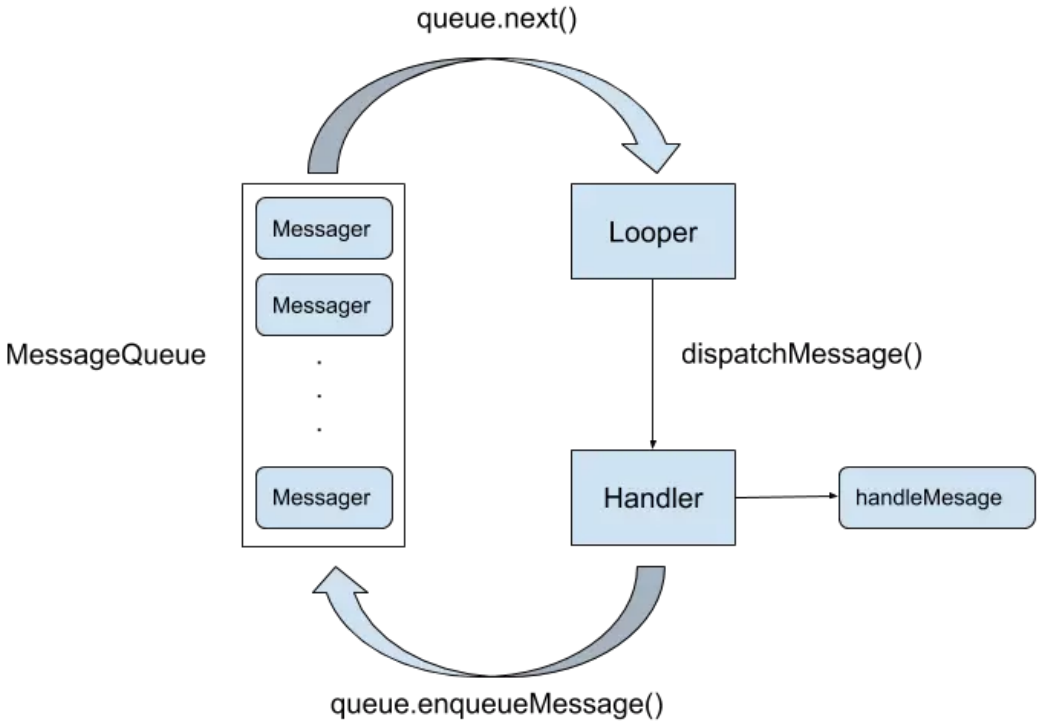
Handler

1、谈谈消息机制Handler作用 ？有哪些要素 ？流程是怎样的 ？

- 参考回答：
  - 负责**跨线程通信**，这是因为**在主线程不能做耗时操作，而子线程不能更新UI**，所以当子线程中进行耗时操作后需要更新UI时，通过Handler将有关UI的操作切换到主线程中执行。
  - 具体分为四大要素
    - **Message（消息）**：需要被传递的消息，消息分为硬件产生的消息（如按钮、触摸）和软件生成的消息。
    - **MessageQueue（消息队列）**：负责消息的存储与管理，负责管理由 Handler 发送过来的Message。读取会自动删除消息，单链表维护，插入和删除上有优势。在其next()方法中会无限循环，不断判断是否有消息，有就返回这条消息并移除。
    - **Handler（消息处理器）**：负责Message的发送及处理。主要向消息池发送各种消息事件（Handler.sendMessage()）和处理相应消息事件（Handler.handleMessage()），按照先进先出执行，内部使用的是单链表的结构。
    - **Looper（消息池）**：负责关联线程以及消息的分发，在该线程下从 MessageQueue获取 Message，分发给Handler，Looper创建的时候会创建一个 MessageQueue，调用loop()方法的时候消息循环开始，其中会不断调用 messageQueue的next()方法，当有消息就处理，否则阻塞在messageQueue的next()方法中。当Looper的quit()被调用的时候会调用messageQueue的quit()，此时next()会返回null，然后loop()方法也就跟着退出。
  - 具体流程如下



- 在主线程创建的时候会创建一个Looper，同时也会在在Looper内部创建一个消息队列。而在创键Handler的时候取出当前线程的Looper，并通过该Looper对象获得消息队列，然后Handler在子线程中通过**MessageQueue.enqueueMessage**在消息队列中添加一条Message。
- 通过**Looper.loop()** 开启消息循环不断轮询调用 **MessageQueue.next()**，取得对应的Message并且通过**Handler.dispatchMessage**传递给Handler，最终调用**Handler.handlerMessage**处理消息。

2、一个线程能否创建多个Handler，Handler跟Looper之间的对应关系 ？

- 参考回答：
  - **一个Thread只能有一个Looper，一个MessageQueen，可以有多个Handler**
  - **以一个线程为基准，他们的数量级关系是： Thread(1) : Looper(1) : MessageQueue(1) : Handler(N)**

3、软引用跟弱引用的区别

- 参考回答：
  - **软引用（SoftReference）**：如果一个对象只具有软引用，则内存空间充足时，垃圾回收器就不会回收它；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，该对象就可以一直被程序使用。

回收它的内存。

- 两者之间**根本区别**在于：只具有弱引用的对象拥有更短暂的生命周期，可能随时被回收。而只具有软引用的对象只有当内存不够的时候才被回收，在内存足够的时候，通常不被回收。

- 推荐文章： Java中的四种引用类型：强引用、软引用、弱引用和虚引用

4、Handler 引起的内存泄露原因以及最佳解决方案

- 参考回答：
  - 泄露原因：
    - Handler 允许我们发送延时消息，如果在延时期间用户关闭了 Activity，那么该 Activity 会泄露。 这个泄露是因为 Message 会持有 Handler，而又因为 Java 的特性，内部类会持有外部类，使得 Activity 会被 Handler 持有，这样最终就导致 Activity 泄露。
  - 解决方案：
    - **将 Handler 定义成静态的内部类，在内部持有Activity的弱引用，并在Acitivity的onDestroy()中调用handler.removeCallbacksAndMessages(null)及时移除所有消息。**

5、为什么系统不建议在子线程访问UI？

- 参考回答：
  - Android的UI控件不是**线程安全**的，如果在多线程中并发访问可能会导致UI控件处于不可预期的状态
  - 这时你可能会问为何系统不对UI控件的访问加上锁机制呢？因为

6、Looper死循环为什么不会导致应用卡死？

- 参考回答：
  - 主线程的主要方法就是**消息循环**，一旦退出消息循环，那么你的应用也就退出了，Looer.loop（ ）方法可能会引起主线程的阻塞，但只要它的消息循环没有被阻塞，能一直处理事件就不会产生ANR异常。
  - 造成**ANR**的不是主线程阻塞，而是主线程的Looper消息处理过程发生了**任务阻塞**，无法响应手势操作，不能及时刷新UI。
  - **阻塞与程序无响应**没有必然关系，虽然主线程在没有消息可处理的时候是阻塞的，但是只要保证有消息的时候能够立刻处理，程序是不会无响应的。

7、使用Handler的postDealy后消息队列会有什么变化？

- 参考回答：
  - 如果队列中只有这个消息，那么消息不会被发送，而是计算到时唤醒的时间，先将Looper阻塞，到时间就唤醒它。但如果此时要加入新消息，该消息队列的对头跟delay时间相比更长，则插入到头部，按照触发时间进行排序，队头的时间最小、队尾的时间最大

8、可以在子线程直接new一个Handler吗？怎么做？

- 参考回答：
  - 不可以，因为在**主线程**中，Activity内部包含一个Looper对象，它会自动管理Looper，处理子线程中发送过来的消息。而对于**子线程**而言，没有任何对象帮助我们

- 推荐文章：Android异步消息处理机制完全解析，带你从源码的角度彻底理解

9、Message可以如何创建？哪种效果更好，为什么？

- 参考回答：可以通过三种方法创建：
  - 直接生成实例**Message m = new Message**
  - 通过**Message m = Message.obtain**
  - 通过**Message m = mHandler.obtainMessage()**
- 后两者效果更好，因为Android默认的消息池中消息数量是10，而后两者是直接在消息池中取出一个Message实例，这样做就可以避免多生成Message实例。

线程

1、线程池的好处？ 四种线程池的使用场景，线程池的几个参数的理解？

- 参考回答：
  - 使用线程池的好处是减少在创建和销毁线程上所花的时间以及系统资源的开销，解决资源不足的问题。如果不使用线程池，有可能造成系统创建大量同类线程而导致消耗完内存或则“过度切换”的问题，归纳总结就是
    - **重用存在的线程，减少对象创建、消亡的开销，性能佳。**
    - **可有效控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞。**

- Android中的线程池都是直接或间接通过配置ThreadPoolExecutor来实现不同特性的线程池.Android中最常见的类具有不同特性的线程池分别为：
  - **newCachedThreadPool**：只有非核心线程，最大线程数非常大，所有线程都活动时会为新任务创建新线程,否则会利用空闲线程（60s空闲时间,过了就会被回收,所以线程池中有0个线程的可能)来处理任务.
    - 优点：任何任务都会被立即执行(任务队列SynchronousQuue相当于一个空集合);比较适合执行大量的耗时较少的任务.
  - **newFixedThreadPool**：只有核心线程，并且数量固定的，所有线程都活动时，因为队列没有限制大小，新任务会等待执行，当线程池空闲时不会释放工作线程，还会占用一定的系统资源。
    - 优点：更快的响应外界请求
  - **newScheduledThreadPool**：核心线程数固定,非核心线程（闲着没活干会被立即回收数）没有限制.
    - 优点：执行定时任务以及有固定周期的重复任务
  - **newSingleThreadExecutor**：只有一个核心线程,确保所有的任务都在同一线程中按序完成
    - 优点：不需要处理线程同步的问题
- 通过源码可以了解到上面的四种线程池实际上还是利用 ThreadPoolExecutor 类实现的

- 推荐文章：java线程池解析和四种线程池的使用

2、Android中还了解哪些方便线程切换的类？

- **AsyncTask**：底层封装了线程池和Handler，便于执行后台任务以及在子线程中进行UI操作。
- **HandlerThread**：一种具有消息循环的线程，其内部可使用Handler。
- **IntentService**：是一种异步、会自动停止的服务，内部采用HandlerThread。

3、讲讲AsyncTask的原理

- 参考回答：
  - **AsyncTask中有两个线程池（SerialExecutor和THREAD_POOL_EXECUTOR）和一个Handler（InternalHandler），其中线程池SerialExecutor用于任务的排队，而线程池THREAD_POOL_EXECUTOR用于真正地执行任务，InternalHandler用于将执行环境从线程池切换到主线程。**
  - **sHandler是一个静态的Handler对象，为了能够将执行环境切换到主线程，这就要求sHandler这个对象必须在主线程创建。由于静态成员会在加载类的时候进行初始化，因此这就变相要求AsyncTask的类必须在主线程中加载，否则同一个进程中的AsyncTask都将无法正常工作。**

4、IntentService有什么用 ？

- 参考回答：
  - **IntentService**可用于**执行后台耗时**的任务，当任务执行完成后会**自动停止**，同时由于IntentService是服务的原因，不同于普通Service，IntentService可**自动创建子线程**来执行任务，这导致它的优先级比单纯的线程要高，不容易被系统杀死，所以IntentService比较适合执行一些**高优先级**的后台任务。

5、直接在Activity中创建一个thread跟在service中创建一个thread之间的区别？

- 参考回答：
  - **在Activity中被创建**：该Thread的就是为这个Activity服务的，完成这个特定的Activity交代的任务，主动通知该Activity一些消息和事件，Activity销毁后，该Thread也没有存活的意义了。
  - **在Service中被创建**：这是保证最长生命周期的Thread的唯一方式，只要整个Service不退出，Thread就可以一直在后台执行，一般在Service的onCreate()中创建，在onDestroy()中销毁。所以，在Service中创建的Thread，适合长期执行一些独立于APP的后台任务，比较常见的就是：在Service中保持与服务器端的长连接。

- 参考回答：ThreadPoolExecutor执行任务时会遵循如下规则
  - 如果线程池中的线程数量**未达到**核心线程的数量，那么会直接启动一个核心线程来执行任务。
  - 如果线程池中的线程数量**已经达到**或则超过核心线程的数量，那么任务会被插入任务队列中排队等待执行。
  - 如果在第2点无法将任务插入到任务队列中，这往往是由于任务队列已满，这个时候如果在线程数量**未达到线程池规定的最大值**，那么会立刻启动一个非核心线程来执行任务。
  - 如果第3点中线程数量**已经达到线程池规定的最大值**，那么就拒绝执行此任务，ThreadPoolExecutor会调用RejectedExecutionHandler的rejectedExecution方法来通知调用者。

7、Handler、Thread和HandlerThread的差别？

- 参考回答：
  - **Handler**：在android中负责发送和处理消息，通过它可以实现其他支线线程与主线程之间的消息通讯。
  - **Thread**：Java进程中执行运算的最小单位，亦即执行处理机调度的基本单位。某一进程中一路单独运行的程序。
  - **HandlerThread**：一个继承自Thread的类HandlerThread，Android中没有对Java中的Thread进行任何封装，而是提供了一个继承自Thread的类HandlerThread类，这个类对Java的Thread做了很多便利的封装。HandlerThread继承于Thread，所以它本质就是个Thread。与普通Thread的差别就在于，它在内部直接实现了Looper的实现，这是Handler消息机制必不可少的。有了自己的looper，可以让我们在自己的线程中分发和处理消息。如果不用HandlerThread的话，需要手动去调用Looper.prepare()和Looper.loop()这些方法。

8、ThreadLocal的原理

- 参考回答：
  - ThreadLocal是一个关于创建线程局部变量的类。使用场景如下所示：
    - **实现单个线程单例以及单个线程上下文信息存储，比如交易id等。**
    - **实现线程安全，非线程安全的对象使用ThreadLocal之后就会变得线程安全，因为每个线程都会有一个对应的实例。 承载一些线程相关的数据，避免在方法中来回传递参数。**
  - 当需要使用多线程时，有个变量恰巧不需要共享，此时就不必使用synchronized这么麻烦的关键字来锁住，每个线程都相当于在堆内存中开辟一个空间，线程中带有对共

相当于线程内的内存，一个局部变量。每次可以对线程自身的数据读取和操作，并不需要通过缓冲区与 主内存中的变量进行交互。并不会像synchronized那样修改主内存的数据，再将主内存的数据复制到线程内的工作内存。ThreadLocal可以让线程独占资源，存储于线程内部，避免线程堵塞造成CPU吞吐下降。

- 在每个Thread中包含一个ThreadLocalMap，ThreadLocalMap的key是ThreadLocal的对象，value是独享数据。

9、多线程是否一定会高效（优缺点）

- 参考回答：
  - 多线程的优点：
    - **方便高效的内存共享 - 多进程下内存共享比较不便，且会抵消掉多进程编程的好处**
    - **较轻的上下文切换开销 - 不用切换地址空间，不用更改CR3寄存器，不用清空TLB**
    - **线程上的任务执行完后自动销毁**
  - 多线程的缺点：
    - **开启线程需要占用一定的内存空间(默认情况下,每一个线程都占512KB)**
    - **如果开启大量的线程,会占用大量的内存空间,降低程序的性能**
    - **线程越多,cpu在调用线程上的开销就越大**
    - **程序设计更加复杂,比如线程间的通信、多线程的数据共享**
  - 综上得出，多线程**不一定**能提高效率，在内存空间紧张的情况下反而是一种负担，因此在日常开发中，应尽量
    - **不要频繁创建，销毁线程，使用线程池**
    - **减少线程间同步和通信（最为关键）**
    - **避免需要频繁共享写的数据**
    - **合理安排共享数据结构，避免伪共享（false sharing）**
    - **使用非阻塞数据结构/算法**
    - **避免可能产生可伸缩性问题的系统调用（比如mmap）**
    - **避免产生大量缺页异常，尽量使用Huge Page**
    - **可以的话使用用户态轻量级线程代替内核线程**

10、多线程中,让你做一个单例,你会怎么做

- 参考回答：
  - 多线程中建立单例模式考虑的因素有很多，比如线程安全 -延迟加载-代码安全:如防止序列化攻击、防止反射攻击(防止反射进行私有方法调用)、性能因素

- 实现方法有多种，饿汉，懒汉(线程安全，线程非安全)，双重检查(DCL),内部类，以及枚举

- 推荐文章：单例模式的总结

11、除了notify还有什么方式可以唤醒线程

- 参考回答：
  - 当一个拥有Object锁的线程调用 wait()方法时，就会使当前线程加入object.wait 等待队列中，并且释放当前占用的Object锁，这样其他线程就有机会获取这个Object锁，获得Object锁的线程调用notify()方法，就能在Object.wait 等待队列中随机唤醒一个线程（该唤醒是随机的与加入的顺序无关，优先级高的被唤醒概率会高）
  - 如果调用notifyAll（ ）方法就唤醒全部的线程。**注意:调用notify()方法后并不会立即释放object锁，会等待该线程执行完毕后释放Object锁。**

12、什么是ANR？什么情况会出现ANR ？如何避免 ？ 在不看代码的情况下如何快速定位出现ANR问题所在 ？

- 参考回答：
  - ANR（Application Not Responding，应用无响应）：当操作在一段时间内系统无

- 产生ANR可能是因为5s内无响应用户输入事件、10s内未结束BroadcastReceiver、20s内未结束Service
- 想要避免ANR就不要在主线程做耗时操作，而是通过开子线程，方法比如继承Thread或实现Runnable接口、使用AsyncTask、IntentService、HandlerThread等
- 推荐文章：如何快速分析定位ANR

Bitmap

1、Bitmap使用需要注意哪些问题 ？

- 参考回答：
  - **要选择合适的图片规格（bitmap类型）**：通常我们优化Bitmap时，当需要做性能优化或者防止OOM，我们通常会使用RGB_565，因为ALPHA_8只有透明度，显示一般图片没有意义，Bitmap.Config.ARGB_4444显示图片不清楚，Bitmap.Config.ARGB_8888占用内存最多。：
    - ALPHA_8 每个像素占用1byte内存
    - ARGB_4444 每个像素占用2byte内存
    - ARGB_8888 每个像素占用4byte内存（默认）
    - RGB_565 每个像素占用2byte内存
  - **降低采样率**：BitmapFactory.Options 参数inSampleSize的使用，先把options.inJustDecodeBounds设为true，只是去读取图片的大小，在拿到图片的大小之后和要显示的大小做比较通过calculateInSampleSize()函数计算inSampleSize的具体值，得到值之后。options.inJustDecodeBounds设为false读图片资源。
  - **复用内存**：即通过软引用(内存不够的时候才会回收掉)，复用内存块，不需要再重新给这个bitmap申请一块新的内存，避免了一次内存的分配和回收，从而改善了运行效率。
  - **使用recycle()方法及时回收内存。**
  - **压缩图片。**

2、Bitmap.recycle()会立即回收么？什么时候会回收？如果没有地方使用这个Bitmap，为什么垃圾回收不会直接回收？

- 参考回答：
  - 通过源码可以了解到，加载Bitmap到内存里以后，是包含**两部分内存区域**的。简单的说，一部分是**Java部分**的，一部分是**C部分**的。这个Bitmap对象是由Java部分分配的，不用的时候系统就会自动回收了
  - 但是那个对应的**C可用**的内存区域，虚拟机是不能直接回收的，这个只能调用底层的

- bitmap.recycle()方法用于回收该Bitmap所占用的内存，接着将bitmap置空，最后使用System.gc()调用一下系统的垃圾回收器进行回收，调用System.gc()并不能保证立即开始进行回收过程，而只是为了加快回收的到来。

3、一张Bitmap所占内存以及内存占用的计算

- 参考回答：
  - Bitamp 所占内存大小 = 宽度像素 x （ inTargetDensity / inDensity ） x 高度像素 x （ inTargetDensity / inDensity ）x 一个像素所占的内存字节大小
    - 注：这里inDensity表示目标图片的dpi（放在哪个资源文件夹下），inTargetDensity表示目标屏幕的dpi，所以你可以发现inDensity和inTargetDensity会对Bitmap的宽高进行拉伸，进而改变Bitmap占用内存的大小。
  - 在Bitmap里有两个获取内存占用大小的方法。
    - **getByteCount()**：API12 加入，代表存储 Bitmap 的像素需要的最少内存。
    - **getAllocationByteCount()**：API19 加入，代表在内存中为 Bitmap 分配的内存大小，代替了 getByteCount() 方法。
    - 在**不复用 Bitmap** 时，getByteCount() 和 getAllocationByteCount 返回的结果是一样的。在通过**复用 Bitmap** 来解码图片时，那么 getByteCount() 表示新解码图片占用内存的大 小，getAllocationByteCount() 表示被复用 Bitmap 真实占用的内存大小

4、Android中缓存更新策略 ？

- 参考回答：
  - Android的缓存更新策略**没有统一的标准**，一般来说，缓存策略主要包含**缓存的添加、获取和删除**这三类操作，但不管是内存缓存还是存储设备缓存，它们的缓存容量是有限制的，因此删除一些旧缓存并添加新缓存，如何定义缓存的新旧这就是一种策略，**不同的策略就对应着不同的缓存算法**
  - 比如可以简单地根据文件的最后修改时间来定义缓存的新旧，当缓存满时就将最后修改时间较早的缓存移除，这就是一种缓存算法，但不算很完美

5、LRU的原理 ？

- 参考回答：
  - 为减少流量消耗，可采用缓存策略。常用的缓存算法是LRU(Least Recently Used)：

- **LruCache(内存缓存)**：LruCache类是一个线程安全的泛型类：内部采用一个LinkedHashMap以强引用的方式存储外界的缓存对象，并提供get和put方法来完成缓存的获取和添加操作，当缓存满时会移除较早使用的缓存对象，再添加新的缓存对象。
- **DiskLruCache(磁盘缓存)**： 通过将缓存对象写入文件系统从而实现缓存效果

性能优化

1、图片的三级缓存中,图片加载到内存中,如果内存快爆了,会发生什么？怎么处理？

- 参考回答：
  - 首先我们要清楚图片的三级缓存是如何的

如果内存足够时不回收。内存不够时就回收软引用对象

2、内存中如果加载一张500*500的png高清图片.应该是占用多少的内存?

- 参考回答：
  - **不考虑屏幕比的话**：占用内存=500 * 500 * 4 = 1000000B ≈ 0.95MB
  - **考虑屏幕比的的话**：占用内存= 宽度像素 x （ inTargetDensity / inDensity ） x 高度像素 x （ inTargetDensity / inDensity ）x 一个像素所占的内存字节大小
  - inDensity表示目标图片的dpi（放在哪个资源文件夹下），inTargetDensity表示目标屏幕的dpi

3、WebView的性能优化？

- 参考回答：
  - 一个加载网页的过程中，native、网络、后端处理、CPU都会参与，各自都有必要的工作和依赖关系；让他们相互并行处理而不是相互阻塞才可以让网页加载更快：
    - **WebView初始化慢，可以在初始化同时先请求数据，让后端和网络不要闲着。**
    - **常用 JS 本地化及延迟加载，使用第三方浏览内核**
    - **后端处理慢，可以让服务器分trunk输出，在后端计算的同时前端也加载网络静态资源。**
    - **脚本执行慢，就让脚本在最后运行，不阻塞页面解析。**
    - **同时，合理的预加载、预缓存可以让加载速度的瓶颈更小。**
    - **WebView初始化慢，就随时初始化好一个WebView待用。**
    - **DNS和链接慢，想办法复用客户端使用的域名和链接。**
  - 推荐文章：WebView性能、体验分析与优化

4、Bitmap如何处理大图，如一张30M的大图，如何预防OOM？

- 参考回答：避免OOM的问题就需要对大图片的加载进行管理，主要通过缩放来减小图片的内存占用。
  - BitmapFactory提供的加载图片的四类方法（**decodeFile、decodeResource、decodeStream、decodeByteArray**）都支持BitmapFactory.Options参数，通过inSampleSize参数就可以很方便地对一个图片进行采样缩放
  - 比如一张1024*1024的高清图片来说。那么它占有的内存为1024*1024*4，即4MB，如果inSampleSize为2，那么采样后的图片占用内存只有512*512*4,即1MB（**注意：根据最新的官方文档指出，inSampleSize的取值应该总是为2的指数，即1、2、4、8等等，如果外界输入不足为2的指数，系统也会默认选择最接近2的指数代替，比如2**）
  - 综合考虑。通过采样率即可有效加载图片，流程如下
    - **将BitmapFactory.Options的inJustDecodeBounds参数设为true并加载图片**
    - **从BitmapFactory.Options中取出图片的原始宽高信息，它们对应outWidth和outHeight参数**
    - **根据采样率的规则并结合目标View的所需大小计算出采样率inSampleSize**
    - **将BitmapFactory.Options的inJustDecodeBounds参数设为false，重新加载图片**

  - 推荐文章：Android高效加载大图、多图解决方案，有效避免程序OOM

- 参考回答：
  - 内存判定对象可回收有两种机制：
    - **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。然而在主流的Java虚拟机里未选用引用计数算法来管理内存，主要原因是它难以解决对象之间**相互循环引用**的问题，所以出现了另一种对象存活判定算法。
    - **可达性分析法**：通过一系列被称为『GCRoots』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***，*当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的。其中可作为GC Roots的对象：虚拟机栈中引用的对象，主要是指栈帧中的本地变量**、本地方法栈中**Native**方法引用的对象、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
  - GC回收算法有以下四种：
    - **分代收集算法**：是当前商业虚拟机都采用的一种算法，根据对象存活周期的不同，将Java堆划分为新生代和老年代，并根据各个年代的特点采用最适当的收集算法。
    - 新生代：大批对象死去，只有少量存活。使用『复制算法』，只需复制少量存活对象即可。
      - **复制算法**：把可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用尽后，把还存活着的对象『复制』到另外一块上面，再将这一块内存空间一次清理掉。**实现简单，运行高效。在对象存活率较高时就要进行较多的复制操作，效率将会变低**
    - 老年代：对象存活率高。使用『标记—清理算法』或者『标记—整理算法』，只需标记较少的回收对象即可。
      - **标记-清除算法**：首先『标记』出所有需要回收的对象，然后统一『清除』所有被标记的对象。**标记和清除两个过程的效率都不高，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一次垃圾收集动作。**
      - **标记-整理算法**：首先『标记』出所有需要回收的对象，然后进行『整理』，使得存活的对象都向一端移动，最后直接清理掉端边界以外的内存。**标记整理算法会将所有的存活对象移动到一端，并对不存活对象进行处理，因此其不会产生内存碎片**
  - 推荐文章：图解Java 垃圾回收机制

6、内存泄露和内存溢出的区别 ？AS有什么工具可以检测内存泄露

- **内存溢出(out of memory)**：是指程序在申请内存时，没有足够的内存空间供其使用，出现out of memory；比如申请了一个integer，但给它存了long才能存下的数，那就是内存溢出。
- **内存泄露(memory leak)**：是指程序在申请内存后，无法释放已申请的内存空间，一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光。**memory leak会最终会导致out of memory！**
- 查找内存泄漏可以使用Android Studio 自带的**AndroidProfiler**工具或**MAT**

7、性能优化,怎么保证应用启动不卡顿? 黑白屏怎么处理?

- 参考回答：
  - **应用启动速度**，取决于你在application里面时候做了什么事情，比如你集成了很多sdk，并且sdk的init操作都需要在主线程里实现所以会有卡顿的感觉。在非必要的情况下可以把加载延后或则开启子线程处理
  - 另外，影响**界面卡顿**的两大因素，分别是**界面绘制和数据处理。**
    - 布局优化(使用include，merge标签，复杂布局推荐使用ConstraintLayout等)
    - onCreate() 中不执行耗时操作 把页面显示的 View 细分一下，放在 AsyncTask 里逐步显示，用 Handler 更好。这样用户的看到的就是有层次有步骤的一个个的 View 的展示，不会是先看到一个黑屏，然后一下显示所有 View。最好做成动画，效果更自然。
    - 利用多线程的目的就是尽可能的减少 onCreate() 和 onReume() 的时间，使得用户能尽快看到页面，操作页面。
    - 减少主线程阻塞时间。
    - 提高 Adapter 和 AdapterView 的效率。
  - 推荐文章：Android 性能优化之内存检测、卡顿优化、耗电优化、APK瘦身
  - **黑白屏产生原因**：当我们在启动一个应用时，系统会去检查是否已经存在这样一个进程，如果不存在，系统的服务会先检查startActivity中的intent的信息，然后在去创建进程，最后启动Acitivy，即冷启动。而启动出现白黑屏的问题，就是在这段时间内产生的。系统在绘制页面加载布局之前，首先会初始化窗口（Window），而在进行这一步操作时，系统会根据我们设置的Theme来指定它的Theme 主题颜色，我们在Style中的设置就决定了显示的是白屏还是黑屏。
    - windowIsTranslucent和windowNoTitle，将这两个属性都设置成true (会有明显的卡顿体验，不推荐)
    - 如果启动页只是是一张图片，那么为启动页专一设置一个新的主题，设置主题的android:windowBackground属性为启动页背景图即可
    - 使用layer-list制作一张图片launcher_layer.xml，将其设置为启动页专一主题的背景，并将其设置为启动页布局的背景。

8、强引用置为null，会不会被回收？

- 参考回答：
  - **不会立即释放对象占用的内存。** 如果对象的引用被置为null，只是断开了当前线程栈帧中对该对象的引用关系，而 垃圾收集器是运行在后台的线程，只有当用户线程运行到安全点(safe point)或者安全区域才会扫描对象引用关系，扫描到对象没有被引用则会标记对象，这时候仍然不会立即释放该对象内存，因为有些对象是可恢复的（在finalize方法中恢复引用 ）。只有确定了对象无法恢复引用的时候才会清除对象内存。

9、ListView跟RecyclerView的区别

- 参考回答：
  - 动画区别：
    - 在**RecyclerView**中，内置有许多动画API，例如：notifyItemChanged(), notifyDataInserted(), notifyItemMoved()等等；如果需要自定义动画效果，可以通过实现（RecyclerView.ItemAnimator类）完成自定义动画效果，然后调用RecyclerView.setItemAnimator()；
    - 但是**ListView**并没有实现动画效果，但我们可以在Adapter自己实现item的动画效果；
  - 刷新区别：
    - ListView中通常刷新数据是用全局刷新notifyDataSetChanged()，这样一来就会非常消耗资源；**本身无法实现局部刷新**，但是如果要在ListView实现**局部刷新**，依然是可以实现的，当一个item数据刷新时，我们可以在Adapter中，实现一个onItemChanged()方法，在方法里面获取到这个item的position（可以通过getFirstVisiblePosition()），然后调用getView()方法来刷新这个item的数据；
    - RecyclerView中可以实现局部刷新，例如：notifyItemChanged()；
  - 缓存区别：
    - RecyclerView比ListView多两级缓存，支持多个离ItemView缓存，支持开发者自定义缓存处理逻辑，支持所有RecyclerView共用同一个RecyclerViewPool(缓存池)。
    - ListView和RecyclerView缓存机制基本一致，但缓存使用不同
  - 推荐文章：
    - 【腾讯Bugly干货分享】Android ListView 与 RecyclerView 对比浅析—缓存机制
    - ListView 与 RecyclerView 简单对比
    - Android开发：ListView、AdapterView、RecyclerView全面解析

10、ListView的adapter是什么adapter

- 参考回答：
  - **BaseAdapter**：抽象类，实际开发中我们会继承这个类并且重写相关方法，用得最多的一个适配器！
  - **ArrayAdapter**：支持泛型操作，最简单的一个适配器，只能展现一行文字～
  - **SimpleAdapter**：同样具有良好扩展性的一个适配器，可以自定义多种效果！
  - **SimpleCursorAdapter**：用于显示简单文本类型的listView，一般在数据库那里会用到，不过有点过时，不推荐使用！

11、LinearLayout、FrameLayout、RelativeLayout性能对比，为什么？

- 参考回答：
  - RelativeLayout会让子View调用2次onMeasure，LinearLayout 在有weight时，也会调用子 View 2次onMeasure
  - RelativeLayout的子View如果高度和RelativeLayout不同，则会引发效率问题，当子View很复杂时，这个问题会更加严重。如果可以，尽量使用padding代替margin

- 在不影响层级深度的情况下,使用LinearLayout和FrameLayout而不是RelativeLayout。

JNI

1、对JNI是否了解

- 参考回答：
  - Java的优点是**跨平台**，但也因为其跨平台的的特性导致其**本地交互的能力不够强大**，一些和操作系统相关的的特性Java无法完成，于是**Java提供JNI专门用于和本地代码交互，通过JNI，用户可以调用C、C++编写的本地代码**
  - NDK是Android所提供的一个工具集合，通过NDK可以在Android中更加方便地通过JNI访问本地代码，其优点在于
    - 提高代码的安全性。由于so库反编译困难，因此NDK提高了Android程序的安全性
    - 可以很方便地使用目前已有的C/C++开源库
    - 便于平台的移植。通过C/C++实现的动态库可以很方便地在其它平台上使用
    - 提高程序在某些特定情形下的执行效率，但是并不能明显提升Android程序的性能

2、如何加载NDK库 ？如何在JNI中注册Native函数，有几种注册方法 ？

- 参考回答：

```
public class JniTest{
    //加载NDK库
    static{
        System.loadLirary("jni-test");
    }
}
```

- 注册JNI函数的两种方法
  - **静态方法**
  - **动态注册**
- 推荐文章：
  - 注册JNI函数的两种方式
  - Android JNI 篇 - 从入门到放弃

3、你用JNI来实现过什么功能 ？怎么实现的 ？（加密处理、影音方面、图形图像处理）

- 参考回答：
  - 推荐文章：Android JNI 篇 - ffmpeg 获取音视频缩略图

设计模式

1、你所知道的设计模式有哪些？

- 参考回答：
  - **创建型模式，共五种**：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。
  - **结构型模式，共七种**：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。
  - **行为型模式，共十一种**：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录 模式、状态模式、访问者模式、中介者模式、解释器模式。

2、谈谈MVC、MVP和MVVM，好在哪里，不好在哪里 ？

- 参考回答：
  - **MVC**:
    - 视图层(View) 对应于xml布局文件和java代码动态view部分
    - 控制层(Controller) MVC中Android的控制层是由Activity来承担的，Activity本来主要是作为初始化页面，展示数据的操作，但是因为XML视图功能太弱，所以Activity既要负责视图的显示又要加入控制逻辑，承担的功能过多。
    - 模型层(Model) 针对业务模型，建立数据结构和相关的类，它主要负责网络请求，数据库处理，I/O的操作。
  - 总结
    - 具有一定的分层，model彻底解耦，controller和view并没有解耦层与层之间的交互尽量使用回调或者去使用消息机制去完成，尽量避免直接持有 controller和view在android中无法做到彻底分离，但在代码逻辑层面一定要分清业务逻辑被放置在model层，能够更好的复用和修改增加业务。
  - **MVP**
    - 通过引入接口BaseView，让相应的视图组件如Activity，Fragment去实现BaseView，实现了视图层的独立，通过中间层Preseter实现了Model和View的完全解耦。MVP**彻底解决**了MVC中View和Controller傻傻分不清楚的问题，但

- **MVVM**
  - MVP中我们说过随着业务逻辑的增加，UI的改变多的情况下，会有非常多的跟UI相关的case，这样就会造成View的接口会很庞大。而MVVM就解决了这个问题，通过双向绑定的机制，实现数据和UI内容，只要想改其中一方，另一方都能够及时更新的一种设计理念，这样就省去了很多在View层中写很多case的情况，只需要改变数据就行。
- 三者如何选择？
  - 如果项目简单，没什么复杂性，未来改动也不大的话，那就不要用设计模式或者架构方法，只需要将每个模块封装好，方便调用即可，不要为了使用设计模式或架构方法而使用。
  - 对于偏向展示型的app，绝大多数业务逻辑都在后端，app主要功能就是展示数据，交互等，建议使用mvvm。
  - 对于工具类或者需要写很多业务逻辑app，使用mvp或者mvvm都可。
- 推荐文章：MVC、MVP、MVVM，我到底该怎么选？

3、封装p层之后.如果p层数据过大,如何解决?

- 参考回答：
  - 对于MVP模式来说，P层如果数据逻辑过于臃肿，建议引入**RxJava**或则**Dagger**，越是复杂的逻辑，越能体现RxJava的优越性
  - 推荐文章：（仿有道精品课）RxJava+OkHttp+Retrofit+Dagger2+MVP框架(kotlin版本)

4、是否能从Android中举几个例子说说用到了什么设计模式 ？

- 参考回答：
  - **AlertDialog、Notification源码中使用了Builder（建造者）模式完成参数的初始化**
  - **Okhttp内部使用了责任链模式来完成每个Interceptor拦截器的调用**
  - **RxJava的观察者模式；单例模式；GridView的适配器模式；Intent的原型模式**
  - **日常开发的BaseActivity抽象工厂模式**

5、装饰模式和代理模式有哪些区别 ？

- 参考回答：
  - 装饰器模式与代理模式的区别就在于
    - 两者都是对类的方法进行**扩展**，但**装饰器模式**强调的是增强自身,在被装饰之后你

- 而**代理模式**则强调要让别人帮你去做一些本身与你业务没有太多关系的职责（记录日志、设置缓存）代理模式是为了实现对象的控制，因为被代理的对象往往难以直接获得或者是其内部不想暴露出来。

6、实现单例模式有几种方法 ？懒汉式中双层锁的目的是什么 ？两次判空的目的又是什么 ？

- 参考回答：
  - 单例模式实现方法有多种：**饿汉，懒汉(线程安全，线程非安全)，双重检查(DCL),内部类，以及枚举**
  - 所谓**双层检验锁（在加锁前后对实例对象进行两次判空的检验）**：加锁是为了第一次对象实例化的线程同步，而锁内还要有第二层判空是因为可能会有多个线程进入第一层if判断内部，而在加锁代码块外排队等候，如果锁内不进行第二次检验，仍然会出现实例化多个对象的情况。
  - 推荐文章：单例模式的总结

7、用到的一些开源框架，介绍一个看过源码的，内部实现过程。

- 参考回答：
  - 面试常客：Okhttp，Retrofit，Glide，RxJava，GreenDao，Dagger等
  - 推荐文章：
    - Android OkHttp源码解析入门教程（一）
    - Android OkHttp源码解析入门教程（二）

8、Fragment如果在Adapter中使用应该如何解耦？

- 参考回答：
  - 接口回调
  - 广播

你当前所处：Android篇：2019初中级Android开发社招面试解答（中）

Android篇：2019初中级Android开发社招面试解答（下）

Android篇：2019初中级Android开发社招面试解答（上）

关注下面的标签，发现更多相似文章