涉及类的概念，而面向对象最大的特点就是类，封装继承和多态是核心，同样的以播放器为例，一面向对象的方式来实现，将会针对每一个功能封装出一个对象，比如说 `Muxer`，获取视频信息，`Decoder`,解码，格式转换器，视频播

放器，音频播放器等，每一个功能对应一个对象，由这个对象来完成对应的功能，并且遵循单一职责原则，一个对象只做它相关的事情

### 3. 说下 java 中的线程创建方式，线程池的工作原理。

java 中有三种创建线程的方式，或者说四种

1.继承 Thread 类实现多线程

2.实现 Runnable 接口

3.实现 Callable 接口

4.通过线程池

线程池的工作原理：线程池可以减少创建和销毁线程的次数，从而减少系统资源的消耗，当一个任务提交到线程池时

- a. 首先判断核心线程池中的线程是否已经满了，如果没满，则创建一个核心线程执行任务，否则进入下一步
- b. 判断工作队列是否已满，没有满则加入工作队列，否则执行下一步
- c. 判断线程数是否达到了最大值，如果不是，则创建非核心线程执行任务，否则执行饱和策略，默认抛出异常

### 4. 说下 handler 原理

Handler, Message, looper 和 MessageQueue 构成了安卓的消息机制，handler 创建后可以通过 sendMessage 将消息加入消息队列，然后 looper 不断的将消息从 MessageQueue 中取出来，回调到 Handler 的 handleMessage 方法，从而实现线程的通信。

从两种情况来说，第一在 UI 线程创建 Handler,此时我们不需要手动开启 looper，因为在应用启动时，在 ActivityThread 的 main 方法中就创建了一个当前主线程的 looper，并开启了消息队列，消息队列是一个无限循环，为什么无限循环不会 ANR?因为可以说，应用的整个生命周期就是运行在这个消息循环中的，安卓是由事件驱动的，Looper.loop 不断的接收处理事件，每一个点击触摸或者 Activity 每一个生命周期都是在 Looper.loop 的控制之下的，looper.loop 一旦结束，应用程序的生命周期也就结束了。我们可以想想什么情况下会发生 ANR，第一，事件没有得到处理，第二，事件正在处理，但是没有及时完成，

而对事件进行处理的就是 **looper**，所以只能说事件的处理如果阻塞会导致 **ANR**，而不能说 **looper** 的无限循环会 **ANR**

另一种情况就是在子线程创建 **Handler**，此时由于这个线程中没有默认开启的消息队列，所以我们需要手动调用 **looper.prepare()**，并通过 **looper.loop** 开启消息主线程 **Looper** 从消息队列读取消息，当读完所有消息时，主线程阻塞。子线程往消息队列发送消息，并且往管道文件写数据，主线程即被唤醒，从管道文件读取数据，主线程被唤醒只是为了读取消息，当消息读取完毕，再次睡眠。因此 **loop** 的循环并不会对 **CPU** 性能有过多的消耗。

## 5. 内存泄漏的场景和解决办法

### 1.非静态内部类的静态实例

非静态内部类会持有外部类的引用，如果非静态内部类的实例是静态的，就会长期的维持着外部类的引用，组织被系统回收，解决办法是使用静态内部类

### 2.多线程相关的匿名内部类和非静态内部类

匿名内部类同样会持有外部类的引用，如果在线程中执行耗时操作就有可能发生内存泄漏，导致外部类无法被回收，直到耗时任务结束，解决办法是在页面退出时结束线程中的任务

### 3.Handler 内存泄漏

**Handler** 导致的内存泄漏也可以被归纳为非静态内部类导致的，**Handler** 内部 **message** 是被存储在 **MessageQueue** 中的，有些 **message** 不能马上被处理，存在的时间会很长，导致 **handler** 无法被回收，如果 **handler** 是非静态的，就会导致它的外部类无法被回收，解决办法是 1.使用静态 **handler**，外部类引用使用弱引用处理 2.在退出页面时移除消息队列中的消息

### 4.Context 导致内存泄漏

根据场景确定使用 **Activity** 的 **Context** 还是 **Application** 的 **Context**，因为二者生命周期不同，对于不必须使用 **Activity** 的 **Context** 的场景（**Dialog**），一律采用 **Application** 的 **Context**，单例模式是最常见的发生此泄漏的场景，比如传入一个 **Activity** 的 **Context** 被静态类引用，导致无法回收

### 5.静态 View 导致泄漏

使用静态 **View** 可以避免每次启动 **Activity** 都去读取并渲染 **View**，但是静态 **View**

会持有 Activity 的引用，导致无法回收，解决办法是在 Activity 销毁的时候将静态 View 设置为 null（View 一旦被加载到界面中将会持有一个 Context 对象的引用，在这个例子中，这个 context 对象是我们的 Activity，声明一个静态变量引用这个 View，也就引用了 activity）

#### 6.WebView 导致的内存泄漏

WebView 只要使用一次，内存就不会被释放，所以 WebView 都存在内存泄漏的问题，通常的解决办法是为 WebView 单开一个进程，使用 AIDL 进行通信，根据业务需求在合适的时机释放掉

#### 7.资源对象未关闭导致

如 Cursor，File 等，内部往往都使用了缓冲，会造成内存泄漏，一定要确保关闭它并将引用置为 null

#### 8.集合中的对象未清理

集合用于保存对象，如果集合越来越大，不进行合理的清理，尤其是入股集合是静态的

#### 9.Bitmap 导致内存泄漏

bitmap 是比较占内存的，所以一定要在不使用的时候及时进行清理，避免静态变量持有大的 bitmap 对象

#### 10.监听器未关闭

很多需要 register 和 unregister 的系统服务要在合适的时候进行 unregister,手动添加的 listener 也需要及时移除

### 6. 如何避免 OOM?

#### 1.使用更加轻量的数据结构：如使用 ArrayMap/SparseArray 替代

HashMap,HashMap 更耗内存，因为它需要额外的实例对象来记录 Mapping 操作，SparseArray 更加高效，因为它避免了 Key Value 的自动装箱，和装箱后的解箱操作

#### 2.便面枚举的使用，可以用静态常量或者注解@IntDef 替代

#### 3.Bitmap 优化:

a.尺寸压缩：通过 InSampleSize 设置合适的缩放

b.颜色质量：设置合适的 format，ARGB\_6666/RBG\_545/ARGB\_4444/ALPHA\_6，

存在很大差异

**c.inBitmap:**使用 inBitmap 属性可以告知 Bitmap 解码器去尝试使用已经存在的内存区域，新解码的 Bitmap 会尝试去使用之前那张 Bitmap 在 Heap 中所占据的 pixel data 内存区域，而不是去问内存重新申请一块区域来存放 Bitmap。利用这种特性，即使是上千张图片，也只会仅仅只需要占用屏幕所能够显示的图片数量的内存大小，但复用存在一些限制，具体体现在：在 Android 4.4 之前只能重用相同大小的 Bitmap 的内存，而 Android 4.4 及以后版本则只要后来的 Bitmap 比之前的小即可。使用 inBitmap 参数前，每创建一个 Bitmap 对象都会分配一块内存供其使用，而使用了 inBitmap 参数后，多个 Bitmap 可以复用一块内存，这样可以提高性能

**4.StringBuilder 替代 String:** 在有些时候，代码中会需要使用到大量的字符串拼接的操作，这种时候有必要考虑使用 StringBuilder 来替代频繁的“+”

**5.避免在类似 onDraw 这样的方法中创建对象，因为它会迅速占用大量内存，引起频繁的 GC 甚至内存抖动**

**6.减少内存泄漏也是一种避免 OOM 的方法**

**7. 说下 Activity 的启动模式，生命周期，两个 Activity 跳转的生命周期，如果一个 Activity 跳转另一个 Activity 再按下 Home 键在回到 Activity 的生命周期是什么样的**

启动模式

**Standard 模式:**Activity 可以有多个实例，每次启动 Activity，无论任务栈中是否已经有这个 Activity 的实例，系统都会创建一个新的 Activity 实例

**SingleTop 模式:**当一个 singleTop 模式的 Activity 已经位于任务栈的栈顶，再去启动它时，不会再创建新的实例,如果不位于栈顶，就会创建新的实例

**SingleTask 模式:**如果 Activity 已经位于栈顶，系统不会创建新的 Activity 实例，和 singleTop 模式一样。但 Activity 已经存在但不位于栈顶时，系统就会把该 Activity 移到栈顶，并把它上面的 activity 出栈

**SingleInstance 模式:**singleInstance 模式也是单例的，但和 singleTask 不同，

singleTask 只是任务栈内单例，系统里是可以有多个 singleTask Activity 实例的，而 singleInstance Activity 在整个系统里只有一个实例，启动一个 singleInstanceActivity 时，系统会创建一个新的任务栈，并且这个任务栈只有他一个 Activity

生命周期

onCreate onStart onResume onPause onStop onDestroy

两个 Activity 跳转的生命周期

1.启动 A

onCreate - onStart - onResume

2.在 A 中启动 B

ActivityA onPause

ActivityB onCreate

ActivityB onStart

ActivityB onResume

ActivityA onStop

3.从 B 中返回 A（按物理硬件返回键）

ActivityB onPause

ActivityA onRestart

ActivityA onStart

ActivityA onResume

ActivityB onStop

ActivityB onDestroy

4.继续返回

ActivityA onPause

ActivityA onStop

ActivityA onDestroy

8. onRestart 的调用场景

- (1) 按下 home 键之后，然后切换回来，会调用 `onRestart()`。
  - (2) 从本 Activity 跳转到另一个 Activity 之后，按 back 键返回原来 Activity，会调用 `onRestart()`；
  - (3) 从本 Activity 切换到其他的应用，然后再从其他应用切换回来，会调用 `onRestart()`；
- 说下 Activity 的横竖屏的切换的生命周期，用那个方法来保存数据，两者的区别。触发在什么时候在那个方法里可以获取数据等。

## 9. 是否了 SurfaceView，它是什么？他的继承方式是什么？他与 View 的区别(从源码角度，如加载，绘制等)。

SurfaceView 中采用了双缓冲机制，保证了 UI 界面的流畅性，同时 SurfaceView 不在主线程中绘制，而是另开辟一个线程去绘制，所以它不妨碍 UI 线程；

SurfaceView 继承于 View，他和 View 主要有以下三点区别：

- (1) View 底层没有双缓冲机制，SurfaceView 有；
- (2) view 主要适用于主动更新，而 SurfaceView 适用与被动的更新，如频繁的刷新
- (3) view 会在主线程中去更新 UI，而 SurfaceView 则在子线程中刷新；

SurfaceView 的内容不在应用窗口上，所以不能使用变换（平移、缩放、旋转等）。也难以放在 ListView 或者 ScrollView 中，不能使用 UI 控件的一些特性比如 `View.setAlpha()`

**View：**显示视图，内置画布，提供图形绘制函数、触屏事件、按键事件函数等；必须在 UI 主线程内更新画面，速度较慢。

**SurfaceView：**基于 view 视图进行拓展的视图类，更适合 2D 游戏的开发；是 view 的子类，类似使用双缓机制，在新的线程中更新画面所以刷新界面速度比 view 快，Camera 预览界面使用 SurfaceView。

**GLSurfaceView：**基于 SurfaceView 视图再次进行拓展的视图类，专用于 3D 游戏开发的视图；是 SurfaceView 的子类，openGL 专用。

## 10. 如何实现进程保活

a: Service 设置成 START\_STICKY kill 后会被重启(等待 5 秒左右), 重传 Intent, 保持与重启前一样

b: 通过 startForeground 将进程设置为前台进程, 做前台服务, 优先级和前台应用一个级别, 除非在系统内存非常缺, 否则此进程不会被 kill

c: 双进程 Service: 让 2 个进程互相保护对方, 其中一个 Service 被清理后, 另外没被清理的进程可以立即重启进程

d: 用 C 编写守护进程(即子进程): Android 系统中当前进程(Process)fork 出来的子进程, 被系统认为是两个不同的进程。当父进程被杀死的时候, 子进程仍然可以存活, 并不受影响(Android5.0 以上的版本不可行)联系厂商, 加入白名单

e.锁屏状态下, 开启一个一像素 Activity

## 11. 说下冷启动与热启动是什么, 区别, 如何优化, 使用场景等。

app 冷启动: 当应用启动时, 后台没有该应用的进程, 这时系统会重新创建一个新的进程分配给该应用, 这个启动方式就叫做冷启动(后台不存在该应用进程)。冷启动因为系统会重新创建一个新的进程分配给它, 所以会先创建和初始化 Application 类, 再创建和初始化 MainActivity 类(包括一系列的测量、布局、绘制), 最后显示在界面上。

app 热启动: 当应用已经被打开, 但是被按下返回键、Home 键等按键时回到桌面或者是其他程序的时候, 再重新打开该 app 时, 这个方式叫做热启动(后台已经存在该应用进程)。热启动因为会从已有的进程中来启动, 所以热启动就不会走 Application 这步了, 而是直接走 MainActivity(包括一系列的测量、布局、绘制), 所以热启动的过程只需要创建和初始化一个 MainActivity 就行了, 而不必创建和初始化 Application

### 冷启动的流程

当点击 app 的启动图标时, 安卓系统会从 Zygote 进程中 fork 创建出一个新的进程分配给该应用, 之后会依次创建和初始化 Application 类、创建 MainActivity 类、加载主题样式 Theme 中的 windowBackground 等属性设置给 MainActivity 以及配置 Activity 层级上的一些属性、再 inflate 布局、当 onCreate/onStart/onResume 方法都走完了后最后才进行 contentView 的 measure/layout/draw 显示在界面上



冷启动的生命周期简要流程：

Application 构造方法 → attachBaseContext() → onCreate → Activity 构造方法 → onCreate() → 配置主体中的背景等操作 → onStart() → onResume() → 测量、布局、绘制显示

冷启动的优化主要是视觉上的优化，解决白屏问题，提高用户体验，所以通过上面冷启动的过程。能做的优化如下：

- 1、减少 onCreate() 方法的工作量
- 2、不要让 Application 参与业务的操作
- 3、不要在 Application 进行耗时操作
- 4、不要以静态变量的方式在 Application 保存数据
- 5、减少布局的复杂度和层级
- 6、减少主线程耗时

## 12. 为什么冷启动会有白屏黑屏问题？

原因在于加载主题样式 Theme 中的 windowBackground 等属性设置给 MainActivity 发生在 inflate 布局当 onCreate/onStart/onResume 方法之前，而 windowBackground 背景被设置成了白色或者黑色，所以我们进入 app 的第一个界面的时候会造成先白屏或黑屏一下再进入界面。解决思路如下

1. 给他设置 windowBackground 背景跟启动页的背景相同，如果你的启动页是张图片那么可以直接给 windowBackground 这个属性设置该图片那么就不会有一闪的效果了

```
<style name="" "Splash_Theme"
    parent="" "@android:style/Theme.NoTitleBar" "" >

    <item
name="" "android:windowBackground" "" >@drawable/splash_bg</item>

    <item name="" "android:windowNoTitle" "" >true</item></style>
```

2. 采用世面的处理方法，设置背景是透明的，给人一种延迟启动的感觉。将背景颜色设置为透明色，这样当用户点击桌面 APP 图片的时候，并不会"立即"进入 APP，而且在桌面上停留一会，其实这时候 APP 已经是启动的了，只是我们心

机的把 Theme 里的 windowBackground 的颜色设置成透明的，强行把锅甩给了手机应用厂商（手机反应太慢了啦）

```
<style name="" "Splash_Theme"
`parent="" "@android:style/Theme.NoTitleBar""`>

    <item name="" "android:windowIsTranslucent""`>`true`</item>`

    <item name="" "android:windowNoTitle""`>`true`</item>`</style>`
```

3.以上两种方法是在视觉上显得更快，但其实只是一种表象，让应用启动的更快，有一种思路，将 Application 中的不必要的初始化动作实现懒加载，比如，在 SplashActivity 显示后再发送消息到 Application，去初始化，这样可以将初始化的动作放在后边，缩短应用启动到用户看到界面的时间

### 13. Android 中的线程有那些, 原理与各自特点

AsyncTask,HandlerThread,IntentService

AsyncTask 原理：内部是 Handler 和两个线程池实现的，Handler 用于将线程切换到主线程，两个线程池一个用于任务的排队，一个用于执行任务，当

AsyncTask 执行 execute 方法时会封装出一个 FutureTask 对象，将这个对象加入队列中，如果此时没有正在执行的任务，就执行它，执行完成之后继续执行队列中下一个任务，执行完成通过 Handler 将事件发送到主线程。AsyncTask 必须在主线程初始化，因为内部的 Handler 是一个静态对象，在 AsyncTask 类加载的时候他已经被初始化了。在 Android3.0 开始，execute 方法串行执行任务的，一个一个来，3.0 之前是并行执行的。如果要在 3.0 上执行并行任务，可以调用 executeOnExecutor 方法

HandlerThread 原理：继承自 Thread，start 开启线程后，会在其 run 方法中会通过 Looper 创建消息队列并开启消息循环，这个消息队列运行在子线程中，所以可以将 HandlerThread 中的 Looper 实例传递给一个 Handler，从而保证这个 Handler 的 handleMessage 方法运行在子线程中，Android 中使用 HandlerThread 的一个场景就是 IntentService

IntentService 原理：继承自 Service，它的内部封装了 HandlerThread 和 Handler，可以执行耗时任务，同时因为它是一个服务，优先级比普通线程高很多，所以更适合执行一些高优先级的后台任务，HandlerThread 底层通过 Looper

消息队列实现的，所以它是顺序的执行每一个任务。可以通过 Intent 的方式开启 IntentService，IntentService 通过 handler 将每一个 intent 加入 HandlerThread 子线程中的消息队列，通过 looper 按顺序一个个的取出并执行，执行完成后自动结束自己，不需要开发者手动关闭

## 14. ANR 的原因

- 1.耗时的网络访问
- 2.大量的数据读写
- 3.数据库操作
- 4.硬件操作（比如 camera）
- 5.调用 thread 的 join()方法、sleep()方法、wait()方法或者等待线程锁的时候
- 6.service binder 的数量达到上限
- 7.system server 中发生 WatchDog ANR
- 8.service 忙导致超时无响应
- 9.其他线程持有锁，导致主线程等待超时
- 10.其它线程终止或崩溃导致主线程一直等待

## 15. 三级缓存原理

当 Android 端需要获得数据时比如获取网络中的图片，首先从内存中查找（按键查找），内存中没有的再从磁盘文件或 sqlite 中去查找，若磁盘中也没有才通过网络获取

## 16. LruCache 底层实现原理：

LruCache 中 Lru 算法的实现就是通过 LinkedHashMap 来实现的。LinkedHashMap 继承于 HashMap，它使用了一个双向链表来存储 Map 中的 Entry 顺序关系，对于 get、put、remove 等操作，LinkedHashMap 除了要做 HashMap 做的事情，还做些调整 Entry 顺序链表的工作。

LruCache 中将 LinkedHashMap 的顺序设置为 LRU 顺序来实现 LRU 缓存，每次调用 get(也就是从内存缓存中取图片)，则将该对象移到链表的尾端。调用 put 插入新的对象也是存储在链表尾端，这样当内存缓存达到设定的最大值时，将链表头部的对象（近期最少用到的）移除。

## 17. 说一下你对 Collection 这个类的理解。

Collection 是集合框架的顶层接口，是存储对象的容器，Collection 定义了接口的公用方法如 add remove clear 等等，它的子接口有两个，List 和 Set，List 的特点有元素有序，元素可以重复，元素都有索引（角标），典型的有

Vector:内部是数组数据结构，是同步的（线程安全的）。增删查询都很慢。

ArrayList:内部是数组数据结构，是不同步的（线程不安全的）。替代了 Vector。查询速度快，增删比较慢。

LinkedList:内部是链表数据结构，是不同步的（线程不安全的）。增删元素速度快。

而 Set 的特点是元素无序，元素不可以重复

HashSet: 内部数据结构是哈希表，是不同步的。

Set 集合中元素都必须是唯一的，HashSet 作为其子类也需保证元素的唯一性。

判断元素唯一性的方式：

通过存储对象（元素）的 hashCode 和 equals 方法来完成对象唯一性的。

如果对象的 hashCode 值不同，那么不用调用 equals 方法就会将对象直接存储到集合中；

如果对象的 hashCode 值相同，那么需调用 equals 方法判断返回值是否为 true，

若为 false，则视为不同元素，就会直接存储；

若为 true，则视为相同元素，不会存储。

如果要使用 HashSet 集合存储元素，该元素的类必须覆盖 hashCode 方法和 equals 方法。一般情况下，如果定义的类会产生很多对象，通常都需要覆盖 equals，hashCode 方法。建立对象判断是否相同的依据。

TreeSet: 保证元素唯一性的同时可以对内部元素进行排序，是不同步的。

判断元素唯一性的方式：

根据比较方法的返回结果是否为 0，如果为 0 视为相同元素，不存；如果非 0 视为不同元素，则存。

TreeSet 对元素的排序有两种方式：

方式一：使元素（对象）对应的类实现 Comparable 接口，覆盖 compareTo 方

法。这样元素自身具有比较功能。

方式二：使 TreeSet 集合自身具有比较功能，定义一个比较器 Comparator，将该类对象作为参数传递给 TreeSet 集合的构造函数

## 18. JVM 老年代和新生代的比例

Java 中的堆是 JVM 所管理的最大的一块内存空间，主要用于存放各种类型的实例对象。

在 Java 中，堆被划分成两个不同的区域：新生代 (Young)、老年代 (Old)。

新生代 (Young) 又被划分为三个区域：Eden、From Survivor、To Survivor。

这样划分的目的是为了使用 JVM 能够更好的管理堆内存中的对象，包括内存的分配以及回收。

堆大小 = 新生代 + 老年代。其中，堆的大小可以通过参数 -Xms、-Xmx 来指定。

(本人使用的是 JDK1.6，以下涉及的 JVM 默认值均以该版本为准。)

默认的，新生代 (Young) 与老年代 (Old) 的比例的值为 1:2 (该值可以通过参数 -XX:NewRatio 来指定)，即：新生代 (Young) = 1/3 的堆空间大小。老年代 (Old) = 2/3 的堆空间大小。其中，新生代 (Young) 被细分为 Eden 和两个 Survivor 区域，这两个 Survivor 区域分别被命名为 from 和 to，以示区分。

默认的，Eden : from : to = 8 : 1 : 1 (可以通过参数 -XX:SurvivorRatio 来设定)，即：Eden = 8/10 的新生代空间大小，from = to = 1/10 的新生代空间大小。

JVM 每次只会使用 Eden 和其中的一块 Survivor 区域来为对象服务，所以无论什么时候，总是有一块 Survivor 区域是空闲着的。

因此，新生代实际可用的内存空间为 9/10 (即 90%) 的新生代空间

## 19. jvm, jre 以及 jdk 三者之间的关系？JDK (Java

Development Kit) 是针对 Java 开发员的产品，是整个 Java 的核心，包括了 Java 运行环境 JRE、Java 工具和 Java 基础类库。

Java Runtime Environment (JRE) 是运行 JAVA 程序所必须的环境的集合，包含 JVM 标准实现及 Java 核心类库。

JVM 是 Java Virtual Machine（Java 虚拟机）的缩写，是整个 java 实现跨平台的最核心的部分，能够运行以 Java 语言写作的软件程序。

## 20. 谈谈你对 JNIEnv 和 JavaVM 理解？

### 1.JavaVm

JavaVM 是虚拟机在 JNI 层的代表，一个进程只有一个 JavaVM，所有的线程共用一个 JavaVM。

### 2.JNIEnv

JNIEnv 表示 Java 调用 native 语言的环境，是一个封装了几乎全部 JNI 方法的指针。

JNIEnv 只在创建它的线程生效，不能跨线程传递，不同线程的 JNIEnv 彼此独立。

native 环境中创建的线程，如果需要访问 JNI，必须要调用 `AttachCurrentThread` 关联，并使用 `DetachCurrentThread` 解除链接。

## 21. Serializable 与 Parcelable 的区别？

### 1.Serializable（java 自带）

方法：对象继承 `Serializable` 类即可实现序列化，就是这么简单，也是它最吸引我们的地方

2.Parcelable（Android 专用）：Parcelable 方式的实现原理是将一个完整的对象进行分解，用起来比较麻烦

1）在使用内存的时候，Parcelable 比 Serializable 性能高，所以推荐使用 Parcelable。

2）Serializable 在序列化的时候会产生大量的临时变量，从而引起频繁的 GC。

3）Parcelable 不能使用在要将数据存储在磁盘上的情况，因为 Parcelable 不能很好的保证数据的持续性,在外界有变化的情况下。尽管 Serializable 效率低点，但此时还是建议使用 Serializable。

4）android 上应该尽量采用 Parcelable，效率至上，效率远高于 Serializable