性能优化

线程池的好处、原理、类型?

- 技术点：线程池
- 参考回答：
- （1）线程池的好处：
    - 重用线程池中的线程，避免线程的创建和销毁带来的性能消耗；
    - 有效控制线程池的最大并发数，避免大量的线程之间因互相抢占系统资源而导致阻塞现象；
    - 进行线程管理，提供定时/循环间隔执行等功能
- （2）线程池的分类：
    - FixThreadPool：线程数量固定的线程池，所有线程都是核心线程，当线程空闲时不会被回收；能快速响应外界请求。
    - CachedThreadPool：线程数量不定的线程池（最大线程数为Integer.MAX_VALUE），只有非核心线程，空闲线程有超时机制，超时回收；适合于执行大量的耗时较少的任务
    - ScheduledThreadPool：核心线程数量固定，非核心线程数量不定；可进行定时任务和固定周期的任务。
    - SingleThreadExecutor：只有一个核心线程，可确保所有的任务都在同一个线程中按顺序执行；好处是无需处理线程同步问题。
- （3）线程池的原理：实际上通过ThreadPoolExecutor并通过一系列参数来配置各种各样的线程池，具体的参数有：
    - corePoolSize核心线程数：一般会在线程中一直存活
    - maximumPoolSize最大线程数：当活动线程数达到这个数值后，后续的任务将会被阻塞
    - keepAliveTime非核心线程超时时间：超过这个时长，闲置的非核心线程就会被回收
    - unit：用于指定keepAliveTime参数的时间单位
    - workQueue任务队列：通过线程池的execute()方法提交的Runnable对象会存储在这个参数中。
    - threadFactory：线程工厂，可创建新线程
    - handler：在线程池无法执行新任务时进行调度

hreadPoolExecutor的工作策略?

- 技术点：线程池
- 参考回答：ThreadPoolExecutor的默认工作策略：
    - 若程池中的线程数量未达到核心线程数，则会直接启动一个核心线程执行任务。
    - 若线程池中的线程数量已达到或者超过核心线程数量，则任务会被插入到任务列表等待执行。
        - 若任务无法插入到任务列表中，往往由于任务列表已满，此时如果
            - 线程数量未达到线程池最大线程数，则会启动一个非核心线程执行任务；
            - 线程数量已达到线程池规定的最大值，则拒绝执行此任务，ThreadPoolExecutor会调用RejectedExecutionHandler的rejectedExecution方法来通知调用者。

加载图片的时候需要注意什么?

- 技术点：Bitmap高效加载
- 参考回答：
    - 直接加载大容量的高清Bitmap很容易出现显示不完整、内存溢出OOM的问题，所以最好按一定的采样率将图片缩小后再加载进来
    - 为减少流量消耗，可对图片采用内存缓存策略，又为了避免图片占用过多内存导致内存溢出，最好以软引用方式持有图片
    - 如果还需要网上下载图片，注意要开子线程去做下载的耗时操作

LRU算法的原理?

- 技术点：LRU算法
- 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)：
    - 核心思想：当缓存满时, 会优先淘汰那些近期最少使用的缓存对象。主要是两种方式：
        - 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

项目中如何做性能优化的?

- 技术点：性能优化实例
- 思路：举例说明项目注意了哪些方面的性能优化，如布局优化、绘制优化、内存泄漏优化、 响应速度优化、列表优化、Bitmap优化、 线程优化.....

布局上如何优化?

- 技术点：布局优化
- 参考回答：布局优化的核心就是尽量减少布局文件的层级，常见的方式有：

- 多嵌套情况下可使用RelativeLayout减少嵌套。
- 布局层级相同的情况下使用LinearLayout，它比RelativeLayout更高效。
- 使用标签重用布局、标签减少层级、标签懒加载。

内存泄漏是什么？为什么会发生？常见哪些内存泄漏的例子？都是怎么解决的?

- 技术点：内存泄漏
- 参考回答：内存泄漏(Memory Leak)是指程序在申请内存后，无法释放已申请的内存空间。简单地说，发生内存泄漏是由于长周期对象持有对短周期对象的引用，使得短周期对象不能被及时回收。常见的几个例子和解决办法：
  - 单例模式导致的内存泄漏：单例传入参数this来自Activity，使得持有对Activity的引用。
    - 解决办法：传参context.getApplicationContext()
  - Handler导致的内存泄漏：Message持有对Handler的引用，而非静态内部类的Handler又隐式持有对外部类Activity的引用，使得引用关系会保持至消息得到处理，从而阻止了Activity的回收。
    - 解决办法：使用静态内部类+WeakReference弱引用；当外部类结束生命周期时清空消息队列。
  - 线程导致的内存泄漏：AsyncTask/Runnable以匿名内部类的方式存在，会隐式持有对所在Activity的引用。
    - 解决办法：将AsyncTask和Runnable设为静态内部类或独立出来；在线程内部采用弱引用保存Context引用
  - 资源未关闭导致的内存泄漏：未及时注销资源导致内存泄漏，如BraodcastReceiver、File、Cursor、Stream、Bitmap等。
    - 解决办法：在Activity销毁的时候要及时关闭或者注销。
      - BraodcastReceiver：调用unregisterReceiver()注销；
      - Cursor，Stream、File：调用close()关闭；
      - 动画：在Activity.onDestroy()中调用Animator.cancel()停止动画

内存泄漏和内存溢出的区别?

- 技术点：内存泄漏、内存溢出
- 参考回答：
  - 内存泄漏(Memory Leak)是指程序在申请内存后，无法释放已申请的内存空间。是造成应用程序OOM的主要原因之一。
  - 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。

什么情况会导致内存溢出?

- 技术点：内存溢出
- 参考回答：内存泄漏是导致内存溢出的主要原因；直接加载大图片也易造成内存溢出
- 引申：谈谈如何避免内存溢出（如何避免内存泄漏、避免直接加载大图片)