目录

[一. 基础知识](#_Toc440076609)

[一.1. Activity](#_Toc433887195)

[一.1.1. activity生命周期](#_Toc1635104800)

[一.1.2. activity启动模式](#_Toc2005626588)

[一.2. 广播](#_Toc1662741204)

[一.3. Service](#_Toc486717217)

[一.3.1. service启动](#_Toc491054696)

[一.4. 动画](#_Toc376620251)

[一.5. binder机制](#_Toc1222250848)

[一.6. View绘制流程](#_Toc1688918781)

[二. 技术原理](#_Toc219106221)

[二.1. 事件分发机制](#_Toc1731285389)

[二.2. android多线程模型](#_Toc1457599720)

[二.3. view和SerfaceView](#_Toc1532532711)

[二.4. WindowManager](#_Toc358411659)

[二.5. 组件化](#_Toc133122978)

[二.6. Mvp、mvc、mvvm](#_Toc1867414719)

[三. 技术实现](#_Toc165681328)

[四. 源码](#_Toc1467273184)

[四.1. Glide](#_Toc905684987)

[四.2. Handler](#_Toc483486573)

[四.3. Okhttp](#_Toc2028195810)

[四.4. Retrofit](#_Toc879049839)

[五. 技巧](#_Toc1650636360)

[五.1. 如何保证service不被杀死](#_Toc1051550574)

[五.2. 内存优化](#_Toc1767566055)

[五.3. 如何避免oom](#_Toc1341397434)

[五.4. 屏幕适配](#_Toc583347032)

[五.5. 性能优化1](#_Toc1050718269)

[五.6. 性能优化2](#_Toc663917802)

[五.7. 热更新热修复](#_Toc141468402)

[五.8. UI优化](#_Toc395035185)

[五.9. 进程保活](#_Toc1484401418)

[六. AndroidStudio](#_Toc1017105127)

[六.1. build.gradle中的各种版本](#_Toc516039369)

[七. 网络](#_Toc1534708197)

[七.1. http](#_Toc414582862)

[七.2. http和https的区别](#_Toc1457210766)

[八. 集合](#_Toc1437833774)

[八.1. SparseArray](#_Toc38759927)

[九. Other](#_Toc750548048)

[九.1. android图片格式](#_Toc142100258)

[九.2. android系统框架](#_Toc277220742)

[九.3. android线程](#_Toc1356980451)

[九.4. android进程](#_Toc494108817)

[九.5. Mipmap](#_Toc167624370)

[九.6. LRU](#_Toc1911725373)

[九.7. Davik和Linux](#_Toc1865501244)

[九.8. Cpu](#_Toc218161708)

[九.9. 签名加密](#_Toc889240927)

[九.9.1. 对称加密](#_Toc1133560616)

[九.9.2. 非对称加密](#_Toc1425840575)

[九.10. 关于对称加密和非对称加密的形象描述](#_Toc332527152)

[十. 集合](#_Toc1031394170)

[十.1. ArrayList](#_Toc153816606)

[十.2. ArrayList与LinkedList的区别](#_Toc1772869701)

[十.3. Arrays](#_Toc285462582)

[十.4. Collections](#_Toc291148276)

[十.5. HashMap](#_Toc1361326866)

[十.6. HashSet](#_Toc529861724)

[十.7. HashTable](#_Toc1918794806)

[十.8. LinkedHashMap](#_Toc422377443)

[十.9. LinkedList](#_Toc1464231166)

[十.10. 队列](#_Toc1318095989)

[十.11. BlockQueue阻塞队列](#_Toc1945468318)

[十.12. PriorityQueue优先级队列](#_Toc2047495051)

[十一. Kotlin](#_Toc971362629)

[十一.1. 常用语法](#_Toc521021109)

[十一.2. Kotlin的优缺点](#_Toc1510950144)

[十一.3. 总览](#_Toc544944433)

# 基础知识

## Activity的介绍

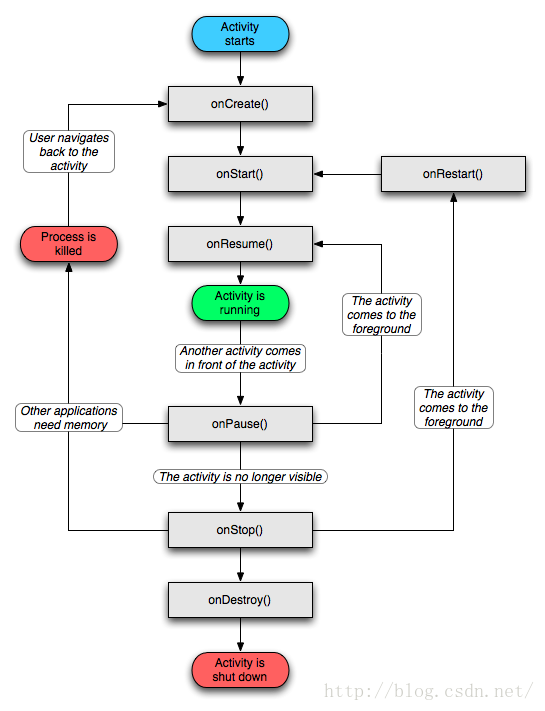
### Activity的几种状态

### android系统通过Activity栈的方式管理Activity，Activity自己通过生命周期的方式管理创建和销毁

* activity的状态
  + Running：活跃状态，activity处于栈顶，可见状态，可以与用户交互
  + Paused：暂停状态：当activity失去焦点、被一个非全屏activity覆盖或者 透明activity覆盖的时候会转化为Paused状态，此时Activity不能与用户交互
  + Stopped：停止状态:当一个Activity被另外一个Activity完全覆盖的时候会进入Stopped状态，
  + Killed:当Activity被系统销毁的时候就会进入Killed状态

## activity生命周期

### 生命周期流程图



### 生命周期简介

* onCreate 初始化Activity，加载布局加载所 需要的数据
* onStart该方法表示activity正在被启动，此时activity处于可见状态（看不见布局），但是不能与用户进行交互
* onRestart 表示activity正在重新启动，且activity正在启动，但是没有在前台展示，不可以交互
* onResume 可以与用户进行交互（界面可以看到了）
* onPause 这个方法表示activity正在被暂停，如果这个时候用户快速回调当前activity那么onResume会被调用，在onpause我们可以做一些存储数据、停止动画、资源回收，但是不能太耗时，只有onPause执行完新的activity的onResume才会被执行
* onStop：这个方法一般在onPause方法执行完成执行，表示activity被停止，此时activity是不可见的在后台运行，onStop也可以做一些资源释放的操作，内存紧张的时候activity被回收onStop可能不回被执行（不是一定不执行）
* onDestroy表示activity即将被销毁，可以做一些回收操作和资源释放

### 生命周期的调用顺序

参考：<https://www.jianshu.com/p/96fa44c550eb>

* 启动activity onCreate=>onStart=>onResume
* 按home键回到桌面 onPause=>onStop
* 从桌面回到activity或者解锁 onRestart=>onStart=>onResume
* 从A页面调到B页面
  + A页面 onPause=>onStop
  + B页面 onCreate=>onStart=>onResume
  + 全部 A\_onPause => B\_onCreate=>B\_onStart=>B\_onResume=>A\_onStop
* 返回上一个activity
  + B页面 onPause=>onStop=>onDestory
  + A页面 onRestart=>onStart=>onResume
  + 全部 B\_onPause=>A\_onRestart=>A\_onStart=>A\_onResume=>B\_onStop=>B\_onDestory
* 退出activity onPause=>onStop=>onDestory

### 关于生命周期需要注意的事

参考：<https://www.jianshu.com/p/8de5562b65fe>

* 如果所有操作都在onCreate中进行有什么问题
  + onCreate调用的时候视图还不存在，所有如果启动动画是不合理的
* 如果所有操作都在onStart中有什么问题
  + 首先 所有操作都在onStart中，activity也是可以正常工作的，但是onCreate中的注释明确表明setContentView和findViewById等操作要放在onCreate中更好
* 打开相机的操作最好放在onResume，关闭相机的操作放在onPause
* **为什么activity跳转的时候A页面的onPause先被调用，然后B页面进行onCreate=>onStart=>onResume 最后调用A页面的onStop（下面有两个回答，意思一样）**
  + 答 当跳转新的activity的时候我们肯定希望能尽快进入新的地图进行操作，当A跳转到B的时候现在A的onPause里面关闭动画、关闭相机等独占设备，防止出现B也需要这些资源A却没有释放的情况发生，A中的资源释放也可以也可让B的初始化更加流畅
  + 文章中回答==当AActivity中比较消耗资源的部分在onPause()中关闭后，再执行BActivity的初始化，显示视图与用户交互。然后，系统在后台默默执行AActivity的onStop()操作，去回收AActivity占用的其余资源。即使onStop()中会有些比较耗时的操作，也没有关系，这是在后台执行也不会影响到用户的体验。（设计的非常好!good！！！）
* 当我们再页面的EditText中输入值然后反转屏幕，EditTex中的数据会变成空，如果我们给Edittext设置一个id再去反转屏幕，EditText中的值会被保留
* 关于onSaveInstanceState和onRestoreInstanceState
  + onSaveInstanceState会在onPause或者onStop之前执行
  + onRestoreInstanceState会在onStart和onResume之间执行
* activity保存数据有两种方式 onPause和onSaveInstanceState,恢复数据也有两种方式onCreate和onRestoreInstanceState 默认情况下onSaveInstanceState和onRestoreInstanceState对UI进行保存和恢复（控件必须设置id），如果保存的数据是持久化等耗时操作要在onPause中进行
* onStart和onStop是从activity是否可见的角度来回调的，onResume和onPause是从activity能否交互的角度进行回调的，onPause和onResume都不能执行耗时操作
* activity异常状态会调用onSaveInstanceState来保存状态，onSaveInstanceState会在onStop之前调用，可能在onPause之前调用也可能在onPause之后调用
* 异常状态下退出activity再重新启动后，onCreate和onRestoreSaveInstance两个方法的参数Bundle都不为空，但是官方推荐我们使用onRestoreSaveInstance来进行恢复数据
* 当我们旋转屏幕的时候我们不想重新创建activity，我们需要在manifest中使用属性**android:configChanges="orientation"**，这样旋转后就不会创建新的activity不会走onCreate方法
* 使用startActivityForResult方法从A启动B，再返回的生命周期（这个知识点目前自己还没实践过，后续需要实践验证）
  + 参考：<https://blog.csdn.net/cbzcbzcbzcbz/article/details/76060986>
  + 从A使用startActivityForResult跳转到B，再从B返回A时候的生命周期：onActivityResult => onStart => onResume
  + 如果将B设置为singleInstance或者singleTask那么会和上一条不太一样，在调用完A的onPause后就会调用onActivityResult，从B返回后不会再次调用onActivity方法，原因分析：因为B是独自在一个任务栈里面的，当从B再次返回的时候返回的页面可能变化了不再是A（比如如下调用顺序:A=>B=>C=>B,此时签名两个B是同一个B，但是当我们返回的时候回到的页面缺失C），所以因为这种不确定性只能在启动的时候就调用onActivityResult

### android进程优先级

前台进程>可见进程>Service进程>后台进程>空进程>

* 前台进程
  + 当前activity正在与用户进行交互
  + 当前进程service正在与activity交互或者当前service调用了startForround属于前台进程
  + 当前进程持有一个Receiver，这个receiver正在执行onReceive方法
* 可见进程
  + 进程持有一个activity，这个activity不在前台，处于onPause下面，
  + 进程持有一个Service，这个Service与一个可见进程的Activity进行绑定
* 后台进程
  + Activity的onStop被调用，但是onDestroy还没有被调用，该进程属于后台进程
* 空进程
  + 进程没有任何可运行的数据了，且保留内存空间，并且没有被系统killed，这种进程属于空进程，空进程非常容易被系统杀死

## activity启动模式

### affinity

affinity是activity的一个属性，默认情况下所有的activity有相同的属性，具有相同affinity属性的activity在同一个任务栈中，

可以在manifest的application和activity标签进行设置android:taskAffinity="gf.zy"

application默认的affinity属性是manifest中的包名

### 启动模式4种

* standard
* singleTop：栈顶复用模式，如果栈顶的activity就是我们要启动的activity那么会复用，否则不会
* singleTask：栈内复用模式，将栈内该activity上面的所以activity全部弹出栈
* singleInstance：单例模式，单独存在一个任务栈中，新的任务栈中有切只有一个activity，

## 广播

### 广播种类

广播接收器中不可以进行耗时逻辑，不要

* 无序广播

通过sendBroadCastReceiver(intent)发送，多要接受者基本同一时间接收到

类似新闻联播，不管有没有看，都会发送

特点：1、无法终止广播 2、无法修改数据

* 无序广播

类似于中央红头文件，一级一级的下发

通过sendOrderedBroadCastReceiver(intent)发送（前一个接受者执行完逻辑后后一个接受者才能开始受到），如果前面的广播接收器结束了广播，那么后面的无法接收（可以通过abortBroadCast结束广播）

特点：1、可以终止广播 2、可以修改数据

* 本地广播

本地广播不会离开我们的程序，因此不比担心内存泄露，其它的程序也无法发送广播到我们的程序中，因此本地广播更安全，本地广播比系统全局广播效率更高

## Service

### Service生命周期

* onCreate首次调用会回调此方法，只会调用一次
* onStartCommand通过startService方法启动的时候，将会调用此方法
* onDestroy销毁的时候会回调此方法
* 当另一个组件通过bindService方法与服务绑定的时候，将会回调此方法
* onUnBind当组件与服务解绑的时候会回调此方法
* onReBind

当Activity退出的时候，Service的onUnbind()方法就会被调用，但Sercvice并不会停止，然后我们可以再进入Activity重新绑定该Service，这个时候 Service就会调用onRebind()方法。

但是onRebind()方法被调用还有个前提是先前的onUnbind()方法返回值为true，但是如果使用默认的 super.onUnbind(intent)是不行的，这时候我们要手动的使其返回true，再次绑定时onRebind()就会执行了

### 几种生命周期启动情景

* 单次：startService=>onCreate=>onStartCommand
* 多次：startService=>onCreate=>onStartCommand=>onStartCommand ……
* 停止：stopService=>onDestroy
* 绑定服务：bindService=>onCreate=>onBind
* startService和bindService一起用：

### service启动

1、服务分为本地服务（LocalService）和远程服务（RemoteService）

本地服务：依附于主线程，主进程杀死后服务便被杀死

远程服务：远程服务为独立的进程，在manifest中加入android:process，由于是独立的进程当activity所在进程被杀死的时候Service依然可以运行

2、启动方法：

startService：主要用于启动一个服务执行后台任务，结束用stopService

bindService：该方法启动的服务可以和服务进行通信，停止服务用unbindService

startService和bindService一起用：停止服务的时候同时stopServie和unbindService

3、service的生命周期方法

a、onCreate：如果service没有被创建过，那么会回调onCreate方法，否则不会回调onCreate

b、onStartCommend：每次用startService都会回调这个方法，

c、onBind：每次绑定都会调用

d、onDestory：销毁服务的时候调用

4、bindService启动的服务是典型的client-server关系，server只有一个，但是绑定者可以是多个

当activity销毁的时候activity会自动解除与server的绑定关系，也可以主动调用unbindService方法解除绑定，当没有任何activity与service绑定的时候service会销毁

bindService后发生的事情：

a、如果Service不存在则执行onCreate onBind

b、client实例ServiceConnection执行onServiceConnected

5、如何保证service不被杀死

a、onStartCommend中返回START\_STICKIY

b、在配置文件中android:priority = "1000"提高service的优先级到最高

c、将service设置为前台任务

d、在onDestory里面重启服务

e、监听系统广播

f、root了的手机可以将app打包成系统应用

g、联系厂商设置白名单

## 动画

1、android中有哪几种动画

a、属性动画：在动画进行过程中动态改变控件属性来达到动态效果

ObjectAnimator，ValueAnimator用于指定单个动画。

AnimatorSet，动画集合里面可以放上面两种单个动画

b、补间动画

c、帧动画：一张张图片组合起来，通过指定图片的展示顺序，达到动画展示的动态效果

2、差值器类型：

a、LinearInterpolator :线性差值器

b、AccelerateInterpolator：加速度差值器

c、DecelerateInterpolator：减速差值器

3、系统提供的五种补间动画：

a、平移动画：TranslateAnimation

b、缩放动画：ScaleAnimation

c、旋转动画：RotateAnimation：

d、透明度动画：AlphaAnimation

e：组合动画：AnimationSet

4、差值器，估值器

差值器：根据动画进度计算出当前属性改变的百分比

估值器：设置属性值从初始值到结束值的变化具体数值

## binder机制

1、Binder是Android系统中进程间通讯（IPC）的方式，

2、Binder机制原理

a、首先需要注册服务端，只有注册了服务端客户端才有通讯目标，注册的过程就是向Binder驱动的全局链表中插入服务端的信息，然后向ServiceManager的列表中缓存注册的服务

b、有了服务端客户端就可以跟服务端通信了，通信之前先拿到服务，就是链表中的引用

c、有了服务端的引用就可以向服务端发送请求了，通过BinderProxy将我们的请求参数发送给ServiceManager，通过共享内存的方式将我们的参数复制到内核空间，然后Binder驱动向服务端插入一条事务，执行完成后把内核中的结果复制到用户的空间，唤醒等待的客户端并且把结果响应回来，这样就是一次通信

## View绘制流程

performTraversal()

performMeasure()

performLayout()

performDraw()

measure()

onMeasure()

layout()

onLayout()

draw()

onDraw()

1、绘制过程

measure：根据父view传入的MeasureSpec进行大小的计算（measure是final的，会在里面调用onMeasure）

layout：根据measure获得的布局参数和大小把view放在合适的位置上，（调用layout的时候会先判断

view是否正在执行动画，如果动画已经执行完成了则调用requestLayout，如果动画没有执行或者已经执行完成则调用layout）

draw：把view绘制到屏幕上

draw的绘制分为7步：

a、drawBackground（canvans）：绘制View的背景

b、onDraw（canvans）绘制View的内容，是空方法需要自己实现

c、dispatchDraw（canvans）：遍历子View进行绘制（仅限于ViewGroup，这个方法一般不需要复写）

d、onDrawForground（canvans）：对前景色和滚动条进行绘制

e、drawDefaultFocusHighLight（canvans）：绘制默认焦点高亮

2、一个activity包含一个Window，Window是Activity和整个View系统交互的接口，只有一个子类实现PhoneWindow。一个PhoneWindow对应一个DecorView和一个

viewRootImpl，DecorView是视图树里面的顶层布局

3、measurespeac:MeasureSpec封装了从父View传递到子View的布局需求，每个MeasureSpec代表宽度或者高度的要求，包含了size和mode （MeasureSpec的32位中，前两位代表mode后面30位代表size）

## 系统服务

## android数据库相关

## android跨进程通信

## SharePreference的存储格式和位置在哪里

## 内部存储和外部存储相关东西

## fragment的生命周期

## android手机启动过程和app启动过程

## AIDL

## 进程和application的生命周期

## 系统启动流程

## android的数字签名

# 技术原理

## 事件分发机制

dispatchTouchEvent()

onInterceptTouchEvent()

onTouchEvent()

requestDisallowInterceptTouchEvent(boolean)

1、当一个Touch事件到达了根节点它会进行下发，首先调用ViewGroup的dispatchTouchEvent方法，

viewgroup遍历它包含的子view，调用每个view的dispatchTouchEvent方法，

dispatchTouchEvent方法分发的对象就是它自己，它可以决定是否自己处理事件，处理的方法就是onTouchEvent事件

当Viewgroup的所有View都返回false的时候那么就由Viewgroup自己处理这个事件

这里的onTouchEvent仅限于down事件，如果onTouchevent返回的true那么之后的action\_move,action\_up都可以响应，否则是不会接收到后续的事件的

## android多线程模型

当我们启动一个app的时候android系统会启动一个Linux process，这个进程包含一个线程就是我们的UI Thread

可以再manifest中加入android:process指定组件运行在不同进程中，这样当一个组件启动的时候如果进程已经存在那么组件直接通过这个进程被启动起来

如果卡顿时间超过5秒就会出现anr application not response

android提供了4种常用的操作多线程的方式

1、Handler+Thread

2、AsyncTask

3、ThreadPoolExecutor

4、IntentService

1、android的主线程中包含一个消息队列MessageQueue，该队列可以存放一系列Message Runnable对象，我们可以通过Hanlder向消息队列中放入Message 或Runnable。handler会在创建的时候绑定它所在的线程（不一定非要再主线程），再出队列的时候会在Handler绑定的线程中处理Message Runnable

handler的缺点是代码冗余较多

2、AsyncTask是android提供的异步类，通过继承AsyncTask可以再doInbackground方法里面处理异步任务，然后在主线程中更新ui

优点：处理单个异步任务的能力强

可以通过cancel取消任务

3、ThreadPoolExecutor 提供了一组线程池，共四种

a、newFixedThreadPool（） 创建一个定长的线程池，每次新任务创建一个新线程直到达到线程池的长度

b、newCacheThreadPool() 创建一个可缓存的线程池，如果线程的长度超过了处理的需要会重复利用已经闲置的线程，不会对线程的长度做出限制

c、newScheduleThreadPool() 创建一个定长的线程池，支持以定时和周期的方式执行

d、newSingleThreadExecutor() 创建一个单线程化的executor，它只创建单一的工作线程来执行任务

优点：批量处理任务

4、IntentService 是一个轻量级的service，用来处理异步请求，客户端通过调用startService启动service处理完异步任务后自动关闭service

## view和SerfaceView

1、surfaceview有独立的绘图表面，因此其ui可以在一个独立的线程中绘制，由于不占用主线程资源它的绘制比较高效，因为android的绘制是一层一层的所以普通的veiw绘制效率比较低

2、android系统每隔16ms会通知view绘制，在一些页面如果执行很多刷新逻辑导致刷新时间超过了16ms就会导致丢贞或者卡顿，很容易导致anr问题

3、SurfaceView继承自View，Surfaceview有两个子类GLSurfaceview和VideoView

4、在SurfaceView中可以通过获得SurfaceHolder操作SurfaceView，绘制的时候就是调动Canvas绘制

SurfaceView通过继承SurfaceHolder接受生命周期的回调，三个生命周期分别是：

surfaceCreated(holder)

surfaceChanged(holder,format,width,height)

surfaceDestroyed(holder)

这三个生命周期都是在主线程回调的

5、SurfaceView的双缓冲：

SurfaceView更新视图的时候使用双缓冲，frontCanvans和backFrontCanvans，每次显示的都是frontCanvans，backCanvans显示的是上一次的视图，当使用lockCanvans（）获取画布的时候实际上获取的是bacCanvans，当绘制完成的时候再调用unlockCanvansAndPost（canvans）比较视图，这样把正在展示的视图变换为backCanvans

这样做的好处是绘制期间不会出现黑屏的情况

## WindowManager

1、Window是通过WindowManager来管理的

2、Android系统中有很多服务，这些服务在app执行的时候会将所有的服务注册到一个Map中，等我们想用某个服务的时候可以通过Context.getSystemService来获取服务

3、WindowManager主要是管理Window的一个接口类，主要定义了一些对View的操作和一些常量（例如添加View，移除View，删除View），WMS是WindowManagerServer的缩写，主要是连接WindowManager与Window之间的Binder通信，即WindowManager通过WMS操作Window

4、Window的类型大致包括三类：应用程序窗口，子窗口，系统窗口

a、应用程序窗口：对应一个activity，

b、子窗口：不能独立的存在必须依附于父窗口，例如dialog

c、系统窗口：系统窗口等级较高，例如Toast，输入法，状态栏，系统错误提示窗

## 组件化

1、Window是通过WindowManager来管理的

2、Android系统中有很多服务，这些服务在app执行的时候会将所有的服务注册到一个Map中，等我们想用某个服务的时候可以通过Context.getSystemService来获取服务

3、WindowManager主要是管理Window的一个接口类，主要定义了一些对View的操作和一些常量（例如添加View，移除View，删除View），WMS是WindowManagerServer的缩写，主要是连接WindowManager与Window之间的Binder通信，即WindowManager通过WMS操作Window

4、Window的类型大致包括三类：应用程序窗口，子窗口，系统窗口

a、应用程序窗口：对应一个activity，

b、子窗口：不能独立的存在必须依附于父窗口，例如dialog

c、系统窗口：系统窗口等级较高，例如Toast，输入法，状态栏，系统错误提示窗

## Mvp、mvc、mvvm

本总结借鉴文章：https://blog.csdn.net/lihaoxiang123/article/details/78977181

https://blog.csdn.net/victoryzn/article/details/78392128

mvc：

model view controller

视图层（布局文件）

控制层 Activity 主要作为view和model之间的桥梁作用

model 是新建的一个类，处理数据和请求网络 activity会持有model的引用

mvp：

model view presenter

view、负责ui绘制、于用户进行交互

model、负责存储 检索 操纵数据

presenter、view和model的中间纽带 控制中心而不是单纯处理view操作的

mvp优点：

a、代码不臃肿

b、model的改动不会影响activity

## 三级缓存

## android动态加载dex

## Dalvik和ART的理解

# 技术实现

# 源码

## Glide

都有哪些图片加载库

glide fresco picasoo

为什么使用glide，其他的如何

把glide的使用和实现总结一下

glide的优势

1、glide可以从多个源加载图片，网络 本地 Uri等，并且为我们封装了很好的缓存机制能够保持一个较低的内存消耗

2、加载同样一张图片Glide比Picass占用内存小，因为Picass默认图片格式argb\_8888 glide默认 rgb\_565（glide的图片格式是可以修改的，自定义一个对象在配置文件中用

meta-data配置）

3、picass默认加载完整尺寸的图片大小，glide会根据imageveiw的大小加载图片（所以就算图片格式一样，glide依然比picass省内存）

4、glide会为不同尺寸的imageview缓存不同的图片大小（这样的坏处是占用磁盘空间较多，但是这种方式在图片已经被缓存的情况下可以加快图片的加载速度）

5、glide可以加载gif图片，picass不可以（加载出来的gif是不可以动的）

6、glide是绑定了activity和fragment的生命周期的，因此glide可以随这activity的生命周期暂停

7、glide可以配置一个展示图片的动画

8、glide可以清除磁盘缓存和内存缓存

glide源码:

with(context) 如果是在非主线程里面操作，那么会把context转成applicationContext

## Handler

自我解读handler源码：

Handler：发送消息sendMessage

Message：异步消息的内容

MessageQueue：消息队列，用于存放异步消息

Looper：维持一个无限循环，轮询消息

1、这里只讲解核心代码：

方法的调用链条：handler发送一条消息（sendMessage）->sendMessageDelayed->sendMessageAtTime->enqueueMessage->queue.enqueueMessage

enqueueMessage：这个方法中会把Handler赋值给Message的target成员变量，也就是说message是持有他handler的引用的

queue.enqueueMessage：最终就是在Messagequeue方法中把消息添加到消息队列中的。

将消息放到队列中会有两种情况：（推送的消息为msg1，当前队列第一个消息为msg2）

a、msg1.when==0或者msg1.when<msg2.when：这个方法中首先会判断推送过来的消息和当前队列第一个消息的执行时间，如果msg1.when==0或者msg1.when<msg2.when那么立即把当前消息放到队列的头部。

b、msg1.when>msg2.when ：这种情况说明消息并不是第一个要被执行的消息，应该把消息插入到队列中合适的位置。所以会遍历队列，将msg1插入到合适的位置

2、Message：Message只作为我们传递消息过程中内容的载体，在这里不做深入讨论

3、MessageQueue：handler中的核心代码同 1

4、Looper：Looper是整个机制的核心。为了便于理解这里不做过多扩展，只讲解在主线程Looper的工作方式。

Looper的准备和轮询状态是在ActivityThread类中的main方法完成的：Looper.prepareMainLooper();和Looper.loop();

先方法调用链条： Looper.prepareMainLooper->prepare(boolean)->sThreadLocal.set(new Looper(quitAllowed))->Looper.loop()

注意点：

sThreadLocal.set(new Looper(quitAllowed)) 创建一个新的Looper，这里重点关注Looper的构造方法。在Looper的构造方法中创建了一个消息队列MessageQueue并且获取了当前线程

Looper.prepareMainLooper方法中有一个同步代码块sMainLooper = myLooper();是为了避免重复创建Looper

重点说一下Looper.loop()这个方法：

在这个方法里面维持了一个死循环用于轮询消息。方法里面有个重要的方法msg.target.dispatchMessage(msg); (msg：Message rarget：Handler)这个方法用于分发消息，最终会把消息回调给Handler的Callback的handlerMessage方法（Handler本身的handlerMessage也会被回调）

## Okhttp

先把okhttp的使用总结一遍

1.okhttp源码

a、首先看两个方法 executed和equeue

executed： client.dispatcher().executed(this); Dispatcher调用execute方法直接把请求放到runningAsyncCalls队列里面

executed请求的形参是一个RealCall（）executed用于进行同步请求

equeue：Dispatcher里面还有一个enqueue方法，这个方法的形参与executed不同是AsyncCall用于进行异步请求

Dispatcher里面有两个队列runningAsyncCalls和readyAsyncCalls（前者是正在运行的请求队列，后者是阻塞的请求队列）

在enqueue方法里面先判断 runningAsyncCalls的请求数是否大于最大请求数量 是：将请求加入readyAsyncCalls执行线程池 否：将请求加入runningAsyncCalls

b、getResponseWithInterceptorChain

Response response = getResponseWithInterceptorChain(forWebSocket);

这两个方法最终的本质都是调用getResponseWithInterceptorChain方法获得响应

getResponseWithInterceptorChain方法里面会新建一个ApplicationInterceptorChain调用ApplicationInterceptorChain的proceed方法获得响应

ApplicationInterceptionChain的processd方法里面通过OkhttpClient拿到所有的拦截器查看是否有拦截器里面可以进行返回（有些情况本地有缓存可能会在这里被拦截），如果请求没有被拦截方法继续向后走

c、RealCall getResponse

最终请求获得返回是在这里完成的（这个方法里面包含给请求添加默认请求头）

这个方法里面会生成一个HttpEngine

HttpEngine包含两个重要的方法sendRequest（） ，readResponse（）前者用于生成请求，后者用于读取请求

Response通过engine.getResponse() \*\*Response response = engine.getResponse()\*\*方法得到

## Retrofit

## Handler和RxJava的区别

# 技巧

## 如何保证service不被杀死

1、Service设置成START\_STICKY kill后会被重启等待五秒左右。重新传intent和杀死前一样

2、通过startForbackground将进程设置为前台进程，做前台服务，优先级和前台应用一样，除非非常缺内存否则不会轻易被杀死

3、双进程Service互相守护

4、已经root的手机可以把app包装成系统应用（场景太少）

5、用c编写守护进程（子进程）循环查询是否service被销毁

6、联系厂商把应用加入白名单

## 内存优化

主要为了避免oom和频繁GC导致性能下降

a、流要及时关闭，包括查询ContentProvider的指针

b、加载图片的时候合理选择编码方式、使用inSampleSize

c、使用静态内部类和WeakRefreence代替内部类，主要是handler AsyncTask上面

d、使用Webview记得在onDestory中要及时移除和销毁，webview.removeAllViews() webview.destory()

e、使用LeakCanary检测内存泄露

activity5秒内没有响应出现anr broadcastreceiver 10秒 service20秒

出现ANR时，adb pull data/anr/tarces.txt 结合log分析

其他性能优化：

a、常量使用static final修饰

b、使用SplashArray代替HashMap

e、不要过度使用枚举因为枚举占用内存空间大，可以使用静态常量代替

f、数据存储使用批量插入查询，避免多次插入查询

内存泄露本质：长生命周期的对象引用短生命周期的对象导致短生命周期不能及时释放

容易引起这些的情况：

a、静态集合类引起内存泄露

b、handler引用

c、各种连接没有关闭：io socket cursor

d、java中的匿名内部类持有宿主类的引用

## 如何避免oom

## 屏幕适配

1、android屏幕分辨率碎片化：320\*480 、480\*800 720\*1280 1080\*1920

2、当android系统屏幕尺寸、屏幕密度出现碎片化的时候很容易出现同一个元素在不同手机显示不同的问题、例如：4.3寸的ui设计图运行在5.0设备上可能会右侧和下侧有很大的空白

3、像素：就是手机屏幕上的最小构成单位

4、分辨率：手机在横向、纵向上的像素点数总和

5、屏幕尺寸：手机对角线的物理尺寸

6、屏幕像素密度：每英寸的像素密度，单位dpi 标准屏幕像素密度为160dpi

密度类型：

低密度：ldpi 120

中密度：mdpi 160

高密度：hdpi 240

超高密度： xhdpi 320

超超高密度： xxhdpi 480

7、dp与px的关系：

android规定：以160dp为基准，1dp=1px。

1. sp主要用于文字大小、可以根据文字大小首选项进行缩放。谷歌推荐以偶数为单位，因为基数会影响精度

## 性能优化1

1、android屏幕分辨率碎片化：320\*480 、480\*800 720\*1280 1080\*1920

2、当android系统屏幕尺寸、屏幕密度出现碎片化的时候很容易出现同一个元素在不同手机显示不同的问题、例如：4.3寸的ui设计图运行在5.0设备上可能会右侧和下侧有很大的空白

3、像素：就是手机屏幕上的最小构成单位

4、分辨率：手机在横向、纵向上的像素点数总和

5、屏幕尺寸：手机对角线的物理尺寸

6、屏幕像素密度：每英寸的像素密度，单位dpi 标准屏幕像素密度为160dpi

密度类型：

低密度：ldpi 120

中密度：mdpi 160

高密度：hdpi 240

超高密度： xhdpi 320

超超高密度： xxhdpi 480

7、dp与px的关系：

android规定：以160dp为基准，1dp=1px。

1. sp主要用于文字大小、可以根据文字大小首选项进行缩放。谷歌推荐以偶数为单位，因为基数会影响精度

## 性能优化2

https://www.jianshu.com/p/797395731747

内存优化：

1、android中经常出现内存优化的点

a、使用单例模式

b、使用匿名内部类

c、使用异步事件handler

d、使用静态变量

e、资源未关闭

f、设置监听

g、bitmap

2、内存抖动的危害

一般是在循环中创建大量对象造成的

内存抖动可能会造成ui卡顿，因为大量的对象创建会很快消耗剩余内存，导致gc回收，gc会占用大量绘制时间导致ui卡顿

3、常用内存分析工具

Heap Snapshot

Heap Viewer

MAT

Allaction Tracking

TraceView

LeakCanary

Lint

UI渲染性能优化

\*\*android每16ms进行一次绘制，所以绘制操作的计算要在16ms内完成，如果16ms内没有完成则会掉帧，导致UI卡顿

cpu负责measure、layout等操作

gpu负责栅格化操作

在cpu方面最常见的问题是不必要的布局和失效，一般两个原因：

a、布局层次太深导致cpu计算太多

b、花费太多时间进行不必要的绘制

在gpu方面最常见的是过度绘制

优化建议

a、没有用的父布局（没有背景没有大小限制）使用merge标签合并减少ui层次

b、使用LinearLayout排版导致UI层次变深，可以使用RelativeLayout优化

c、不常用的UI可以使用ViewStub标签

GPU:

过度绘制是屏幕上某一个点被绘制了多次，在错层次的ui中如果不可见的ui也在绘制会导致某些像素区域被绘制多次。这样就会浪费大量的cpu和gpu的资源

GPU优化建议：

移除window默认的background window.setbackground(null)

移除xml中非必要的background

计算性能优化

运算性能差分为两种情况：

a、执行耗时较长的方法

b、代码执行时间短，但是执行频次高

优化建议：

a、将不需要在主线程中进行的操作放到子线程中

电量优化

耗电的操作：

a、频繁唤醒屏幕

b、网络请求、定位

电量优化建议：

a、耗时任务充电的时候执行

b、因为wifi的耗电比移动网络低，所以建议wifi连接的时候进行后台网络任务

c、大频次需要唤醒cpu的操作集中批处理（网络请求也可以批次处理）

d、定位要及时关闭

## 热更新热修复

1、android屏幕分辨率碎片化：320\*480 、480\*800 720\*1280 1080\*1920

2、当android系统屏幕尺寸、屏幕密度出现碎片化的时候很容易出现同一个元素在不同手机显示不同的问题、例如：4.3寸的ui设计图运行在5.0设备上可能会右侧和下侧有很大的空白

3、像素：就是手机屏幕上的最小构成单位

4、分辨率：手机在横向、纵向上的像素点数总和

5、屏幕尺寸：手机对角线的物理尺寸

6、屏幕像素密度：每英寸的像素密度，单位dpi 标准屏幕像素密度为160dpi

密度类型：

低密度：ldpi 120

中密度：mdpi 160

高密度：hdpi 240

超高密度： xhdpi 320

超超高密度： xxhdpi 480

7、dp与px的关系：

android规定：以160dp为基准，1dp=1px。

1. sp主要用于文字大小、可以根据文字大小首选项进行缩放。谷歌推荐以偶数为单位，因为基数会影响精度

## UI优化

a、LinearLayout是最省性能的，在布局嵌套不多的情况下应该使用LinearLayout

b、使用标签include merge ViewStub

c、减少层级嵌套，可以通过Gpu过度绘制查看

d、去除不必要的背景getWindow().setBackGroundDrawable(null)

e、文字和图片一起的时候使用drawableLeft这种方式实现

1、android 16ms进行一次绘制，所以我们在绘制的时候如果超过了16ms那么就会发生卡顿产生掉帧现象

2、过度绘制：在用户看不到的地方做了太多绘制

3、动画较多导致过多消耗cpu gpu资源

## 进程保活

常见保活方案：

1、监听广播：例（时间更新广播，开机广播，锁屏广播）

2、双进程守护，NDK实现，目前高版本已经失效

3、提高service优先级（这种方法只能缓解应用被杀死）

具体保活方案：

1、aidl方式单进程、双进程方式保活service

2、使用一像素的activity或者后台无限播放无声音频

3、监听锁屏广播，使app始终在前台

4、使用自定义锁屏页面覆盖系统锁屏页面，例如qq音乐

5、将app加入到系统白名单

app被杀死后复活：

1、推送唤醒复活 （极光友盟）

2、同派系广播互相唤醒 比如今日头条系，阿里系

这篇文章实现了上面的集中保活方式非常值得借鉴：

https://www.jianshu.com/p/b5371df6d7cb

# AndroidStudio

## build.gradle中的各种版本

compileSdkVersion：项目的编译版本告诉你用哪个sdk版本编译你的apk，仅仅是编译的时候用到而已不会打包到app中

minSdkVersion：最小的sdk版本，运行的最低要求，也是判断手机是否可以安装该apk的依据

targetSdkVersion：向前兼容，在targetSdkVerson没有更新前系统不会向前兼容，比如targetSdkVersion设置为22的时候系统不会进行运行时权限检查，如果设置为23就会有

充要条件：minSdkVersion<targetSdkVersion<compileSdkVersion

# 网络

## http

这东西简历里面你敢写就有人敢问，所有必须预习

1、http常用的方法：GET POST PUT HEAD DELETE OPTIONS

2、get是从服务器获取数据，post是向服务器传送数据（get请求将参数放在url中上传，post是将参数放在请求实体中上传）

3、get传输数据小（不能大于2kb）但是速度快，post不受限制

4、get安全优先级低，因为参数都在url上

5、get请求方式

6、请求报文：

a、请求行：包含请求方法，首部字段，http版本信息

b、请求首部字段

c、请求内容实体

7、响应报文：

a、状态行：包含http版本，状态码，状态码原因短语

b、响应首部字段

c、响应实体内容

8、常见状态码

200：成功

301：永久性重定向

400:请求报文语法有误，服务器无法识别

404：服务器无法找到对应的资源

500：服务器内部错误

9、首部字段：

请求首部字段

响应首部字段

通用首部字段

10、session cookie 的异同

a、cookie数据存放在客户端浏览器上，session数据存放在服务器上

b、cookie不是安全的，别人可以分析本地的cookie

c、session会一定时间保存在服务器上，考虑到服务器性能的话要用cookie

d、单个cookie大小不能超过4k

11、三次握手

a、客户端像服务器发送一个syn包，客户端进入syn\_send状态等待服务器确认

b、服务器收到syn包确认客户端，同时自己发送一个syn\_ask包，服务器进入sync\_recv状态

c、客户端收到syn+ack包，向服务器发送新的ack确认包，此包发送完毕通信完毕，断开连接

## http和https的区别

http：超文本传输协议：以明文方式传递数据

https：安全套接字层文本传输协议（以安全为目标的http通道），http中加入ssl层，https的安全基础是ssl，因此加密的向下内容就需要ssl

https需要到ca申请证书，一般免费证书比较少，因而需要一定费用

http使用80端口，而https使用的是443端口

https连接的时候服务器要求有公钥和签名证书，

# 集合

## SparseArray

参考：<https://blog.csdn.net/weixin_39460667/article/details/81774128>

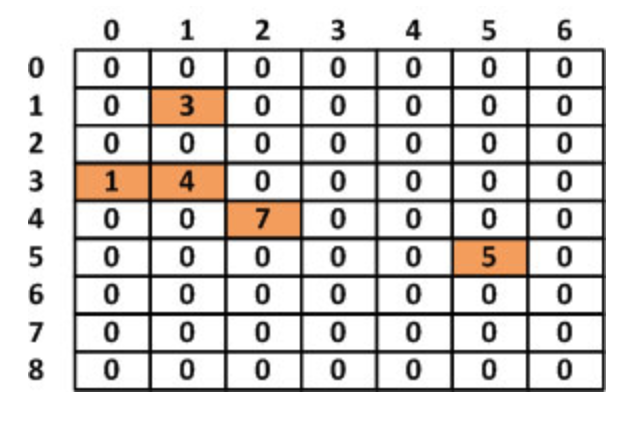
SparseArray是用来代替HashMap的，不过只能替代HashMap<Integer,Object>的类，也就是说SparseArray的key只能是int类型

SparseArray比HashMap更节省内存

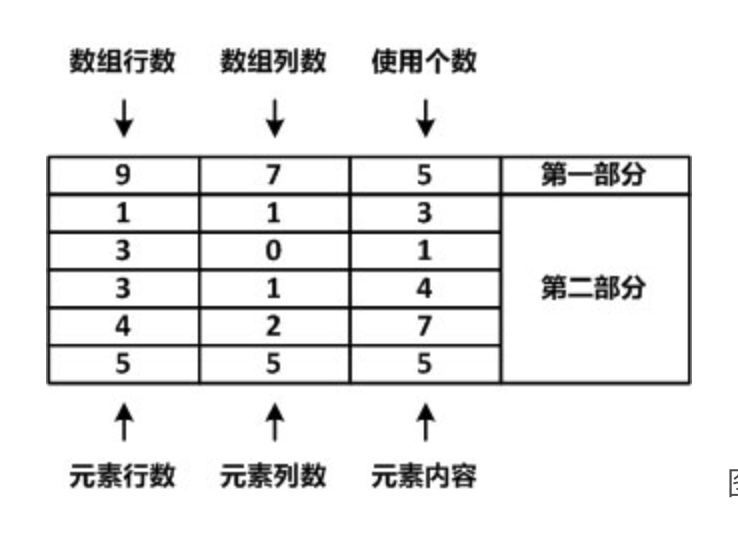
插入数据的时候SparseArray的效率低

SparseArray适合小于100条数据的情况，低于1000条数据的时候使用SparseArray最

如下一个9\*7的数组



数组里面大部分区域都是空的，这就是稀疏数组，为了节省空间我们可以将数组进行压缩，如下：



稀疏数组的第一部分记录数组的行长度和列长度，第二部分记录数据在原数组中的行号和列号已经所存放的数据

SparseArray方法：

1. put(int key,Object val)
2. Append(int key,Object value)
3. Delete(int key)
4. Remove(int value)
5. Removeat(int index)
6. Clear()

Note：

1. SparseArray可以预先设置容器大小，默认的大小是10
2. SparseArray使用二分法查找数据
3. 不仅有SparseArray，还有SparseBooleanArray、SparseIntArray
4. SparseArray结构简单避免了拆装箱（HashMap是有拆装箱的）
5. SparseArray内存比HashMap少（100000数据少了40%）
6. SparseArray的key在SparseArray是按照升序排列的，并不是插入顺序，所以需要二分法查找，还有就是如果插入数据的时候按照key进行正序插入是最快的，逆向会比较慢因为需要重新排列数组（这也证明了为什么SparseArray不适合大量数据的场景，性能消耗太大了）
7. SS

# Other

## android图片格式

alpha\_8:每个像素占用一个byte

argb\_4444:每个像素占用两个byte

argb\_8888:每个像素占用4个byte

rgb\_565:每个像素占用2byte

## android系统框架

linux kernel,lib dalvik vm ,application framework, app

android底层建立在Linux系统之上，采用一种软件堆叠的方式进行构建，这种方式使得层与层之间相互隔离明确各层分工。当下层发生改变的时候上层不需要改变

1、四个层

Application层 （应用层）

ApplicationFrameWork层 （应用程序框架层） 系统服务都位于这个层

Libraries 系统运行库层 （系统运行库层） 系统库和android运行时

Linux kernel（内核层）

Application：包含了很多应用程序 例：短信、电话、时钟、录音、图片

ApplicationFrameWork层： 主要是系统服务

Libraries层： 分为两部分 系统库和安卓运行时，Android运行时库由Android核心库集和Dalvik虚拟机两部分组成。Android核心库集，能够允许开发者使用Java语言来编写Android应用；而Dalvik虚拟机则使得每一个Android应用都能运行在独立的进程中，并且拥有一个自己的Dalvik虚拟机实例。

系统库层：是ApplicationFrameWork层的支撑，是连接应用程序框架层和内核层的重要纽带（主要包括 Surface Manager、Media FrameWork、Sqlite、OpenGl/ES、WebKit、SSL、Libc）

android运行库：android程序采用java语言编写，其运行时分为核心库和Dalvik两部分

核心库提供了java语言的大部分功能

Dalvik：每个应用程序都对应一个Dalvik虚拟机实例

内核层：主要包括：安全、内存管理、进程管理、网络协议、驱动模型

## android线程

在java中线程有五种状态：创建、就绪、运行、阻塞、死亡，当有组件运行的时候UI线程是处于运行状态的。默认状态下所有的组件都是运行在主线程的

所有如果主线程做太多耗时操作就可能会造成anr

工作线程中不能更新ui原因是，invalidate不是线程安全的

android提供了两种线程间通信方式：Handler和AsyncTask

## android进程

在android中应用程序是一个独立的进程（运行在独立进程可以避免其它应用干扰），当启动一个程序的时候系统会fork一个进程出来，并且为这个进程创建一个主线程应用程序的组件

默认都是运行在主线程中的，我们可以通过 android:process(使用了后再Logcat可以看到同一个应用可以有两个进程) 指定组件运行在一个新的进程中

每一个应用都运行在一个单独的进程中的优点是：

是给每个应用一个内存预算，这样可以给主线程更多的内存预算

## Mipmap

mipmap只是用来存放启动图标的（只要放一张图他就会自动生成各个版本的启动图）

## LRU

LruCache是将数据缓存到内存中去，

DiskLruCache是将数据缓存到外部存储中

DiskLrucache如果想使用要直接将这个类复制到项目中，sdk并没有这个类

使用DiskLruCache必须申请运行时权限

LruCache：内部是一个LinkedHashMap使用强引用的方式缓存对象，当缓存满时移除最近最少使用的对象，添加新的对象（最近最少使用）

现在非常流行的内存缓存技术是LruCache（LRU是Least Recently Used 近期最少使用算法）,这个类非常适合用来缓存图片，它的主要算法原理是把最近使用的对象用强引用存储在 LinkedHashMap 中，并且把最近最少使用的对象在缓存值达到预设定值之前从内存中移除。

DiskLruCache是不能new出实例的，如果我们要创建一个DiskLruCache的实例，则需要调用它的open()方法，接口如下所示：

public static DiskLruCache open(File directory, int appVersion, int valueCount, long maxSize)

open()方法接收四个参数，第一个参数指定的是数据的缓存地址，第二个参数指定当前应用程序的版本号，第三个参数指定同一个key可以对应多少个缓存文件，基本都是传1，第四个参数指定最多可以缓存多少字节的数据。

## Davik和Linux

delvik虚拟机基于寄存器，jvm虚拟机基于栈。基于寄存器的虚拟机编译的时候时间更短

dalvik虚拟机执行.dex字节码 jvm虚拟机执行.class字节码

1、Dalvik和Jvm的区别

相同：都是java语法

不同：字节码的格式不同 dalvik是dex Jvm是class

dalvik是基于寄存器的，jvm是给予堆栈的，前者运行程序比后者快

Dalvik可以允许多个实例运行，也就是说每个app都是独立的运行在一个VM中，这样的好处是一个app崩溃只会影响到他自己

Dalvik虚拟机特点

1、运行专有的.dex文件。专有的.dex文件减少了.class文件中的冗余信息，而且会把所有.class文件整合到一个文件中，从而提高运行性能；而且DX工具还会对.dex文件进行一些性能的优化。

2、基于寄存器实现。大多数虚拟机包括JVM都是基于栈的，而Dalvik虚拟机则是基于寄存器的。一般来说，基于寄存器的虚拟机具有更好的性能表现，但在硬件通用上略差。

## Cpu

1、线程里面有死循环，导致子线程不能休息提高了cpu占用率

2、应用处理大量图片，加载音频视频

3、socket线程长时间运转没有休息，也没有被关闭

cpu使用过大会造成什么影响

1、造成anr

2、手机死机，手机发烫

3、应用交互不流畅，卡顿

## 加密

### 对称加密

对称加密指的是加密和解密使用同一个秘钥的加密方式

工作过程：加密方使用秘钥将数据加密，解密方接收到数据后使用同样的钥匙对数据进行解密

常见的加密方式：DES、3DEX、IDEA、RC4、RC5、RC6、AES

对称加密的安全性取决于加密算法本身的破解难度和秘钥的保存方法，秘钥存放在客户端是很危险的

### 非对称加密

非对称加密算法种类：RSA、ECC（移动设备）、DSA

非对称密钥加密的使用过程：

* A要向B发送信息，A和B都要产生一对用于加密和解密的公钥和私钥。
* A的私钥保密，A的公钥告诉B；B的私钥保密，B的公钥告诉A。
* A要给B发送信息时，A用B的公钥加密信息，因为A知道B的公钥。
* A将这个消息发给B（已经用B的公钥加密消息）。
* B收到这个消息后，B用自己的私钥解密A的消息，其他所有收到这个报文的人都无法解密，因为只有B才有B的私钥。
* 反过来，B向A发送消息也是一样。

## 关于对称加密和非对称加密的形象描述

北京的Bob发了一个快递到广州的Alice，途中经过了上海，上海快递中心出现了一个黑客H，他偷偷打开了Bob给Alice的快递，然后偷偷把里边的衣服剪烂，再按照原样包装好发往广州，可以看到对于这样简单包装的传输在中途是可以偷偷修改里边的东西。

HTTP的数据包是明文传输，也即是如果中途某个黑客嗅探到这个HTTP包，他可以偷偷修改里边包的内容，至于Bob跟Alice是互相不知道这个动作的，因此我们必须要有一个方案来防止这种不安全的篡改行为，有个方法就是加密！

非对称加密

Bob将衣服放到一个保险箱里边锁起来，他打了个电话告诉Alice保险箱开柜密码是1234，而黑客H不知道密码，所以他看不到保险箱里边的东西，Alice收到快递后用预先沟通好的密码就可以打开保险箱了。

这里保护的手段就是Bob对物品进行加密，同时给了告诉Alice解密的方法！

那如果现在要求Bob的密码只能通过快递传给Alice呢？如果Bob直接传密码给Alice，H如果嗅探到这个快递，那H也知道密码了，这就无法保护快递的安全性了。因此还需要有个方案，让Bob能够告诉Alice密码的同时，H又无法查看到Bob跟Alice通信的数据。

非对称加密在这个时候就发挥作用了，来看看怎么回事：Bob拥有两把钥匙，一把叫做公钥，一把叫做私钥。公钥是公开让全社会都知道，没关系，Bob告诉所有人，你们要传递数据给我的时候请先用这个密钥(公钥)去加密一下你们的数据，加密后的数据只能通过Bob私自藏着的私钥才能解密。

回到刚刚例子，Bob先发给保险柜（Bob公钥）给Alice，接着Alice把自己的保险柜（Alice公钥）放到Bob的保险柜(即使用Bob的公钥加密Alice的公钥)里边发还给Bob，接着Bob拿到Alice的数据包后，用自己的私钥解开了外层保险柜(Bob的公钥)，拿到了里边Alice保险柜(Alice的公钥)。此时Alice跟Bob都有了各自的公钥(并且都有他们自己的私钥)，接着只要保证每次互相传递数据的时候，把数据放在对方的保险柜里边即可(即每次都用对方的公钥加密数据)，这样无论如何，H都无法解开保险柜（因为只有各自的私钥才能解开各自的保险柜）。

## 两种加密方式对比

* 对称加密使用相同 的秘钥进行加密和解密，所以速度较快
* 非对称加密使用一对秘钥（公钥和私钥）进行加密和解密，但是加密解密速度较慢

## 解决非对称加密速度慢的一种方法

将对称加密的秘钥使用非对称加密的方式进行加密，然后发送给对方，接收方使用非对称私钥解密拿到对称加密的秘钥，然后双方使用对称加密的方式进行数据传输

# 集合

## ArrayList

1、ArrayList的长度是不受限制了，可以存储任意多的对象（不能存储基本类型）

2、ArrayList中重要的两个属性是elementData（存放数组数据）和size（数组的长度）两个属性

3、ArrayList添加数据的时候主要做的是扩容校验，将插入的值放在尾部并且将size+1

4、ArrayList扩容一次容量变为之前的1.5倍

5、ArrayList访问的时间复杂度是O(1)

问题：

1、elementData前为什么用transient修饰

使用transient修饰的数据在序列化的时候会被忽略，而elementData是一个数组，数组的长度是固定的可能有很多空间是没有使用的被序列化会造成空间浪费，所以要用transient修饰

2、ArrayList的插入和删除一定慢吗

不一定：取决于插入删除的位置，如果在尾部进行插入和删除速度会比链表还快

3、ArrayList默认长度为什么是10

sun程序员定的，可能有他们的考量吧（既要考虑数据较少的情况，又要支持扩展）

4、ArrayList适合做队列吗

适合：不过需要用一个定长的ArrayList，作为数组的时候用两个偏移量来标记队头和队尾这样做效率很高（不能理解为在尾部进在头部出，不能这么实现）

5、ArrayList为什么要实现RandomAccess

RandowAccess是一个空接口，用来标记表示这个类可以用来随机访问（当程序遍历的时候发现了这个标记可以依此选择更高效的遍历方式）

6、什么情况下使用ArrayList，什么情况下使用LinkedList

如果查询较多的场景使用ArrayList，如果插入和删除场景较多使用LinkedList

注意点：

1、ArrayList扩容的代价是很高的，所以尽量提前确定好ArrayList的长度，避免发生较多扩容

2、如果插入数据到ArrayList的指定位置那么会触发数据挪动，并且可能引发扩容。如果插入到末尾，那么只可能触发扩容

3、删除指定位置元素也会触发数据挪动

## ArrayList与LinkedList的区别

ArrayList和LinkedList的异同

1、插入顺序：ArrayList快，因为ArrayList是基于数组实现的，而LinkedList需要new一个节点在进行多次引用操作

2、LinkedList占用空间较大

3、遍历效率：ArrayList的遍历效率高

## Arrays

1、数组转计划Arrays.asList(array)

2、集合转数组 list.toArray

3、排序Arrays.sort()排序

4、查找元素Arrays.binarySearch(object[],key)

5、复制数组元素Arrays.copyofRange(array,startIndex,endIndex)

6、填充元素Arrays.fill(array)

## Collections

1、一次性添加多个元素：Collections.addAll(list,"aaa","bbb","ccc");

2、排序 ： Collections.sort(list2);

3、将数组内数据乱序排列Collections.shuffle(list)

4、交换两个元素位置Collections.swap(list,index2,index3)

5、搜索Collections.binarySearch(list,key)

6、统计元素出现次数Collections.frequency(list,obj)

7、替换元素Collections.replaceAll(list,obj)

8、将普通集合包装成线程安全集合Collections.synchronizedCollection(new ArrayList())

## HashMap

note

1、HashMap是线程不安全的

2、HashMap初始容量为16，负载因子为0.75 （扩容见下面的第四条）

3、HashMap使用的是哈希表的结构，（我们会用一个hash（object）的方法生成一个hash值，并且根据hash值来计算下标决定这个数据被放在哪一个链表中）当某一个节点的链表的长度大于一定长度的时候将这个链表转换为红黑树

即：HashMap1.7之前采用 数组+链表 实现 1.8后采用 数组+链表+红黑树的方式实现 目的是为了提高查找效率

一篇hashmap的问答文章，内容非常全：https://www.jianshu.com/p/33b25964798c

1、HashMap采用Entry数组来存储key-value，Entry是一个单向的链表，具有next指针

2、数组是用来确定元素所在链表的位置的（元素下标用元素的hash值和数组长度-1 取模得到）

3、使用数组是因为数组查询速度快

4、当容量大于 容量乘以负载因子（capacity\*factor）的时候就会进行扩容（不是满了再扩容）

5、为什么扩容是2的次幂（详细原因见上面文章的 二章二节），

目的是为了减少碰撞，避免不同的hash值计算出来的数组下标是相同的情况

下面是具体原理比较粗略：

我们知道计算数组下标的原理就是 hash&length-1） ，即：hash值和数组的长度减一取模

如果数组的长度是2的次幂，那么长度减一应该是这样的

4-1=3 对应二进制 011

8-1=7 对应二进制 0111

16-1=15对应二进制 01111

所以当hash值和这样的值进行取模的时候就可以保证最大的减少出现不同的hash值计算出来的数组下标相同的情况

6、HashMap中put的过程：

a、对key的hashcode做hash运算计算出index

b、如果没有碰撞直接放在数组里（数组里面存放的也是Entry节点），如果碰撞了以链表的形式放在链表里

c、如果碰撞后长度过大就转化为红黑树

d、如果节点以及存在就替换节点数据

e、如果长度超过临界点就resize重新分配

7、hash算法的意义 ： 把一个大范围映射到一个小范围，可以节省空间容易保存

8、为什么解决hash冲突的时候不直接使用红黑树，而是长度超过8的时候再由列表转化为红黑树

因为红黑树是一种平衡树，需要不断的左旋右旋保证平衡，而链表不需要

所以如果一开始就用红黑树效率会变低

9、不用红黑树，用二叉树可以吗

可以，但是意义不大，因为二叉树不是一个平衡树（极端情况可能就是一个链表），可能并不能带了效率的提升，用二叉树还不如用链表

10、当链表转化为红黑树的时候什么时候退化为链表

链表长度为6的时候退化为链表，中间有个7是为了避免频繁的进行链表和红黑树的转化

11、HashMap在并发编程中有什么问题（可以使用ConcurrentHashMap和HashTable代替HashMap解决并发的问题）

a、多线程扩容引起死循环

b、多线程put可能导致元素丢失

12、HashMap的key可以为null吗

可以：null的时候index==0

13、一般用什么做HashMap的key

一般用Integer、String这种不可变类当HashMap当key，String最为常用

因为字符串是不可变的，创建的时候hashcode就被缓存了

14、用可变的类做HashMap的key有什么问题

hashcode值可能改变，导致put进去的值，无法get出来

15、HashMap中数据越多发生碰撞的几率就越大，所以需要进行扩容

16、如果map.get("key")得到的数据是null，那么可能存储的就是null或者对应的key不存在HashMap中

## HashSet

1、Set内部元素不重复、无序、允许null、线程不安全

2、HashSet是基于HashMap实现的，HashSet里面有一个成员变量HashMap，将HashSet的值做完HashMap的key保存，HashMap的值是同一个Object对象

3、HashMap的key值是不可以重复的，所以HashSet的元素不可以重复

## HashTable

1、HashTable不允许null值（key value都不行）

2、HashTable是同步的，因为方法前加上了synchronized

HashMap和HashTable的区别

1、HashMap是HashTable的轻量级实现，主要区别是HashMap允许空，由于非线程安全效率比HashTable高

2、HashTable继承的是Dictionary同时实现Map接口，HashMap继承的是Map的一个接口

3、单线程中使用HashMap，多线程使用HashTable

## LinkedHashMap

不错的参考文章：https://blog.csdn.net/qq\_26857649/article/details/81713420

1、LinkedHashMap是有序的（HashMap是无序的）

2、LinkedHashMap继承自HashMap并且实现了Map接口

3、LinkedHashMap是数组+双向链表实现的

4、LinkedHashMap可以按照访问顺序访问也可以按照插入顺序访问(有个accessOrder构造参数可以指定不按照访问顺序遍历，默认按照插入顺序)

5、LinkedHashMap的有序增加了时间和空间上的开销

6、和HashMap一样LinkedHashMap可以插入null的key和value

7、LinkedHashMap比HashMap多了一个双向链表的头结点（Header）和一个accessOrder属性，其它的在使用上一样

8、LinkedHashMap中的removeEldestEntry方法是用来支持LRU算法的

9、LinkedHashMap进行扩容的时候与HashMap不同，LinkedHashMap直接遍历双向的链表进行重hash，而HashMap使用顺序访问的方式进行扩容

note：

1、LinkedHashMap采用的hash与HashMap相同，但是重新定义了Entry，增加了指针before和after他们用于维护双向链表

2、相比HashMap而言，LinkedHashMap在向哈希表添加一个键值对的同时，也会将其链入到它所维护的双向链表中，以便设定迭代顺序

## LinkedList

1、LinkedList是一个双向循环链表

2、LinkedList是非同步的（线程不安全）

3、一个空的LinkedList他都前后指针节点都是自己

4、头节点是不保存数据的，里面的内容永远为空

5、LinkedList增加节点逻辑

将插入元素转换为节点、增加该节点的前后引用、增加前后节点对该节点的引用

6、LinkedList删除节点逻辑

将该节点的前后节点进行连接、该节点的前后引用设置为null

7、LinkedList的元素查找使用的是二分，但是只进行一次二分，

if (index < (size >> 1)) {

// 如果index距离list中间位置较近，则从头部向后遍历(next)

for (int i = 0; i <= index; i++)

e = e. next;

} else {

// 如果index距离list中间位置较远，则从头部向前遍历(previous)

for (int i = size; i > index; i--)

e = e. previous;

}

note：

1、链表是由一系列的非连续的节点组成的存储结构

2、单链表

每个节点的指针指向下一个节点从而链起来的结构，最后一个节点的next指向null

单向循环链表

单向循环链表的最后一个节点的next指向head节点，形成一个环

双向循环链表

第一个节点的pre指向最后一个节点，最后一个节点的next指向第一个节点

LinkedList和Array比较

1、LinkedList使用foreach方法遍历比较快，ArrayList使用最普通的for循环比较快

## 队列

1、Queue实现了Collection接口

add/offer添加元素：add队列如果满了会抛出异常、offer则返回false表示失败

remove/poll删除第一个元素：如果队列为空remove方法会抛出异常、二poll会返回null

element/peek查询头部元素：查询元素但是不移除

## BlockQueue阻塞队列

参考：https://www.jianshu.com/p/024a36b83099

方法：

1、add（obj）往队列里面增加一个对象，如果没有就抛出异常

2、offer(obj)往队列中增加一个对象，返回true/false

3、put(obj) 往队里里面增加一个对象，如果没有空间则会阻塞线程，知道有空间

4、poll（time）取出排在首位的对象，如果一定时间没有返回则会返回null

5、take() 取出队列头部的对象，如果没有就阻塞队列知道有对象

6、container（obj） 查询是否包含某个对象

7、peek返回队列头部对象，没有则返回null

特点：

1、容量有限，可以限定队列长度

2、因为集成Collection所以可以使用集合接口，但是接口不能保证立即执行，因为内部维护着锁

阻塞队列的实现类：

阻塞队列：BlockQueue

ArrayBlockQueue：一个由数组支持的有界队列

构造的时候需要制定容量，并且可以设置公平性（公平性指等待时间最长的可以优先得到处理）

LinkedBlockQueue：一个由链表支持的可选有界队列

构造的时候可以指定容量，如果不指定容量容量==Integer.MAX\_VALUE

PriorityBlockQueue：一个由优先级堆支持的无界优先级队列

优先级队列，元素按照优先级顺序进行排列，ProrityBlockQueue是没有容量限制的

DelayQueue：有个由优先级堆支持的、基于时间的调度队列

是一个存放元素的无界阻塞队列，里面的元素知道到达了期限才能被提取，如果队里里面没有到期的元素那么取元素的时候取到的是null，队列被阻塞

## PriorityQueue优先级队列

1、数据存放顺序由指定Compartor或者自然排序决定，而不是插入时候的顺序

2、PriorityQueue：不能插入null值，会抛出异常

3、PriorityQueue是线程不安全的

## Stack

1、stack的方法

* 1. push入栈
  2. pop出栈
  3. empty判断栈是否为空
  4. peek获取栈顶元素
  5. Search(obj)判断元素是否在栈中，存在返回1，不存在返回-1

1. 注意点：push会吧元素弹出栈，peek只是获取不会弹出栈
2. Stack继承Vector，并对Vector进行扩展

## TreeMap

1. TreeMap默认按照key进行排序
2. TreeMap的几个构造函数
   1. TreeMap()
   2. TreeMap(Comparator com) 指定排序方式
   3. TreeMap(map)使用指定的Map创建TreeMap
   4. TreeMap(Sortmap)使用指定的Map创建TreeMap，key顺序使用构造中Map的顺序

3、TreeMap实现了SortedMap接口，并且使用红黑树的算法实现

## HashMap、TreeMap、LinkedHashMap

HashMap可以实现快速存储和检索，但是内部元素是无序的，所以在大量迭代的时候表现不好

LinkedHashMap保存的HashMap的优点，并且使用链表的方式保证了元素的有序性，在大量迭代的情况下表现较好（可以直接通过链表遍历）

TreeMap可以快速的实现对内部元素的排序，但是性能较差

## TreeSet

1. TreeSet有序并且没有重复元素
2. TreeSet是基于TreeMap实现的，
3. TreeSet可以有两种排序方式
4. 自然排序
5. 根据提供的Comparator进行排序
6. TreeSet依靠TreeMap实现，TreeSet内部有一个（private transient NavigableMap<E,Object> m;）的全局变量，这个全局变量是一个TreeMap的实现，当我们调用add方法的时候，会调用m.put(e,PRESENT),e就是要插入TreeMap的key，PERSENT是一个Object对象实例

## Vector

1. Vector可实现自动增长的对象数组
2. Vector实现了List接口，
3. Vector与ArrayList意义维护了一个插入顺序，但是ArrayList比Vector速度快，因为Vector是同步的
4. Vector可以存放不同类型的数据（比如：在不定义泛型的情况下，可以同时插入 int类型1和String类型“aaa”,ArrayList也可以这么做）
5. Vector默认容量是10
6. Vector里面的元素可以重复

## 问题

1. 为什么java的集合可以存放不同类型的数据

例如ArrayList：ArrayList里面真实存放数据的数组是Object[] elementData，因为任何类型的数据实际都是继承子Object所以我们可以同时存入int类型和String类型（不指定泛型的情况下）（还有就是也不能说是不同类型，毕竟他们都继承自Object）

# Java

## Java类加载

## GC回收策略

# Kotlin

## 常用语法

1、操作符介绍

a ? 变量可能为空

b !! 变量不为空的情况下运行

c == 和 ===

== 判断值是否相等

=== 判断值和引用是否都相等

d : 用于继承 变量后面限定变量类型

e .. 区间符号

f \_ 下划线

g @ 限定符this符号 this@MainActivity

@ 跳出for循环或者foreach

h @JvmOverloads 尚未了解

i vararg 尚未了解

2、

in !in 检查值是否在一个范围或者一个区间 例：in 1..10 -> print("x is in the range")

3、kotiln字符串拼接 使用 '$+变量'' 就可以取这个变量的值

val firstName = "Amit"

val lastName = "Shekhar"

val message = "My name is: $firstName $lastName"

4、for循环

for (i in 1..10) { }

for (i in 1 until 10) { }

for (i in 10 downTo 0) { }

for (i in 1..10 step 2) { }

for (i in 10 downTo 1 step 2) { }

for (item in collection) { }

for ((key, value) in map) { }

5、集合操作

val listOfNumber = listOf(1, 2, 3, 4) list

val keyValue = mapOf(1 to "Amit", Map

2 to "Ali",

3 to "Mindorks")

cars是个集合，找出集合中元素speed>100的所有元素的集合

cars.filter { it.speed > 100 } \*这里it是关键字，必须用it表示集合里面的元素

.forEach { println(it.speed)}

如果想根据下标遍历对象可以用indices

例:

for (i in array.indices)

print(array[i])

6、kotiln声明实体类的方式

data class DataClass2(var b: Any)

可以把所有的实体类放在同一个文件里，非常方

7、kotiln遍历列表拿到每一个属性的方法

val listOf = listOf<Int>(1, 2, 3)

val sumBy = listOf.sumBy { i ->

i

}

Log.e("总和", sumBy.toString())

## Kotlin的优缺点

优点：

a、完全兼容java（可以和java相互调用）

b、null safe（减少空指针异常）

c、对开发的好处：代码量少、开发速度快、更少的空指针

d、语法好，比如字符串拼接

e、天然支持函数式编程，可以不声明类

f、kotiln语言支持动态扩展类（扩展就是在不使用继承的情况下给类增加方法属性等，扩展优于继承）

g、kotiln可以直接给方法赋值给另一个方法作为成员变量

例：

fun test(a: Int,b :Int): Boolean {

return true

}

fun test1(a: Int) = test(a,1)

4、链接推荐 https://www.jianshu.com/p/97eb59ea7eae

## 总览

1、用链表表示线性表的优点是便于插入和删除

2、单链表中头结点的目的（方便运算的实现）

3、栈和队列的相同点（只可以在端点处进行插入和删除）

4、栈通常采用的两种存储结构（链式结构和线性结构）

5、队列具有 ”先进先出“ 栈 ”先进后出“

6、链表的优势劣势（优势：插入删除的时候不需要移动元素， 劣势：不可以随机访问某一个元素）

7、循环列表的优点 （从链表的任意一个节点出发都可以访问整个链表）

8、算法由（顺序、选择、循环）控制结构组合而成

9、算法的时间复杂度（执行算法过程中的需要的基本运算次数）

10、算法的空间复杂度 （执行算法所需要的存储空间）

11、数据的存储结构 (数据的逻辑结构在计算机中的表示)

12、数据的逻辑结构 （反应数据元素之间逻辑关系的数据结构）

13、算法的五个特性：

有限性：算法在执行有限步后必须终止

确定性：算法的每个步骤都需要精确的定义，严格的，无歧义的运行

输入：

输出：

可行性：

14、与单链表相比双向链表更容易访问相邻节点

15、对长度为 N 的线性表进行查找，最坏的 情况需要比较 N 次

16、顺序存储的方法是把逻辑上相邻的节点存储在物理上相邻的存储单元

17、数组和链表的区别：

数组不可以动态的定义大小，只能提前分配一个较大的空间，很可能造成空间浪费

数组是顺序存储，查找方便但是插入和删除不方便（因为需要移动后面的元素）

链表在内存中是链式的存储结构，查找不方便时间复杂度O（n），插入和删除方便，时间复杂度O（1）

集合：结构中的元素除了同属于一个集合的关系外，别无其它关系

线性结构：结构中的元素存在一个对一个的关系

树形结构：结构中的元素存在一个对多个的关系

图形结构或网状结构：结构中的元素存在多个对多个的关系

数据结构在计算机中的表示称为物理结构，又称存储

# 设计模式

# 具体题目

## 分组一

题目链接：<https://www.jianshu.com/p/feb9584b492c>

### wait和sleep区别

### view和SurfaceView的区别

### 内存泄露如何查看和解决

### RecyclerView和ListView区别

### RecyclerView嵌套卡顿如何解决

### SharePreference跨进程使用会怎样，如何保证跨进程使用安全

### 推送到达率如何提高

参考文章：<https://www.jianshu.com/p/e4ddc1686220>

## 分组二

### 八种数据类型和他们的包装类

### 重写和重载的区别

### java中的静态类可以被继承吗

## 分组三

参考：

<https://www.jianshu.com/p/56296dd1d4c7>

### java中==和equals的区别

### int和Integer的区别

### 什么是内部类，内部类的作用

### Serializable和ParceLable的区别

### 静态属性和方法是否可以被继承，是否可以被重写，原因

### String转换成Integer的原理

### 哪些情况下对象会被垃圾回收器回收

### 静态代理和动态代理的区别

### String为什么要设计成不可变的

### List、Set和Map的区别

### Java中run和start方法的区别

### 什么导致线程阻塞，怎么关闭线程

### 如何保证线程安全

### synchronized和volatile关键字的区别

### java中堆和栈的区别

### fragment如何实现类似activity的栈的压栈和出栈的效果

### androidcpu型号有哪些

### merge和viewstab的区别

### android子线程为什么不能更新ui

### activity、window、view的关系

### res/drawable res/raw assets的区别

### android的长连接怎么处理心跳机制

### 极光推送、小米推送、华为推送退出app能继续受到通知吗

## 分组4

参考：<https://blog.csdn.net/m0_37868230/article/details/81457720>

### activity在有dialog的情况下按home的生命周期

### anr是什么，怎么避免anr

## 分组5

参考：<https://msd.misuland.com/pd/3255817997595449674>

### apk安卓过程

### invalidate和postinvalidate区别

### 广播注册后不解除广播有什么问题

### Dalvik与jvm的区别

### Application在多进程情况下会调用多次onCreate吗

### int char long各占几个字节

### 造成anr的原因

### BroadCastReceiver和LocalBroadCastReceiver的区别

### http请求慢 的解决办法

### android怎么检测弱网

### androidstudiobuild后都发生了什么

### 一个应用程序安装到手机上时候都发生了什么

### oom是否可以try catch

### 多进程场景，什么情况下使用多进程

## 分组六

### 强引用设置为null会不会被回收

### application和Activity的Context的区别

## 比较深入比较难的问题

### app如何沙箱化，为什么这么做

### 底层的权限管理系统如何进行gaint的

## 比较简单的

### 如何将一个activity设置成窗口的模式

android:theme="@android:style/Theme.Dialog"