**谈谈你对android系统（体系）架构的理解**

Linux操作系统为核心，从下往上，依赖关系。

1. 应用程序层：包括系统应用以及第三方应用。
2. 应用程序框架：提供应用开发所必须的一些API框架，是软件复用的重要手段
3. 库：android运行时（核心包（相当于JDK提供的包），虚拟机（优化过的JVM））；C/C++的一些库
4. Linux核心：提供了电源管理、进程调度、内存管理、网络协议栈、驱动模型等核心系统服务

**android中的四大组件以及应用场景**

1. Activity：在Android应用中负责与用户交互的组件。
2. Service：常用于为其他组件提供后台服务或者监控其他组件的运行状态。经常用来执行一些耗时操作。
3. BroadcastReceiver：用于监听应用程序中的其他组件。
4. ContentProvider：Android应用程序之间实现实时数据交换。

1、Activity的生命周期

生命周期：对象什么时候生，什么时候死，怎么写代码，代码往那里写。

**注意：**

1. 当打开新的Activity，采用透明主题的时候，当前Activity不会回调onStop
2. onCreate和onDestroy配对，onStart和onStop配对（是否可见），onResume和onPause配对（是否在前台，可以与用户交互）
3. 打开新的Activity的时候，相关的Log为：

Main1Activity: onPause

Main2Activity: onCreate

Main2Activity: onStart

Main2Activity: onResume

MainA1ctivity: onStop

**异常状态下的生命周期：**

资源相关的系统配置发生改变或者资源不足：例如屏幕旋转，当前Activity会销毁，并且在onStop之前回调onSaveInstanceState保存数据，在重新创建Activity的时候在onStart之后回调onRestoreInstanceState。其中Bundle数据会传到onCreate（不一定有数据）和onRestoreInstanceState（一定有数据）。

   防止屏幕旋转的时候重建，在清单文件中添加配置：

 android:configChanges="orientation"

2、Fragment的生命周期

**正常启动**

   Activity: onCreate

   Fragment: onAttach

   Fragment: onCreate

   Fragment: onCreateView

   Fragment: onActivityCreated

   Activity: onStart

   Activity: onResume

**正常退出**

 Activity: onPause

   Activity: onStop

   Fragment: onDestroyView

   Fragment: onDestroy

   Fragment: onDetach

   Activity: onDestroy

3、Activity的启动模式

1. standard：每次激活Activity时(startActivity)，都创建Activity实例，并放入任务栈；
2. singleTop：如果某个Activity自己激活自己，即任务栈栈顶就是该Activity，则不需要创建，其余情况都要创建Activity实例；
3. singleTask：如果要激活的那个Activity在任务栈中存在该实例，则不需要创建，只需要把此Activity放入栈顶，即把该Activity以上的Activity实例都pop，并调用其onNewIntent；
4. singleInstance：应用1的任务栈中创建了MainActivity实例，如果应用2也要激活MainActivity，则不需要创建，两应用共享该Activity实例。

4、Activity与Fragment之间的传值

1. 通过findFragmentByTag或者getActivity获得对方的引用（强转）之后，再相互调用对方的public方法，但是这样做一是引入了“强转”的丑陋代码，另外两个类之间各自持有对方的强引用，耦合较大，容易造成内存泄漏。
2. 通过Bundle的方法进行传值，例如以下代码：

//Activity中对fragment设置一些参数

fragment.setArguments(bundle);

//fragment中通过getArguments获得Activity中的方法

Bundle arguments = getArguments()

1. 利用eventbus进行通信，这种方法实时性高，而且Activity与Fragment之间可以完全解耦。

//Activity中的代码

EventBus.getDefault().post("消息");

//Fragment中的代码

EventBus.getDefault().register(this);

@Subscribe

public void test(String text) {

    tv\_test.setText(text);

}

5、Service

**Service分为两种：**

1. 本地服务，属于同一个应用程序，通过startService来启动或者通过bindService来绑定并且获取代理对象。如果只是想开个服务在后台运行的话，直接startService即可，如果需要相互之间进行传值或者操作的话，就应该通过bindService。
2. 远程服务（不同应用程序之间），通过bindService来绑定并且获取代理对象。

**对应的生命周期如下：**

context.startService() ->onCreate()- >onStartCommand()->Service running--调用context.stopService() ->onDestroy()

   context.bindService()->onCreate()->onBind()->Service running--调用>onUnbind() -> onDestroy()

**注意**

Service默认是运行在main线程的，因此Service中如果需要执行耗时操作（大文件的操作，数据库的拷贝，网络请求，文件下载等）的话应该在子线程中完成。

！特殊情况是：Service在清单文件中指定了在其他进程中运行。

6、Android中的消息传递机制

**为什么要使用Handler？**

因为屏幕的刷新频率是60Hz，大概16毫秒会刷新一次，所以为了保证UI的流畅性，耗时操作需要在子线程中处理，子线程不能直接对UI进行更新操作。因此需要Handler在子线程发消息给主线程来更新UI。

这里再深入一点，Android中的UI控件不是线程安全的，因此在多线程并发访问UI的时候会导致UI控件处于不可预期的状态。Google不通过锁的机制来处理这个问题是因为：

1. 引入锁会导致UI的操作变得复杂
2. 引入锁会导致UI的运行效率降低

因此，Google的工程师最后是通过单线程的模型来操作UI，开发者只需要通过Handler在不同线程之间切花就可以了。

**概述一下Android中的消息机制？**

Android中的消息机制主要是指Handler的运行机制。Handler是进行线程切换的关键，在主线程和子线程之间切换只是一种比较特殊的使用情景而已。其中消息传递机制需要了解的东西有Message、Handler、Looper、Looper里面的MessageQueue对象。

如上图所示，我们可以把整个消息机制看作是一条流水线。其中：

1. MessageQueue是传送带，负责Message队列的传送与管理
2. Looper是流水线的发动机，不断地把消息从消息队列里面取出来，交给Handler来处理
3. Message是每一件产品
4. Handler就是工人。但是这么比喻不太恰当，因为发送以及最终处理Message的都是Handler

**为什么在子线程中创建Handler会抛异常？**

Handler的工作是依赖于Looper的，而Looper（与消息队列）又是属于某一个线程（ThreadLocal是线程内部的数据存储类，通过它可以在指定线程中存储数据，其他线程则无法获取到），其他线程不能访问。因此Handler就是间接跟线程是绑定在一起了。因此要使用Handler必须要保证Handler所创建的线程中有Looper对象并且启动循环。因为子线程中默认是没有Looper的，所以会报错。

正确的使用方法是：

 handler = null;

   new Thread(new Runnable() {

       private Looper mLooper;

       @Override

       public void run() {

           //必须调用Looper的prepare方法为当前线程创建一个Looper对象，然后启动循环

           //prepare方法中实质是给ThreadLocal对象创建了一个Looper对象

           //如果当前线程已经创建过Looper对象了，那么会报错

           Looper.prepare();

           handler = new Handler();

           //获取Looper对象

           mLooper = Looper.myLooper();

           //启动消息循环

           Looper.loop();

           //在适当的时候退出Looper的消息循环，防止内存泄漏

           mLooper.quit();

       }

   }).start();

主线程中默认是创建了Looper并且启动了消息的循环的，因此不会报错：  
应用程序的入口是ActivityThread的main方法，在这个方法里面会创建Looper，并且执行Looper的loop方法来启动消息的循环，使得应用程序一直运行。

子线程中可以通过Handler发送消息给主线程吗？

可以。有时候出于业务需要，主线程可以向子线程发送消息。子线程的Handler必须按照上述方法创建，并且关联Looper。

7、事件传递机制以及自定义View相关

**Android的视图树**

Android中View的机制主要是Activity的显示，每个Activity都有一个Window（具体在手机中的实现类是PhoneWindow），Window以下有DecorView，DecorView下面有TitleVie以及ContentView，而ContentView就是我们在Activity中通过setContentView指定的。

**事件传分发机制**

ViewGroup有以下三个与事件分发的方法，而View只有dispatchTouchEvent和onTouchEvent。

 @Override

   public boolean dispatchTouchEvent(MotionEvent ev) {

       return super.dispatchTouchEvent(ev);

   }

   @Override

   public boolean onInterceptTouchEvent(MotionEvent ev) {

       return super.onInterceptTouchEvent(ev);

   }

   @Override

   public boolean onTouchEvent(MotionEvent event) {

       return super.onTouchEvent(event);

   }

事件总是从上往下进行分发，即先到达Activity，再到达ViewGroup，再到达子View，如果没有任何视图消耗事件的话，事件会顺着路径往回传递。其中：

1. dispatchTouchEvent是事件的分发方法，如果事件能够到达该视图的话，就首先一定会调用，一般我们不会去修改这个方法。
2. onInterceptTouchEvent是事件分发的核心方法，表示ViewGroup是否拦截事件，如果返回true表示拦截，在这之后ViewGroup的onTouchEvent会被调用，事件就不会往下传递。
3. onTouchEvent是最低级的，在事件分发中最后被调用。
4. 子View可以通过requestDisallowInterceptTouchEvent方法去请求父元素不要拦截。

**注意**

1. 事件从Activity.dispatchTouchEvent()开始传递，只要没有被停止或拦截，从最上层的View(ViewGroup)开始一直往下(子View)传递。子View 可以通过onTouchEvent()对事件进行处理。
2. 事件由父View(ViewGroup)传递给子View，ViewGroup 可以通过onInterceptTouchEvent()对事件做拦截，停止其往下传递。
3. 如果事件从上往下传递过程中一直没有被停止，且最底层子View 没有消费事件，事件会反向往上传递，这时父View(ViewGroup)可以进行消费，如果还是没有被消费的话，最后会到Activity 的onTouchEvent()函数。
4. 如果View 没有对ACTION\_DOWN 进行消费，之后的其他事件不会传递过来。
5. OnTouchListener 优先于onTouchEvent()对事件进行消费。

**自定义View的分类**

1. 对现有的View的子类进行扩展，例如复写onDraw方法、扩展新功能等。
2. 自定义组合控件，把常用一些控件组合起来以方便使用。
3. 直接继承View实现View的完全定制，需要完成View的测量以及绘制。
4. 自定义ViewGroup，需要复写onLayout完成子View位置的确定等工作。

**View的测量-onMeasure**

View的测量最终是在onMeasure方法中通过setMeasuredDimension把代表宽高两个MeasureSpec设置给View，因此需要掌握MeasureSpec。MeasureSpec包括大小信息以及模式信息。

**MeasureSpec的三种模式：**

1. EXACTLY模式：精确模式，对应于用户指定为match\_parent或者具体大小的时候（实际上指定为match\_parent实质上是指定大小为父容器的大小）
2. AT\_MOST模式：对应于用户指定为wrap\_content，此时控件尺寸只要不超过父控件允许的最大尺寸即可。
3. UNSPECIFIED模式：不指定大小的测量模式，这种模式比较少用

下面给出模板代码：

   public class MeasureUtils {

       /\*\*

        \* 用于View的测量

        \*

        \* @param measureSpec

        \* @param defaultSize

        \* @return

        \*/

       public static int measureView(int measureSpec, int defaultSize) {

           int measureSize;

           //获取用户指定的大小以及模式

           int mode = View.MeasureSpec.getMode(measureSpec);

           int size = View.MeasureSpec.getSize(measureSpec);

           //根据模式去返回大小

           if (mode == View.MeasureSpec.EXACTLY) {

               //精确模式（指定大小以及match\_parent）直接返回指定的大小

               measureSize = size;

           } else {

               //UNSPECIFIED模式、AT\_MOST模式（wrap\_content）的话需要提供默认的大小

               measureSize = defaultSize;

               if (mode == View.MeasureSpec.AT\_MOST) {

                   //AT\_MOST（wrap\_content）模式下，需要取测量值与默认值的最小值

                   measureSize = Math.min(measureSize, defaultSize);

               }

           }

           return measureSize;

       }

   }

最后，复写onMeasure方法，把super方法去掉：

   @Override

   protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {

       setMeasuredDimension(MeasureUtils.measureView(widthMeasureSpec, 200),

               MeasureUtils.measureView(heightMeasureSpec, 200)

       );

   }

**View的绘制-onDraw**

View绘制，需要掌握Android中View的坐标体系：

View的坐标体系是以左上角为坐标原点，向右为X轴正方向，向下为Y轴正方向。

View绘制，主要是通过Android的2D绘图机制来完成，时机是onDraw方法中，其中包括画布Canvas，画笔Paint。下面给出示例代码。相关API不是介绍的重点，重点是Canvas的save和restore方法，通过save以后可以对画布进行一些放大缩小旋转倾斜等操作，这两个方法一般配套使用，其中save的调用次数可以多于restore。

   @Override

   protected void onDraw(Canvas canvas) {

       super.onDraw(canvas);

       Bitmap bitmap = ImageUtils.drawable2Bitmap(mDrawable);

       canvas.drawBitmap(bitmap, getLeft(), getTop(), mPaint);

       canvas.save();

       //注意，这里的旋转是指画布的旋转

       canvas.rotate(90);

       mPaint.setColor(Color.parseColor("#FF4081"));

       mPaint.setTextSize(30);

       canvas.drawText("测试", 100, -100, mPaint);

       canvas.restore();

   }

**View的位置-onLayout**

与布局位置相关的是onLayout方法的复写，一般我们自定义View的时候，只需要完成测量，绘制即可。如果是自定义ViewGroup的话，需要做的就是在onLayout中测量自身以及控制子控件的布局位置，onLayout是自定义ViewGroup必须实现的方法。

8、性能优化

**布局优化**

1. 使用include标签，通过layout属性复用相同的布局。

 <include

    android:id="@+id/v\_test"

    layout="@layout/include\_view" /

1. 使用merge标签，去除同类的视图
2. 使用ViewStub来进行布局的延迟加载一些不是马上就用到的布局。例如列表页中，列表在没有拿到数据之前不加载，这样做可以使UI变得流畅。

<ViewStub

    android:id="@+id/v\_stub"

    android:layout\_width="match\_parent"

    android:layout\_height="wrap\_content"

    android:layout="@layout/view\_stub" />

//需要手动调用inflate方法，布局才会显示出来。

stub.inflate();

//其中setVisibility在底层也是会调用inflate方法

//stub.setVisibility(View.VISIBLE);

//之后，如果要使用ViewStub标签里面的View，只需要按照平常来即可。

TextView tv\_1 = (TextView) findViewById(R.id.tv\_1);

1. 尽量多使用RelativeLayout，因为这样可以大大减少视图的层级。

内存优化

**APP设计以及代码编写阶段都应该考虑内存优化：**

1. 珍惜Service，尽量使得Service在使用的时候才处于运行状态。尽量使用IntentService

IntentService在内部其实是通过线程以及Handler实现的，当有新的Intent到来的时候，会创建线程并且处理这个Intent，处理完毕以后就自动销毁自身。因此使用IntentService能够节省系统资源。

1. 内存紧张的时候释放资源（例如UI隐藏的时候释放资源等）。复写Activity的回调方法。

@Override

public void onLowMemory() {

    super.onLowMemory();

}

@Override

public void onTrimMemory(int level) {

    super.onTrimMemory(level);

    switch (level) {

        case TRIM\_MEMORY\_COMPLETE:

            //...

            break;

        case 其他:

    }

}

1. 通过Manifest中对Application配置更大的内存，但是一般不推荐

android:largeHeap="true"

1. 避免Bitmap的浪费，应该尽量去适配屏幕设备。尽量使用成熟的图片加载框架，Picasso，Fresco，Glide等。
2. 使用优化的容器，SparseArray等
3. 其他建议：尽量少用枚举变量，尽量少用抽象，尽量少增加类，避免使用依赖注入框架，谨慎使用library，使用代码混淆，时当场合考虑使用多进程等。
4. 避免内存泄漏（本来应该被回收的对象没有被回收）。一旦APP的内存短时间内快速增长或者GC非常频繁的时候，就应该考虑是否是内存泄漏导致的。

**分析方法**

1. 使用Android Studio提供的Android Monitors中Memory工具查看内存的使用以及没使用的情况。

2. 使用DDMS提供的Heap工具查看内存使用情况，也可以手动触发GC。

3. 使用性能分析的依赖库，例如Square的LeakCanary，这个库会在内存泄漏的前后通过Notification通知你。

**什么情况会导致内存泄漏**

1. 资源释放问题：程序代码的问题，长期保持某些资源，如Context、Cursor、IO 流的引用，资源得不到释放造成内存泄露。
2. 对象内存过大问题：保存了多个耗用内存过大的对象（如Bitmap、XML 文件），造成内存超出限制。
3. static 关键字的使用问题：static 是Java 中的一个关键字，当用它来修饰成员变量时，那么该变量就属于该类，而不是该类的实例。所以用static 修饰的变量，它的生命周期是很长的，如果用它来引用一些资源耗费过多的实例（Context 的情况最多），这时就要谨慎对待了。

解决方案  
1. 应该尽量避免static 成员变量引用资源耗费过多的实例，比如Context。  
2. Context 尽量使用ApplicationContext，因为Application 的Context 的生命周期比较长，引用它不会出现内存泄露的问题。  
3. 使用WeakReference 代替强引用。比如可以使用WeakReference<Context> mContextRef

1. 线程导致内存溢出：线程产生内存泄露的主要原因在于线程生命周期的不可控。例如Activity中的Thread在run了，但是Activity由于某种原因重新创建了，但是Thread仍然会运行，因为run方法不结束的话Thread是不会销毁的。

**解决方案**

1. 将线程的内部类，改为静态内部类（因为非静态内部类拥有外部类对象的强引用，而静态类则不拥有）。

2. 在线程内部采用弱引用保存Context 引用。

**查看内存泄漏的方法、工具**

1. android官方提供的工具：Memory Monitor（当APP占用的内存在短时间内快速增长或者GC变得频繁的时候）、DDMS提供的Heap工具（手动触发GC）
2. Square提供的内存泄漏检测工具，LeakCanary（能够自动完成内存追踪、检测、输出结果），进行演示，并且适当的解说。

**性能优化**

1. 防止过度绘制，通过打开手机的“显示过度绘制区域”即可查看过度绘制的情况。
2. 最小化渲染时间，使用视图树查看节点，对节点进行性能分析。
3. 通过TraceView进行数据的采集以及分析。在有大概定位的时候，使用Android官方提供的Debug类进行采集。最后通过DDMS即可打开这个.trace文件，分析函数的调用情况（包括在指定情况下执行时间，调用次数）

//开启数据采集

Debug.startMethodTracing("test.trace");

//关闭

Debug.stopMethodTracing();

**OOM**

避免OOM的一些常见方法：

1. App资源中尽量少用大图。使用Bitmap的时候要注意等比例缩小图片，并且注意Bitmap的回收。

BitmapFactory.Options options = new BitmapFactory.Option();

options.inSampleSize = 2;

//Options 只保存图片尺寸大小，不保存图片到内存

BitmapFactory.Options opts = new BitmapFactory.Options();

opts.inSampleSize = 2;

Bitmap bmp = null;

bmp = BitmapFactory.decodeResource(getResources(),

mImageIds[position],opts);

//回收

bmp.recycle();

1. 结合组件的生命周期，释放资源
2. IO流，数据库查询的游标等应该在使用完之后及时关闭。
3. ListView中应该使用ViewHolder模式缓存ConverView
4. 页面切换的时候尽量去传递（复用）一些对象

**ANR**

不同的组件发生ANR 的时间不一样，主线程（Activity、Service）是5 秒，BroadCastReceiver 是10 秒。

**ANR一般有三种类型：**

1. KeyDispatchTimeout(5 seconds)  
   主要类型按键或触摸事件在特定时间内无响应
2. BroadcastTimeout(10 seconds)  
   BroadcastReceiver在特定时间内无法处理完成
3. ServiceTimeout(20 seconds)  
   小概率类型Service在特定的时间内无法处理完成

**解决方案：**

1. UI线程只进行UI相关的操作。所有耗时操作，比如访问网络，Socket 通信，查询大量SQL 语句，复杂逻辑计算等都放在子线程中去，然后通过handler.sendMessage、runonUITread、AsyncTask 等方式更新UI。

2. 无论如何都要确保用户界面操作的流畅度。如果耗时操作需要让用户等待，那么可以在界面上显示进度条。

3. BroadCastReceiver要进行复杂操作的的时候，可以在onReceive()方法中启动一个Service来处理。

9、九切图（.9图）、SVG图片

**九切图**

点九图，是Android开发中用到的一种特殊格式的图片，文件名以”.9.png“结尾。这种图片能告诉程序，图像哪一部分可以被拉升，哪一部分不能被拉升需要保持原有比列。运用点九图可以保证图片在不模糊变形的前提下做到自适应。点九图常用于对话框背景图片中。

1. 1、2部分规定了图像的可拉伸部分,当实际程序中设定了对话框的宽高时，1、2部分就会被拉伸成所需要的高和宽，呈现出于设计稿一样的视觉效果。
2. 而3、4部分规定了图像的内容区域。内容区域规定了可编辑区域，例如文字需要被包裹在其内。

**android5.0的SCG矢量动画机制**

1. 图像在方法缩小的时候图片质量不会有损失
2. 使用XML来定义图形
3. 适配不同分辨率

10、Android中数据常见存储方式

1. 文件（包括XML、SharePreference等）
2. 数据库
3. Content Provider
4. 保存在网络

11、进程间通信

**操作系统进程间通信的方法，android中有哪些？**

**操作系统：**

1. Windows：剪贴板、管道、邮槽等
2. Linux：命名管道、共享内存、信号量

**Android中的进程通信方式并不是完全继承于Linux：**

1. Bundle
2. 文件共享
3. AIDL
4. Messenger
5. Content Provider
6. Socket

12、常见的网络框架

**常用的http框架以及他们的特点**

1. HttpURLConnection:在Android 2.2版本之前，HttpClient拥有较少的bug，因此使用它是最好的选择。而在Android 2.3版本及以后，HttpURLConnection则是最佳的选择。它的API简单，体积较小，因而非常适用于Android项目。压缩和缓存机制可以有效地减少网络访问的流量，在提升速度和省电方面也起到了较大的作用。对于新的应用程序应该更加偏向于使用HttpURLConnection，因为在以后的工作当中我们也会将更多的时间放在优化HttpURLConnection上面。特点：比较轻便，灵活，易于扩展，在3.0后以及4.0中都进行了改善，如对HTTPS的支持，在4.0中，还增加了对缓存的支持。
2. HttpClient：高效稳定，但是维护成本高昂，故android 开发团队不愿意在维护该库而是转投更为轻便的
3. okHttp：okhttp 是一个 Java 的 HTTP+SPDY 客户端开发包，同时也支持 Android。需要Android 2.3以上。特点：OKHttp是Android版Http客户端。非常高效，支持SPDY、连接池、GZIP和 HTTP 缓存。默认情况下，OKHttp会自动处理常见的网络问题，像二次连接、SSL的握手问题。如果你的应用程序中集成了OKHttp，Retrofit默认会使用OKHttp处理其他网络层请求。从Android4.4开始HttpURLConnection的底层实现采用的是okHttp。
4. volley：早期使用HttpClient，后来使用HttpURLConnection，是谷歌2013年推出的网络请求框架，非常适合去进行数据量不大，但通信频繁的网络操作，而对于大数据量的网络操作，比如说下载文件等，Volley的表现就会非常糟糕。
5. xutils：缓存网络请求数据
6. Retrofit：和Volley框架的请求方式很相似，底层网络请求采用okhttp（效率高，android4.4底层采用okhttp），采用注解方式来指定请求方式和url地址，减少了代码量。
7. AsyncTask

13、常用的图片加载框架以及特点、源码

1. Picasso：PicassoSquare的网络库一起能发挥最大作用，因为Picasso可以选择将网络请求的缓存部分交给了okhttp实现。
2. Glide：模仿了Picasso的API，而且在他的基础上加了很多的扩展(比如gif等支持)，支持图片流，因此在做爱拍之类的视频应用用得比较多一些。
3. Fresco：Fresco中设计有一个叫做image pipeline的模块。它负责从网络，从本地文件系统，本地资源加载图片。 为了最大限度节省空间和CPU时间，它含有3级缓存设计（2级内存，1级文件）。Fresco中设计有一个叫做Drawees模块， 方便地显示loading图，当图片不再显示在屏幕上时，及时地释放内存和空间占用。

**Fresco是把图片缓存放在了Ashmem（系统匿名内存共享区）**

1. Heap-堆内存：Android中每个App的 Java堆内存大小都是被严格的限制的。每个对象都是使用Java的new在堆内存实例化，这是内存中相对安全的一块区域。内存有垃圾回收机制，所以当 App不在使用内存的时候，系统就会自动把这块内存回收。不幸的是，内存进行垃圾回收的过程正是问题所在。当内存进行垃圾回收时，内存不仅仅进行了垃圾回收，还把 Android 应用完全终止了。这也是用户在使用 App 时最常见的卡顿或短暂假死的原因之一。
2. Ashmem：Android 在操作 Ashmem 堆时，会把该堆中存有数据的内存区域从 Ashmem 堆中抽取出来，而不是把它释放掉，这是一种弱内存释放模式；被抽取出来的这部分内存只有当系统真正需要更多的内存时（系统内存不够用）才会被释放。当 Android 把被抽取出来的这部分内存放回 Ashmem 堆，只要被抽取的内存空间没有被释放，之前的数据就会恢复到相应的位置。

不管发生什么，垃圾回收器都不会自动回收这些 Bitmap。当 Android 绘制系统在渲染这些图片，Android 的系统库就会把这些 Bitmap 从 Ashmem 堆中抽取出来，而当渲染结束后，这些 Bitmap 又会被放回到原来的位置。如果一个被抽取的图片需要再绘制一次，系统仅仅需要把它再解码一次，这个操作非常迅速。

14、在Android开发里用什么做线程间的通讯工具？

传统点的方法就是往同步代码块里些数据，然后使用回调让另外一条线程去读。在Android里我一般会创建Looper线程，然后Hanlder传递消息。

15、Android新特性相关

1. 5.0：Material Design、多种设备的支持、支持64位ART虚拟机、Project Volta电池续航改进计划等
2. 6.0：动态权限管理、过度动画、支付、指纹等
3. 7.0：分屏、通知消息快捷回复、夜间模式、流量保护模式等

16、网络请求优化

**网络请求优化**

1. 能够缓存起来的尽量去缓存起来，减轻服务器的压力。例如APP中首页的一些数据，又例如首页的图标、文案都是缓存起来的，而且这些数据通过网络来指定可以使app具有更大的灵活性。
2. 不用域名，用 IP 直连，省去了DNS域名解析。
3. 连接复用、请求合并、请求数据Body可以利用压缩算法Gzip来进行压缩，使用JSON 代替 XML

**网络请求的安全性**

这块了解的不多。我给你说说我的思路吧，利用哈希算法，比如MD5，服务器给我们的数据可以通过时间戳和其他参数做个加密，得到一个key，在客户端取出数据后根据数据和时间戳再去生成key与服务端给的做个对比。