# Android学习总结

[Android学习总结 1](#_Toc90393315)

[一.Activity 11](#_Toc90393316)

[1.Activity的启动流程 11](#_Toc90393317)

[2.onSaveInstanceState(),onRestoreInstanceState的调用时机 11](#_Toc90393318)

[3.activity的启动模式和使用场景 12](#_Toc90393319)

[4.Activity A跳转Activity B，再按返回键，生命周期执行的顺序 12](#_Toc90393320)

[5.横竖屏切换,按home键,按返回键,锁屏与解锁屏幕,跳转透明Activity界面,启动一个 Theme 为 Dialog 的 Activity，弹出Dialog时Activity的生命周期 13](#_Toc90393321)

[6.onStart 和 onResume、onPause 和 onStop 的区别 13](#_Toc90393322)

[7.Activity之间传递数据的方式Intent是否有大小限制，如果传递的数据量偏大，有哪些方案 13](#_Toc90393323)

[8.Activity的onNewIntent()方法什么时候会执行 13](#_Toc90393324)

[二.Service 13](#_Toc90393325)

[1.service 的生命周期，两种启动方式的区别 13](#_Toc90393326)

[2.Service启动流程 13](#_Toc90393327)

[3.Service与Activity怎么实现通信 13](#_Toc90393328)

[4.IntentService是什么,IntentService原理，应用场景及其与Service的区别 14](#_Toc90393329)

[5.Service 的 onStartCommand 方法有几种返回值?各代表什么意思? 14](#_Toc90393330)

[6.bindService和startService混合使用的生命周期以及怎么关闭 14](#_Toc90393331)

[3.BroadcastReceiver 14](#_Toc90393332)

[1.广播的分类和使用场景 14](#_Toc90393333)

[2.广播的两种注册方式的区别 14](#_Toc90393334)

[3.广播发送和接收的原理 14](#_Toc90393335)

[4.本地广播和全局广播的区别 14](#_Toc90393336)

[4.ContentProvider 15](#_Toc90393337)

[1.什么是ContentProvider及其使用 15](#_Toc90393338)

[2.ContentProvider,ContentResolver,ContentObserver之间的关系 15](#_Toc90393339)

[3.ContentProvider的实现原理 15](#_Toc90393340)

[4.ContentProvider的优点 15](#_Toc90393341)

[5.Uri 是什么 15](#_Toc90393342)

[五.Handler 15](#_Toc90393343)

[1.Handler的实现原理 15](#_Toc90393344)

[2.子线程中能不能直接new一个Handler,为什么主线程可以主线程的Looper第一次调用loop方法,什么时候,哪个类 16](#_Toc90393345)

[3.Handler导致的内存泄露原因及其解决方案 16](#_Toc90393346)

[4.一个线程可以有几个Handler,几个Looper,几个MessageQueue对象 16](#_Toc90393347)

[5.Message对象创建的方式有哪些 & 区别？Message.obtain()怎么维护消息池的 16](#_Toc90393348)

[6.Handler 有哪些发送消息的方法 16](#_Toc90393349)

[7.Handler的post与sendMessage的区别和应用场景 16](#_Toc90393350)

[8.handler postDealy后消息队列有什么变化，假设先 postDelay 10s, 再postDelay 1s, 怎么处理这2条消息 16](#_Toc90393351)

[9.MessageQueue是什么数据结构 17](#_Toc90393352)

[10.Handler怎么做到的一个线程对应一个Looper，如何保证只有一个MessageQueue 17](#_Toc90393353)

[11.ThreadLocal在Handler机制中的作用 17](#_Toc90393354)

[六.View绘制 17](#_Toc90393355)

[1.View绘制流程 17](#_Toc90393356)

[2.MeasureSpec是什么 17](#_Toc90393357)

[3.子View创建MeasureSpec创建规则是什么 17](#_Toc90393358)

[4.自定义Viewwrap\_content不起作用的原因 18](#_Toc90393359)

[5.在Activity中获取某个View的宽高有几种方法 18](#_Toc90393360)

[6.为什么onCreate获取不到View的宽高 18](#_Toc90393361)

[7.View#post与Handler#post的区别 18](#_Toc90393362)

[8.Android绘制和屏幕刷新机制原理 18](#_Toc90393363)

[9.Choreography原理 18](#_Toc90393364)

[10.什么是双缓冲 18](#_Toc90393365)

[七.View事件分发 19](#_Toc90393366)

[1.View事件分发机制 19](#_Toc90393367)

[2.view的onTouchEvent，OnClickListerner和OnTouchListener的onTouch方法 三者优先级 19](#_Toc90393368)

[3.onTouch 和onTouchEvent 的区别 19](#_Toc90393369)

[4.ACTION\_CANCEL什么时候触发 19](#_Toc90393370)

[5.事件是先到DecorView还是先到Window 19](#_Toc90393371)

[6.点击事件被拦截，但是想传到下面的View，如何操作 19](#_Toc90393372)

[7.如何解决View的事件冲突 19](#_Toc90393373)

[8.在 ViewGroup 中的 onTouchEvent 中消费 ACTION\_DOWN 事件，ACTION\_UP事件是怎么传递 20](#_Toc90393374)

[9.Activity ViewGroup和View都不消费ACTION\_DOWN,那么ACTION\_UP事件是怎么传递的 20](#_Toc90393375)

[10.同时对父 View 和子 View 设置点击方法，优先响应哪个 20](#_Toc90393376)

[11.requestDisallowInterceptTouchEvent的调用时机 20](#_Toc90393377)

[八.RecycleView 20](#_Toc90393378)

[1.RecyclerView的多级缓存机制,每一级缓存具体作用是什么,分别在什么场景下会用到哪些缓存 20](#_Toc90393379)

[2.RecyclerView的滑动回收复用机制 20](#_Toc90393380)

[3.RecyclerView的刷新回收复用机制 20](#_Toc90393381)

[4.RecyclerView 为什么要预布局 21](#_Toc90393382)

[5.ListView 与 RecyclerView区别 21](#_Toc90393383)

[6.RecyclerView性能优化 21](#_Toc90393384)

[九.Viewpager&Fragment 21](#_Toc90393385)

[1.Fragment的生命周期 & 结合Activity的生命周期 21](#_Toc90393386)

[2.Activity和Fragment的通信方式， Fragment之间如何进行通信 21](#_Toc90393387)

[3.为什么使用Fragment.setArguments(Bundle)传递参数 21](#_Toc90393388)

[4.FragmentPageAdapter和FragmentStatePageAdapter区别及使用场景 21](#_Toc90393389)

[5.Fragment懒加载 22](#_Toc90393390)

[6.ViewPager2与ViewPager区别 22](#_Toc90393391)

[7.Fragment嵌套问题 22](#_Toc90393392)

[十.WebView 22](#_Toc90393393)

[1.如何提高WebView加载速度 22](#_Toc90393394)

[2.WebView与 js的交互 22](#_Toc90393395)

[3.WebView的漏洞 22](#_Toc90393396)

[4.JsBridge原理 22](#_Toc90393397)

[十一.动画 23](#_Toc90393398)

[1.动画的类型 23](#_Toc90393399)

[2.补间动画和属性动画的区别 23](#_Toc90393400)

[3.ObjectAnimator，ValueAnimator及其区别 23](#_Toc90393401)

[4.TimeInterpolator插值器，自定义插值器 23](#_Toc90393402)

[5.TypeEvaluator估值器 23](#_Toc90393403)

[十二.Bitmap 23](#_Toc90393404)

[1.Bitmap 内存占用的计算 23](#_Toc90393405)

[2.getByteCount() & getAllocationByteCount()的区别 23](#_Toc90393406)

[3.Bitmap的压缩方式 24](#_Toc90393407)

[4.LruCache & DiskLruCache原理 24](#_Toc90393408)

[5.如何设计一个图片加载库 24](#_Toc90393409)

[6.有一张非常大的图片,如何去加载这张大图片 24](#_Toc90393410)

[7.如果把drawable-xxhdpi下的图片移动到drawable-xhdpi下，图片内存是如何变的。 24](#_Toc90393411)

[8.如果在hdpi、xxhdpi下放置了图片，加载的优先级。如果是400800，10801920，加载的优先级。 24](#_Toc90393412)

[十三.mvc&mvp&mvvm 25](#_Toc90393413)

[1.MVC及其优缺点 25](#_Toc90393414)

[2.MVP及其优缺点 25](#_Toc90393415)

[3.MVVM及其优缺点 25](#_Toc90393416)

[4.MVP如何管理Presenter的生命周期，何时取消网络请求 25](#_Toc90393417)

[十四.Binder 25](#_Toc90393418)

[1.Android中进程和线程的关系,区别 25](#_Toc90393419)

[2.为何需要进行IPC,多进程通信可能会出现什么问题 25](#_Toc90393420)

[3.Android中IPC方式有几种、各种方式优缺点 25](#_Toc90393421)

[4.为何新增Binder来作为主要的IPC方式 26](#_Toc90393422)

[5.什么是Binder 26](#_Toc90393423)

[6.Binder的原理 26](#_Toc90393424)

[7.Binder Driver 如何在内核空间中做到一次拷贝的？ 26](#_Toc90393425)

[7.使用Binder进行数据传输的具体过程 26](#_Toc90393426)

[8.Binder框架中ServiceManager的作用 26](#_Toc90393427)

[十五.内存泄漏&内存溢出 26](#_Toc90393428)

[1.什么是OOM & 什么是内存泄漏以及原因 26](#_Toc90393429)

[2.Thread是如何造成内存泄露的，如何解决？ 27](#_Toc90393430)

[3.Handler导致的内存泄露的原因以及如何解决 27](#_Toc90393431)

[4.如何加载Bitmap防止内存溢出 27](#_Toc90393432)

[5.MVP中如何处理Presenter层以防止内存泄漏的 27](#_Toc90393433)

[十六.性能优化 27](#_Toc90393434)

[1.内存优化 27](#_Toc90393435)

[2.启动优化 27](#_Toc90393436)

[3.布局加载和绘制优化 27](#_Toc90393437)

[4.卡顿优化 28](#_Toc90393438)

[5.网络优化 28](#_Toc90393439)

[十七.Window&WindowManager 28](#_Toc90393440)

[1.什么是Window 28](#_Toc90393441)

[2.什么是WindowManager 28](#_Toc90393442)

[3.什么是ViewRootImpl 28](#_Toc90393443)

[4.什么是DecorView 28](#_Toc90393444)

[5.Activity，View，Window三者之间的关系 28](#_Toc90393445)

[6.DecorView什么时候被WindowManager添加到Window中 29](#_Toc90393446)

[十八.WMS 29](#_Toc90393447)

[1.什么是WMS 29](#_Toc90393448)

[2.WMS是如何管理Window的 29](#_Toc90393449)

[3.IWindowSession是什么，WindowSession的创建过程是怎样的 29](#_Toc90393450)

[4.WindowToken是什么 29](#_Toc90393451)

[5.WindowState是什么 29](#_Toc90393452)

[6.Android窗口大概分为几种？分组原理是什么 29](#_Toc90393453)

[7.Dialog的Context只能是Activity的Context，不能是Application的Context 30](#_Toc90393454)

[8.App应用程序如何与SurfaceFlinger通信的View 的绘制是如何把数据传递给 SurfaceFlinger 的 30](#_Toc90393455)

[9.共享内存的具体实现是什么 30](#_Toc90393456)

[10.relayout是如何向SurfaceFlinger申请Surface 30](#_Toc90393457)

[11.什么是Surface 30](#_Toc90393458)

[十九.AMS 30](#_Toc90393459)

[1.ActivityManagerService是什么？什么时候初始化的？有什么作用？ 30](#_Toc90393460)

[2.ActivityThread是什么?ApplicationThread是什么?他们的区别 31](#_Toc90393461)

[3.Instrumentation是什么？和ActivityThread是什么关系？ 31](#_Toc90393462)

[4.ActivityManagerService和zygote进程通信是如何实现的 31](#_Toc90393463)

[5.ActivityRecord、TaskRecord、ActivityStack，ActivityStackSupervisor，ProcessRecord 31](#_Toc90393464)

[6.ActivityManager、ActivityManagerService、ActivityManagerNative、ActivityManagerProxy的关系 31](#_Toc90393465)

[7.手写实现简化版AMS 31](#_Toc90393466)

[二十.系统启动 31](#_Toc90393467)

[1.android系统启动流程 31](#_Toc90393468)

[2.SystemServer，ServiceManager，SystemServiceManager的关系 32](#_Toc90393469)

[3.孵化应用进程这种事为什么不交给SystemServer来做，而专门设计一个Zygote 32](#_Toc90393470)

[4.Zygote的IPC通信机制为什么使用socket而不采用binder 32](#_Toc90393471)

[二十一.App启动&打包&安装 32](#_Toc90393472)

[1.应用启动流程 32](#_Toc90393473)

[2.apk组成和Android的打包流程 32](#_Toc90393474)

[3.Android的签名机制，签名如何实现的,v2相比于v1签名机制的改变 32](#_Toc90393475)

[4.APK的安装流程 32](#_Toc90393476)

[二十二.序列化 33](#_Toc90393477)

[1.什么是序列化 33](#_Toc90393478)

[2.为什么需要使用序列化和反序列化 33](#_Toc90393479)

[3.序列化的有哪些好处 33](#_Toc90393480)

[4.Serializable 和 Parcelable 的区别 33](#_Toc90393481)

[5.什么是serialVersionUID 33](#_Toc90393482)

[6.为什么还要显示指定serialVersionUID的值? 33](#_Toc90393483)

[二十三.Art & Dalvik 及其区别 34](#_Toc90393484)

[1.Art & Dalvik 及其区别 34](#_Toc90393485)

[二十四.模块化&组件化 34](#_Toc90393486)

[1.什么是模块化 34](#_Toc90393487)

[2.什么是组件化 34](#_Toc90393488)

[3.组件化优点和方案 34](#_Toc90393489)

[4.组件独立调试 34](#_Toc90393490)

[5.组件间通信 34](#_Toc90393491)

[6.Aplication动态加载 34](#_Toc90393492)

[7.ARouter原理 35](#_Toc90393493)

[二十五.热修复&插件化 35](#_Toc90393494)

[1.插件化的定义 35](#_Toc90393495)

[2.插件化的优势 35](#_Toc90393496)

[3.插件化框架对比 35](#_Toc90393497)

[4.插件化流程 35](#_Toc90393498)

[5.插件化类加载原理 35](#_Toc90393499)

[6.插件化资源加载原理 35](#_Toc90393500)

[7.插件化Activity加载原理 36](#_Toc90393501)

[8.热修复和插件化区别 36](#_Toc90393502)

[9.热修复原理 36](#_Toc90393503)

[二十六.AOP 36](#_Toc90393504)

[1.AOP是什么 36](#_Toc90393505)

[2.AOP的优点 36](#_Toc90393506)

[3.AOP的实现方式,APT,AspectJ,ASM,epic,hook 36](#_Toc90393507)

[二十七.jectpack 37](#_Toc90393508)

[1.Navigation 37](#_Toc90393509)

[2.DataBinding 37](#_Toc90393510)

[3.Viewmodel 37](#_Toc90393511)

[4.livedata 37](#_Toc90393512)

[5.liferecycle 37](#_Toc90393513)

[二十八.开源框架 37](#_Toc90393514)

[1.Okhttp源码流程,线程池 37](#_Toc90393515)

[2.Okhttp拦截器,addInterceptor 和 addNetworkdInterceptor区别 37](#_Toc90393516)

[3.Okhttp责任链模式 38](#_Toc90393517)

[4.Okhttp缓存怎么处理 38](#_Toc90393518)

[5.Okhttp连接池和socket复用 38](#_Toc90393519)

[6.Glide怎么绑定生命周期 38](#_Toc90393520)

[7.Glide缓存机制,内存缓存，磁盘缓存 38](#_Toc90393521)

[8.Glide与Picasso的区别 38](#_Toc90393522)

[9.LruCache原理 38](#_Toc90393523)

[10.Retrofit源码流程,动态代理 39](#_Toc90393524)

## 一.Activity

#### 1.Activity的启动流程

#### 2.onSaveInstanceState(),onRestoreInstanceState的调用时机

**(1) onSaveInstanceState(Bundle outState)在什么时机会被调用呢？**

答案是当activity有可能被系统回收的情况下,而且是在onStop()之前。注意是有可能，如果是已经确定会被销毁，比如用户按下了返回键，或者调用了finish()方法销毁activity，则onSaveInstanceState不会被调用。

总结下，onSaveInstanceState(Bundle outState)会在以下情况被调用：

* 当用户按下HOME键时。
* 从最近应用中选择运行其他的程序时。
* 按下电源按键（关闭屏幕显示）时。
* 从当前activity启动一个新的activity时。
* 屏幕方向切换时(无论竖屏切横屏还是横屏切竖屏都会调用)。

在前4种情况下，当前activity的生命周期为：

onPause -> onSaveInstanceState -> onStop。

**(2) onRestoreInstanceState什么时机被调用？**

onRestoreInstanceState(Bundle savedInstanceState)只有在activity确实是被系统回收，重新创建activity的情况下才会被调用。

比如第5种情况屏幕方向切换时，activity生命周期如下：

onPause -> onSaveInstanceState -> onStop -> onDestroy -> onCreate -> onStart -> onRestoreInstanceState -> onResume

在这里onRestoreInstanceState被调用，是因为屏幕切换时原来的activity确实被系统回收了，又重新创建了一个新的activity。

而按HOME键返回桌面，又马上点击应用图标回到原来页面时，activity生命周期如下：

onPause -> onSaveInstanceState -> onStop -> onRestart -> onStart -> onResume

因为activity没有被系统回收，因此onRestoreInstanceState没有被调用。

如果onRestoreInstanceState被调用了，则页面必然被回收过，则onSaveInstanceState必然被调用过。

**(3) onCreate()里也有Bundle参数，可以用来恢复数据，它和onRestoreInstanceState有什么区别？**

因为onSaveInstanceState 不一定会被调用，所以onCreate()里的Bundle参数可能为空，如果使用onCreate()来恢复数据，一定要做非空判断。

而onRestoreInstanceState的Bundle参数一定不会是空值，因为它只有在上次activity被回收了才会调用。

而且onRestoreInstanceState是在onStart()之后被调用的。有时候我们需要onCreate()中做的一些初始化完成之后再恢复数据，用onRestoreInstanceState会比较方便。下面是官方文档对onRestoreInstanceState的说明：

注意这个说明的最后一句是什么意思？

to allow subclasses to decide whether to use your default implementation.

它是说，用onRestoreInstanceState方法恢复数据，你可以决定是否在方法里调用父类的onRestoreInstanceState方法，即是否调用super.onRestoreInstanceState(savedInstanceState);

而用onCreate()恢复数据，你必须调用super.onCreate(savedInstanceState);

#### activity的启动模式和使用场景

https://blog.csdn.net/black\_bird\_cn/article/details/79764794

**(1)基本描述**

standard：标准模式：如果在mainfest中不设置就默认standard；每次启动时都会创建一个新的实例

singleTop：栈顶复用模式：**使用场景：展示推送过来的消息**，如果栈中没有activity创建并入栈，如果栈中已经有Activity且此activity在栈顶直接复用，如果栈中有Activity但是并不在栈顶新建activity并入栈，与standard相比栈顶复用可以有效减少activity重复创建对资源的消耗，但是这要根据具体情况而定，不能一概而论；

singleTask：栈内单例模式，**使用场景：程序入口等启动页面，**栈内只会有一个activity实例，栈内已存activity实例，如果在栈顶则直接复用，不在栈顶，会把该activity上所有activity实例出栈以后再复用此activity

singleInstance :堆内单例：**使用场景：完全独立的，类似闹钟的提示**，整个手机操作系统里面只有一个实例存在就是内存单例；若该实例不存在,则要启动一个新activity实例,并且会存在于一个单独的任务栈中

##### **(2)Intent中标志位设置启动模式**

启动模式可以在AndroidMainfest的xml文件中进行配置，GoogleAndroid团队同时提供另种级别更高的设置方式，即通过Intent.setFlags(int flags)设置启动模式；

FLAG\_ACTIVITY\_CLEAR\_TOP : 等同于mainfest中配置的singleTask；

FLAG\_ACTIVITY\_SINGLE\_TOP: 同样等同于mainfest中配置的singleTop;

FLAG\_ACTIVITY\_EXCLUDE\_FROM\_RECENTS: 其对应在AndroidManifest中的属性为android:excludeFromRecents=“true”,当用户按了“最近任务列表”时候,该Task不会出现在最近任务列表中，可达到隐藏应用的目的。

FLAG\_ACTIVITY\_NO\_HISTORY: 对应在AndroidManifest中的属性为:android:noHistory=“true”，这个FLAG启动的Activity，一旦退出，它不会存在于栈中。

FLAG\_ACTIVITY\_NEW\_TASK: 这个属性需要在被start的目标Activity在AndroidManifest.xml文件配置taskAffinity的值【必须和startActivity发起者Activity的包名不一样，如果是跳转另一个App的话可以taskAffinity可以省略】，则会在新标记的Affinity所存在的taskAffinity中压入这个Activity。

##### **(二)、taskAffinity属性**

taskAffinity只有和SingleTask启动模式匹配使用时，启动的Activity才会运行在名字和taskAffinity相同的任务栈中

1.如果不指定taskAffinity属性，应用程序的所有Activity都存放于默认task（single instance启动的activity除外，因为single instance启动的activity独占一个task）

2.指定taskAffinity，只有当Activity设置FLAG\_ACTIVITY\_NEW\_TASK或者在AndroidMainfest.xml启动模式设置为singleTask才起作用，否则不起作用。

### **Activity四种启动模式常见使用场景**



#### 4.Activity A跳转Activity B，再按系统返回键，生命周期执行的顺序

（1）Activity A，Activity B都是启动模式都是标准模式（standard）

A onCreate-->A onStart()-->A onResume()-->A onpasue()-->B onCreate()-->B onStart()-->B onResume()-->A onStop()-->B onPause()-->A onRestart()-->A onstart()-->A onResume()--->B onStop()-->B onDestory()

#### 5.横竖屏切换,按home键,按返回键,锁屏与解锁屏幕,跳转透明Activity界面,启动一个 Theme 为 Dialog 的 Activity，弹出Dialog时Activity的生命周期

Android4.0以前配置横竖屏切换属性是：android:configChanges="keyboardHidden|orientation”

Android4.0以后配置横竖屏切换属性是：

android:configChanges="keyboardHidden|orientation|screenSize"

1、未配置android:configChanges="keyboardHidden|orientation|screenSize"切换横竖屏生命周期

onCreate()-->onStart()-->onResume()-->onPasue()-->切换横竖屏onStop()--onSaveInstanceState()-->onDestory()-->onCreate()-->onStart()--onRestoreInstanceState()-->onResume()

2、配置android:configChanges="keyboardHidden|orientation|screenSize"切换横竖屏生命周期

onCreate()-->onStart()-->onResume()-->切换竖屏onConfigurationChanged()切换横屏onConfigurationChanged()

熄屏 onCreate()-->onStart()-->onResume()熄屏-->onPasue()-->onStop()-->onSaveInstanceState()

解锁屏幕 解锁屏幕后如果页面不可见执行的生命周期(如锁屏后网易云音乐界面在前台，解锁后可见页面也是网易云音乐)onRestart（）-->onStart()

解锁屏幕后如果页面可见执行的生命周期（解锁后可见页面是网易云音乐，右滑关闭网易云页面，app页面可见执行的生命周期）onRestart（）-->onStart()-->onResume()

按home键 执行的生命周期 onPause()-->onstop()--->onSaveInstanceState()

从最近任务选择app执行的生命周期是：onRestart()-->onstart()-->onResume()

按返回键 onPasue()-->onStop()-->onDestory()

启动一个 Theme 为 Dialog 的 Activity，弹出Dialog时Activity的生命周期

onCreate()-->onStart()-->onResume()--->启动一个 Theme 为 Dialog 的 Activity -->onPasue()按返回键--->onResume()

#### 6.onStart 和 onResume、onPause 和 onStop 的区别

onStart

onCreate或者onRestart方法执行后会调用此方法，该方法表明Activity准备展示给用户了

This is a good place to begin drawing visual elements, running animations

这是开始绘制视觉元素、运行动画的好地方

可以在这个方法里调用finish，会跳过onResume, onPause直接调用onStop

Dispatch onStart() to all fragments.

onResume

在onRestoreInstanceState, onRestart, 或者onPause之后被调用，以便页面开始和用户进行交互

这时activity已经激活并且准备接受输入内容，此时activity在栈顶并且用户可见

Build.VERSION\_CODES.Q（Android 10）之前，这也是尝试打开独占访问设备或访问单例资源的好地方（如相机设备等），Android 10以后，Android具有多个任务同时工作的情况，所以多个Activity的onResume可能同时恢复工作，但是要判断谁在栈顶可以通过onTopResumedActivityChanged(boolean)来判断

将onResume()分派fragments， 注意：为了更好的与旧版本的平台进行交互操作，再此调用fragments attached 到activity不会恢复

onPause

当用户不在主动与activity交互时调用，但是仍然在屏幕可见，与之相对应的是重新开始

当在activity A页面启动activity B时，将在A上调用此回调，在A的onPause返回之前，不会创建B，因此请确保不要在此处执行任何冗长的操作

此回调主要用于保存activity 正在编辑的任何持久状态，向用户提供“原地编辑”模型，并确保在没有足够资源启动新活动而不首先终止此活动时不会丢失任何内容。这也是一个关闭一些耗时CPU较高的工作的好地方，以便尽可能快地切换到下一个activity

Build.VERSION\_CODES.Q（Android 10）之前，这也是关闭独占访问设备或释放单例资源的好地方（如相机设备等），Android 10以后，Android具有多个任务同时工作的情况，所以多个Activity的onResume可能同时恢复工作，但是要判断谁在栈顶可以通过onTopResumedActivityChanged(boolean)来判断

如果activity在栈顶，在接收到这个调用以后，你一般将会收到onstop调用（在下一个activity已经被恢复并且显示）。然而在某些情况下，会直接回调onResume，而不会进入stopped状态。在某些情况下，当处于多窗口模式时，活动也可以处于暂停状态，用户仍然可以看到

onStop

用户不在可见时调用，方法后续可能会执行onRestart, onDestroy或者什么也没有，具体取决于后面用户的activity。这里是一个停止刷新ui、运行动画和其他视觉效果的好地方

#### 7.Activity之间传递数据的方式Intent是否有大小限制，如果传递的数据量偏大，有哪些方案

Intent 传递数据的大小是有限制的，它大概能传的数据是1M-8K，原因是Binder锁映射的内存大小就是1M-8K.一般activity间传递数据会要使用到binder，因此这个就成为了数据传递的大小的限制

##### Activity之间传递大量数据主要有如下几种方式实现：

1、使用全局单例： 你可以创建一个单例类来存储需要在多个Activity之间共享的数据

2、持久化（sqlite、file等）

3、匿名共享内存

##### 持久化数据

那就是sqlite、file等方式。将需要传递的数据写在临时文件或者数据库中，再跳转到另外一个组件的时候再去读取这些数据信息，这种处理方式会由于读写文件较为耗时导致程序运行效率较低。这种方式特点如下：

优势：

（1）应用中全部地方均可以访问

（2）即便应用被强杀也不是问题了

缺点：

（1）操做麻烦

（2）效率低下

##### ****匿名共享内存****

在跨进程传递大数据的时候，我们一般会采用binder传递数据，但是Binder只能传递1M一下的数据，所以我们需要采用其他方式完成数据的传递，这个方式就是匿名共享内存。

「**Anonymous Shared Memory 匿名共享内存**」是 Android 特有的内存共享机制，它可以将指定的物理内存分别映射到各个进程自己的虚拟地址空间中，从而便捷的实现进程间内存共享。

Android 上层提供了一些内存共享工具类，就是基于 Ashmem 来实现的，比如 MemoryFile、 SharedMemory。

#### 8.Activity的onNewIntent()方法什么时候会执行

‌**Activity的[onNewIntent()](https://www.baidu.com/s?rsv_dl=re_dqa_generate&sa=re_dqa_generate&wd=onNewIntent()&rsv_pq=afe5fe1600836d6f&oq=Activity%E7%9A%84onNewIntent()%E6%96%B9%E6%B3%95%E4%BB%80%E4%B9%88%E6%97%B6%E5%80%99%E4%BC%9A&rsv_t=821dTbAvRMaxQLJq8aaGJpIa8dNP+Wt0e9PiBF/54IRBvQYK0k/yHGDMXj6po2vvpFpF&tn=baiduhome_pg&ie=utf-8" \t "https://www.baidu.com/_blank)方法会在以下几种情况下被调用**‌：

**‌启动模式为[singleTop](https://www.baidu.com/s?rsv_dl=re_dqa_generate&sa=re_dqa_generate&wd=singleTop&rsv_pq=afe5fe1600836d6f&oq=Activity%E7%9A%84onNewIntent()%E6%96%B9%E6%B3%95%E4%BB%80%E4%B9%88%E6%97%B6%E5%80%99%E4%BC%9A&rsv_t=821dTbAvRMaxQLJq8aaGJpIa8dNP+Wt0e9PiBF/54IRBvQYK0k/yHGDMXj6po2vvpFpF&tn=baiduhome_pg&ie=utf-8" \t "https://www.baidu.com/_blank)时‌：**当Activity已经位于任务栈的顶部，并且用户尝试通过[Intent](https://www.baidu.com/s?rsv_dl=re_dqa_generate&sa=re_dqa_generate&wd=Intent&rsv_pq=afe5fe1600836d6f&oq=Activity%E7%9A%84onNewIntent()%E6%96%B9%E6%B3%95%E4%BB%80%E4%B9%88%E6%97%B6%E5%80%99%E4%BC%9A&rsv_t=821dTbAvRMaxQLJq8aaGJpIa8dNP+Wt0e9PiBF/54IRBvQYK0k/yHGDMXj6po2vvpFpF&tn=baiduhome_pg&ie=utf-8" \t "https://www.baidu.com/_blank)再次启动这个Activity时（无论是通过[startActivity()](https://www.baidu.com/s?rsv_dl=re_dqa_generate&sa=re_dqa_generate&wd=startActivity()&rsv_pq=afe5fe1600836d6f&oq=Activity%E7%9A%84onNewIntent()%E6%96%B9%E6%B3%95%E4%BB%80%E4%B9%88%E6%97%B6%E5%80%99%E4%BC%9A&rsv_t=821dTbAvRMaxQLJq8aaGJpIa8dNP+Wt0e9PiBF/54IRBvQYK0k/yHGDMXj6po2vvpFpF&tn=baiduhome_pg&ie=utf-8" \t "https://www.baidu.com/_blank)还是startActivityForResult()），onNewIntent()方法会被调用，而不是创建这个Activity的一个新实例。Intent对象会作为参数传递给onNewIntent()方法‌。

**‌使用[FLAG\_ACTIVITY\_SINGLE\_TOP标志](https://www.baidu.com/s?rsv_dl=re_dqa_generate&sa=re_dqa_generate&wd=FLAG_ACTIVITY_SINGLE_TOP%E6%A0%87%E5%BF%97&rsv_pq=afe5fe1600836d6f&oq=Activity%E7%9A%84onNewIntent()%E6%96%B9%E6%B3%95%E4%BB%80%E4%B9%88%E6%97%B6%E5%80%99%E4%BC%9A&rsv_t=821dTbAvRMaxQLJq8aaGJpIa8dNP+Wt0e9PiBF/54IRBvQYK0k/yHGDMXj6po2vvpFpF&tn=baiduhome_pg&ie=utf-8" \t "https://www.baidu.com/_blank)启动Activity时‌：**即使Activity的启动模式不是singleTop，如果在启动Activity的Intent中设置了FLAG\_ACTIVITY\_SINGLE\_TOP标志，并且这个Activity已经位于任务栈的顶部，那么同样会调用onNewIntent()方法，而不是创建新的实例‌。

‌**从通知启动Activity时**‌：如果你的应用从通知栏中的通知启动了一个Activity，onNewIntent()方法也会被调用‌。

**onNewIntent() 方法的使用场景主要集中在：**

单一活动模式：当 Activity 是单一实例时（singleTop/singleTask），新的 Intent 会调用此方法。适合需要更新数据，但不需要重新初始化 Activity 的场景，适用于首页或主菜单等需要避免多次创建实例的场景。

数据更新：比如需要根据新的 Intent 数据更新 UI 内容，避免重复初始化，onNewIntent()方法允许在不重新创建Activity的情况下更新Intent，这对于处理动态变化的用户请求非常有用。

保留状态：可以通过 onNewIntent() 把 Activity 状态与新的 Intent 进行结合，保证用户的操作连续性。

‌通知处理‌：从通知启动Activity时传递数据

## 二.Service

#### 1.service 的生命周期，两种启动方式的区别



区别：

1. 生命周期不同

**‌startService()‌：**使用该方法开启Service时，执行的生命周期方法依次为‌onCreate()‌、‌onStartCommand()‌、‌onDestroy()‌。如果Service已经在运行，再次调用startService()只会调用onStartCommand()，而不会重复调用onCreate()，如果组件通过改方法启动服务，则服务会一直运行，直到其使用 stopSelf() 自行停止运行，或由其他组件通过调用 stopService() 将其停止为止‌。

**‌bindService()‌：**使用该方法开启Service时，执行的生命周期方法依次为‌onCreate()‌、‌onBind()‌、‌onUnbind()‌、‌onDestroy()‌。如果服务已经被创建，多次调用bindService绑定服务，并不会重复调用onBind()方法，如果通过bindService()绑定Service，组件一旦被销毁，该服务也会被销。

2、停止服务的方式不同

**startService()‌：**通过调用‌stopService()‌或‌stopSelf()‌方法停止服务‌。

**‌bindService()‌：**通过调用‌unbindService()‌方法停止服务‌。

3、组件的关联不同

**‌startService()**‌：服务与开启该组件没有关联，即使开启服务的组件被销毁，服务依旧运行‌。

#### 2.Service启动流程

#### 3.Service与Activity怎么实现通信

通过Binder或者广播通信

#### 4.IntentService是什么,IntentService原理，应用场景及其与Service的区别

 IntentService是 Service 的子类，其使用工作线程逐一处理所有启动请求。如果不要求服务同时处理多个请求，此类为最佳选择。实现 onHandleIntent()，该方法会接收每个启动请求的 Intent，以便执行后台工作  
  
IntentService原理   
原理是继承Service,并使用Handler开启一个子线程处理请求

#### 5.Service 的 onStartCommand 方法有几种返回值?各代表什么意思?

* [START\_NOT\_STICKY](https://developer.android.com/reference/android/app/Service?hl=zh-cn" \l "START_NOT_STICKY)

如果系统在 onStartCommand() 返回后终止服务，系统不会自动重启该服务。

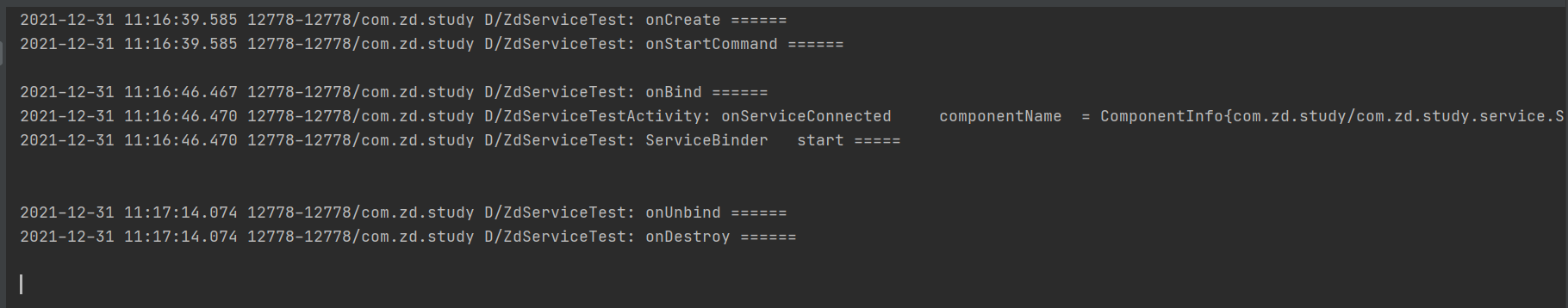
* [START\_STICKY](https://developer.android.com/reference/android/app/Service?hl=zh-cn" \l "START_STICKY)

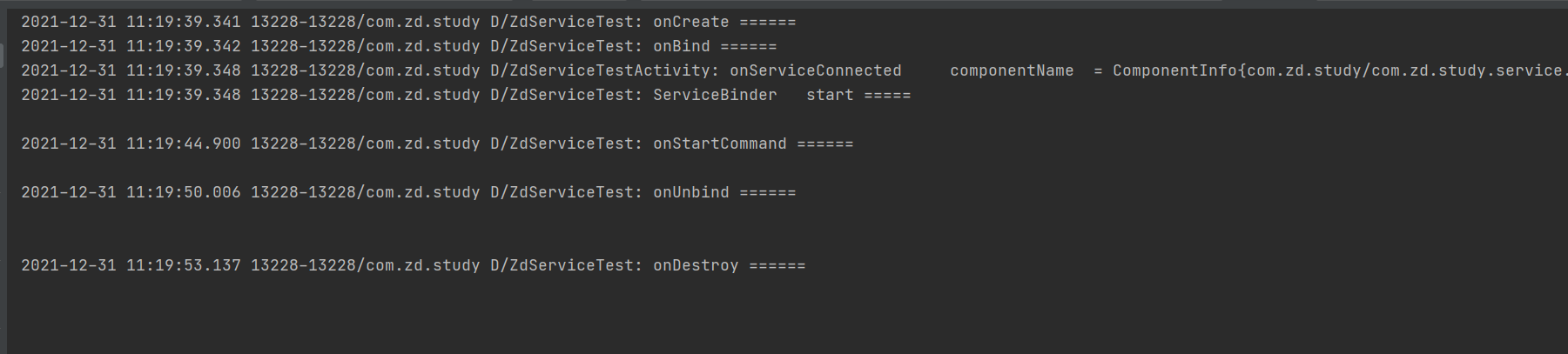
如果系统在 onStartCommand() 返回后终止服务，则其会重建服务并调用 onStartCommand()，但不保留 Intent

* [START\_REDELIVER\_INTENT](https://developer.android.com/reference/android/app/Service?hl=zh-cn" \l "START_REDELIVER_INTENT)

重传Intent。使用这个返回值时，如果在执行完onStartCommand后，服务被异常kill掉，系统会自动重启该服务，并将Intent的值传入。

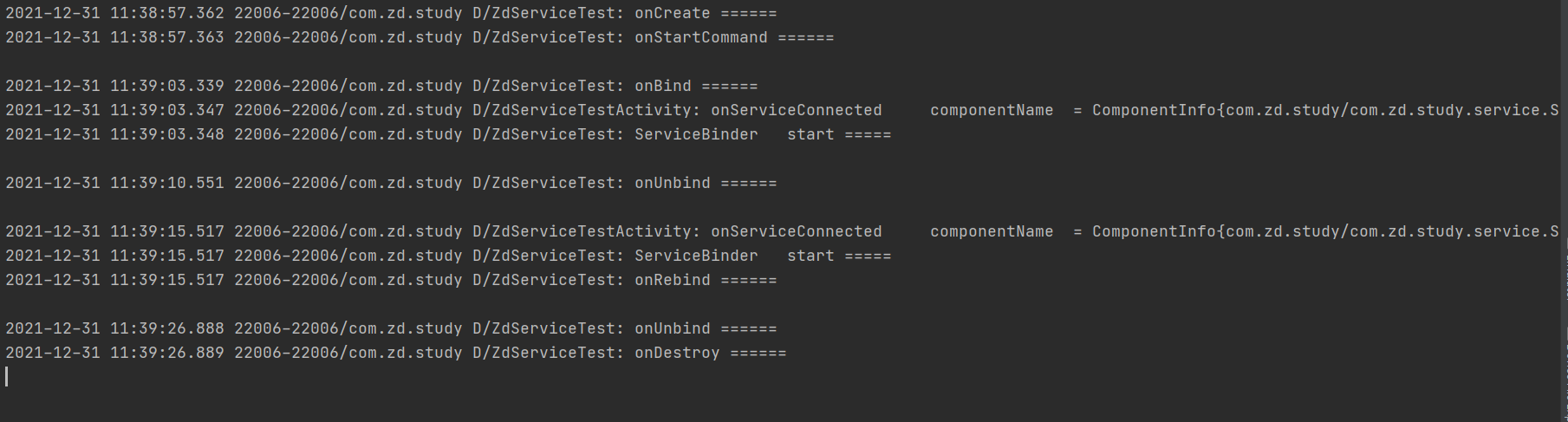
#### 6.bindService和startService混合使用的生命周期以及怎么关闭





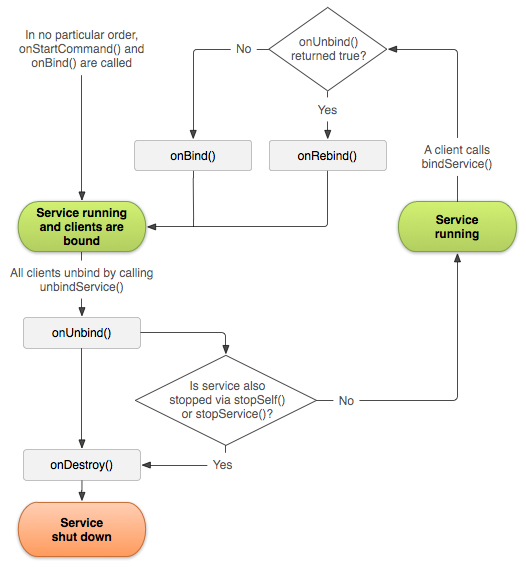
bindService和startService混合使用，无论是先启动服务还是绑定服务，stopService()或者stopSelf()只需要调一次即可，但是解绑服务需要把所有绑定在服务上的的组键都解绑，服务才会被销毁

onUnBind()返回true 混合使用生命周期如图：



onUnBind()返回true或者false只是决定重新绑定service的时候是否调用onRebind(),返回true重新绑定service的时候调用onRebind(),返回false不调用

已启动并且还允许绑定的服务的生命周期，如下所示：



## 三.BroadcastReceiver

#### 1.广播的分类和使用场景

广播的类型主要分为:

* 普通广播
* 系统广播
* 无序广播
* 有序广播
* 粘性广播
* App应用内广播
* 普通广播  
  普通广播对于多个接收者来说是完全异步的，通常每个接收者都无需等待即可以接收到广播，接收者相互之间不会有影响。对于这种广播，接收者无法终止广播，即无法阻止其他接收者的 接收动作
* 系统广播  
  Android中内置了很多系统广播：只要涉及到手机的基本操作(开机、网络变化、插入耳机等),都可以通过发送系统广播来监听变化，通过发送对应的intent-filter(包括action)  
    
  注：当使用系统广播时，只需要在注册广播接收者时定义相关的action即可，并不需要手动发送广播，当系统有相关操作时会自动进行系统广播
* 无序广播  
    
  无序广播即为我们平时经常使用的广播，其主要是通过public abstract void sendBroadcast (Intent intent)方法进行发送，并通过intent传递数据。  
    
  无序广播会被注册了的相应的感兴趣（intent-filter匹配）接收，且顺序是无序的。如果发送广播时有相应的权限要求，BroadCastReceiver如果想要接收此广播，也需要有相应的权限。  
  无序广播不可以被拦截，不可以被终止，不可以被修改，无序广播任何接收者只要匹配条件都可以接收到，无优先级问题。
* 有序广播  
    
  有序广播比较特殊，每次发送广播会先发送到优先者高的地方，然后再通过优先者高的往低的发送，优先者高的可以截断广播，那么之后的接收者就接收不到广播了，可以在广播注册时使用intent-filter里面的android: priority=”xxx”去解决或在java代码中用setPriority（xxx）来设置。
* 粘性广播  
    
  粘性消息在发送后就一直存在于系统的消息容器里面，等待对应的处理器去处理，如果暂时没有处理器处理这个消息则一直在消息容器里面处于等待状态，粘性广播的Receiver如果被销毁，那么下次重建时会自动接收到消息数据。
* App内应用广播(本地广播)  
    
  1. App应用内广播可理解为一种局部广播，广播的发送者和接收者都同属于一个App。  
  2. 相比于全局广播（普通广播），App应用内广播优势体现在：安全性高 & 效率高  
    
    
    
   具体使用1 - 将全局广播设置成局部广播  
    
  1. 注册广播时将exported属性设置为false，使得非本App内部发出的此广播不被接收；  
  2. 在广播发送和接收时，增设相应权限permission，用于权限验证；  
  3. 发送广播时指定该广播接收器所在的包名，此广播将只会发送到此包中的App内与之相匹配的有效广播接收器中。  
  通过intent.setPackage(packageName)指定包名  
    
    
    
  具体使用2 - 使用封装好的LocalBroadcastManager类  
    
  使用方式上与全局广播几乎相同，只是注册/取消注册广播接收器和发送广播时将参数的context变成了LocalBroadcastManager的单一实例  
    
  > 注：对于LocalBroadcastManager方式发送的应用内广播，只能通过LocalBroadcastManager动态注册，不能静态注册

#### 2.广播的两种注册方式的区别

动态注册和静态注册两种  
  
区别：  
动态注册是在代码中注册  
  
静态注册是在清单文件中注册  
  
在代码中进行注册后，当应用程序关闭后，就不再进行监听。我们都知道，应用程序是否省电，决定了该应用程序的受欢迎程度，所以，对于那些没必要在程序关闭后仍然进行监听的Receiver，在代码中进行注册，无疑是一个明智的选择。  
  
  
在AndroidManifest中进行注册后，不管改应用程序是否处于活动状态，都会进行监听，比如某个程序时监听 内存的使用情况的，当在手机上安装好后，不管改应用程序是处于什么状态，都会执行改监听方法中的内容。

#### 3.广播发送和接收的原理

#### 4.本地广播和全局广播的区别

由于之前的广播都是全局的，所有应用程序都可以接收到，这样就很 容易会引起安全性 的问题，比如说我们发送一些携带关键性数据的广播有可能 被其他的应用程序截获 ，或者其他的程序不停地 向我们的广播接收器里发送各种垃圾广播 。  
  
为了能够简单地解决广播的安全性问题，Android引入了一套 本地广播机制 ，使用这个机制发出的广播 只能够在应用程序的内部进行传递 ，并且 广播接收器也只能接收来自应用程序发出的广播 ，这样所有的安全性问题就都不存在了。  
  
1注册方式不同  
2安全性不同（一个内，一个外）  
   
（1）本地广播无法通过静态注册来接收，相比起系统全局广播更加高效  
  
（2）在广播中启动activity的话，需要为intent加入FLAG \_ACTIVITY \_NEW \_TASK的标记，不然会报错，因为需要一个栈来存放新打开的activity。  
  
（3）广播中弹出AlertDialog的话，需要设置对话框的类型为:TYPE \_SYSTEM \_ALERT不然是无法弹出的。

## 四.ContentProvider

#### 1.什么是ContentProvider及其使用

ContentProvider是Android四大组件之一，其底层通过Binder进行数据共享。如果我们要对第三方应用提供数据，可以考虑使用ContentProvider实现。

android中ContenProvider借助ContentResolver通过Uri与其他的ContentProvider进行匹配通信

#### 2.ContentProvider,ContentResolver,ContentObserver之间的关系

* ContentProvider：管理数据，提供数据的增删改查操作，数据源可以是数据库、文件、XML、网络等，ContentProvider为这些数据的访问提供了统一的接口，可以用来做进程间数据共享。
* ContentResolver：ContentResolver可以不同URI操作不同的ContentProvider中的数据，外部进程可以通过ContentResolver与ContentProvider进行交互。
* ContentObserver：观察ContentProvider中的数据变化，并将变化通知给外界。

#### 3.ContentProvider的实现原理

#### 4.ContentProvider的优点

透明地提供内容  
使用 ContentProvider 允许应用透明地将数据开放给其它应用，无论底层数据采用何种实现方式（网络、内存、文件或数据库），外界对于数据的访问方式都是统一的 & 固定的

#### 5.Uri 是什么

URI为系统中的每一个资源赋予一个名字，比方说通话记录。每一个ContentProvider都拥有一个公共的URI，用于表示ContentProvider所提供的数据。 Android所提供的ContentProvider都位于android.provider包中， 可以将URI分为A、B、C、D 4个部分来理解。如对于content://com.wang.provider.myprovider/tablename/id：

  a、标准前缀——content://，用来说明一个Content Provider控制这些数据；

  b、URI的标识——com.wang.provider.myprovider，用于唯一标识这个ContentProvider，外部调用者可以根据这个标识来找到它。对于第三方应用程序，为了保证URI标识的唯一性，它必须是一个完整的、小写的类名。这个标识在元素的authorities属性中说明，一般是定义该ContentProvider的包.类的名称；

  c、路径——tablename，通俗的讲就是你要操作的数据库中表的名字，或者你也可以自己定义，记得在使用的时候保持一致就可以了；

  d、记录ID——id，如果URI中包含表示需要获取的记录的ID，则返回该id对应的数据，如果没有ID，就表示返回全部；

  对于第三部分路径（path）做进一步的解释，用来表示要操作的数据，构建时应根据实际项目需求而定。如：

a、操作tablename表中id为11的记录，构建路径：/tablename/11；

b、操作tablename表中id为11的记录的name字段：tablename/11/name；

c、操作tablename表中的所有记录：/tablename；

d、操作来自文件、xml或网络等其他存储方式的数据，如要操作xml文件中tablename节点下name字段：/ tablename/name；

e、若需要将一个字符串转换成Uri，可以使用Uri类中的parse()方法，如：

Uri uri = Uri.parse("content://com.wang.provider.myprovider/tablename")；

## 五.Handler

#### 1.Handler的实现原理

handler由消息处理的五大组成部分：Message，Handler，Message Queue，Looper和ThreadLocal

**Message**:  需要传递的消息，可以传递数据；  
它的作用仅限于线程之间通信的时候传递消息，他可以携带少量数据，用于线程之间传递信息，常用的四个字段target，what，obj，arg;  
 **target**:消息回调后的作用域类，通常是一个handler。  
 **what**:是一个区分不同消息的标识符。  
 **obj**：这是obj是一个对象类型，可以携带自定义的类。  
 **arg**：int类型，携带的参数  
  
 **Handler**: 消息辅助类，主要功能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)  
**MessageQueue**: 消息队列，但是它的内部实现并不是用的队列，实际上是通过一个单链表的数据结构来维护消息列表，因为单链表在插入和删除上比较有优势。主要功能向消息池投递消息（MessageQueue.enqueueMessage）和取走消息池的消息（MessageQueue.next）  
每一个线程只有一直MessageQueue队列  
-Looper:不断循环执行（Looper.loop），从MessageQueue中读取消息，按分发机制将消息分发给目标处理者  
 每个线程中只会有一个Looper对象  
  
MessageQueue，Handler和Looper三者之间的关系：  
**每个线程中只能存在一个Looper**,Looper是保存在ThreadLocal中的。主线程（UI线程）已经创建了一个Looper,所以主线程不需要在创建Looper，但是在其他线程中需要创建Looper。**每个线程中可以有多个Handler**，即一个Looper可以处理来自多个Handler的消息。**Looper中维护一个MessageQueue**，来维护消息队列，消息队列中的Message可以来自不同的Handler.

#### 2.子线程中能不能直接new一个Handler,为什么主线程可以，主线程的Looper第一次调用loop方法,什么时候,哪个类

子线程不能直接new一个handler，要在子线程中调用Looper.prepare();以及 getLooper();方法并且将get到的looper传递到handler的构造函数中，并且调用

Looper.loop();开启循环

例如：

Looper.prepare();

Looper mLooper = getLooper();

Looper looper = Looper.myLooper();

Handler mHandler = new Handler(Looper.myLooper()) {

@Override

public void handleMessage(@NonNull Message msg) {

super.handleMessage(msg);

Log.d(TAG, "MyThread ,handleMessage " + Utils.getPid());

Log.d(TAG,"MyThread ,handleMessage == " + msg);

}

};

Looper.loop();

主线程可以直接new一个handler是因为主线程会自动创建Looper对象

主线程的Looper第一次调用loop方法是在ActivityThread.java中的main函数

#### 3.Handler导致的内存泄露原因及其解决方案

1.须知：

主线程Looper生命周期和Activity的生命周期一致。

非静态内部类，或者匿名内部类。默认持有外部类引用。

2.原因：

Handler造成内存泄露的原因。非静态内部类，或者匿名内部类。使得Handler默认持有外部类的引用。在Activity销毁时，由于Handler可能有未执行完/正在执行的Message。导致Handler持有Activity的引用。进而导致GC无法回收Activity。

3.可能造成内存泄漏

匿名内部类

非静态内部类

4.解决方法：

Activity销毁时，清空Handler中，未执行或正在执行的Callback以及Message。

静态内部类+弱引用

总结：

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。  
解决该问题的最有效的方法是： 将 Handler 定义成静态的内部类， 在内部持有  
Activity 的弱引用， 并及时移除所有消息。并且再在 Activity.onDestroy() 前移除消息mWorkHandler.removeCallbacksAndMessages(null);， 加一层保障，这样双重保障， 就能完全避免内存泄露了。  
注意： 单纯的在 onDestroy 移除消息并不保险， 因为 onDestroy 并不一定执行。  
  
主线程的 Looper 不允许退出，其实原因很简单， 主线程不允许退出， 退出就意味 APP 要挂。

#### 4.一个线程可以有几个Handler,几个Looper,几个MessageQueue对象

一个线程可以多个Handler，一个Looper,一个MessageQueue

#### 5.Message对象创建的方式有哪些 & 区别？Message.obtain()怎么维护消息池的

Message对象创建的方式有两种：

1、Message msg = new Message();

2、Message message = Message.obtain();

3、Message message = new Handler().obtainMessage();

方式1每次都会新建一个Message对象，不断的新建新对象会对内存造成压力

方式2和3可以循环使用Message，3最终调用还是回归到方式2

#### 6.Handler 有哪些发送消息的方法

post(Runnable)

postAtTime(Runnable，long)

postDelayed(Runnable long)

sendEmptyMessage(int)

sendMessage(Message)

sendMessageAtTime(Message，long)

sendMessageDelayed(Message，long)

#### 7.Handler的post与sendMessage的区别和应用场景

#### 8.handler postDealy后消息队列有什么变化，假设先 postDelay 10s, 再postDelay 1s, 怎么处理这2条消息

#### 9.MessageQueue是什么数据结构

#### 10.Handler怎么做到的一个线程对应一个Looper，如何保证只有一个MessageQueue

#### 11.ThreadLocal在Handler机制中的作用

## 六.View绘制

#### 1.View绘制流程

#### 2.MeasureSpec是什么

Measure是View绘制三个过程中的第一步，提到Measure就不得不提MeasureSpac它是一个32位int类型数值，高两位SpacMode代表测量模式，低30位SpacSize代表测量尺寸，是View的内部类

内部也包含三种测量模式：

* UNSPECIFIED ：父布局不会对子View做任何限制，例如我们常用的ScrollView就是这种测量模式。
* EXACTLY ：精确数值，比如使用了match\_parent或者xxxdp，表示父布局已经决定了子View的大小，通常在这种情况下View的尺寸就是SpacSize
* AT\_MOST ：自适应，对应wrap\_content子View可以根据内容设置自己的大小，但前提是不能超出父ViewGroup的宽高。

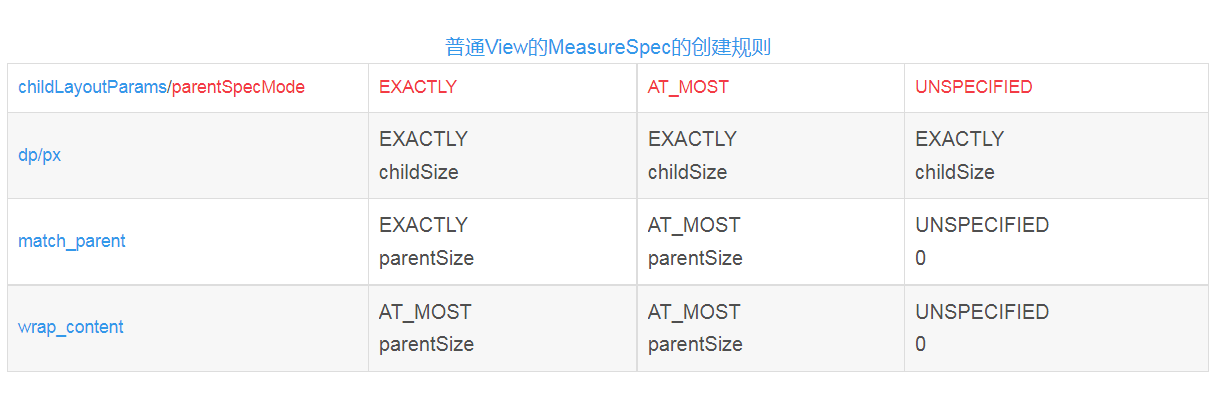
#### **注意点：**

在我们自定义View的过程中都会在onMeasure中进行宽高的测量，这个方法会从父布局中接收两个参数widthMeasureSpac和heightMeasureSpac，所以子布局的宽高大小需要受限于父布局。

#### **LayoutParams**

在刚接触Android的时候经常有一个疑问，为什么View设置自己的宽高，还要创建一个xxx.LayoutParams？前面也提到了，子View的宽高是要受限于父布局的，所以不能通过setWidth或者setHeight直接设置宽高的，另外 LayoutParams的作用不仅如此，比如一个View的父布局是RelativeLayout，可以通过设置RelativeLayout.LayoutParams的above，below等属性来调整在父布局中的位置。

#### 3.子View创建MeasureSpec创建规则是什么



#### 4.自定义View wrap\_content不起作用的原因

 public static int getDefaultSize(int size, int measureSpec) {    
  
      //参数说明：  
      // 第一个参数size：提供的默认大小  
      // 第二个参数：宽/高的测量规格（含模式 & 测量大小）  
  
       //设置默认大小  
      int result = size;   
  
      //获取宽/高测量规格的模式 & 测量大小  
      int specMode = MeasureSpec.getMode(measureSpec);    
      int specSize = MeasureSpec.getSize(measureSpec);    
  
      switch (specMode) {    
        // 模式为UNSPECIFIED时，使用提供的默认大小  
        // 即第一个参数：size   
        case MeasureSpec.UNSPECIFIED:    
            result = size;    
            break;    
        // 模式为AT\_MOST,EXACTLY时，使用View测量后的宽/高值  
        // 即measureSpec中的specSize  
        case MeasureSpec.AT\_MOST:    
        case MeasureSpec.EXACTLY:    
            result = specSize;    
            break;    
       }    
  
      //返回View的宽/高值  
      return result;    
    }  
  
从上面发现：  
  
在getDefaultSize（）的默认实现中，当View的测量模式是AT\_MOST或EXACTLY时，View的大小都会被设置成子View MeasureSpec的specSize。  
  
因为AT\_MOST对应wrap \_ content；EXACTLY对应match \_ parent，所以，默认情况下，wrap \_ content和match \_ parent是具有相同的效果的。  
  
  
由于在getDefaultSize（）的默认实现中，当View被设置成wrap \_ content和match \_ parent时，View的大小都会被设置成子View MeasureSpec的specSize。  
所以，这个问题的关键在于\*\*子View MeasureSpec的specSize的值是多少\*\*  
  
  
我们知道，子View的MeasureSpec值是根据子View的布局参数（LayoutParams）和父容器的MeasureSpec值计算得来，具体计算逻辑封装在getChildMeasureSpec()里。  
  
接下来，我们看\*\*生成子View MeasureSpec的方法:getChildMeasureSpec()的源码\*\*分析：  
  
//作用：  
/ 根据父视图的MeasureSpec & 布局参数LayoutParams，计算单个子View的MeasureSpec  
//即子view的确切大小由两方面共同决定：父view的MeasureSpec 和 子view的LayoutParams属性   
  
  
   public static int getChildMeasureSpec(int spec, int padding, int childDimension) {    
  
    //参数说明  
    \* @param spec 父view的详细测量值(MeasureSpec)   
    \* @param padding view当前尺寸的的内边距和外边距(padding,margin)   
    \* @param childDimension 子视图的布局参数（宽/高）  
  
    //父view的测量模式  
    int specMode = MeasureSpec.getMode(spec);       
  
    //父view的大小  
    int specSize = MeasureSpec.getSize(spec);       
  
    //通过父view计算出的子view = 父大小-边距（父要求的大小，但子view不一定用这个值）     
    int size = Math.max(0, specSize - padding);    
  
    //子view想要的实际大小和模式（需要计算）    
    int resultSize = 0;    
    int resultMode = 0;    
  
    //通过父view的MeasureSpec和子view的LayoutParams确定子view的大小    
  
  
    // 当父view的模式为EXACITY时，父view强加给子view确切的值  
    //一般是父view设置为match\_parent或者固定值的ViewGroup   
    switch (specMode) {    
    case MeasureSpec.EXACTLY:    
        // 当子view的LayoutParams>0，即有确切的值    
        if (childDimension >= 0) {    
            //子view大小为子自身所赋的值，模式大小为EXACTLY    
            resultSize = childDimension;    
            resultMode = MeasureSpec.EXACTLY;    
  
        // 当子view的LayoutParams为MATCH\_PARENT时(-1)    
        } else if (childDimension == LayoutParams.MATCH\_PARENT) {    
            //子view大小为父view大小，模式为EXACTLY    
            resultSize = size;    
            resultMode = MeasureSpec.EXACTLY;    
  
        // 当子view的LayoutParams为WRAP\_CONTENT时(-2)        
        } else if (childDimension == LayoutParams.WRAP\_CONTENT) {    
            //子view决定自己的大小，但最大不能超过父view，模式为AT\_MOST    
            resultSize = size;    
            resultMode = MeasureSpec.AT\_MOST;    
        }    
        break;    
  
    // 当父view的模式为AT\_MOST时，父view强加给子view一个最大的值。（一般是父view设置为wrap\_content）    
    case MeasureSpec.AT\_MOST:    
        // 道理同上    
        if (childDimension >= 0) {    
            resultSize = childDimension;    
            resultMode = MeasureSpec.EXACTLY;    
        } else if (childDimension == LayoutParams.MATCH\_PARENT) {    
            resultSize = size;    
            resultMode = MeasureSpec.AT\_MOST;    
        } else if (childDimension == LayoutParams.WRAP\_CONTENT) {    
            resultSize = size;    
            resultMode = MeasureSpec.AT\_MOST;    
        }    
        break;    
  
    // 当父view的模式为UNSPECIFIED时，父容器不对view有任何限制，要多大给多大  
    // 多见于ListView、GridView    
    case MeasureSpec.UNSPECIFIED:    
        if (childDimension >= 0) {    
            // 子view大小为子自身所赋的值    
            resultSize = childDimension;    
            resultMode = MeasureSpec.EXACTLY;    
        } else if (childDimension == LayoutParams.MATCH\_PARENT) {    
            // 因为父view为UNSPECIFIED，所以MATCH\_PARENT的话子类大小为0    
            resultSize = 0;    
            resultMode = MeasureSpec.UNSPECIFIED;    
        } else if (childDimension == LayoutParams.WRAP\_CONTENT) {    
            // 因为父view为UNSPECIFIED，所以WRAP\_CONTENT的话子类大小为0    
            resultSize = 0;    
            resultMode = MeasureSpec.UNSPECIFIED;    
        }    
        break;    
    }    
    return MeasureSpec.makeMeasureSpec(resultSize, resultMode);    
  
  
 问题总结:  
  
在onMeasure()中的getDefaultSize（）的默认实现中，当View的测量模式是AT\_MOST或EXACTLY时，View的大小都会被设置成子View MeasureSpec的specSize。  
  
  
因为AT\_MOST对应wrap\_content；EXACTLY对应match\_parent，所以，默认情况下，wrap\_content和match\_parent是具有相同的效果的。  
  
  
因为在计算子View MeasureSpec的getChildMeasureSpec()中，子View MeasureSpec在属性被设置为wrap\_content或match\_parent情况下，子View MeasureSpec的specSize被设置成parenSize = 父容器当前剩余空间大小  
  
  
所以：wrap\_content起到了和match\_parent相同的作用：等于父容器当前剩余空间大小

#### 5.在Activity中获取某个View的宽高有几种方法

#### 6.为什么onCreate获取不到View的宽高

#### 7.View#post与Handler#post的区别

#### 8.Android绘制和屏幕刷新机制原理

#### 9.Choreography原理

#### 10.什么是双缓冲

## 七.View事件分发

#### 1.View事件分发机制

#### 2.view的onTouchEvent，OnClickListerner和OnTouchListener的onTouch方法 三者优先级

#### 3.onTouch 和onTouchEvent 的区别

#### 4.ACTION\_CANCEL什么时候触发

#### 5.事件是先到DecorView还是先到Window

#### 6.点击事件被拦截，但是想传到下面的View，如何操作

#### 7.如何解决View的事件冲突

#### 8.在 ViewGroup 中的 onTouchEvent 中消费 ACTION\_DOWN 事件，ACTION\_UP事件是怎么传递

#### 9.Activity ViewGroup和View都不消费ACTION\_DOWN,那么ACTION\_UP事件是怎么传递的

#### 10.同时对父 View 和子 View 设置点击方法，优先响应哪个

#### 11.requestDisallowInterceptTouchEvent的调用时机

## 八.RecycleView

#### 1.RecyclerView的多级缓存机制,每一级缓存具体作用是什么,分别在什么场景下会用到哪些缓存

* 一级缓存：mAttachedScrap 和 mChangedScrap
* 二级缓存：mCachedViews
* 三级缓存：ViewCacheExtension
* 四级缓存：RecycledViewPool

#### 2.RecyclerView的滑动回收复用机制

#### 3.RecyclerView的刷新回收复用机制

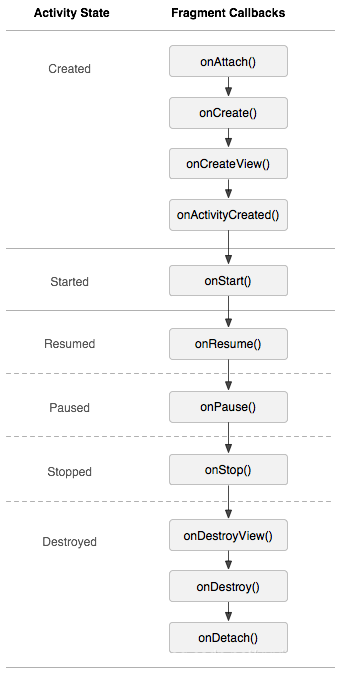
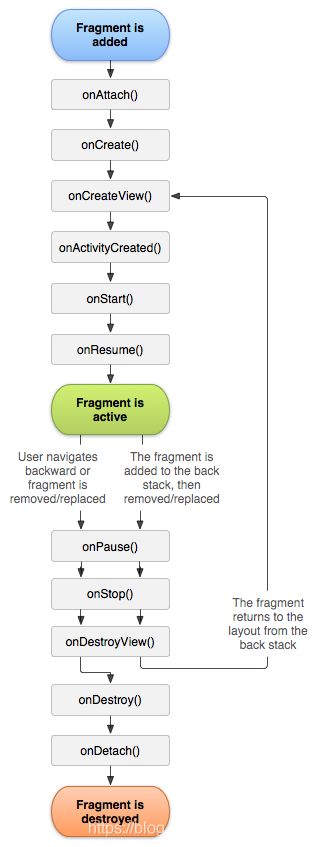
#### 4.RecyclerView 为什么要预布局

#### 5.ListView 与 RecyclerView区别

#### 6.RecyclerView性能优化

## 九.Viewpager&Fragment

#### 1.Fragment的生命周期 & 结合Activity的生命周期



onAttach() 在Fragment 和 Activity 建立关联是调用（Activity 传递到此方法内）  
onCreateView() 当Fragment 创建视图时调用  
onActivityCreated() 在相关联的 Activity 的 onCreate() 方法已返回时调用。  
onDestroyView() 当Fragment中的视图被移除时调用  
onDetach() 当Fragment 和 Activity 取消关联时调用

#### 2.Activity和Fragment的通信方式， Fragment之间如何进行通信

#### 3.为什么使用Fragment.setArguments(Bundle)传递参数

#### 4.FragmentPageAdapter和FragmentStatePageAdapter区别及使用场景

#### 5.Fragment懒加载

#### 6.ViewPager2与ViewPager区别

#### 7.Fragment嵌套问题

## 十.WebView

#### 1.如何提高WebView加载速度

#### 2.WebView与 js的交互

#### 3.WebView的漏洞

#### 4.JsBridge原理

## 十一.动画

#### 1.动画的类型

#### 2.补间动画和属性动画的区别

#### 3.ObjectAnimator，ValueAnimator及其区别

#### 4.TimeInterpolator插值器，自定义插值器

#### 5.TypeEvaluator估值器

## 十二.Bitmap

#### 1.Bitmap 内存占用的计算

#### 2.getByteCount() & getAllocationByteCount()的区别

#### 3.Bitmap的压缩方式

#### 4.LruCache & DiskLruCache原理

#### 5.如何设计一个图片加载库

#### 6.有一张非常大的图片,如何去加载这张大图片

#### 7.如果把drawable-xxhdpi下的图片移动到drawable-xhdpi下，图片内存是如何变的。

#### 8.如果在hdpi、xxhdpi下放置了图片，加载的优先级。如果是400800，10801920，加载的优先级。

## 十三.mvc&mvp&mvvm

#### 1.MVC及其优缺点

#### 2.MVP及其优缺点

#### 3.MVVM及其优缺点

#### 4.MVP如何管理Presenter的生命周期，何时取消网络请求

## 十四.Binder

#### 1.Android中进程和线程的关系,区别

#### 2.为何需要进行IPC,多进程通信可能会出现什么问题

#### 3.Android中IPC方式有几种、各种方式优缺点

#### 4.为何新增Binder来作为主要的IPC方式

#### 5.什么是Binder

#### 6.Binder的原理

#### 7.Binder Driver 如何在内核空间中做到一次拷贝的？

#### 7.使用Binder进行数据传输的具体过程

#### 8.Binder框架中ServiceManager的作用

## 十五.内存泄漏&内存溢出

#### 1.什么是OOM & 什么是内存泄漏以及原因

#### 2.Thread是如何造成内存泄露的，如何解决？

#### 3.Handler导致的内存泄露的原因以及如何解决

#### 4.如何加载Bitmap防止内存溢出

#### 5.MVP中如何处理Presenter层以防止内存泄漏的

## 十六.性能优化

#### 1.内存优化

#### 2.启动优化

#### 3.布局加载和绘制优化

#### 4.卡顿优化

#### 5.网络优化

## 十七.Window&WindowManager

#### 1.什么是Window

#### 2.什么是WindowManager

#### 3.什么是ViewRootImpl

#### 4.什么是DecorView

#### 5.Activity，View，Window三者之间的关系

#### 6.DecorView什么时候被WindowManager添加到Window中

## 十八.WMS

#### 1.什么是WMS

#### 2.WMS是如何管理Window的

#### 3.IWindowSession是什么，WindowSession的创建过程是怎样的

#### 4.WindowToken是什么

#### 5.WindowState是什么

#### 6.Android窗口大概分为几种？分组原理是什么

#### 7.Dialog的Context只能是Activity的Context，不能是Application的Context

#### 8.App应用程序如何与SurfaceFlinger通信的View 的绘制是如何把数据传递给 SurfaceFlinger 的

#### 9.共享内存的具体实现是什么

#### 10.relayout是如何向SurfaceFlinger申请Surface

#### 11.什么是Surface

## 十九.AMS

#### 1.ActivityManagerService是什么？什么时候初始化的？有什么作用？

#### 2.ActivityThread是什么?ApplicationThread是什么?他们的区别

#### 3.Instrumentation是什么？和ActivityThread是什么关系？

#### 4.ActivityManagerService和zygote进程通信是如何实现的

#### 5.ActivityRecord、TaskRecord、ActivityStack，ActivityStackSupervisor，ProcessRecord

#### 6.ActivityManager、ActivityManagerService、ActivityManagerNative、ActivityManagerProxy的关系

#### 7.手写实现简化版AMS

## 二十.系统启动

#### 1.android系统启动流程

#### 2.SystemServer，ServiceManager，SystemServiceManager的关系

#### 3.孵化应用进程这种事为什么不交给SystemServer来做，而专门设计一个Zygote

#### 4.Zygote的IPC通信机制为什么使用socket而不采用binder

## 二十一.App启动&打包&安装

#### 1.应用启动流程

#### 2.apk组成和Android的打包流程

#### 3.Android的签名机制，签名如何实现的,v2相比于v1签名机制的改变

#### 4.APK的安装流程

## 二十二.序列化

#### 1.什么是序列化

#### 2.为什么需要使用序列化和反序列化

#### 3.序列化的有哪些好处

#### 4.Serializable 和 Parcelable 的区别

#### 5.什么是serialVersionUID

#### 6.为什么还要显示指定serialVersionUID的值?

## 二十三.Art & Dalvik 及其区别

#### 1.Art & Dalvik 及其区别

## 二十四.模块化&组件化

#### 1.什么是模块化

#### 2.什么是组件化

#### 3.组件化优点和方案

#### 4.组件独立调试

#### 5.组件间通信

#### 6.Aplication动态加载

#### 7.ARouter原理

## 二十五.热修复&插件化

#### 1.插件化的定义

#### 2.插件化的优势

#### 3.插件化框架对比

#### 4.插件化流程

#### 5.插件化类加载原理

#### 6.插件化资源加载原理

#### 7.插件化Activity加载原理

#### 8.热修复和插件化区别

#### 9.热修复原理

## 二十六.AOP

#### 1.AOP是什么

#### 2.AOP的优点

#### 3.AOP的实现方式,APT,AspectJ,ASM,epic,hook

## 二十七.jectpack

#### 1.Navigation

#### 2.DataBinding

#### 3.Viewmodel

#### 4.livedata

#### 5.liferecycle

## 二十八.开源框架

#### 1.Okhttp源码流程,线程池

#### 2.Okhttp拦截器,addInterceptor 和 addNetworkdInterceptor区别

#### 3.Okhttp责任链模式

#### 4.Okhttp缓存怎么处理

#### 5.Okhttp连接池和socket复用

#### 6.Glide怎么绑定生命周期

#### 7.Glide缓存机制,内存缓存，磁盘缓存

#### 8.Glide与Picasso的区别

#### 9.LruCache原理

#### 10.Retrofit源码流程,动态代理