### 1，低版本 SDK 实现高版本 api

两种情况：

1） 一般很多高版本的新的API都会在兼容包里面找到替代的实现。比如fragment。Notification，在v4兼容包里面有NotificationCompat类。5.0+出现的backgroundTint，minSdk小于5.0的话会包检测错误，v4兼容包DrawableCompat类。

2） 没有替代实现就自己手动实现。比如：控件的水波纹效果—第三方实现。或者直接在低版本去除这个效果。

3） 补充:如果设置了minSDK但是代码里面使用了高版本的API，会出现检测错误。需要在代码里面使用声明编译检测策略，比如：@SuppressLint和@TargetApi注解提示编译器编译的规则。@SuppressLint是忽略检测；@TargetApi=23，会根据你函数里面使用的API，严格地匹配SDK版本，给出相应的编译错误提示。

4） 为了避免位置的错误，最好不要使用废弃api。（一般情况下不会有兼容性问题，后面可能会随时删除这个API方法；性能方面的问题。）

5） <http://chinagdg.org/2016/01/picking-your-compilesdkversion-minsdkversion-targetsdkversion/>

### 2，Android 的安全问题？

1，参数校验不严格 2，webview 引入js注入 3，明文存储，山寨加密 4，滥用权限，内存泄露，使用debug签名 5，不混淆，不防二次打包，不加固等

### 3，Android与服务器交互的加密方式

对称加密，AES，DES，加密解密用的key是一样的。

非对称加密，RSA，加密解密使用的不同的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，ssh和ssl都是典型的非对称加密。

对极大整数做因数分解的难度决定了RSA的算法的可靠性；对一个极大整数做因数分解越困难，RSA算法就越可靠。

如何破解RSA算法，除非你能找到一种快速因数分解的算法，否则破解的可能非常小

### 4，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

### 5，Android 为每个应用程序分配的内存大小是多少？

看具体的手机平台，常见的有：64M/32M等

### 6，App启动流程

App启动时，AMS会检查这个应用程序所需要的进程是否存在，不存在就会请求Zygote进程启动需要的应用程序进程，

Zygote进程接收到AMS请求并通过fock自身创建应用程序进程，这样应用程序进程就会获取虚拟机的实例，还会创建Binder线程池（ProcessState.startThreadPool()）和消息循环(ActivityThread looper.loop),

然后App进程，通过Binder IPC向sytem\_server进程发起attachApplication请求；

system\_server进程在收到请求后，进行一系列准备工作后，再通过Binder IPC向App进程发送scheduleLaunchActivity请求；

App进程的binder线程（ApplicationThread）在收到请求后，通过handler向主线程发送LAUNCH\_ACTIVITY消息；

主线程在收到Message后，通过反射机制创建目标Activity，并回调Activity.onCreate()等方法。

到此，App便正式启动，开始进入Activity生命周期，执行完onCreate/onStart/onResume方法，UI渲染结束后便可以看到App的主界面。

### 7， 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

### 8，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>

2.采用市面上的处理方法，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>

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SpashActivity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

### 9，Android单线程模型

Android单线程模型的核心原则就是：只能在UI线程(Main Thread)中对UI进行处理。当一个程序第一次启动时，Android会同时启动一个对应的 主线程（Main Thread），主线程主要负责处理与UI相关的事件，如：用户的按键事件，用户接触屏幕的事件以及屏幕绘图事件，并把相关的事件分发到对应的组件进行处理。所以主线程通常又被叫做UI线程。在开发Android应用时必须遵守单线程模型的原则： Android UI操作并不是线程安全的并且这些操作必须在UI线程中执行。

Android的单线程模型有两条原则：

1.不要阻塞UI线程。 2.不要在UI线程之外访问Android UI toolkit（主要是这两个包中的组件：android.widget and android.view

### 10，Java虚拟机和Dalvik虚拟机的区别

Java虚拟机：

1、java虚拟机基于栈。 基于栈的机器必须使用指令来载入和操作栈上数据，所需指令更多更多。 2、java虚拟机运行的是java字节码。（java类会被编译成一个或多个字节码.class文件）

Dalvik虚拟机：

1、dalvik虚拟机是基于寄存器的 2、Dalvik运行的是自定义的.dex字节码格式。（java类被编译成.class文件后，会通过一个dx工具将所有的.class文件转换成一个.dex文件，然后dalvik虚拟机会从其中读取指令和数据 3、常量池已被修改为只使用32位的索引，以 简化解释器。 4、一个应用，一个虚拟机实例，一个进程（所有android应用的线程都是对应一个linux线程，都运行在自己的沙盒中，不同的应用在不同的进程中运行。每个android dalvik应用程序都被赋予了一个独立的linux PID(app\_\*)）