**Android面试帮助篇**

**1、要做一个尽可能流畅的ListView，你平时在工作中如何进行优化的？①Item布局，层级越少越好，使用hierarchyview工具查看优化。**

②复用convertView

③使用ViewHolder

④item中有图片时，异步加载

⑤快速滑动时，不加载图片

⑥item中有图片时，应对图片进行适当压缩

⑦实现数据的分页加载

**2、对于Android 的安全问题，你知道多少**

①错误导出组件

② 参数校验不严

③WebView引入各种安全问题,webview中的js注入

④不混淆、不防二次打包

⑤明文存储关键信息

⑦ 错误使用HTTPS

⑧山寨加密方法

⑨滥用权限、内存泄露、使用debug签名

**3、如何缩减APK包大小？**

代码

保持良好的编程习惯，不要重复或者不用的代码，谨慎添加libs，移除使用不到的libs。

使用proguard混淆代码，它会对不用的代码做优化，并且混淆后也能够减少安装包的大小。

native code的部分，大多数情况下只需要支持armabi与x86的架构即可。如果非必须，可以考虑拿掉x86的部分。

资源

使用Lint工具查找没有使用到的资源。去除不使用的图片，String，XML等等。 assets目录下的资源请确保没有用不上的文件。

生成APK的时候，aapt工具本身会对png做优化，但是在此之前还可以使用其他工具如tinypng对图片进行进一步的压缩预处理。

jpeg还是png，根据需要做选择，在某些时候jpeg可以减少图片的体积。 对于9.png的图片，可拉伸区域尽量切小，另外可以通过使用9.png拉伸达到大图效果的时候尽量不要使用整张大图。

策略

有选择性的提供hdpi，xhdpi，xxhdpi的图片资源。建议优先提供xhdpi的图片，对于mdpi，ldpi与xxxhdpi根据需要提供有差异的部分即可。

尽可能的重用已有的图片资源。例如对称的图片，只需要提供一张，另外一张图片可以通过代码旋转的方式实现。

能用代码绘制实现的功能，尽量不要使用大量的图片。例如减少使用多张图片组成animate-list的AnimationDrawable，这种方式提供了多张图片很占空间。

**4、Android与服务器交互的方式中的对称加密和非对称加密是什么?**对称加密，就是加密和解密数据都是使用同一个key，这方面的算法有DES。

非对称加密，加密和解密是使用不同的key。发送数据之前要先和服务端约定生成公钥和私钥，使用公钥加密的数据可以用私钥解密，反之。这方面的算法有RSA。ssh 和 ssl都是典型的非对称加密。

**5、设备横竖屏切换的时候，接下来会发生什么？**

1、不设置Activity的android:configChanges时，切屏会重新调用各个生命周期，切横屏时会执行一次，切竖屏时会执行两次

2、设置Activity的android:configChanges=”orientation”时，切屏还是会重新调用各个生命周期，切横、竖屏时只会执行一次

3、设置Activity的android:configChanges=”orientation|keyboardHidden”时，切屏不会重新调用各个生命周期，只会执行onConfigurationChanged方法

**6、Android启动Service的两种方式是什么? 它们的适用情况是什么?**

如果后台服务开始后基本可以独立运行的话，可以用startService。音乐播放器就可以这样用。它们会一直运行直到你调用 stopSelf或者stopService。你可以通过发送Intent或者接收Intent来与正在运行的后台服务通信，但大部分时间，你只是启动服务并让它独立运行。如果你需要与后台服务通过一个持续的连接来比较频繁地通信，建议使用bind()。比如你需要定位服务不停地把更新后的地理位置传给UI。Binder比Intent开发起来复杂一些，但如果真的需要，你也只能使用它。

startService：生命周期与调用者不同。启动后若调用者未调用stopService而直接退出，Service仍会运行

bindService：生命周期与调用者绑定，调用者一旦退出，Service就会调用unBind->onDestroy

**7、谈谈你对Android中Context的理解?**

Context:包含上下文信息(外部值) 的一个参数. Android 中的 Context 分三种,Application Context ,Activity Context ,Service Context.

它描述的是一个应用程序环境的信息，通过它我们可以获取应用程序的资源和类，也包括一些应用级别操作，例如：启动一个Activity，发送广播，接受Intent信息等

**8、Service的onCreate回调在UI线程中吗？**

Service生命周期的各个回调和其他的应用组件一样，是跑在主线程中，会影响到你的UI操作或者阻塞主线程中的其他事情

**9、请介绍下AsyncTask的内部实现，适用的场景是？**

AsyncTask内部也是Handler机制来完成的，只不过Android提供了执行框架来提供线程池来执行相应地任务，因为线程池的大小问题，所以AsyncTask只应该用来执行耗时时间较短的任务，比如HTTP请求，大规模的下载和数据库的更改不适用于AsyncTask，因为会导致线程池堵塞，没有线程来执行其他的任务，导致的情形是会发生AsyncTask根本执行不了的问题。

**10、谈谈你对binder机制的理解?**

binder是一种IPC机制,进程间通讯的一种工具.

Java层可以利用aidl工具来实现相应的接口.

**11、Android中进程间通信有哪些实现方式？**

Intent，Binder（AIDL），Messenger，BroadcastReceiver

**12、介绍下实现一个自定义view的基本流程**

1、自定义View的属性 编写attr.xml文件

2、在layout布局文件中引用，同时引用命名空间

3、在View的构造方法中获得我们自定义的属性 ，在自定义控件中进行读取（构造方法拿到attr.xml文件值）

4、重写onMesure

5、重写onDraw

**13、Android中touch事件的传递机制是怎样的?**

1、Touch事件传递的相关API有dispatchTouchEvent、onTouchEvent、onInterceptTouchEvent

2、Touch事件相关的类有View、ViewGroup、Activity

3、Touch事件会被封装成MotionEvent对象，该对象封装了手势按下、移动、松开等动作

4、Touch事件通常从Activity#dispatchTouchEvent发出，只要没有被消费，会一直往下传递，到最底层的View。

5、如果Touch事件传递到的每个View都不消费事件，那么Touch事件会反向向上传递,最终交由Activity#onTouchEvent处理.

6、onInterceptTouchEvent为ViewGroup特有，可以拦截事件.

7、Down事件到来时，如果一个View没有消费该事件，那么后续的MOVE/UP事件都不会再给它

**14、Android多线程的实现方式有哪些?**

Thread & AsyncTask

Thread 可以与Loop 和 Handler 共用建立消息处理队列

AsyncTask 可以作为线程池并行处理多任务

**15、Android开发中何时使用多进程？使用多进程的好处是什么？**

要想知道如何使用多进程，先要知道Android里的多进程概念。一般情况下，一个应用程序就是一个进程，这个进程名称就是应用程序包名。我们知道进程是系统分配资源和调度的基本单位，所以每个进程都有自己独立的资源和内存空间，别的进程是不能任意访问其他进程的内存和资源的。

那如何让自己的应用拥有多个进程？

很简单，我们的四大组件在AndroidManifest文件中注册的时候，有个属性是android:process，

1、这里可以指定组件的所处的进程。默认就是应用的主进程。指定为别的进程之后，系统在启动这个组件的时候，就先创建（如果还没创建的话）这个进程，然后再创建该组件。你可以重载Application类的onCreate方法，打印出它的进程名称，就可以清楚的看见了。再设置android:process属性时候，有个地方需要注意：如果是android:process=”:deamon”，以:开头的名字，则表示这是一个应用程序的私有进程，否则它是一个全局进程。私有进程的进程名称是会在冒号前自动加上包名，而全局进程则不会。一般我们都是有私有进程，很少使用全局进程。他们的具体区别不知道有没有谁能补充一下。

2、使用多进程显而易见的好处就是分担主进程的内存压力。我们的应用越做越大，内存越来越多，将一些独立的组件放到不同的进程，它就不占用主进程的内存空间了。当然还有其他好处，有心人会发现Android后台进程里有很多应用是多个进程的，因为它们要常驻后台，特别是即时通讯或者社交应用，不过现在多进程已经被用烂了。典型用法是在启动一个不可见的轻量级私有进程，在后台收发消息，或者做一些耗时的事情，或者开机启动这个进程，然后做监听等。还有就是防止主进程被杀守护进程，守护进程和主进程之间相互监视，有一方被杀就重新启动它。应该还有还有其他好处，这里就不多说了。

3、坏处的话，多占用了系统的空间，大家都这么用的话系统内存很容易占满而导致卡顿。消耗用户的电量。应用程序架构会变复杂，应为要处理多进程之间的通信。这里又是另外一个问题了。

**16、ANR是什么？怎样避免和解决ANR?**

ANR:Application Not Responding，即应用无响应

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

超时的原因一般有两种：

(1)当前的事件没有机会得到处理（UI线程正在处理前一个事件没有及时完成或者looper被某种原因阻塞住）

(2)当前的事件正在处理，但没有及时完成

UI线程尽量只做跟UI相关的工作，耗时的工作（数据库操作，I/O，连接网络或者其他可能阻碍UI线程的操作）放入单独的线程处理，尽量用Handler来处理UI thread和thread之间的交互。

UI线程主要包括如下：

Activity:onCreate(), onResume(), onDestroy(), onKeyDown(), onClick()

AsyncTask: onPreExecute(), onProgressUpdate(), onPostExecute(), onCancel()

Mainthread handler: handleMessage(), post(runnable r)

other

**17、Android下解决滑动冲突的常见思路是什么?**

相关的滑动组件 重写onInterceptTouchEvent，然后判断根据xy值，来决定是否要拦截当前操作

**18、如何把一个应用设置为系统应用？**

成为系统应用，首先要在 对应设备的 Android 源码 SDK 下编译，编译好之后：

此 Android 设备是 Debug 版本，并且已经 root，直接将此 apk 用 adb 工具 push 到 system/app 或 system/priv-app 下即可。

如果非 root 设备，需要编译后重新烧写设备镜像即可。

有些权限(如 WRITE\_SECURE\_SETTINGS )，是不开放给第三方应用的，只能在对应设备源码中编译然后作为系统 app 使用。

**19、Android内存泄露研究**

Android内存泄漏指的是进程中某些对象（垃圾对象）已经没有使用价值了，但是它们却可以直接或间接地引用到gc roots导致无法被GC回收。无用的对象占据着内存空间，使得实际可使用内存变小，形象地说法就是内存泄漏了。

场景

类的静态变量持有大数据对象

静态变量长期维持到大数据对象的引用，阻止垃圾回收。

非静态内部类的静态实例

非静态内部类会维持一个到外部类实例的引用，如果非静态内部类的实例是静态的，就会间接长期维持着外部类的引用，阻止被回收掉。

资源对象未关闭

资源性对象如Cursor、File、Socket，应该在使用后及时关闭。未在finally中关闭，会导致异常情况下资源对象未被释放的隐患。

注册对象未反注册

未反注册会导致观察者列表里维持着对象的引用，阻止垃圾回收。

Handler临时性内存泄露

Handler通过发送Message与主线程交互，Message发出之后是存储在MessageQueue中的，有些Message也不是马上就被处理的。在Message中存在一个 target，是Handler的一个引用，如果Message在Queue中存在的时间越长，就会导致Handler无法被回收。如果Handler是非静态的，则会导致Activity或者Service不会被回收。

由于AsyncTask内部也是Handler机制，同样存在内存泄漏的风险。

此种内存泄露，一般是临时性的。

**20、内存泄露检测有什么好方法？**

检测：

1、DDMS Heap发现内存泄露

dataObject totalSize的大小，是否稳定在一个范围内，如果操作程序，不断增加，说明内存泄露

2、使用Heap Tool进行内存快照前后对比

BlankActivity手动触发GC进行前后对比，对象是否被及时回收

定位：

1、MAT插件打开.hprof具体定位内存泄露：

查看histogram项，选中某一个对象，查看它的GC引用链，因为存在GC引用链的，说明无法回收

2、AndroidStudio的Allocation Tracker：

观测到期间的内存分配，哪些对象被创建，什么时候创建，从而准确定位