# 内存优化

**核心问题就是要解决内存泄露、内存溢出**.

## 什么是内存泄漏？什么情况下会导致内存泄漏？

内存泄漏（Memory Leak）是指程序中己动态分配的堆中无用内存由于某种原因程序未释放或无法释放，导致程序运行速度减慢甚至系统崩溃等严重后果。

大多数情况下导致内存泄漏的根本原因是：长生命周期对象引用短生命周期对象而导致短生命周期不能被及时回收。

造成内存泄漏的对象一定满足如下两个条件：

1. **从GCRoot可达。**
2. **是无用的对象。**

常见场景**：**

（1）**静态集合对象**引起内存泄漏。主要是HashMap,Vector等。如果是静态集合这些集合没有及时set null的话。短周期对象就无法及时释放  
 （2）**观察者模式**：我们在使用监听器的时候，往往是addxxxlistener，但是当我们不需要的时候，忘记removexxxlistener，就容易内存leak。  
 （3）**广播**没有unRegisterRecevier  
 （4）**数据连接**没有关闭，数据库、ContentProvider，IO，等  
 （5）**内部类**：Java中的内部类，会持有宿主的强引用this。如果是new Thread这种，后台线程的操作，当线程没有执行结束时，activity不会被回收。

（6）**单例**是一个全局的静态对象，当持有某个对象是该对象无法释放

**总结为三类**：

1. 赃数据未及时清理。

2、长生命周期对象持用短生命周期对象，导致短生命周期对象无法被及时回收。

3、各种资源在使用完后未及时释放。

## OOM是什么？什么情况下会导致OOM?如何避免OOM异常：

操作系统给虚拟机分配的内存大小有限，当应用程序需要申请一段大内存，但是虚拟机没有办法及时的给到，严重的内存溢出会导致OOM

**原因：**（1）严重的内存泄漏。

（2）大内存对象，因为GC回收机制会尽量避免回收大内存对象。

**解决方案**：

**[1]减少内存对象的占用**：减少bitmap的内存占用，减少资源图片的大小，避免在Android里面使用Enum，优化布局层次，减少内存消耗。

**[2]内存对象的重复利用：**listView/GridView/RecycleView中对contentView的复用，避免在onDraw方法里面new对象，stringBuilder代替+，复用系统自带的资源，Bitmap对象的复用、**对象池**

**[3]增加可用内存：**

(1) AndroidManifest ->android:largeHeap=”true”

(2) 通过C、C++实现

**[4]使用合适分辨率的图片**

## 以下同避免内存泄漏：

[3] 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

[4] 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。

[5]利用静态内部类不会持有外部对象引用优化Handler

[6] 关注线程创建及销毁：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

## ANR是什么？怎么避免和解决ANR：

在规定的时间内，**主线程**无法及时响应用户交互事件,不同的组件发生ANR 的时间不一样：

(1)service 前台20s后台200s未完成启动

(2)Activity是 5 秒

(3)BroadCastReceiver 是 10 秒  
ANR的关键是避免在UI线程处理耗时操作。

常见发生场景：

1. UI线程发生耗时操作。
2. UI线程等待其它某个线程释放锁。
3. CPU负载过重。

**UI卡顿原理及原因？**

**原理：**60fps，每隔1/60秒绘制一帧，错过就会丢帧。

**原因**：1、主线程被轻微阻塞。

2、View过于复杂，未能及时完成绘制。

3、过度绘制，重叠太多，无法及时绘制。

4、GC活动频繁，阻碍绘制。

5、过度使用动画或其它原因导致CPU、GPU负载过重。

6、代码逻辑本身存在问题。

# 线程与进程

## Android线程间通信有哪几种方式?

1、共享变量(静态变量、volatile)

2、handler机制

3、Activity#runOnUiThread方法

4、广播

5、AsyncTask方法

6、View#poast（Runable runable）

7、EventBus

## Android进程的五个常用等级:

**一、前台进程（Foreground process）**

　　 前台进程是用户当前做的事所必须的进程，如果满足下面各种情况中的一种，一个进程被认为是在前台：

进程持有一个正在**与用户交互的Activity**。

进程持有一个Service，这个Service处于这几种状态:①Service与用户正在交互的Activity绑定。②Service是在前台运行的，即它调用了 startForeground()。③Service正在执行它的生命周期回调函数（onCreate(), onStart(), or onDestroy()）。

进程持有一个**BroadcastReceiver**，这个BroadcastReceiver正在执行它的 onReceive() 方法。

杀死前台进程需要用户交互，因为前台进程的优先级是最高的。

二、**可见进程（Visible process）**  
　　如果一个进程不含有任何前台的组件，但仍可被用户在屏幕上所见。当满足如下任一条件时，进程被认为是可见的：

进程持有一个Activity，这个Activity不在前台，但是仍然被用户可见（处于onPause()调用后又没有调用onStop()的状态，比如，前台的activity打开了一个对话框，这样activity就会在其后可见）。

进程持有一个Service，这个Service和一个可见的Activity绑定。

可见的进程也被认为是很重要的，一般不会被销毁，除非是为了保证所有前台进程的运行而不得不杀死可见进程的时候。

三、**服务进程 （Service process）**  
　　如果一个**进程中运行着一个service**，这个service是通过 startService() 开启的，并且不属于上面两种较高优先级的情况，这个进程就是一个服务进程。

尽管服务进程没有和用户可以看到的东西绑定，但是它们一般在做的事情是用户关心的，比如后台播放音乐，后台下载数据等。所以系统会尽量维持它们的运行，除非系统内存不足以维持前台进程和可见进程的运行需要。

**四、后台进程 （Background process）**  
　　如果进程不属于上面三种情况，但是进程**持有一个用户不可见的activity**（activity的onStop()被调用，但是onDestroy()没有调用的状态），就认为进程是一个后台进程。

后台进程不直接影响用户体验，系统会为了前台进程、可见进程、服务进程而任意杀死后台进程。通常会有很多个后台进程存在，它们会被保存在一个LRU (least recently used)列表中，这样就可以确保用户最近使用的activity最后被销毁，即最先销毁时间最远的activity。

**五．空进程**  
如果一个进程不包含任何活跃的应用组件，则认为是空进程。  
例如：一个进程当中已经没有数据在运行了，但是内存当中还为这个应用驻留了一个进程空间。  
保存这种进程的唯一理由是为了缓存的需要，为了加快下次要启动这个进程中的组件时的启动时间。系统为了平衡进程缓存和底层内核缓存的资源，经常会杀死空进程

# Android中的动画有哪几种，他们的特点和区别是什么

[1]视图动画或者说补间动画。只是视觉上的一个效果，实际view属性没有变化，性能好，支持方式少  
[2]属性动画，通过改变属性来达到动画的效果，性能略差，支持点击等事件。Android3.0加入  
[3]帧动画，通过drawable一帧帧画出来。

# 属性动画是什么？插值器、估值器是什么？

# SurfaceView和view的区别

view的更新必须在UI thread中进行SurfaceView会单独有一个线程做UI的更新SurfaceView支持open GL绘制,surfaceChanged大小发生变化，surfaceCreated创建的时候激发，surfaceDestroyed()销毁时激发 。

被动动画使用VIew，如触摸产生的动画用view，比如打消消乐,被动动画使用SurfaceView，如一直在动的动画用SurfaceView，比如有一款跑步的app里面的效果

# Java基础知识

## 标准IO与NIO的区别？

<https://blog.csdn.net/g6U8W7p06dCO99fQ3/article/details/92802325?utm_source=distribute.pc_relevant.none-task>

## Class.forName( )的作用是什么？

使用Class.forName( )静态方法的目的是为了动态加载类。在加载完成后，一般还要调用Class下的newInstance( )静态方法来实例化对象以便操作

## 深拷贝、浅拷贝有什么区别？

1、深拷贝：深拷贝就是对象所在的对象数中所有的对象都进行拷贝，对基本数据类型进行值传递，对引用数据类型，创建一个新的对象，并复制其内容。

2、浅拷贝：对基本数据类型进行值传递，对引用数据类型进行引用传递般的拷贝。

## 强引用、软引用、弱引用、虚引用的区别？

**强引用**：如果一个对象具有强引用，那垃圾回收器绝不会回收它，当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。

**软引用**（SoftReference）：如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

**弱引用**：弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象将要被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

当你想引用一个对象，但是这个对象有自己的生命周期，你不想介入这个对象的生命周期，这时候你就是用弱引用。

**虚引用**：

“虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。

虚引用主要用来跟踪对象被垃圾回收器回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之 关联的引用队列中。

## CopyOnWriteArryList原理?

## **数组和集合类的区别?**

数组长度固定，集合长度可变。数组中只能存储同种基本类型的对象，集合在未指定泛型的情况下可以存储多种类型的对象。

## 面向对象的特征有哪些方面？

**抽象**：把一类事物的明确的属性和行为提取出来，形成一个模型，这种研究问题的方法称为抽象

**封装**：就是把对象的属性和行为（数据）有机地整合为一个独立的整体，并隐藏对象的内部实现细节，把可以告诉别人的公开。

**继承**：是面向对象最显著的一个特性，继承是从已有的类中派生出新的类称为子类，子类继承父类的数据属性和行为，并能根据自己的需求扩展出新的行为，提高了代码的复用性。

**多态**：指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。是通过动态绑定（dynamic binding）实现，动态绑定是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法

## 访问修饰符public private protected 以及不写时的区别？

public 公共的修饰符  
private 当前类的修饰符  
protected 当前类，子类，同包修饰符  
default 当前类和同包修饰符

## Math函数方法

Math.round()四舍五入  
Math.ceil()得到不小于某数的最小整数:>=  
Math.floor()得到不大于某数的最大整数:<=

## 16.最有效的方法计算2乘以8？

2 << 3;

## String与StringBuilder和StringBuffer区别？

[1]String字符串常量：String是不可变的对象，因此在每次对String类型进行改变的时候其实都等于生成了一个新的String对象

[2]StringBuffer字符串变量（线程安全）：每次改变结果都会对StringBuffer对象本身进行操作，而不是生成新的对象

[3]StringBuilder字符串变量（非线程安全）

总结：

String：适用于少量的字符串操作的情况

StringBuilder：适用于**单线程下在字符缓冲区进**行大量操作的情况

StringBuffer：适用**多线程下在字符缓冲区**进行大量操作的情况

## 重载（overload）和重写（override）的区别？

**重载**：方法重载是让类以统一的方式处理不同类型数据的一种手段。多个同名函数同时存在，具有不同的参数个数/类型。重载是一个类中多态性的一种表现。就是在类中可以创建多个方法，他们具有相同的名字，但具有不同的参数和不同的定义。调用方法时通过传递给它们不同参数个数和参数类型来决定具体使用哪个方法。方法名要一样，但是参数类型和个数不一样，返回值类型可以相同也可以不相同。

**重写：**父类与子类之间的多态性，对父类的函数进行重新定义。如果在子类中定义某方法与其父类有相同的名称和参数，我们说该方法被重写。

## 静态与非静态的区别？

[1]静态类不能实例化，静态类中不能创建非静态的方法。

[2]静态方法属于类所有，在类实例化前即可使用。非静态方法可以访问类中的任何成员，静态方法只能访问类中的静态成员，因为静态方法在类实例化前就使用，而类中的非静态变量必须在实例化之后才能分配内存，

[3]静态变量不自动进行销毁，而实例化的则可以做销毁

[4]静态成员不能被序列化

## Java中的final关键字有哪些用法？

[1]final修饰的类不能被继承

[2]final修饰的方法不可重写

[3]final修饰的变量只能赋值一次

## try{}里面一个return语句，那么finally{}里的代码会不会被执行，什么时候被执行？在return前还是后？

会执行，在执行return前执行

## 描述final，finally，finalize的区别？

final：Java中的关键字，修饰符  
如果一个类被声明为final，就意味着它不能再派生出新的子类，不能作为父类被继承。因此一个类不能同时被声明为absrtac抽象类和final的类，如果将变量或者方法声明为final，可以保证它们在使用中不被改变  
finally：Java的一种异常处理机制  
finally是对Java异常处理模型的最佳补充。finally结构使代码总会被执行，而不管有无异常发生。使用finally可以维护对象的内部类状态，并可以清理非内存资源。  
finalize：Java中的一个方法名  
当对象变成(GC Roots)不可达时，GC会判断该对象是否覆盖了finalize方法，若未覆盖，则直接将其回收。否则，若对象未执行过finalize方法，将其放入F-Queue队列，由一低优先级线程执行该队列中对象的finalize方法。执行finalize方法完毕后，GC会再次判断该对象是否可达，若不可达，则进行回收，否则，对象“复活”。

**对象再生问题：finalize方法中，可将待回收对象赋值给GC Roots可达的对象引用，从而达到对象再生的目的**

## 关于finally()方法,有哪些情况会导致它不会被执行?

1、在try/catch分支中调用System.exit()

2、线程提前死亡

## 线程有哪几种状态？

1. **新建状态(New)**: 线程对象被创建后，就进入了新建状态。例如，Thread thread = new Thread()。  
2. **就绪状态(Runnable)**: 也被称为“可执行状态”。线程对象被创建后，其它线程调用了该对象的start()方法，从而来启动该线程。例如，thread.start()。处于就绪状态的线程，随时可能被CPU调度执行。  
3. **运行状态(Running)** : 线程获取CPU权限进行执行。需要注意的是，线程只能从就绪状态进入到运行状态。  
4. **阻塞状态(Blocked)** : 阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：  
    (1) 等待阻塞 -- 通过调用线程的wait()方法，让线程等待某工作的完成。  
    (2) 同步阻塞 -- 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态。

(3) 其他阻塞 -- 通过调用线程的sleep()或join()或发出了I/O请求时，线程会进入到阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

5. **死亡状态(Dead)**    : 线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

## ****介绍一下wait(), notify(), notifyAll()方法****

在Object.java中，定义了wait(), notify()和notifyAll()等接口。wait()的作用是让当前线程进入等待状态，同时，wait()也会让当前线程释放它所持有的锁。而notify()和notifyAll()的作用，则是唤醒当前对象上的等待线程进入就绪状态；notify()是唤醒单个线程，而notifyAll()是唤醒所有的线程。

Object类中关于等待/唤醒的API详细信息如下：  
**notify()**-- 唤醒在此对象监视器上等待的单个线程（随机）-进入就绪状态。  
**notifyAll()**  -- 唤醒在此对象监视器上等待的所有线程-进入就绪状态。  
**wait()**  -- 让当前线程处于“等待(阻塞)状态”，“直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法”，当前线程被唤醒(进入“就绪状态”)。  
**wait(long timeout)** -- 让当前线程处于“等待(阻塞)状态”，“直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量”，当前线程被唤醒(进入“就绪状态”)。  
**wait(long timeout, int nanos)**-- 让当前线程处于“等待(阻塞)状态”，“直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量”，当前线程被唤醒(进入“就绪状态”)。

## 为什么notify(), wait()等函数定义在Object中，而不是Thread中？

Object中的wait(), notify()等函数，和synchronized一样，会对“对象的同步锁”进行操作。

wait()会使“当前线程”等待，因为线程进入等待状态，所以线程应该释放它锁持有的“同步锁”，否则其它线程获取不到该“同步锁”而无法运行！  
线程调用wait()之后，会释放它锁持有的“同步锁”；而且，根据前面的介绍，我们知道：等待线程可以被notify()或notifyAll()唤醒。现在，请思考一个问题：notify()是依据什么唤醒等待线程的？或者说，wait()等待线程和notify()之间是通过什么关联起来的？答案是：依据“对象的同步锁”。

负责唤醒等待线程的那个线程(我们称为“**唤醒线程**”)，它只有在获取“该对象的同步锁”(**这里的同步锁必须和等待线程的同步锁是同一个**)，并且调用notify()或notifyAll()方法之后，才能唤醒等待线程。虽然，等待线程被唤醒；但是，它不能立刻执行，因为唤醒线程还持有“该对象的同步锁”。必须等到唤醒线程释放了“对象的同步锁”之后，等待线程才能获取到“对象的同步锁”进而继续运行。

总之，notify(), wait()依赖于“同步锁”，而“同步锁”是对象锁持有，并且每个对象有且仅有一个！这就是为什么notify(), wait()等函数定义在Object类，而不是Thread类中的原因。

# ****yield()有什么作用？****

**yield()的作用是让步。它能让当前线程由“运行状态”进入到“就绪状态”（），从而让其它具有相同优先级的等待线程获取执行权；但是，并不能保证在当前线程调用yield()之后，其它具有相同优先级的线程就一定能获得执行权；也有可能是当前线程又进入到“运行状态”继续运行！**

**yield() 与 wait()的比较**

**我们知道，**[**wait()**](http://www.cnblogs.com/skywang12345/p/3479224.html)**的作用是让当前线程由“运行状态”进入“等待(阻塞)状态”的同时，也会释放同步锁。而yield()的作用是让步，它也会让当前线程离开“运行状态”。它们的区别是：**

**wait()是让线程由“运行状态”进入到“等待(阻塞)状态”，而yield()是让线程由“运行状态”进入到“就绪状态”**

## Thread类的sleep()方法和对象的wait()方法都可以让线程暂停执行，它们有什么区别？

（1）sleep（）是由操作系统实现的而wait（）是由虚拟机实现的

（2）sleep()方法是线程类的静态方法，调用此方法会让当前线程暂停执行指定的时间，将执行机会让给其他线程，但是对象的锁依然保持，在休眠结束后会自动回复。

（3）wait()是object类的方法，调用对象的wait()方法导致当前线程放弃对象的锁（线程暂停执行），**进入对象的等待池，重新进入阻塞状态**，线程得到了notify()或者notifyAll()消息，线程才会进入就绪状态。

（4）wait（）方法必需在同步方法或同步块中调用

## Java四类八种数据类型

基本数据类型：

byte 1 byte

short 2 byte

int 4 byte

long 8 byte

float 4 byte

double 8 byte

char 2 byte

Boolean jvm规范并没有明确定义其大小,通常是4个字节,如果是boolean数组则是1个字节  
应用数据类型：class interface 数组  
short从-32768到32767  
int从-2147483648到2147483647共10位  
long从-9223372036854775808到9223372036854775807共19位

# 什么是范型?它的意义?

# 什么是注解?元注解,注解的处理方式有哪些?

# ListView错乱原因：

1.listView item的缓存机制：  
为了使用性能更优，listView会缓存行item。listView通过adapter的getView函数获得每行item，如果某行item已经滑出屏幕，若该item不在缓存内，则put进缓存，否则更新缓存；获取滑入屏幕的行item之前会先判断缓存中是否有可用的item，如果有做为convertView参数传递给adapter的getView  
2.行item图片显示重复异步加载图片滑动快加载慢，造成显示重复  
3.行item图片显示错乱  
比如ListView滑动到第2行会异步加载某个图片，但是加载很慢，加载过程中listView已经滑动到了第14行，第2行已不在屏幕内，根据上面介绍的缓存原理，第2行的view可能被第14行复用，第14行显示了第2行的View，这时之前的图片加载结束，就会显示在第14行，造成错乱  
4.行item图片显示闪烁  
有很快加载结束，先显示缓存的图片，立马又显示了自己的图片进行覆盖造成闪烁错乱

**解决方案**?  
通过上面的分析我们知道了出现错乱的原因是异步加载及对象被复用造成的，如果每次getView能给对象一个标识，在异步加载完成时比较标识与当前行item的标识是否一致，一致则显示，否则不做处理即可listView中加载不同的item使用geiItemViewType和使用getViewTypeCount设置item样式数

# 四大组件

## 请介绍一下ContentProvider是如何实现数据共享的

## intentFilter的匹配规则?

在<intent-filter></intent-filter>中可以包含action，category、data。action不能单独使用，必须配合category、或data使用，如果只想在intent中添加一个action就能匹配，可以在intent-filter中加入一个：

<category android:name="android.intent.category.DEFAULT" />

**action**：intent-filter中可以同时存在多个，能与其中任意一个匹配即可(在intent中只能添加一个action)。

**category：**intent-filter中可以同时存在多个，在intent中添加无论多少个category一定是intent-filter中category集合的子集才行。

**data**：intent-filter中可以有多个，只要匹配其中一个即可。

## 如何修改activity进入和退出动画

overridePendingTransition

## Activity的启动模式：

1.standard：标准

2.singleTop：（栈顶复用）

3.singleTask：（栈内复用）

4.singleInstance：单例

设置Android的启动模式在AndroidManifest.xml里面对应的<activity>标签设置launchMode属性  
**1.standard：默认模式**，可以不用写配置。因此在这个模式下可以有多个相同的实例，也可以有多个相同的activity叠加。点击back键会依照栈的顺序依次退出  
**2.singleTop**：允许有多个实例但是不允许有多个activity叠加。即如果activity在栈顶的时候，启动相同的activity不会创建新的实例，而会调用其onNewIntent方法  
**3. singleTask**：只会有一个实例，在同一个应用程序中启动它的时候，若activity不存在，则在当前的task创建一个新的实例。若存在则会把task中之上的其他activity destroy掉并调用它的onNewIntent方法  
如果在其他应用中启动它，则会新建一个task，并在这个task中启动这个activity。singleTask允许别的activity在一个task中存在  
**4. singleInstance：**只有一个实例，并且这个实例独立运行在一个task中，这个task只有这个实例，不允许有别的activity存在

## Android四大组件，如何在android系统中看待它们，为何设计它们？

**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。

## Service和Thead有何不同？

1. Service是android四大组件之一，具有自己独特生命的周期，可以方便地进行管控，它在主线程运行，有Service活动的APP比没有的有较高优先级。
2. Thread则是CPU调度的最小单元，它与Service是完全不同性质的东西，与Service相比它不易管控，很难在一个组件中对另个组件中运行的线程进行管理。

## Service生命周期？

必需**分两种情况**讨论：

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

[2]context.bindService()->onCreate()->onBind()->Service running--调用context.unbindService()->onUnbind() -> onDestroy()

## 屏幕旋转时Activity的生命周期？

## IntentService的特点?可以通过bindService()启动吗?

IntentService是Service的子类.使用时必须实现抽象方法onHandleIntent(),在该方法中可以进行耗时操作,该方法执行完后会自行调用stopSelf()方法关闭服务,其内部是通过在onCreate()中创建一个HandlerThread对象并启动它,并且会使用HandlerThread中的Looper对象来构建一个Handler,在onStartCommand会调onStart()方法并在中会发送一个消息给这个Hanlder,而在这个Handler的hanldeMessage()方法中会调用onHandleMessage()以及stopSelf()

IntentService在onBind()返回null,并且startCommand()不会被调用.最终onHandleIntent()也不会被调用,所以bindService()不能正常启动IntentService.

## 如何判断一个Intent是否能成功启动Activity?

## 四大组件生命周期

1. **Activity：**常用的生命周期方法有 ；  
   onCreat()->onStart()->onResume->running->onPuase()->onStop()->onDestroy()

当已经存在的Activity由不可见到可见会走onRestart方法。  
意外销毁时会走onSaveInstanceState方法结束，由意外销毁恢复走onRestoreInstanceState方法。

发生意外销毁的原因有两类:1、内存不足 2、系统配置发生变化:如 **local(用户所在区域发生变化，一般是用户切换了语言时，切换后的语言会显示出来)、orientation、keyboardHidden(键盘的可用性发生了改变) fontScale(**字体比例发生了变化----选择了不同的全局字体**)**

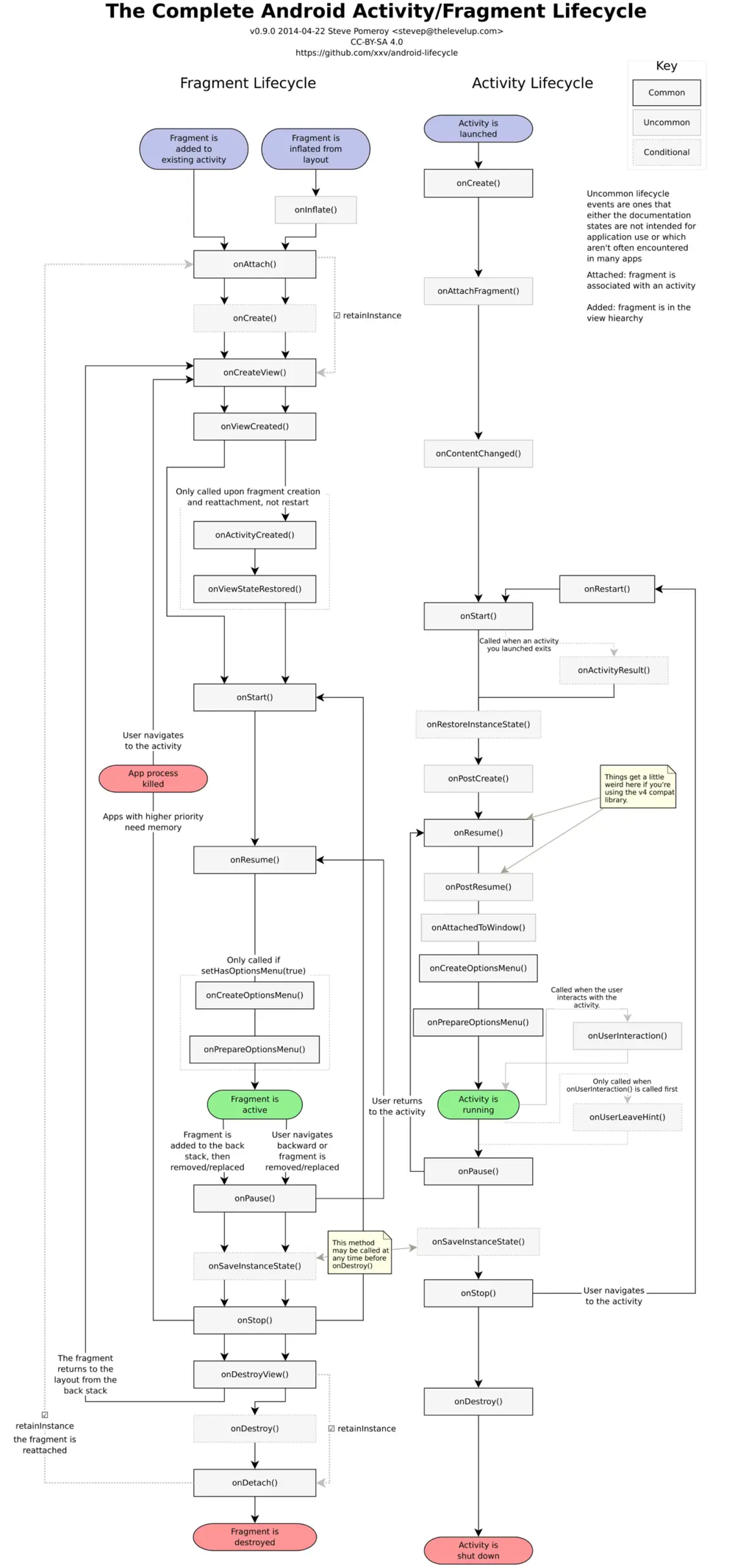
意外销毁时onPause 、onSaveInstanceState、onStop、onDestroy方法均会被调用。onSaveInstanceState方法会在onStop方法之前调用，但是他和onPause方法调用没有特定先后顺序，可前可后。

**2）BroadcastReceive：**在接收到广播是执行onReceive方法在该方法返回结果后结束，如果在onReceive中做了超过10s的事情就会报ANR。  
**3）Service：**

* 当**start**一个Service时的生命周期方法：  
  onCreate->onStart->onDestroy
* 当**bind**一个Serive时的生命周期方法：  
  onCreate->onBind->onUnbind->onDestroy

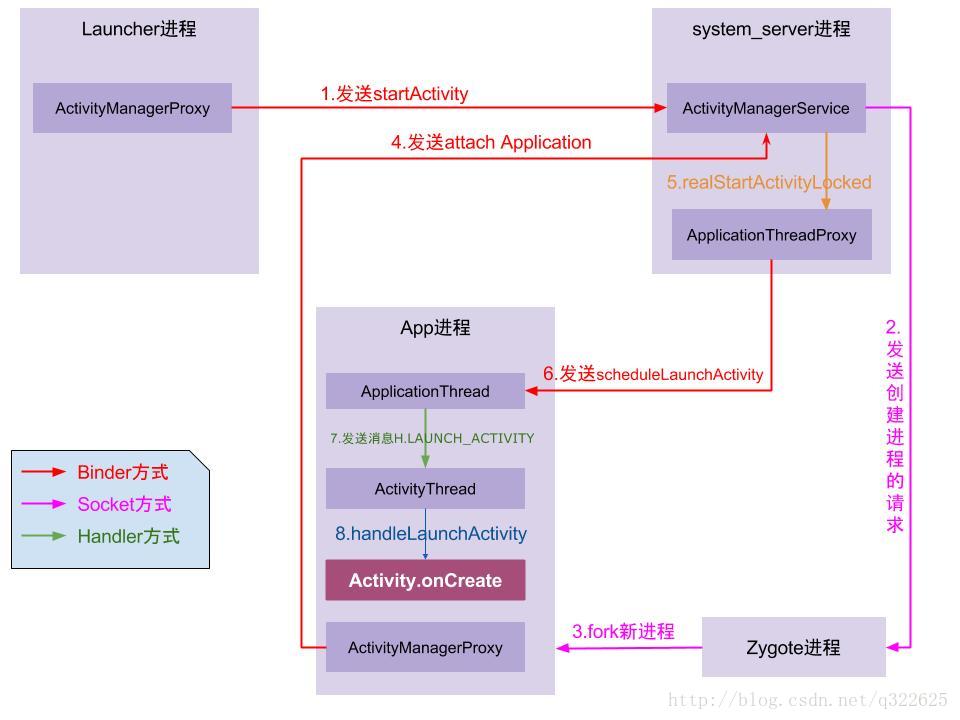
**4)ContentProvider:**在Android平台上多个应用程序共享数据时会用到。 在访问ContentProvider的时候，可以通过Activity或者Context的**getContentRresolver**来访问。每一个ContentProvider都有一个唯一的URI来识别。  
其中的方法有：query，insert，update，delete；  
ContentProvider在接收到**ContentResolver**发出的请求后被激活，也仅在响应ContentResolver提出的请求是激活，我们**普遍认为它在请求完毕后关闭**。

## Fragment生命周期？onAttach()->onCearte()->onCreateView()->onActivityCreate()->onStart()->onResume()—活动—>onStop()->onDestroyView()->onDestroy()->onDetach()



## Service通过startService()与bindService()不同顺序启动有何不同?

## 描述一下Activity启动流程



# Android视图绘制流程？自定义View如何考虑机型适配；自定义View的事件

View的绘制是从ViewRoot的performTraversals（）方法开始的，经过measure，layout，draw这个3大步骤。

##### measure的过程：

measure过程是对整个view树的所有控件计算宽高  
measure是从ViewRoot类中的host.measure开始的，内部调用的是View的measure（int widthMeasureSpec,int heightMeasureSpec）方法，measure方法里面调用了onMeasure（int widthMeasureSpec,int heightMeasureSpec）方法，方法中的两个参数都是是MeasureSpec类型（指父控件对子控件宽高的期望值，它是一个32位的int类型数，前两位表示测量模式，后30位表示测量大小）  
测量模式一共有3种：  
1）EXACTLY 精确测量模式，xml文件中写200dp，march\_parent等代表使用该模式，  
2）AT\_MOST 最大模式，xml文件中写wrap\_content表示使用该模式。  
3）UNSPECIFIED 无限大测量模式，只有在绘制特定自定义View时才用的到这个模式。  
真正代表测量结束的方法是setMeasuredDimension方法，该方法传入的两个参数是宽高的SpecSize。测量结束后我们可以通过getMeasureHeight和getMeasureWidth来获取测量宽高。  
自定义ViewGroup一定要重写onMeasure方法，用于测量子View的宽高，不重写的话子View没有宽高。  
自定义View如果在xml中使用了wrap\_content属性，就需要重写onMeasure方法来设置wrap\_content的默认大小，不然会显示出match\_parent的效果。

##### layout的过程：

ViewGroup用来将子View放在合适的位置上。  
layout是从ViewRoot类中的host.layout开始的，内部调用的是ViewGroup的layout方法。在ViewGroup的layout方法中，先调用setFrame来确定自己的左上右下的位置，再调用onLayout来确定子View的位置。  
自定义ViewGroup一定要重写layout方法来确定子View的位置，自定义View一般不需要重写该方法，它的位置是父控件确定的。

##### draw过程:

此过程是真正将内容展示在屏幕上让我们能够看到的过程。  
draw是从ViewRoot类中的host.draw开始的，内部调用的是View的draw方法。  
draw的步骤：  
1）绘制背景。  
2）绘制内容，也就是调用onDraw方法。  
3）绘制子View，调用的是dispatchDraw方法。  
4）绘制装饰，如listview的滚动条等。

对于View的绘制过程，既可以说是简单的，也可以说是复杂的，简单的在于Google已经帮我们将draw框架写好了，我们在自定义ViewGroup时不用管draw过程，只需要实现measure和layout过程。复杂在于，我们写继承View的自定义控件的时候需要重写onDraw方法，这样才能绘制出你自定义的View的内容，onDraw（Canvas canvas）方法中最重要的两个东西是Paint和Canvas，这个使用起来算是比较复杂的。

# 如何实现自定义View？

定义方式有：

1. 继承View重写onDraw（）方法。
2. 继承ViewGroup重写onMeasure（）与onLayout（）方法。
3. 继承已有View或ViewGroup子类，实现功能扩展。

4、自定义组合控件，把常用一些控件组合起来以方便使用。

# 自定义View如何考虑机型适配

（1）获取屏幕宽度px等数据，然后根据这些去设置 view的宽度px，字体的大小

（2）使用像素密度无关的单位，如dp、sp，然后在代码中通过工具类转换为px。

# px、dp、sp、dpi分别是指什么？为什么使用dp还是不能解决适配问题？

**1、px**

　　即像素，1px代表屏幕上的一个物理像素点。

**2、dp**

　　dip：device independent pixels（设备独立像素），dp与dip一样，而dp也叫dip，是device independent pixels。设备不依赖像素的一个单位。在不同的像素密度的设备上会自动适配，比如:

在320x480分辨率，像素密度为160,1dp=1px （这个是**基准**）

在480x800分辨率，像素密度为240,1dp=1.5px

**3、sp**

　　sp:scale-independent pixels（缩放独立像素），与缩放无关的抽象像素，它和dp很相似，但唯一的区别在于，Android系统允许用户自定义文字尺寸大小（小，正常，大，超大等），当文字尺寸是“正常”时，1sp=1dp=0.00625inch（英寸），当文字尺寸是“大”或“超大”时，1sp>1dp=0.00625inch

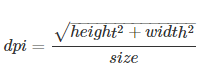
**4、dpi**

　　dpi：dots per inch（每英寸点数，或者说每英寸像素数目 更易懂），是根据设备的屏幕真实分辨率和尺寸大小进行计算得到的，每个设备可能不一样，这也是Android设备进行碎片化的原因，可以通过context.getResources().getDisplayMetrics().densityDpi获取到。当然，也可以通过以下方法算出：

dpi=横向分辨率/横向英寸数=纵向分辨率/纵向英寸数

以5.5寸1920 \* 1080的手机为例：

5.5寸是指手机对角线的长度，分辨率是1920 \* 1080，也就是知道长像素是1920，宽像素是1080，那么对角线多长呢，其实就是求斜边的长度: 1920^2+1080^2=2202^2



所以 dpi=2202/5.5=400

height和width即为长宽的像素，平方和即为对角线的像素个数，size即我们常说的5寸手机、4寸手机中的5和4，即对角线的长度

android中在渲染屏幕时，都会将我们在xml中的**dp单位转化为px**,去渲染到设备中，用到的**转换公式**：（density 的意思就是 1 dp 占当前设备多少像素）

density=dpi/160;

px =dp \* density;

dp=(px/dpi)\*160;

**dp=(px/dpi)\*160**，dpi是由屏幕本身物理属性决定的，因此屏幕不同对应的总宽dp也不同。但是我们布局中的dp是相同的，这样就会导致控件在不同设备上所占比例不同出现变形，因此我们才需要适配

# 你使用过哪些适配方案？

1、Google的百分百布局

2、今日头条适配方案，今日头条适配方案默认项目中只能以高或宽中的一个作为基准，进行适配。这里width可以是宽或高，原理是：修改**density**计算方法使得dp永远保持不变。

屏幕的总 px 宽度 / density = 屏幕的总 dp 宽度 ， 可以看出px肯定是变化的，那么只能改变density 值才能保证dp不变。因此

**density=当前设备屏幕总宽度（单位为px）/ 设计图总宽度（单位为 dp)**

1，优点

使用起来简单，可以全局配置省去很多人工成本。

使用系统API实现，减少性能损耗。

2，缺点

如果是在老项目的基础上做会造成很大影响，不适合老项目。

如果是第三方控件和自己的设计图不一致也会有影响。

# 如何进行屏幕适配？

屏幕适配问题主要解决两个问题：

1、**图片资源适配**：

* 尽量使用点9图片。
* 切图的时候切大分辨率的图，应用到布局当中。在小分辨率的手机上也会有很好的显示效果。

2、**布局适配：**

* 使用与密度无关的像素单位dp，sp
* 引入android的百分比布局。
* 合理使用warp\_content，match\_parent
* 针对不同的机型，使用不同的布局文件放在对应的目录下，android会自动匹配。

# JVM相关

## 简述类加载流程，Class Loader有哪些模型。Android类加载器有何不同？

在java应用程序开发中，只有被jvm装载的class类型才能在程序中使用。只要生成的字节码符合jvm指令集和文件格式，就可以在jvm上运行，这为java的跨平台性提供条件。类加载器就是负责将class文件加载到内存中.

字节码的装载过程分为3个阶段：加载，连接，初始化。其中连接包括3个步骤（验证，准备，解析）。

类的生命周期:

加载->验证->准备->解析->初始化->使用->卸载**(为了实现动态绑定,其中“解析”可能出现在“初始化”后面)**

1）加载：

1、根据全限定类名获取二进制数据流

2、将二进制静态存储数据结构转化为**方法区**中的运行时数据结构.

3、在**堆**中创建一个代表该类的java.lang.Class对象,作为访问**方法区中**对应数据的入口.

2）连接：

* 验证：验证字节码文件，保证加载的字节码是符合规范的。
* 准备：正式为static修饰的变量分配内存，并设置变量的初始值，这些内存都将在方法区进行分配。
* 解析：将类的二进制数据中的符号引用替换成直接引用（符号引用是用一组符号描述所引用的目标；直接引用是指向目标的指针）。

1. 初始化：如果前面的步骤都没有出现问题，那么表示类可以顺利的装载到系统中。这个时候才会执行java字节码。初始化阶段的主要工作是执行类的初始化方法。

参考: <https://blog.csdn.net/ln152315/article/details/79223441>

**ClassLoader**是一个对象，主要是对类的请求提供服务，当jvm需要某个类是它根据名称向ClassLoader请求这个类，然后ClassLoader返回这个类的class对象。

Java类加载器分类:

**1）引导类加载器（**B**ootstrap Class Loader**）:它用来加载**java的核心库**，使用源生代码(C++)来实现的(因此它不是java.lang.ClassLoader的子类)。它属于jVM的一部分不要加载,而其他类加载器需要需要B**ootstrapClassLoader先加载才能使用.**  
**2）扩展类加载器（Extensions Class Loader**）:用来加载**java的扩展库**。jvm的实现会提供一个扩展库目录，该类加载器在目录中寻找并加载java类。  
**3）系统类加载器**（**APP** **Class Loader**）：一般来讲，**java应用的类**都是由它加载完成的，是根据java的应用类路径来加载java类。

**Android类加载器：**

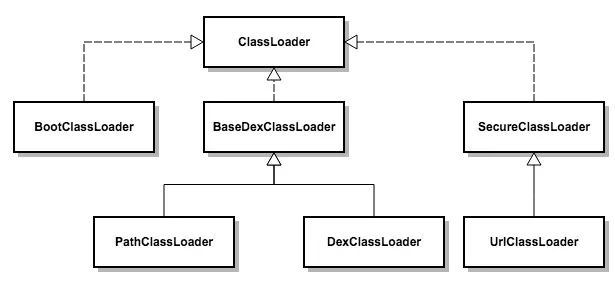
 BootClassLoader，Android系统启动时会使用BootClassLoader来预加载常用类，与Java中的Bootstrap ClassLoader不同的是，它并不是由C/C++代码实现，而是由Java实现的。**BootClassLoader是ClassLoader的一个内部类。**

 PathClassLoader，全名是dalvik/system.PathClassLoader，可以加载已经安装的Apk，也就是/data/app/package 下的apk文件，也可以加载/vendor/lib, /system/lib下的nativeLibrary。

 DexClassLoader，全名是dalvik/system.DexClassLoader，可以加载一个未安装的apk文件。

PathClassLoader和DexClasLoader都是继承自dalviksystem.**BaseDexClassLoader**，它们的类加载逻辑全部写在BaseDexClassLoader中。

下图展示了Android中的ClassLoader中的继承体系，其中SecureClassLoader和UrlClassLoader是在Java中的类加载器，在Android中是没法办使用的。



## 对象创建的流程是什么？

1、当虚拟机遇到new指令时，首先检查该指令的参数在常量池中是否找到一个对应类的符号引用。

2、检查该符号对应的类是否已经加载过，如果没有就先完成加载过程。

3、检查加载通过后，为对象分配内存。

## 请描述一下java对象内存布局

<https://blog.csdn.net/yan_wenliang/article/details/82802490>

## 什么是内存对齐？

<https://www.jianshu.com/p/37409be16a37>

## 请简述字节码文件的组成

<https://www.jianshu.com/p/e5062d62a3d1>

## GC是什么？为什么要有GC?

GC是垃圾收集的意思，内存处理是编程人员容易出现问题的地方。忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java提供的GC功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的。垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。垃圾回收器通常是作为一个单独的低优先级的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收

可以调用下面的方法之一：System.gc()或Runtime.getRuntime().gc()，但JVM可以屏蔽掉显示的垃圾回收调用。

## 常见的GCRoot有哪些？android中最大的GCRoot节点是什么？

a.虚拟机栈(栈桢中的本地变量表)中的引用的对象。

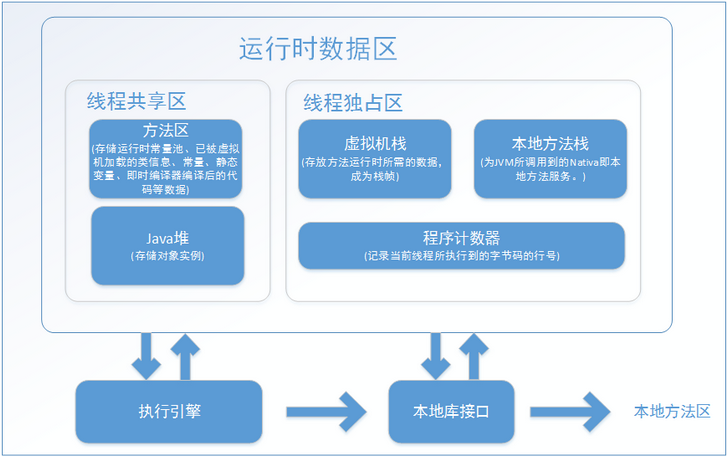
b.方法区中的类静态属性引用的对象。

c.方法区中的常量引用的对象，如：final String =“” ，final int i=0 ，非方法区的常量不会是GCRoot，如：final Object obj=new Object（）。

d.本地方法栈中JNI的引用的对象。

# jvm运行时数据区域有哪几部分组成，各自作用。

jvm运行时数据区通常包括：程序计数器（Program Counter Register）,Java栈（VM Stack）,本地方法栈（Native Method Stack）,方法区（Method Area），堆（Heap）。  
作用：



* 栈： 栈中存放的是一个个的栈帧，每个栈帧对应一个被调用的方法，在**栈帧中包括局部变量表，操作数栈**，指向**当前方法所属的类的运行时常量池的引用**，**方法返回地址**和一些额外的附加信息。当线程执行一个方法时，就会随之创建一个对应的栈帧必定位于java栈的顶部。
* 堆:在java中堆是用来存储对象本身的以及数组（数组引用是存放在java栈中的），在java中程序员不用去关心空间释放的问题，java的垃圾回收机制会自动进行处理。因此这部分空间也是java垃圾收集管理的主要区域。堆是被所有线程共享的，在jvm中只有一个堆。
* 方法区：方法区在jvm中也是一个非常重要的区域，它与堆一样，是被线程共享的区域。在方法区中，**存储了每个类的信息（包括类的名称，方法信息，字段信息），静态变量，常量以及编译器编译后的代码**等。还有一项信息是**常量池**，用来存储编译期间生成的字面量和符号引用。
* 本地方法栈：为使用到的本地操作系统方法服务。
* 程序计数器：一块较小的内存空间，它是当**前线程所执行的字节码的行号指示器**。字节码解释器工作时通过改变该计数器的值来选择下一条要**执行的字节码指令。分支，跳转，循环**等功能都依靠它来实现。**每条线程都有一个独立的计数器，它们之间互不影响**，因此该区域是线程私有的。

# gc算法有哪些？，gc收集器有哪些？

**定义：gc是java的垃圾回收机制**

* **引用计数法**（Reference Counting Collector）:使用计数器来区分存活对象和不再使用的对象。一般来说，堆中的每个对象对应一个引用计数器。**当每一次创建一个对象并赋给一个变量时，引用计数器置为1。当对象被赋给任意变量时，引用计数器每次加1当对象出了作用域后(该对象丢弃不再使用)，引用计数器减1**，一旦引用计数器为0，对象就满足了垃圾收集的条件。
* **标记算法**（Tracing Collector):使用了根集的概念，基于tracing算法的**垃圾收集器从GCRoot开始扫描**，**识别出哪些对象可达，哪些对象不可达**，并用某种方式标记可达对象。
* **整理算法**（Compacting Collecotr）：该算法会将所有的对象移到堆的一端。能解决堆碎片的问题。
* **复制算法**：将内存分为两个区域（from space 和 to space）。所有的对象都分配到from space。**清理时先将所有标为活动对象copy到to space，然后清除from space空间。然后互换from space和to apce的身份**，每次清理都重复上述过程。

gc收集器：

* **serial收集器**：单线程，工作时必须暂停其他工作线程，多用于client机器上，使用**复制算法**。
* **ParNew收集器**：serial的多线程版本，server模式下jvm首选的新生代收集器。**复制算法**。
* **Parallel Scavenge收集器**：可控制吞吐量的收集器，吞吐量指有效运行时间。复制算法。
* **Serial Old收集器**：serial的老年代版本，使用整理算法。
* **Parallel Old收集器**：Parallel Scavenge收集器的老版本，多线程，标记整理。
* **CMS收集器**：整理算法。最短回收停顿时间，缺点是产生碎片。
* GI收集器：基本思想是化整为零，将堆分为多个Region，优先回收价值最大的Region。**并行并发，分代收集，空间整合。整理算法**。

**ART运行时与Dalvik虚拟机一样，都使用了Mark-Sweep算法进行垃圾回收**

## GC过程中是否每次都要重新遍历GCRoot？

## JNI是怎么实现底层与上层调用的？

## Dvm进程与Linux进程，应用程序的进程是否为同一概念?

Dvm进程是dalvik虚拟机进程，，每个Android程序都运行在自己进程里，每个Android程序系统都会给他分配一个独立的Linux uid(user id),每个dvm都是linux里面的一个进程.所以说这两个进程是一个进程.

## dalvik和Java运行环境的区别？（其实就是问dalvik与普通jvm的区别）

dalvik主要完成对象生命周期管理，堆栈管理，线程管理，安全和异常管理，以及垃圾回收等重要功能

2.Dalvik负责进程隔离和线程管理，每一个Android应用在底层都会对应一个独立的Dalvik虚拟机实例，其代码在Dalvik虚拟机的解释下得以执行。

3.不同于Java虚拟机运行Java字节码，Dalvik虚拟机运行的是其专有的文件格式Dex

4.所有的Android应用的线程都对应一个Linux线程，虚拟机因而可以更多的依赖操作系统和线程调度和管理

## Dex优点？它与ODex的区别？

1.dex文件格式可以减少整体文件尺寸，提高I/O操作的类查找速度  
2. dex和odex区别：

其实一个APK是一个程序压缩包，里面有个执行程序包含dex文件，ODEX优化就是把包里面的执行程序提取出来，就变成ODEX文件。因为你提取出来了，系统第一次启动的时候就不用去解压程序压缩包，少了一个解压的过程。这样的话系统启动就加快了。为什么说是第一次呢？是因为DEX版本的也只有第一次会解压执行程序到 /data/dalvik-cache（针对PathClassLoader）或者optimizedDirectory(针对DexClassLoader）目录，之后也是直接读取目录下的的dex文件，所以第二次启动就和正常的差不多了。当然这只是简单的理解，实际生成的ODEX还有一定的优化作用。ClassLoader只能加载内部存储路径中的dex文件，所以这个路径必须为内部路径。

## Dalvik和ART的区别?

[1]ART能够把应用程序的字节码转为机器码，是Android所使用的一种新的虚拟机，与Dalvik不同在于：Dalvik采用的是JIT技术而ART采用Ahead-of-time(AOT)技术。ART同时也改善了性能，垃圾回收机制，应用程序出错以及性能分析

[2]ART引入了AOT这种预编译技术，在应用程序安装的过程中，ART就已经将大多数的字节码重新编译成机器码。应用程序运行过程中无需进行实时的编译工作，只需要进行直接调用，因此ART极大的**提高了应用程序的运行效率，同时也减少了手机的电量消耗**，在**垃圾回收机制等机制上也有了较大的提升**。

相对于Dalvik虚拟机模式。ART模式下Android应用程序的安**装需要消耗更多的时间**，同时也会占用更大的存储空间（指内部存储，用于存储编译后的代码），但节省了很多dalvik虚拟机用于实时编译的时间

**内存抖动是什么?哪些情况会导致内存抖动?**

内存抖动是由于短时间内有大量对象进出Young Generiation区导致的，它伴随着频繁的GC

避免发生内存抖动的几点建议：

- 尽量避免在循环体内创建对象，应该把对象创建移到循环体外。

- 注意自定义View的onDraw()方法会被频繁调用，所以在这里面不应该频繁的创建对象。

- 当需要大量使用Bitmap的时候，试着把它们缓存在数组中实现复用。

- 对于**能够复用**的对象，同理可以使用对象池将它们缓存起来

**热更新是什么?**

# 操作系统相关

## 进程与线程之间的区别

**定义：**

（1）**进程**是具有一定独立功能的程序关于某个数据集合上的一次运行活动，而线程是其中的一个执行分支，进程是一个系统进行资源分配和调度的一个独立单位

（2）**线程**是进程的一个实体，是CPU调度和分派的基本单位，它是比进程更小的能独立运行的基本单位，线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源，

**资源共享性**：线程与同属一个进程的其他的线程共享进程所拥有的资源，进程间资源相互隔离。

**独立性：**进程独立运行在地址空间，但线程之间没有单独地址空间，线程必需依赖进程才能存活。

## 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

**Binder机制的原理?**

原理：  
跨进程通信是需要内核空间做支持的。传统的 IPC 机制如管道、Socket 都是内核的一部分，因此通过内核支持来实现进程间通信自然是没问题的。但是 Binder 并不是 Linux 系统内核的一部分，那怎么办呢？这就得益于 Linux 的动态内核可加载模块（Loadable Kernel Module，LKM）的机制；模块是具有独立功能的程序，它可以被单独编译，但是不能独立运行。它在运行时被链接到内核作为内核的一部分运行。这样，Android 系统就可以通过动态添加一个内核模块运行在内核空间，用户进程之间通过这个内核模块作为桥梁来实现通信。  
在 Android 系统中，这个运行在内核空间，负责各个用户进程通过 Binder 实现通信的内核模块就叫 Binder 驱动（Binder Dirver）。

内存映射：  
Binder IPC 机制中涉及到的内存映射通过 mmap() 来实现，mmap() 是操作系统中一种内存映射的方法。内存映射简单的讲就是将用户空间的一块内存区域映射到内核空间。映射关系建立后，用户对这块内存区域的修改可以直接反应到内核空间；反之内核空间对这段区域的修改也能直接反应到用户空间。

Binder IPC 通信过程：  
首先 Binder 驱动在内核空间创建一个数据接收缓存区；  
接着在内核空间开辟一块内核缓存区，建立内核缓存区和内核中数据接收缓存区之间的映射关系，以及内核中数据接收缓存区和接收进程用户空间地址的映射关系；  
发送方进程通过系统调用 copyfromuser() 将数据 copy 到内核中的内核缓存区，由于内核缓存区和接收进程的用户空间存在内存映射，因此也就相当于把数据发送到了接收进程的用户空间，这样便完成了一次进程间的通信。

Binder通讯模型：  
Binder是基于C/S架构的，**包含4个角色：Client、Server、Binder驱动ServiceManager**。

1、Binder驱动：类似网络通信中的**路由器**，负责将Client的请求转发到具体的Server中执行，并将Server返回的数据传回给Client。  
2️、ServiceManager：类似网络通信中的**DNS服务器**，负责将Client请求的Binder描述符转化为具体的Server地址，以便Binder驱动能够转发给具体的Server。Server如需提供Binder服务，需要向ServiceManager注册。

通信过程：  
1、Server向ServiceManager注册。Server通过Binder驱动向ServiceManager注册，声明可以对外提供服务。ServiceManager中会保留一份映射表。  
2️、Client向ServiceManager请求Server的Binder引用。Client想要请求Server的数据时，需要先通过Binder驱动向ServiceManager请求Server的Binder引用（代理对象）。  
3、向具体的Server发送请求。Client拿到这个Binder代理对象后，就可以通过Binder驱动和Server进行通信了。  
4、Server返回结果。Server响应请求后，需要再次通过Binder驱动将结果返回给Client。

ServiceManager是一个**独立的进程**，那么Server与ServiceManager通讯是靠什么呢:  
当Android系统启动后，会创建一个名称为ServiceManager的进程，这个进程通过一个约定的命令BINDERSETCONTEXT\_MGR向Binder驱动注册，**申请**成为为ServiceManager，**Binder驱动会自动为ServiceManager创建一个Binder实体。并且这个Binder实体的引用在所有的Client中都为0**，也就说各个Client通过这个0号引用就可以和ServiceManager进行通信。Server通过号引用向ServiceManager进行注册，Client通过0号引用就可以获取到要通信的Server的Binder引用。

Binder机制优点：  
1️、只需要进行一次数据拷贝，性能上仅次于共享内存  
2️、基于C/S架构，职责明确，架构清晰，稳定性较好  
3️、为每个App分配UID，UID是鉴别进程身份的标志，安全性较好

## 解释下Android程序运行时权限和文件系统权限的区别

文件的系统权限，是由Linux系统规定的，只读，读写等  
运行时权限，是对于某个系统上的APP来访问权限，允许，拒绝，询问，该功能可以防止非法的程序访问敏感信息。

## FrameWork工作方式及原理，activity是如何生成一个view的，机制是什么

1、Framework是Android系统对Linux，lib库等封装，提供WMS,AMS,Binder等机制，handler-message机制等方式，供APP使用简单来说framework就是提供APP生存的环境。

2、Activity在attach方法的时候，会创建一个phoneWindow。onCreate中的setContentView方法来创建一个window，会通过getDecorView生成view对象，通过addView方法，会把layout中的布局加载进来

## 谈谈你对android系统（体系）架构的理解

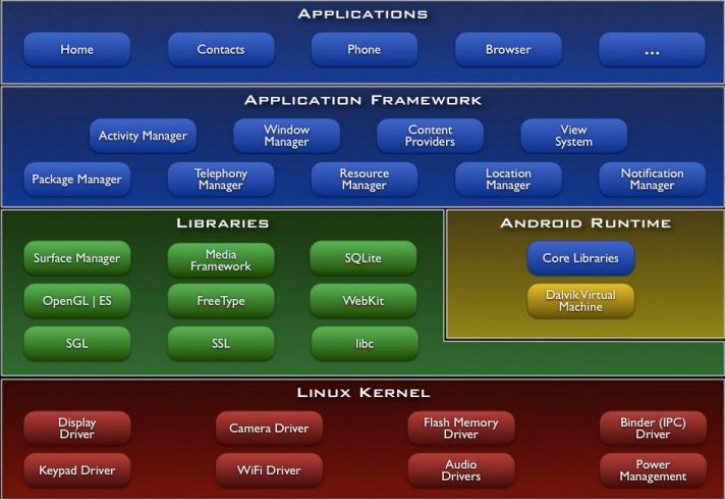
Linux操作系统架构，从下往上:

[1]应用程序层：包括系统应用以及第三方应用。

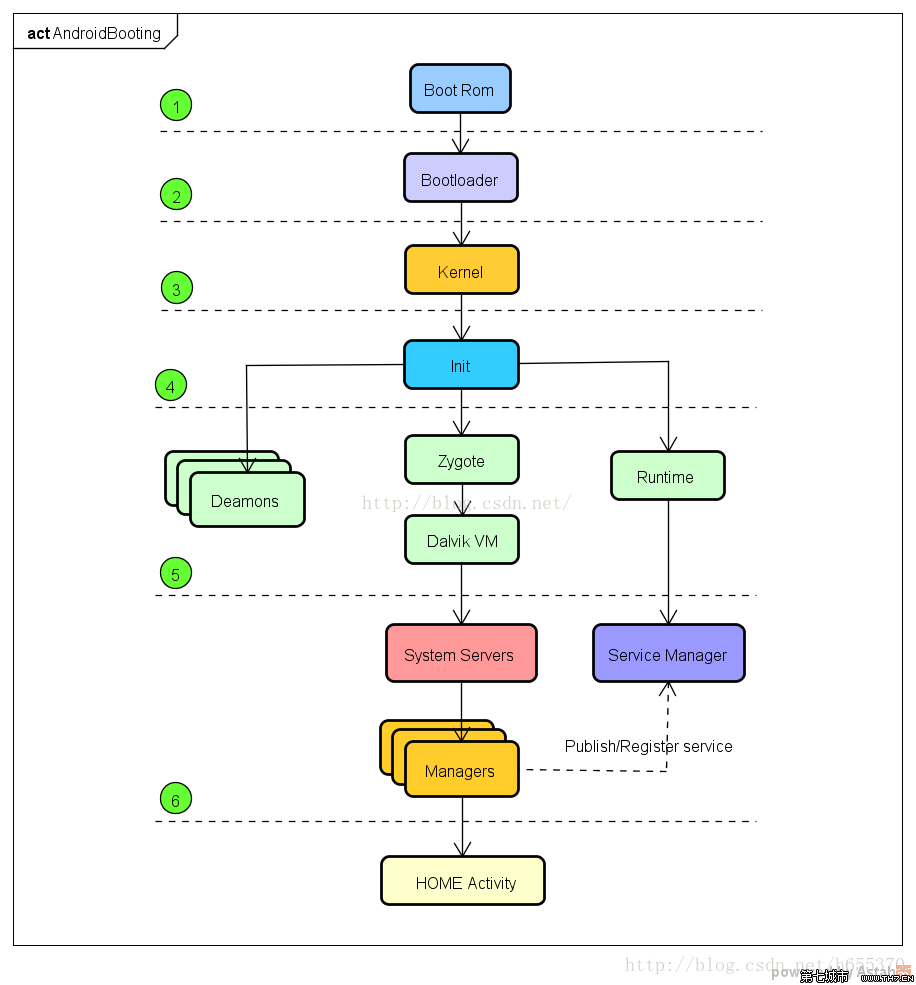
[2]应用程序框架：提供应用开发所必须的一些API框架，是软件复用的重要手段

[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开机启动流程?



**Step 1. Boot Rom**

 当按开机键的时候，引导芯片开始从固化在ROM的预设代码开始执行，然后加载引导程序加载到RAM。

**Step 2. Bootloader**

 BootLoader，又称为引导程序。它是在操作系统运行之前运行的一段程序，是运行的第一个程序。主要有检查RAM，初始化硬件参数等功能，当然它的最终**目的是把操作系统给拉起来**

其实Bootloader主要的必须的作用只有一个：就是把操**作系统映像文件拷贝到RAM中**去，然后跳转到它的入口处去执行，我们称之为**启动加载模式**，该过程没有用户的介入，是它正常工作的模式。它的步骤如下：

**Stage1:**

硬件设备初始化。

为stage2的执行及随后内核的执行准备好基本的硬件环境

为加载stage2 准备ram空间。

为了获得更好的执行速度，通常吧stage2加载到ram中执行

复制stage2的代码到ram中

设置好堆栈

跳转到stage2的c程序入口

**Stage2：**

初始化本阶段要使用的硬件设备

检测系统内存映射

将内核映像和根文件系统映像从flash读到ram中

为内核设置启动参数

调用内核

**Step 3. 初始化Kernel**

 接着就进入C语言编写的结构无关的代码了。这个入口的函数是start\_kernel函数。start\_kernel 函数完成了内核的大部分初始化工作。实际上，可以将start\_kernel 函数看做内核的main函数。start\_kernel函数执行到最后调用了reset\_init函数进行后续的初始化。 reset\_init函数最主要的任务就是启动内核线程kernel\_init。kernel\_init函数将完成**设备驱动程序的初始化**，并**调用init\_post函数启动用户空间的init进程**。到init\_post函数为止，内核的初始化已经基本完成

**Step 4. init进程**

 当初始化内核之后，就会启动一个相当重要的**祖先进程**，也就是init进程，在Linux中所有的进程都是由init进程直接或间接fork出来的。init进程负责创建系统中最关键的几个核心daemon(守护)进程，尤其是**zygote**和**servicemanager**。前者是android启动的第一个dalvik 虚拟机，它将负责启动Java世界的进程；后者是BInder通信的基础。另外，它还提供了property service（属性服务），类似于windows系统的注册表服务。

 在Android系统中，会有个init.rc脚本。init进程一启动就会读取并解析这个脚本文件，把其中的元素整理成自己的数据结构（链表）。

 换句话说， **init进程负责解析init.rc配置文件，开启系统守护进程**。两个最重要的守护进程是zygote进程和servicemanager，zygote是Android启动的第一个Dalvik虚拟机，servicemanager是**Binder通讯的基础**。

**Step 5. Zygote进程**

 当init进程创建之后，会fork出一个**Zygote进程，这个进程是所有Java进程的父进程**。我们知道，Linux是基于C的，而Android是基于Java的（当然底层也是C）。所以这里就会fork出一个Zygote Java进程用来fork出其他的进程。在zygote开启的时候，会调用ZygoteInit.main()进行初始化。下面我们看一段ZygoteInit.main()源码：

zygote虚拟机启动子进程system\_server，同时也可以看出zygote中定义了一个Socket，用于接收ActivityManagerService启动应用程序的请求。

**Step 6. SystemServer进程**

 前面ZygoteInit.java里面通过startSystemServer() fork出了SystemServer进程，这个进程在整个的Android中非常重要，它和Zygote进程一样，是Android Framework层的两大重要进程。**系统里面重要的服务都是在这个进程里面开启的**，例如AMS, WindowsManager, PackageManagerService等等**都是由这个SystemServer fork出来**的。在下面SystemServer 的代码中可以看到，这些服务如何开启和具体开启了哪些服务。

文件路径：/frameworks/base/services/java/com/android/server/SystemServer.java

 从SystemServer.java文件代码中可以看出，在SystemServer进程开启的时候，就会初始化ActivityManagerService 。同时，会加载本地系统的服务库，调用createSystemContext()创建系统上下文，创建ActivityThread及开启各种服务等等。

 也就是说**在system\_server中开启了核心系统服务，并将系统服务添加到ServiceManager中，然后系统进入SystemReady状态**。

**Step 7. Home Activity**

 上面**ActivityManagerService 开启之后，会调用finishBooting()** ，完成引导过程，**同时发送开机广播**。

文件路径：/frameworks/base/services/java/com/android/server/am/ActivityManagerService.java

 之后就会启动Home程序，完成系统界面的加载与显示。其实这一步远比上面所说的复杂，具体步骤是：

1、在systemReady状态，ActivityManagerService会与zygote的Socket通信，请求启动Home。

2、zygote收到AMS的连接请求后，执行runSelectLoopMode处理请求。

3、zygote处理请求会通过**forkAndSpecialize**启动新的应用进程，并最终启动Home。Android的开机启动流程就完成了。

**开机启动流程总结：**

**Step1 系统加电**，执行Bootloader。Bootloader负责初始化软件运行所需要的最小硬件环境，最后加载内核到内存。

**Step2 内核加载进内存后**，将首先进入内核引导阶段，在内核引导阶段的最后，调用start\_kernel进入内核启动阶段。start\_kernel最终启动用户空间的init程序。

**Step3 init程序负责解析init.rc配置文件，开启系统守护进程**。**两个**最重要的守护进程是**zygote进程**和**ServiceManager**，zygote是Android启动的第一个Dalvik虚拟机，ServiceManager是Binder通讯的基础。

**Step4 zygote虚拟机启动子进程system\_server**，在system\_server中开启了**核心系统服务**，并将系统服务**添加到ServiceManager**中，然后系统进入SystemReady状态。

**Step5 在SystemReady状态**，ActivityManagerService与zygote中的**socket**通信，通过zygote启动home应用，进入系统桌面。

从Step3开始，init启动后，上层的实现。

Step1 init启动的核心Daemon服务包括Android的第一个Dalvik虚拟机zygote。

Step2 zygote定义一个socket，用于接受ActivityManagerService启动应用的请求。

Step3 zygote通过fork系统调用创建system\_server进程

Step4 在system\_server进程中，将会启动系统核心服务以及其他服务。

Step5 系统服务启动后会注册到ServiceManager中，用于Binder通信。

Step6 ActivityManagerService进入systemReady状态。

Step7 在systemReady状态，ActivityManagerService会与zygote的Socket通信，请求启动Home。

Step8 zygote收到AMS的连接请求后，执行runSelectLoopMode处理请求。

Step9 zygote处理请求会通过forkAndSpecialize启动新的应用进程，并最终启动Home。

# Android基础

## SharedPreferences apply与commit的区别？

1、apply没有返回值，而commit有返回值。

2、apply是将修改数据原子提交到内存, 而后异步真正提交到硬件磁盘, 而commit是直接同步提交到硬件磁盘，所以很明显apply的效率要高一些。

3、apply方法不会提示任何失败的提示

## 如何使用AIDL？

1、在aidl文件中定义供客户端调用的接口。

2、在服务端的Service中实现通过aidl生成接口中的静态抽象类Stub，并在onBind（）方法中返回一个相应对象。

3、将aidl文件包括完整的包拷贝至客户端，通过Context#bindServer（）方法绑定服务端Service并将返回的Binder对象转换为adil接口对象（通过Stub.asInterface()方法）即可进行通信。

## View中getRowX和getX的区别？

getRowX：触摸点相对于屏幕左上角的X和Y坐标

getX:触摸点相对于当前View的左上角的X和Y坐标

## Android的五大布局（虽然时过境迁，但还是有些傻逼要问）

**LinearLayout**：线性布局按照垂直或者水平的顺序依次排列子元素  
**FrameLayout**：单帧布局所以的子元素都不能被指定放置位置，全部放于这块区域的左上角，后面的子元素直接覆盖在前面的子元素之上  
**RelativeLayout**：相对布局按照各个子元素之间的位置关系完成布局  
**AbsolutLayout**：绝对布局子元素的layout\_x和layout\_y属性将生效，用于描述该子元素的坐标位置  
**TableLayout**：表格布局一个TableLayout由许多TableRow组成，一个TableRow就代表一行

## 怎么优化ListView性能优化

1.在adapter中的getView方法中尽量少使用逻辑尽最大可能避免GC

2.滑动的时候不加载图片

3.使用viewHolder

4.将ListView的scrollingCache和animateCache设置为false  
5.item的布局层级越少越好  
6.Adapter的getView方法里面convertView使用setTag和getTag方式

## 设计一个图片缓存加载机制

先检查在内存中是否存在该图片，在检查在硬盘中是否存在该图片，最后在网络上获取图片

## 视频播放插件

（1）MediaPlayer  
（2）SurfaceView

## 42.监听listView和GridView页面滑动到底部

重写onScrollStateChanged方法获取可见item的数量  
重写onScroll方法获取可见item的数量和项目列表中适配器的数量

## Android与js如何交互？

**一、JS调用Java**：

1、通过WebView的

/\*\* **@param** object the Java object to inject into this WebView's JavaScript

\* context. Null values are ignored.

\* **@param** name the name used to expose the object in JavaScript

\*/

public void addJavascriptInterface(Object object, String name)

将一个java对象暴露给JS。

2、通过重写WebViewClient 的shouldOverrideUrlLoading (WebView view, String url)方法回调拦截 url，根据url中的信息作出响应。

3、通过重写WebChromeClient的**onJsAlert**（）、**onJsConfirm**（）、**onJsPrompt**（）方法回调拦截JS对话框alert（）、confirm（）、prompt（）消息。

【1】交互的接口函数：mWebView.addJavascriptInerface（getHtmlObject（），“interface”）（注：getHtmlObject（）是获取定义有@JavascriptInterface注解方法的对象，“interface”供js调用的，如在js中调用某方法interface.xxx()）。

（注：Js 调用 Java 方法时，不是在主线程 (Thread Name：main) 中运行的，而是在一个名为 JavaBridge 的线程中执行的）。

**二Android调JS：**

**方式1：**

1、 Android允许实现脚本设置：webSettings.addJavascriptEnabled（true）

2、webView.loadUrl("javascript:androidCallJs()")(注：“javascript:”是必备标识。 “androidCallJs”是在HTML里定义的方法，调用JS方法时一定要在onPageFinished（）回调之后才能调用，否则不会调用)

**方式2：**

webView. evaluateJavascript(String script, ValueCallback<String> resultCallback)执行该方法**不会使页面刷新**，而第一种方法（loadUrl ）的执行则会，并且更容易**获取返回值**。

mWebView.evaluateJavascript("javascript:callJSReturn('方式二调用JS')", new ValueCallback<String>() {

@Override

public void onReceiveValue(String value) {

//此处为 js 返回的结果

}

});

## 更新单条的listView属性

使用getFirstVisiblePosition()\getLastVisiblePosition()\getItemAtPosition(int position)：该方法返回当前状态下position位置上listView的convertView

## 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。

Handler工作原理

1、MessageQueue是传送带，负责Message队列的传送与管理

2、Looper不断地把消息从消息队列里面取出来，交给Handler来处理

3、Message是每一个具体的消息

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。

# 性能优化

# ddms和traceView的区别

1， ddms：是android开发环境中的dalvik虚拟机调试监控服务；  
ddms能够提供，测试设备截屏，针对特定的进程查看正在运行的线程以及堆信息，Logcat，广播状态信息，模拟电话呼叫，接收sms，虚拟地理坐标等。  
2，traceView是android平台配备的性能分析的工具；它可以通过图形化让我们了解要跟踪的程序的性能，并且能具体到方法。  
区别：ddms是一个程序执行查看器，在里面可以看见线程和堆栈等信息，traceView是程序性能分析器

## 如何检测布局或内存有问题，运用到哪些工具？

1. **布局分析：**

（1）AndroidStudio3.0中**Layout Inspector** 替代Hierarchy Viewer(或者使用 adb shell dumpsys activity top)

（2）开发者模式->调试GPU过度绘制->显示过度绘制区域

**没有颜色**： 意味着没有overdraw。像素只画了一次。

**蓝色**： 意味着overdraw 1倍。像素绘制了两次。大片的蓝色还是可以接受的（若整个窗口是蓝色的，可以摆脱一层）。

**绿色：** 意味着overdraw 2倍。像素绘制了三次。中等大小的绿色区域是可以接受的但你应该尝试优化、减少它们。

**浅红**： 意味着overdraw 3倍。像素绘制了四次，小范围可以接受。

**暗红**： 意味着overdraw 4倍。像素绘制了五次或者更多。这是错误的，要修复它们。



Overdraw产生的原因无外乎三种：

**1、复杂的Layout层级**

**2、重叠的View**

**3、重叠的背景**

**2、内存分析：**

（1）使用Android Studio提供的Memory Profiler

（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。

## 性能优化有哪些手段？（至少要提及布局优化、内存优化，其它还有IO优化、电池优化、数据库操作优化、网络请求优化、流量分析…）

**一、布局优化**

防止过度绘制：

1、使用include标签，通过layout属性复用相同的布局。

2、使用merge标签，去除同类的视图。

3、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。

4、优先使用ConstraintLayout或RelativeLayout，因为这样可以大大减少视图的层级。

5、去除布局中重叠的背景或View

**二、内存优化（三个角度分析：1、减少内存占用 2、复用资源 3、增加可用内存）**

APP设计以及代码编写阶段都应该考虑内存优化：

**1、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。

**2、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（**HandlerThread**）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。

**3、内存紧张的时候释放资源**（例如UI隐藏的时候释放资源等）。

四大组件都实现了**ComponentCallbacks**/**ComponentCallbacks2（4.0后推荐使用，它更精细）**接口，我们可以重写onTrimMemory/onLowMemory来做一些资源释放操作，当然也可以自定义实现此接口，然后在Context中注册（要记得反注册），例如：

private MemoryTrim mMemoryTrim;

@Override

public void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

mMemoryTrim = new MemoryTrim();

registerComponentCallbacks(mMemoryTrim);

}

@Override

protected void onDestroy() {

super.onDestroy();

unregisterComponentCallbacks(mMemoryTrim);

}

public class MemoryTrim implements ComponentCallbacks2 {

@Override

public void onTrimMemory(int level) {

/\*\*

在onTrimMemory (int level)中可以根据系统的内存状况及时调整App内存占用，**提升用户体验或让App存活更久**

应用运行在前台 level从低到高

**TRIM\_MEMORY\_RUNNING\_MODERATE（5）** 系统处于低内存状态，app正在运行且不会被杀死

**TRIM\_MEMORY\_RUNNING\_LOW（10）** 系统处于更低内存状态，app正在运行且不会被杀死，可以释放不用的资源提升app性能

**TRIM\_MEMORY\_RUNNING\_CRITICAL（15）** 系统处于极低的内存状态，app仍然不会被杀死。但如果不是放资源，系统开始杀后台进程，此时app应该清理一些资源

应用可见性变化时

**TRIM\_MEMORY\_UI\_HIDDEN（20）** 你的用户界面不再可见，此时应该释放仅仅使用在UI上的大资源

应用进程位于后台LRU（least-recently used）列表中时

**TRIM\_MEMORY\_BACKGROUND（40）** 系统正处于低内存状态，app进程位于LRU List开始附近，尽管app进程被杀死的概率低，系统可能已经开始杀在LRU List中的后台进程,此时**应该释放一些资源(释放哪些呢？)**，防止被杀的几率

**TRIM\_MEMORY\_MODERATE（60）** 系统正处于低内存状态，app进程位于LRU List中部附近，如果系统进一步内存紧张，app进程可能会被杀掉

**TRIM\_MEMORY\_COMPLETE（80）** 系统正处于低内存状态，app进程是首先被杀进程之一，如果系统现在没有恢复内存，应该立即释放对app不重要的所有内容

\*/

}

@Override

public void onConfigurationChanged(Configuration newConfig) {}

@Override

public void onLowMemory() {}

}

**4、通过Manifest中对Application配置更大的内存**，但是一般不推荐

android:largeHeap="true"

**5、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。

6、使用优化的容器，SparseArray等

7、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。

android5.0的SVG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

9、适当时机使用线程池。

**三、网络优化**

**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**、

4、请求数据Body可以利用**压缩**算法Gzip来进行压缩，

5、使用J**SON 代替 XML**

6、图片下载使用缩略图

7、图片（文件）的上传失败率比较高，不仅仅因为大文件，同时带宽、时延、稳定性等因素在此场景下的影响也更加明显；避免整文件传输，采用分片传输；根据网络类型以及传输过程中的变化动态的修改分片大小；每个分片失败重传的机会。

8、动态修改图片，应用要求按指定的渲染大小来从网络上请求图片

9、处于Wifi状态下时我们可以提前获取更多的数据等

10、检测网络改变，然后修改APP的行为

11、针对轮询请求要考虑activty生命周期以及设计适当的轮询机制  
网络质量不是固定不变的，它会随着地理位置，网络流量和当地人口密度发生改变。APP 应该检测网络中的改变并且相应地调整带宽，让APP可以更好地适应网络质量，可以实现下面的这些方法检测网络状态：

ConnectivityManager> getActiveNetworkInfo()

ConnectivityManager> getNetworkCapabilities(Network)

TelephonyManager> getDataState()

随着网络质量的下降，减少请求的数量，随着网络质量的提升，你可以提高你的请求量到最优级别。

**四、电池优化**

**Android系统上App的电量消耗主要由cpu、wakelock、数据传输（流量和wifi）、wifi运行、gps、other senior组成，而耗电异常也是由于这几个模块的使用不当**

**分析工具：**

1. Battery Historian分析手机耗电神器
2. Profiler-Energy

# 简述一下Android事件分发机制

**事件分发就是指对TouchEvent的分发、拦截、消费处理的过程**。

一个完整的touch事件，由一个down事件，多个move事件，一个up事件组成。

Touch事件的一般传递流程Activity-->window(唯一实现类PhoneWindow)-->顶级View(DecorView)-->ViewGroup-->View。

**三个重要方法**：  
**1、dispatchTouchEvent（）**：该方法是对点击事件的分发，在这个方法中我们一般return super.dispatchTouchEvent,讲该事件分发下去。

**2、onInterceptTouchEvent（）**：该方法表示对Touch事件进行拦截，这是ViewGroup特有的方法，View没有。在ViewGroup中如果onInterceptTouchEvent返回true，表示将该事件拦截，那么事件将传递给ViewGroup的onTouchEvent方法处理。如果onInterceptTouchEvent方法返回true，事件将传递个子View的dispatchTouchEvent进行分发。

**3、onTouchEvent（）**:该方法表示对Touch事件进行消费，返回true表现消费，返回false表示不消费，那么该事件该事件将传递给父控件的onTouchEvent处理

**监听Touch事件**有两种方式，setOnTouchListener和直接重写三个方法：dispatchTouchEvent,onInterceptTouchEvent,onTouchEvent。  
使用setOnTouchListenre方式设置监听事件，该方式的优先级较高，如果在onTouchListener的onTouch方法中return true的话，那么onTouchEvent方法是接收不到该Touch事件的。而且因为onClickListener中的onClick方法实际上是在onTouchEvent中被调用的，所以Touch事件走不到onTouchEvent，Touch事件就不会生效

# 如何解决滑动冲突问题？

**第一步**：找到滑动冲突规则

**第二步**：决定采用内部拦截法还是外部拦截法

**第三步：**在自定义View的修改重写onTouchEnvent（）与onInterceptTouchEvent（）

# Apk打包流程及Apk安装流程：

**一、Apk打包流程：**

**1. 使用aapt工具打包资源文件，生成R.java文件**

打包资源的**工具是aapt**（The Android Asset Packaing Tool）(.\sdk\build-tools\25.0.0\aapt.exe)。

在这个过程中，项目中的AndroidManifest.xml文件和布局文件XML都会编译，然后生成相应的R.java，另外AndroidManifest.xml会被aapt编译成二进制。

存放在APP的res目录下的资源，该类资源在APP打包前大多会被编译，变成二进制文件，并会为每个该类文件赋予一个resource id。**对于该类资源的访问，应用层代码则是通过resource id进行访问的**。Android应用在编译过程中aapt工具会对资源文件进行编译，并生成一个**resource.arsc**文件，resource.arsc文件相当于一个文件索引表，记录了很多跟资源相关的信息。

**2. 处理aidl文件，生成相应的Java文件**

这一过程中使用到的**工具是aidl**（Android Interface Definition Language），即Android接口描述语言（~\sdk\build-tools\25.0.0\aidl.exe）。

aidl工具解析接口定义文件然后生成相应的Java代码接口供程序调用。如果在项目没有使用到aidl文件，则可以跳过这一步。

**3. 编译项目源代码，生成class文件**

项目中所有的Java代码，包括*R.java*和*.aidl*文件，都会变J**ava编译器（javac）**编译成*.class*文件，生成的class文件位于工程中的bin/classes目录下。

**4. 转换所有的class文件，生成classes.dex文件**

**dx工具**生成可供Android系统Dalvik虚拟机执行的classes.dex文件，该工具位于（E:\Documents\Android\sdk\build-tools\25.0.0\dx.bat）。

任何第三方的*libraries*和*.class*文件都会被转换成*.dex*文件。dx工具的主要工作是将Java字节码转成成Dalvik字节码、压缩常量池、消除冗余信息等。

**5. 打包生成APK文件**

所有没有编译的资源，如images、assets目录下资源（**该类文件是一些原始文件，APP打包时并不会对其进行编译，而是直接打包到APP中，对于这一类资源文件的访问，应用层代码需要通过文件名对其进行访问**）；编译过的资源和*.dex*文件都会被**apkbuilder**工具打包到最终的*.apk*文件中。

打包的工具apkbuilder位于 android-sdk/tools目录下。apkbuilder为一个脚本文件，实际调用的是（E:\Documents\Android\sdk\tools\lib）文件中的com.android.sdklib.build.ApkbuilderMain类。

**6. 对APK文件进行签名**

一旦APK文件生成，它必须被签名才能被安装在设备上。

在开发过程中，主要用到的就是两种签名的keystore。一种是用于调试的debug.keystore，它主要用于调试，在Eclipse或者Android Studio中直接run以后跑在手机上的就是使用的debug.keystore。另一种就是用于发布正式版本的keystore。

Android提供了两种对Apk的签名方式，一种是基于JAR的签名方式，另一种是基于Apk的签名方式，它们的主要区别在于使用的签名文件不一样：**jarsigner**使用keystore文件进行签名；**apksigner**除了支持使用keystore文件进行签名外，还支持直接指定pem证书文件和私钥进行签名。

**7. 对签名后的APK文件进行对齐处理**

为什么要对齐？<https://blog.csdn.net/zkf11387/article/details/7662450>

**zipalign优化**的最根本目的是帮助操作系统更高效率的根据请求索引资源，通过将apk中的未压缩数据进行字节对齐（一般为4字节对齐），允许系统使用mmap方法直接映射文件至内存空间，降低内存的消耗。

**APK安装流程：**

（1）**拷贝apk到指定的目录**：默认情况下，用户安装的apk首先会拷贝到/data/app下，**普通用户有访问/data/app目录的权限**，但系统出厂的apk文件会被放到/system分区下，包括/system/app，/system/vendor/app，以及/system/priv-app等，该分区需要root权限的用户才能访问。

（2）**加载apk、拷贝文件、创建应用的数据目录**：为了加快APP的启动速度，apk在安装的时候，会首先将APP的可执行文件（dex）拷贝到/data/dalvik-cache目录下，缓存起来。再在/data/data/目录下创建应用程序的数据目录（以应用包名命令），用来存放应用的**数据库、xml文件、cache、二进制的so动态库**等。

（3）**解析apk的AndroidManifest.xml文件**：在安装apk的过程中，会解析apk的AndroidManifest.xml文件，将apk的权限、应用包名、apk的安装位置、版本、userID等重要信息保存在/data/system/packages.xml文件中。这些操作都是在**PackageManagerService**中完成

的。

（4）发出广播，显示icon图标：应用程序经过PMS中的逻辑处理后，相当于已经注册好了，如果想要在Android桌面上看到icon图标，则需要Launcher将系统中已经安装的程序展现在桌面上。

# 并发编程

## 说说你了解的一个线程安全队列。

分为BlockingQueQue（阻塞队列）和 ConcurrentLinkedQueQue（非阻塞队列）。  
以BlockingQueQue为例，BlockingQueQue作为线程容器，可以为线程同步提供有力的保障。  
BlockingQueQue有两种实现：  
1）基于数组的阻塞队列实现ArrayBlockingQueQue，内部维护了一个定长数组，以便缓存队列中的数据对象。  
2）基于链表的阻塞队列LinkedBlockingAQueQue，它内部维护了一个链表结构的数据缓冲队列。  
BlockingQueQue按照FIFO（先进先出）排列元素。  
注意：1，必须要使用take（）方法在获取时达成阻塞结果。  
2，使用poll（）方法将产生非阻塞效果。

## 如何保证线程安全：

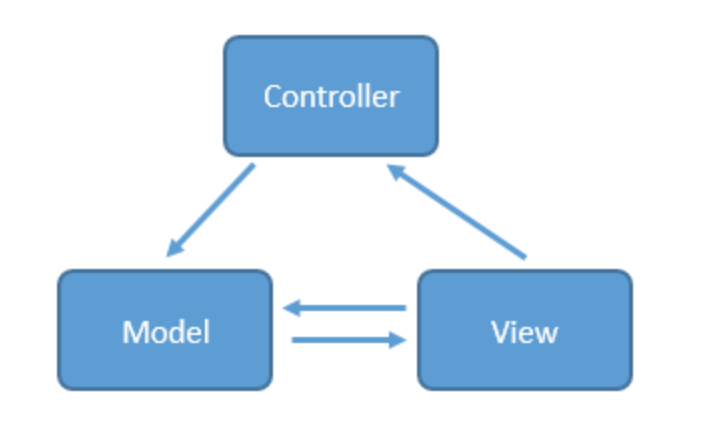
## 死锁的必要条件以及解决方案：

# APP架构设计

## MVC、MVP、MVVM的区别？

### MVC

MVC，Model View Controller，是软件架构中最常见的一种框架，简单来说就是通过controller的控制去**操作model层**的数据，并且返回给view层展示，呈现三角关系，具体见下图



当用户出发事件的时候，view层会发送指令到controller层，接着controller去通知model层更新数据，model层更新完数据以后直接显示在view层上，这就是MVC的工作原理。

那具体到Android上是怎么样一个情况呢？

1. 对于原生的Android项目来说，layout.xml里面的xml文件就对应于MVC的view层，里面都是一些view的布局代码，
2. 而各种java bean，还有一些类似repository类就对应于model层，
3. 至于controller，当然就是各种activity咯。大家可以试着套用我上面说的MVC的工作原理是理解。比如你的界面有一个按钮，按下这个按钮去网络上下载一个文件，这个按钮是view层的，是使用xml来写的，而那些和网络连接相关的代码写在其他类里，比如你可以写一个专门的networkHelper类，这个就是model层，那怎么连接这两层呢？是通过button.setOnClickListener()这个函数，这个函数就写在了activity中，对应于controller层。是不是很清晰。

**MVC存在的问题:**

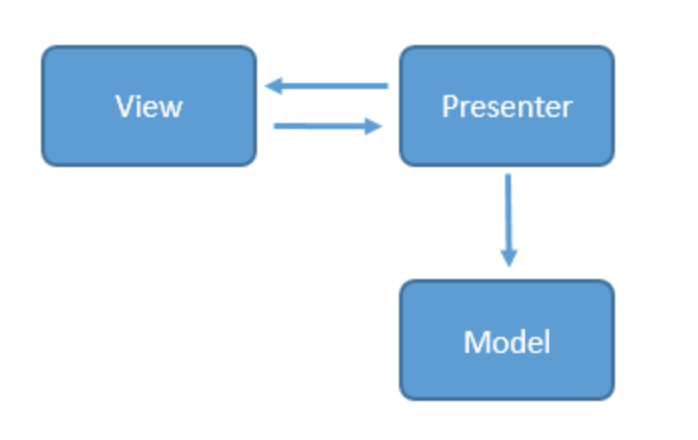
（1）**xml作为view层，控制能力实在太弱了**，你想去动态的改变一个页面的背景，或者动态的隐藏/显示一个按钮，这些都没办法在xml中做，**只能把代码写在activity中，造成了activity既是controller层**，又是view层的这样一个窘境。大家回想一下自己写的代码，如果是一个逻辑很复杂的页面，activity或者fragment是不是动辄上千行呢？这样不仅写起来麻烦，维护起来更是噩梦。（当然看过Android源码的同学其实会发现上千行的代码不算啥，一个RecyclerView.class的代码都快上万行了呢）

（2）MVC还有一个重要的缺陷，大家看上面那幅图，**view层和model层是相互可知的，这意味着两层之间存在耦合**，耦合对于一个大型程序来说是非常致命的，因为这表示开发，测试，维护都需要花大量的精力。

正因为MVC有这样那样的缺点，所以才演化出了MVP和MVVM这两种框架。

### MVP

MVP作为MVC的演化，解决了MVC不少的缺点，对于Android来说，MVP的model层相对于MVC是一样的，**而activity和fragment不再是controller层**，**而是纯粹的view层**，所有关于**用户事件的转发全部交由presenter层处理**。下面还是让我们看图



从图中就可以看出，**MVP与MVC最明显的差别就是view层和model层不再相互可知**，完全的解耦，取而代之的presenter层充当了桥梁的作用，用于操作**view层发出的事件传递到presenter层中，presenter层去操作model层，并且将数据返回给view层**，整个过程中view层和model层完全没有联系。看到这里大家可能会问，虽然view层和model层解耦了，**但是view层和presenter层不是耦合在一起了吗？**其实不是的，对于**view层和presenter层的通信，我们是通过接口实现的**，具体的意思就是说我们的**activity，fragment可以去实现实现定义好的接口，而在对应的presenter中通过接口调用方法**。不仅如此，我们还可以编写测试用的View，模拟用户的各种操作，从而实现对Presenter的测试。这就解决了MVC模式中测试，维护难的问题。

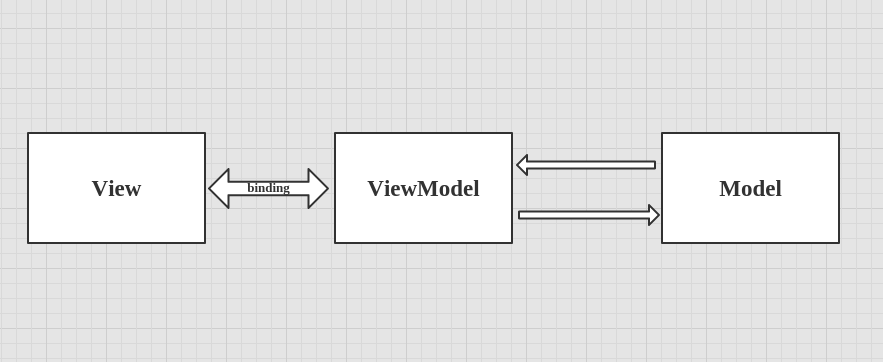
和MVC最大的不同，MVP把activity作为了view层，通过代码也可以看到，整个activity没有任何和model层相关的逻辑代码，取而代之的是把代码放到了presenter层中，presenter获取了model层的数据之后，通过接口的形式将view层需要的数据返回给

这样的好处是什么呢？首先，activity的代码逻辑减少了，其次，view层和model层完全解耦，具体来说，如果你需要测试一个http请求是否顺利，你不需要写一个activity，只需要写一个java类，实现对应的接口，presenter获取了数据自然会调用相应的方法，相应的，你也可以自己在presenter中mock数据，分发给view层，用来测试布局是否正确。

当然，其实**最好的方式是使用fragment作为view层，而activity则是用于创建view层(fragment)和presenter层(presenter)的一个控制器**。

### MVVM

MVVM最早是由微软提出的



从图中看出，它和MVP的区别貌似不大，只不过是**presenter层换成了viewmodel**层，还有一点就是**view层和viewmodel层是相互绑定的关系，这意味着当你更新viewmodel层的数据的时候，view层会相应的变动ui**。

### MVP

MVP的问题在于，由于我们使用了接口的方式去连接view层和presenter层，这样就导致了一个问题，如果你有一个逻辑很复杂的页面，你的接口会有很多，十几二十个都不足为奇。想象一个app中有很多个这样复杂的页面，维护接口的成本就会非常的大。

这个问题的解决方案就是你得根据自己的业务逻辑去斟酌着写接口。你可以定义一些基类接口，把一些公共的逻辑，比如网络请求成功失败，toast等等放在里面，之后你再定义新的接口的时候可以继承自那些基类，这样会好不少。

### MVVM

MVVM的问题呢，其实和MVC有一点像。data binding框架解决了数据绑定的问题，但是view层还是会过重，**activity在MVVM中应该是view层**的，但是里面却和MVC一样写了对model的处理。有人会说你可以把对model的处理放到viewmodel层中，这样不是更符合MVVM的设计理念吗？这样确实可以，但是progressDialog的show和dismiss呢？你怎么在viewmodel层中控制？这是view层的东西啊，而且在xml中也没有，我相信会有解决的方案，但是我们有没有一种更加便捷的方式呢？

其实，真正的最佳实践都是人想出来的，我们为何不结合一下MVP和MVVM的特点呢？其实谷歌已经做了这样的事，大家可以看下[这个](https://github.com/googlesamples/android-architecture)。没错，就是**MVP+data binding**，我们可以使用presenter去做和model层的通信，并且使用data binding去轻松的bind   
作者：潇湘沐  
链接：http://www.imooc.com/article/254232  
来源：慕课网

# 算法

## 对称加密与非对称加密有什么区别，它们常见的算法有哪些？

**对称加密：**

**对称加密算法的特点是算法公开、计算量小、加密速度快、加密效率高**

加密（encryption）与解密（decryption）用的是同样的密钥（secret key）。对称加密有很多种算法，由于它效率很高，所以被广泛使用在很多加密协议的核心当中。

对称加密通常使用的是相对较小的密钥，一般小于256 bit。因为密钥越大，加密越强，但加密与解密的过程越慢。如果你只用1 bit来做这个密钥，那黑客们可以先试着用0来解密，不行的话就再用1解；但如果你的密钥有1 MB大，黑客们可能永远也无法破解，但加密和解密的过程要花费很长的时间。密钥的大小既要照顾到安全性，也要照顾到效率。

**对称加密的一大缺点是密钥的管理与分配**。

**原理应用**：对称加密算法的优点在于加解密的高速度和使用长密钥时的难破解性。假设两个用户需要使用对称加密方法加密然后交换数据，则用户最少需要2个密钥并交换使用，如果企业内用户有n个，则整个企业共需要n×(n-1) 个密钥，密钥的生成和分发将成为企业信息部门的恶梦。对称加密算法的安全性取决于加密密钥的保存情况，但要求企业中每一个持有密钥的人都保守秘密是不可能的，他们通常会有意无意的把密钥泄漏出去--如果一个用户使用的密钥被入侵者所获得，入侵者便可以读取该用户密钥加密的所有文档，如果整个企业共用一个加密密钥，那整个企业文档的保密性便无从谈起。

**具体算法**：DES算法，3DES算法，TDEA算法，Blowfish算法，RC5算法，IDEA算法。

**非对称加密**

非对称加密为数据的加密与解密提供了一个非常安全的方法，它使用了一对密钥，公钥（public key）和私钥（private key）。私钥只能由一方安全保管，不能外泄，而公钥则可以发给任何请求它的人。非对称加密使用这对密钥中的一个进行加密，而解密则需要另一个密钥。比如，你向银行请求公钥，银行将公钥发给你，你使用公钥对消息加密，那么**只有私钥的持有人--银行才能对你的消息解密**。**与对称加密不同的是，银行不需要将私钥通过网络发送出去，因此安全性大大提高**

**主要算法：**

RSA、Elgamal、背包算法、Rabin、HD,ECC（椭圆曲线加密算法）。

使用最广泛的是RSA算法，Elgamal是另一种常用的非对称加密算法。

## 实现冒泡排序

## 实现链表反转

public class Node {

private Object data;//数据域

private Node next;//指针域

public Node(Object data){

this.data = data;

}

public Node(Object data,Node next){

this.data = data;

this.next = next;

}

public Object getData() {

return data;

}

public void setData(Object data) {

this.data = data;

}

public Node getNext() {

return next;

}

public void setNext(Node next) {

this.next = next;

}

}

**反转**：

public static Node reverseListNode(Node head){

//单链表为空或只有一个节点，直接返回原单链表

if (head == null || head.getNext() == null){

return head;

}

//前一个节点指针

Node preNode = null;

//当前节点指针

Node curNode = head;

//下一个节点指针

Node nextNode = null;

while (curNode != null){

nextNode = curNode.getNext();//nextNode 指向下一个节点

curNode.setNext(preNode);//将当前节点next域指向前一个节点

preNode = curNode;//preNode 指针向后移动

curNode = nextNode;//curNode指针向后移动

}

return preNode;

}

## 如何实现一个定时调度和循环调度?

* 1. 使用Timer：

Timer myTimer = new Timer(); myTimer.schedule(new Worker（），1000)；//1秒后执行

myTimer.scheduleAtFixedRate(new Worker(),5000,1000);

//5秒后执行第一次，之后每隔1秒执行一次。

2、使用线程池

3、使用Handler、RxJava

# URL与URI的区别？

**URI 是统一资源标识符，而 URL 是统一资源定位符。**因此，笼统地说，每个 URL 都是 URI，但不一定每个 URI 都是 URL。这是因为 URI 还包括一个子类，即统一资源名称 (URN)，它命名资源但不指定如何定位资源。上面的 mailto、news 和 isbn URI 都是 URN 的示例。

**URI—Uniform Resource Identifier通用资源标志符**Web上可用的每种资源如HTML文档、图像、视频片段、程序等都是一个来URI来定位的  
URI一般由三部组成  
①访问资源的命名机制：Scheme  
②存放资源的主机名：Host  
③资源自身的名称，由路径表示，着重强调于资源。（Path）

**URL—Uniform Resource Location统一资源定位符**URL是Internet上用来描述信息资源的字符串，主要用在各种WWW客户程序和服务器程序上，特别是著名的Mosaic。  
采用URL可以用一种统一的格式来描述各种信息资源，包括文件、服务器的地址和目录等。  
URL一般由三部组成  
①协议(或称为服务方式)  
②存有该资源的主机IP地址(有时也包括端口号)  
③主机资源的具体地址。如目录和文件名等

## Unicode、GBK、UTF-8、UTF-1的区？

# Synchronized 修饰类，对象，块、静态方法的区别：

# synchronized与volatile、lock间的区别?

1. synchronized能修饰类(静态方法、变量)、方法、变量,而volatile只能修饰变量.
2. synchronized能对整个对象产生影响而volatile只能对一个对象产生影响.
3. volatile它被称为轻量级的 synchronized, 它比synchronized的使用和执行成本会更低，因为它不会引起线程的阻塞从而导致线程上下文的切换和调度
4. **实现原理不同:**

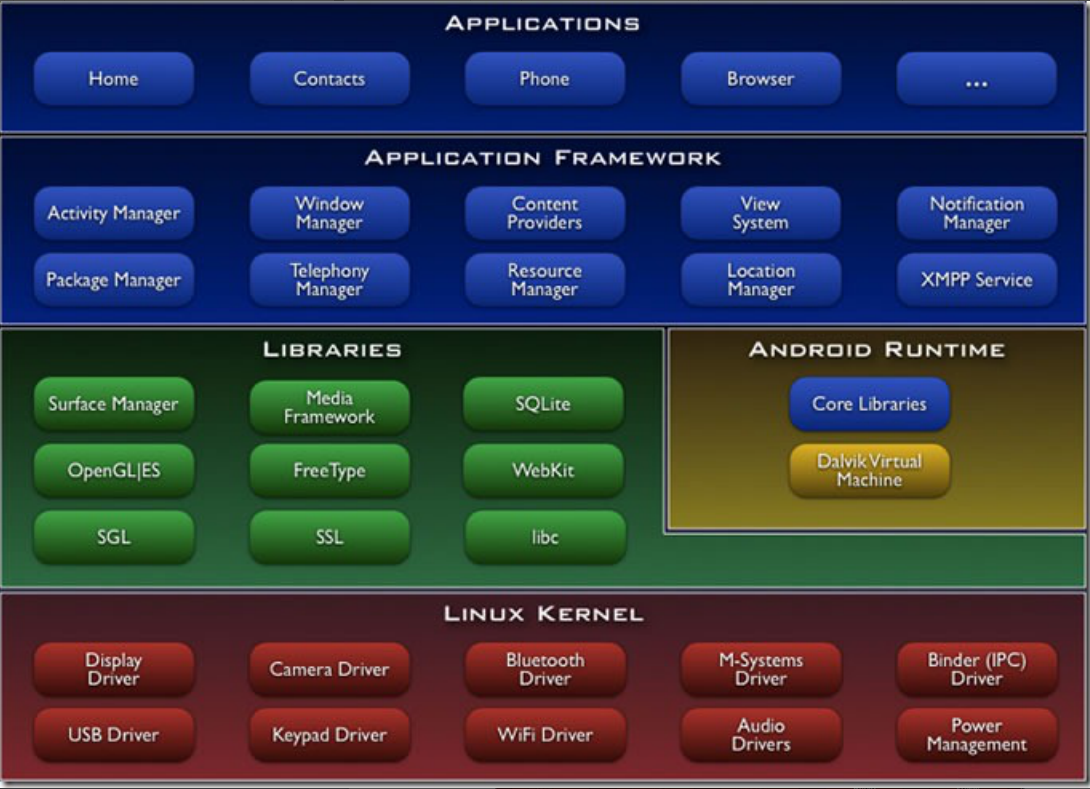
volatile 通过写操作会将线程本地内存值立即刷新到 主内存的共享变量中.读操作会从主内存的共享变量中读取,再更新本地内存变量的值,从而实现同步.

synchronized是通过锁机制实现,每个对象都有一个锁,只有获得锁的线程能访问同步块内,而其他线程只能阻塞等待.

1. synchronized满足了原子性，可见性和有序性，是并发编程正确执行的有效保障，而volatile只保证了可见性和有序性（禁止指令重排）
2. Lock不是Java语言内置的，synchronized是Java语言的关键字,由虚拟机为每个对象维护一个锁，因此是内置特性。Lock是一个普通类，通过这个类可以实现同步访问
3. 在使用上Lock需要手动添加到代码块中,而且必须通过手动释放锁(通常需要在finally中释放),而synchronized不需要,也不能它更加轻便灵活.( 当有多个线程读写文件时，读操作和写操作会发生冲突现象，写操作和写操作会发生冲突现象，但是读操作和读操作不会发生冲突现象。但是采用synchronized关键字来实现同步的话，就会导致一个问题：如果多个线程都只是进行读操作，所以当一个线程在进行读操作时，其他线程只能等待无法进行读操作)
4. Lock可以知道是否获取到锁(tryLock()返回boolean)而synchronied是无法办到的
5. Lock可以让等待锁的线程响应中断(lockInterruptibly())，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断

。

# 请绘制Android架构图：



# 简述synchronized,object,Monitor机制.

synchronized主要是用来进行同步操作，能帮助我们设计更安全的多线程程序。

1)当两个并发线程访问同一个对象object中的synchronized同步代码块时,一个时间内只能有一个线程得到执行。另一个线程只能等到当前线程执行完这个代码块才能执行该代码块。  
2）当一个线程访问object的一个synchronized同步代码块时，其他线程对object中所有其它synchronized同步代码块的访问将被阻塞。  
3）一个线程访问object中的synchronized代码块时，另外一个线程仍然可以访问object中的其它代码。

* 对于普通的方法同步，锁是当前实例对象。
* 对于静态方法同步，锁是当前类的class对象。
* 对于方法块同步，锁是synchronized括号里的对象。

monitor是一个**同步工具**，**相当于操作系统中的互斥量**。它内置于每一个object对象中，**相当于一个许可证。拿到许可证即可以进行操作**，没有拿到需要阻塞等待。

使用synchronized时，其实是通过锁对象的monitor的取用与释放来实现的。

### 简述happen-before规则。

happen-before其实就是一个保证而已，它保证一套语句对内存的写操作对于另一条语句是可见的。

**规则：**

1）程序顺序规则：一个线程中的每个操作，happen-befor于该线程中的任意后续操作。  
2）监视锁规则：对一个监视器锁的解锁，happen-before于随后对这个监视器锁的加锁。  
3)volatileb变量规则：对于一个volatile域的写，happen-before于任意后续对这个volatile域的读。  
4）传递性：如果A happen-before B ，且B happen-before C，那么A happen-before C。

# 简述AQS。

AQS全称为（Abstract Queued Synchronizer），这个类是在java.util.concurrent.locks下面。  
AQS的核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个就是AQS机制。

# 简述常用的JDK命令行以及工具。

jak常用工具在安装目录的bin目录下。  
工具

--编译工具：javac  
--运行工具：java  
--压缩工具：jar  
--文档生成工具：javadoc  
--字符编码转换工具：native2ascii

命令  
1，编译命令  
编译一个文件：在目录下面输入javac a.java(java文件名),编译多个文件中间用空格隔开；javac -d cls a.java是讲编译的文件放到同目录的cls文件夹中。  
2，运行命令  
java my.hello运行my包下面的hello编译文件。  
java -calsspath cl my.java  
3，压缩命令  
jar -cvf：压缩 //jar -cvf hello.jar a.class b.class c.class  
jar -xvf：解压 //jar -xvf cl\hello.jar 解压cl目录下的hello.jar  
4,文档生成工具 （将源文件中的doc帮助文件生成HTML文档）  
javadoc hello.java  
javadoc -d java\doc -version -author lcz\src\lcz.java  
5.字符编码转换工具  
native2ascii -encoding 编码 源文件名字 改变后的名字

# Proguard的作用?EntryPiont是什么?哪些东西不能混淆?

作用:

1、**压缩**(Shrink)：检测并移除代码中无用的类、字段、方法和特性

2、**优化**(Optimize)：对字节码进行优化，移除无用的指令。

3、**混淆**(Obfuscate)：使用a，b，c，d这样简短而无意义的名称，对类、字段和方法进行重命名。

4、**预检**(Preveirfy)：在Java平台上对处理后的代码进行预检，确保加载的class文件是可执行的。

**Entry Point:**

Entry Point是在ProGuard过程中不会被处理的类或方法。在压缩的步骤中，ProGuard会从上述的Entry Point开始递归遍历，搜索哪些类和类的成员在使用，对于没有被使用的类和类的成员，就会在压缩段丢弃，在接下来的优化过程中，那些非Entry Point的类、方法都会被设置为private、static或final，不使用的参数会被移除，此外，有些方法会被标记为内联的，在混淆的步骤中，ProGuard会对非Entry Point的类和方法进行重命名。

那么这个入口点怎么来呢？就是从ProGuard的配置文件来，只要这个配置了，那么就不会被移除。

**不能混淆:**

1、在AndroidManifest中配置的类，比如四大组件

2、JNI调用的方法

3、反射用到的类

4、WebView中JavaScript调用的方法

5、Layout文件引用到的自定义View

6、实体类

# 对AIDL的理解

AIDL:android interface definition language的缩写。  
AIDL是用来实现进程间通信的，可以帮我们实现发布以及调用远程服务。  
使用：  
1）服务端：创建一个Service用来监听客户端的连接请求，然后创建一个AIDL文件，将服务端暴露给客户端的接口在这个文件中声明，最后在Service中实现这个AIDL接口。  
2）客户端：首先绑定服务端的Service，绑定成功后将服务端返回的Binder对象转成AIDL接口所属的类型，接着就可以调用AIDL中的方法

# Serializable 和Parcelable 的区别；

（1）Parcelabe代码比Serializable多一些。  
（2）Parcelabe比Serializable速度高十倍以上。  
（3）Serializable只需要对某个类以及它的属性实现Serializable接口即可，无需实现方法。缺点是使用的反射，序列化的过程较慢，这种机制会在序列化的时候创建许多的临时对象。容易触发GC。  
（4）Parcable方法实现的原理是将一根完整的对象进行分解，而分解后的每一部分都是Intent所支持的数据类型，这样也就实现传递对象的功能。

（5）Serializable 实现是基于磁盘而Parcelabe则是基于内存，因此Serializable对象可以通过网络或其它方式传输而Parcelabe对象则不能。

# NDK相关

## JNI实现流程？

**情景1：java调C、C++方法：**

1、在java文件中以native关键字申明本地方法。

2、同过java -h 命名获得相应头文件。

3、根据生成的头文件在C/C++中实现方法

4、通过NDK编译C文件生成so文件

5、在调用native方法前通过System.loadLibrary()加载so文件

**情景2：C中调Java中方法：**

（1）

## 简单介绍一下NDK

一 ： NDK是一系列工具的集合  
NDK提供了一系列的工具，帮助开发者快速开发C或C++的动态库，并能自动将so和java应用一起打包成apk。这些工具对开发者的帮助是巨大的。  
NDK集成了交叉编译器，并提供了相应的mk文件隔离CPU，平台，ABI等差异。开发人员只需要简单修改mk文件（指出哪些文件需要编译，编译特性要求等）就可以创建出so。  
NDK可以自动将so和java应用一起打包，极大的减轻了开发人员的打包工作。  
二 ： NDK提供了一份稳定，功能有限的API头文件声明。  
Google明确声明该API是稳定的，在后续所有版本中都稳定支持当前发布的API。从该版本的NDK中看出，这些API支持的功能非常有限，包含：c标准库，标准数学库，压缩库，Log库。

## JNA是什么？

JNA（Java Native Access ）提供一组Java工具类用于在运行期间动态访问系统本地库（native library）而不需要编写任何Native/JNI代码。开发人员只要在一个java接口中描述目标native library的函数与结构，JNA将自动实现Java接口到native function的映射

# 计算机网络

## HTTP1.0和HTTP1.1和HTTP2.0的区别？

**1 HTTP1.0和HTTP1.1的区别**

1.1 **长连接**(Persistent Connection)

   HTTP1.1支持**长连接**和请求的**流水线处理**，在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟，在HTTP1.1中**默认开启长连接keep-alive**，一定程度上弥补了HTTP1.0每次请求都要创建连接的缺点。

1.2 **节约带宽**

   HTTP1.0中存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持**断点续传**功能。HTTP1.1支持**只发送header信息**（不带任何body信息），如果服务器认为客户端有权限请求服务器，则返回100，客户端接收到100才开始把请求**body**发送到服务器；如果返回401，客户端就可以不用发送请求body了节约了带宽。

1.3 **HOST域**

   在HTTP1.0中认为每台服务器都绑定一个唯一的IP地址，因此，请求消息中的URL并没有传递主机名（**hostname**），HTTP1.0没有host域。随着虚拟主机技术的发展，**在一台物理服务器上可以存在多个虚拟主机**（Multi-homed Web Servers），**并且它们共享一个IP地址**。HTTP1.1的请求消息和响应消息都支持host域，且请求消息中如果没有host域会报告一个错误（400 Bad Request）。

1.4**缓存处理**

    在HTTP1.0中主要使用header里的**If-Modified-Since（**对比的是服务器中文件修改时间，由于精度为秒，可能出现文件内容被修改后而修改时间为变化**）,Expires（这是一个绝对时间**客户端本地时间被修改修改后会造成混乱**）**来做为缓存判断的标准，HTTP1.1则引入了更多的缓存控制策略例如E tag（ETag返回的是较检码，可以保证每一个资源是唯一的），If-Unmodified-Since, If-Match, If-None-Match等更多可供选择的缓存头来控制缓存策略。

1.5**错误通知的管理**

     在HTTP1.1中新增了24个错误状态响应码，如409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。

**2 HTTP1.1和HTTP2.0的区别**

2.1 **多路复用**

    HTTP2.0使用了多路复用的技术，做到同一个连接并发处理多个请求，而且并发请求的数量比HTTP1.1大了好几个数量级。HTTP1.1也可以多建立几个TCP连接，来支持处理更多并发的请求，但是创建TCP连接本身也是有开销的。

2.2 **头部数据压缩**

    在HTTP1.1中，HTTP请求和响应都是由状态行、请求/响应头部、消息主体三部分组成。一般而言，**消息主体**都会经过gzip压缩，或者本身传输的就是压缩过后的二进制文件，**但状态行和头部却没有经过任何压缩**，直接以纯文本传输。随着Web功能越来越复杂，每个页面产生的请求数也越来越多，导致消耗在头部的流量越来越多，尤其是每次都要传输UserAgent、Cookie这类不会频繁变动的内容，完全是一种浪费。

   HTTP1.1不支持header数据的压缩，HTTP2.0使用HPACK算法对header的数据进行压缩，这样数据体积小了，在网络上传输就会更快。

2.3 **服务器推送**

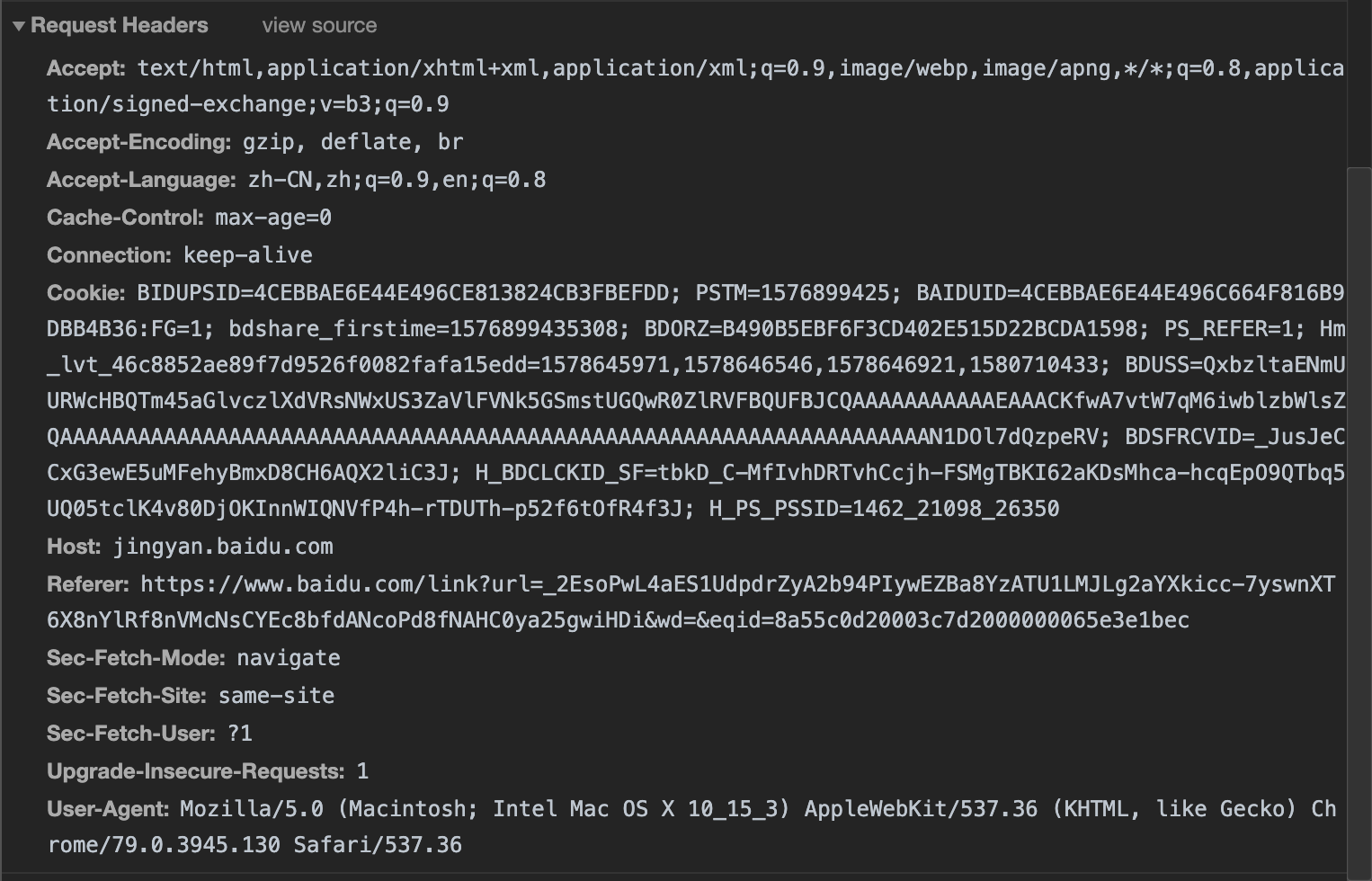
    服务端推送是一种在客户端请求之前发送数据的机制。网页使用了许多资源：HTML、样式表、脚本、图片等等。在HTTP1.1中这些资源每一个都必须明确地请求。这是一个很慢的过程。浏览器从获取HTML开始，然后在它解析和评估页面的时候，增量地获取更多的资源。因为服务器必须等待浏览器做每一个请求，网络经常是空闲的和未充分使用的。

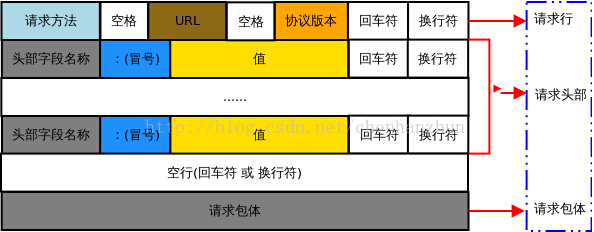
    为了改善延迟，HTTP2.0引入了server push，它允许服务端推送资源给浏览器，在浏览器明确地请求之前，免得客户端再次创建连接发送请求到服务器端获取。这样客户端可以直接从本地加载这些资源，不用再通过网络。

## Http请求及响应报文格式？

HTTP 请求报文由请求行、**请求头部、空行 和 请求包体** 4 个部分组成：

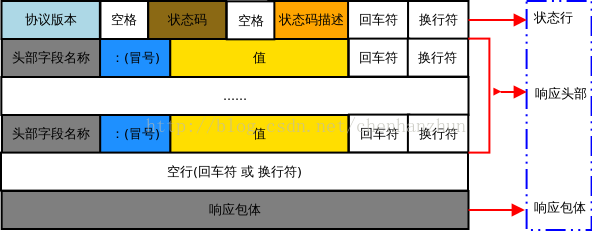
请求方法有 **OPTIONS、GET、HEAD、POST、PUT、DELETE、TRACE、CONNECT**





HTTP 响应报文由**状态行**、**响应头部、空行 和 响应包体** 4 个部分组成





## Get与Post的区别？

**1、从用法上：**

Get强调从服务器获取资源，这意味重复请求不应该有副作用，且资源有被缓存的可能性。Post强调提交数据给服务器，需要服务器处理，这相当于交给服务器一项任务，通常需要考虑重复请求带来的副作用并且数据不应该缓存。

**2、从安全性上**：

从攻击角度看，http协议都是采用明文传输数据的，都不安全。由于Get方法请求参数通常放在url（在浏览器只能放在Url中，通过接口请求还可以放在Header、Body中）中以及可能存在缓存，所以它显得更为不安全。

## Cookie与Session的区别？

1. 存放地点：Cookie又服务器返回给客户端并存放在客户端，Session则是由服务端创建并保存在服务端，只返回给客户端一个SessionId。
2. 内容格式：Cookie只能是键值对，而Session可以是任意内容。

## 输入一个网址到网页显示的流程？

(1)解析URL，解析出协议名称、主机名、端口号等。

  (2)通过DNS解析URL获取对应IP地址。

  (3)与服务器建立 TCP 连接

  (4)向服务器发出请求。

  (5)服务器给出响应，把数据发送客户端。

  (6)TCP 连接释放。

  (7)解析服务器返回内容。

## Http工作流程?（三次握手与四次分手）

<https://baijiahao.baidu.com/s?id=1593714120815701015&wfr=spider&for=pc>

HTTP请求方式中8种请求方法（简单介绍）

<https://www.cnblogs.com/weibanggang/p/9454581.html>

## DNS解析流程是什么？

1、在浏览器中输入www . qq .com 域名，操作系统会先检查自己**本地的hosts文件**是否有这个网址映射关系，如果有，就先调用这个IP地址映射，完成域名解析。

2、如果hosts里没有这个域名的映射，则查找本地**DNS解析器缓存**，是否有这个网址映射关系，如果有，直接返回，完成域名解析。

3、如果hosts与本地DNS解析器缓存都没有相应的网址映射关系，首先会找TCP/ip参数中**设置的首选DNS服务器**，在此我们叫它**本地DNS服务器**，此服务器收到查询时，如果要查询的域名，包含在本地配置区域资源中，则返回解析结果给客户机，完成域名解析，**此解析具有权威性**。

4、如果要查询的域名，不由本地DNS服务器区域解析，但该**服务器已缓存**了此网址映射关系，则调用这个IP地址映射，完成域名解析，**此解析不具有权威性**。

5、如果本地DNS服务器本地区域文件与缓存解析都失效，则根据本地DNS服务器的设置（是否设置转发器）进行查询，如果**未用转发模式**，**本地DNS就把请求发至13台根DNS**，根DNS服务器收到请求后会判断这个域名(.com)是谁来授权管理，并会返回一个负责该**顶级域名服务器的一个IP**。本地DNS服务器收到IP信息后，将会联系负责.com域的这台服务器。这台负责.com域的服务器收到请求后，如果自己无法解析，它就会找一个管理.com域的下一级DNS服务器地址([http://qq.com](https://link.zhihu.com/?target=http%3A//qq.com))给本地DNS服务器。当本地DNS服务器收到这个地址后，就会找[http://qq.com](https://link.zhihu.com/?target=http%3A//qq.com)域服务器，重复上面的动作，进行查询，直至找到www . qq .com主机。

6、如果用的是**转发模式**，此DNS服务器就会把请求转发至**上一级DNS服务器**，由上一级服务器进行解析，上一级服务器如果不能解析，或找根DNS或把转请求转至上上级，以此循环。不管是本地DNS服务器用是是转发，还是根提示，最后都是把结果返回给本地DNS服务器，由此DNS服务器再返回给客户机。

## https中常见的加密算法有哪些？签名、证书的区别是什么？

## Socket是什么？

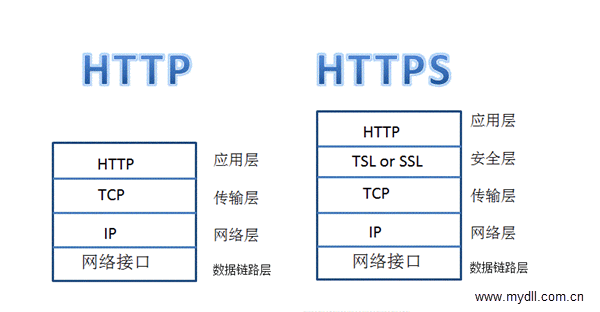
Socket是应用层与传输层（TCP/IP、UDP协议族）中间的软件抽象层，它是一组接口。在设计模式中，Socket其实就是一个门面模式，它把复杂的传输层中协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。

## TCP与UDP有什么区别？

## HTTP与HTTPS有什么区别？

1. HTTP协议是工作在TCP协议之上的，而HTTPS协议是工作在SSL/TLS（称为安全层，位于应用层与传输层之间）之上的。

2、http协议URL以“http:”开头，而https以“https:”开头。



## TSL与SSL有什么区别？

**SSL：**（Secure Socket Layer，安全套接字层），位于可靠的面向连接的网络层协议和应用层协议之间的一种协议层。SSL通过互相认证、使用**数字签名确保完整性**、使用**加密确保私密性**，以实现客户端和服务器之间的安全通讯。该协议由两层组成：SSL记录协议和SSL握手协议。

**TLS：**(Transport Layer Security，传输层安全协议)，用于两个应用程序之间提供保密性和数据完整性。该协议由两层组成：TLS记录协议和TLS握手协议。

**关系**：

SSL由从前的[网景公司](https://www.baidu.com/s?wd=%E7%BD%91%E6%99%AF%E5%85%AC%E5%8F%B8&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)开发有1,2,3三个版本，但现在只使用版本3.0,TLS是SSL的标准化后的产物有1.0 、1.1、 1.2三个版本，默认使用1.0，TLS1.0和SSL3.0几乎没有区别事实上我们现在用的都是TLS，但因为历史上习惯了SSL这个称呼，平常还是以SSL为多。

## SSL/TLS工作流程？

<https://segmentfault.com/a/1190000021559557>

## SASL/TLS区别？

（1）SASL：SASL全称Simple Authentication and Security Layer，是一种用来扩充C/S模式验证能力的机制。在Postfix可以利用SASL来判断用户是否有权使用转发服务，或是辨认谁在使用你的服务器。SASL提供了一个通用的方法为基于连接的协议增加验证支持

（2）TLS：安全传输层协议（TLS）用于在两个通信应用程序之间提供**保密性和数据完整性**。该协议由两层组成： TLS 记录协议（TLS Record）和 TLS 握手协议（TLS Handshake）

## 签名与证书的区别？

**数字签名**的作用就是验证数据来源以及数据完整性，发送方将发送的内容提取摘要（MD5）并用私钥加密，然后将加密的摘要与内容一起发送出去，接收方收到数据后使用对方给的公钥解密摘要，解密成功就能验证对方身份（私钥与公钥是一一对应的），最后将解密的摘要与计算接收内容的摘要进行比较。签名机制的缺点就是公钥可能会被替换可以哦配合证书使用以解决该问题，也就是发送方发送签名+证书+内容，接收方收到数据后用CA的公钥解密这份数字证书，拿到A的公钥，然后验证数字签名。

“证书中心"（certificate authority，简称CA），为**公钥做认证**。证书中心用自己的私钥，对A的公钥和一些相关信息一起加密，生成"数字证书"（Digital Certificate）。

**常见的加密算法有哪些？**

<https://www.cnblogs.com/tuyile006/p/10873975.html>

**非对称**加密算法：

**RSA**：RSA 是一种目前应用非常广泛、历史也比较悠久的非对称秘钥加密技术，在1977年被麻省理工学院的罗纳德·李维斯特（Ron Rivest）、阿迪·萨莫尔（Adi Shamir）和伦纳德·阿德曼（Leonard Adleman）三位科学家提出，由于难于破解，RSA 是目前应用最广泛的数字加密和签名技术，比如国内的支付宝就是通过RSA算法来进行签名验证。它的安全程度取决于秘钥的长度，目前主流可选秘钥长度为 **1024位、2048位、4096**位等，理论上秘钥越长越难于破解，按照维基百科上的说法，**小于等于256位的秘钥，在一台个人电脑上花几个小时就能被破解**，512位的秘钥和768位的秘钥也分别在1999年和2009年被成功破解，虽然目前还没有公开资料证实有人能够成功破解1024位的秘钥，但显然距离这个节点也并不遥远，所以目前业界推荐使用 2048 位或以上的秘钥，不过目前看 2048 位的秘钥已经足够安全了，支付宝的官方文档上推荐也是2048位，当然更长的秘钥更安全，但也意味着会产生更大的性能开销。

**ECDSA**：Elliptic Curve Digital Signature Algorithm，**椭圆曲线签名**算法，是**ECC**（Elliptic curve cryptography，椭圆曲线密码学）和 DSA 的结合，椭圆曲线在密码学中的使用是在1985年由Neal Koblitz和Victor Miller分别独立提出的，相比于RSA算法，ECC 可以使用更小的秘钥，更高的效率，提供更高的安全保障，据称256位的ECC秘钥的安全性等同于3072位的RSA秘钥，和普通DSA相比，ECDSA在计算秘钥的过程中，部分因子使用了椭圆曲线算法**。**

<https://www.cnblogs.com/wushaopei/p/11979200.html>

# 设计模式

## 面向对象五(SOLID)/六大原则?

1. **单一职责原则->Single Responsibilty Principle:**

一个类，只有一个引起它变化的原因。应该只有一个职责。每一个职责都是变化的一个轴线，如果一个类有一个以上的职责，这些职责就[耦合](https://baike.baidu.com/item/%E8%80%A6%E5%90%88/2821124)在了一起。这会导致脆弱的设计。当一个职责发生变化时，可能会影响其它的职责。另外，多个职责耦合在一起，会影响[复用性](https://baike.baidu.com/item/%E5%A4%8D%E7%94%A8%E6%80%A7/12719815)。例如：要实现逻辑和界面的分离。

1. **开闭原则->Open Close Principle:**

**开闭原则**规定“*软件中的对象（类，模块，函数等等）应该对于扩展是开放的，但是对于修改是封闭的*”，这意味着一个实体是允许在不改变它的[源代码](https://baike.baidu.com/item/%E6%BA%90%E4%BB%A3%E7%A0%81)的前提下变更它的行为。该特性在产品化的环境中是特别有价值的，在这种环境中，改变源代码需要[代码审查](https://baike.baidu.com/item/%E4%BB%A3%E7%A0%81%E5%AE%A1%E6%9F%A5)，[单元测试](https://baike.baidu.com/item/%E5%8D%95%E5%85%83%E6%B5%8B%E8%AF%95)以及诸如此类的用以确保产品使用质量的过程。遵循这种原则的代码在扩展时并不发生改变，因此无需上述的过程。

1. **里氏替换原则->Liskov Substitution Principle:**

里氏替换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。

1. **依赖倒置原则->Dependence Inversion Principle:**

[面向过程](https://baike.baidu.com/item/%E9%9D%A2%E5%90%91%E8%BF%87%E7%A8%8B)的开发，上层调用下层，上层依赖于下层，当下层剧烈变动时上层也要跟着变动，这就会导致模块的复用性降低而且大大提高了开发的成本。

依赖倒置原则是程序要**依赖于抽象接口**，不要**依赖于具体实现**。简单的说就是要求对抽象进行编程，不要对实现进行编程，这样就降低了客户与实现模块间的耦合。

1. **接口隔离原则->Interface Segreation Principle:**

客户端不应该依赖它不需要的接口。一个类对另一个类的依赖应该建立在最小的接口上。

1. **迪米特原则->Law of Demeter:**

## 列举常见设计模式,如何实现及列举在Android中的具体应用

1. **工厂法：**

**目的**：解偶，将编译时绑定转换成运行时绑定 。

**定义：**定义一个创建对象的接口，由其子类决定实例化那一类，使得一个类实例化延迟到子类。

**场景**：当一个抽象类或接口需要派生出许多子类时，不希望调用者直接通过new关键字创建复杂多样的具体子类从而造成严重的耦合。

**Android中的应用**：

ThreadFactory，BitmapFactory，Activity

1. **单例模式：**

**目的：**复用对象节约资源或该对象在应用中要求只能存在一个。

**定义：**

**场景：**

**Android中的应用：**

1. **责任链模式：**

**目的：**

**定义：**

**场景：**

**Android中的应用：**

1. **外观模式：**

**目的：**

**定义：**

**场景：**

**Android中的应用：Context**

1. **构建者模式：**

**目的：**

**定义：**

**场景：**

**Android中的应用：**

1. **外观模式：**

**目的：**

**定义：**

**场景：**

**Android中的应用：**

1. **装饰器模式：**

**目的：解决扩张对象的功能同时避免子类膨胀（通过继承转组合）。**

**定义：动态地给一个对象增加额外的职责**

**场景：**

**Android中的应用：ContextWraper**

1. **策略模式：**

**目的：对象使用的某个算法可能有多种多样的实现，经常发生变化，如果将这些算法都硬件到对象中可能会使得对象变得异常复杂，难以维护与扩展，同时冗余的算法也是一种资源、性能的浪费。**

**定义：定义一系列算法将它们一个个的封装起来，并且它们可以相互替换，使得算法独立于客户端而变化（扩展）。**

**场景：某个算法多变。**

**Android中的应用：ListView的Adapter ValueAnimation的TimeInterpolator、TypeEvaluator(TimeInterpolator控制在何时取值，而TypeEvaluator控制在当前时间点需要取多少值),Comprator**

1. **观察者模式：**

**目的：**

**定义：**

**场景：**

**Android中的应用：**

1. **模版法：**

**目的：解决某个算法的整体流程稳定，但各子步骤却存在很多需求变化或因为某些原因无法与整体一起实现的问题。**

**定义：定义一个算法的骨架，将一些步骤延的实现迟到子类，使得子类实现一些特定步骤后便可以实现整个算法。**

**Android中的应用：Activity、Service、AsyncTask、View**

**(11)原型模式：**

**目的：为了安全因素隔离对象与具体的使用者或通过克隆已有的复杂对象以减少资源、内存、时间占用。**

**定义：用原型实例指定创建对象的种类，并通过拷贝这些原型创建新的对象。**

**场景：**

**（1）类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等，通过原型拷贝避免这些消耗。**

**（2）通过new产生的一个对象需要非常繁琐的数据准备或者权限，这时可以使用原型模式。**

**（3）一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用，即保护性拷贝。**

**Android中的应用：Shape、Animator、Intent、ArrayList**

# 其它

## WebRTC原理？

必备知识：

**（1）NAT**：（为了解决IPV4下的IP地址匮乏问题）

NAT（Network Address Translation，网络地址转换）是1994年提出的。当在专用网内部的一些主机本来已经分配到了本地IP地址（即仅在本专用网内使用的专用地址），但现在又想和因特网上的主机通信（并不需要加密）时，可使用NAT方法。

1. **ICE协议框架**：

基于IP的语音、数据、视频等业务在NGN（Next Generation Network）网络中所面临的一个实际困难就是如何有效地穿透各种NAT（Network Address Translator）/FW(Fire Wall)的问题。

对此，SIP（会话初始化协议）以往的解决方法由ALGs（(Application Layer Gateway Service)）SDP、STUN、TURN等方式。帮助ICE共同实现了NAT/防火墙穿越。

**一、什么是信令（SDP）**

信令就是协调通讯的过程，为了建立一个webRTC的通讯过程，客户端需要交换如下信息：

. 会话控制信息，用来开始和结束通话，即开始视频、结束视频这些操作指令。

. 处理错误的消息。

. 元数据，如各自的音视频解码方式、带宽。

. 网络数据，对方的公网IP、端口、内网IP及端口。

信令处理过程需要客户端能够来回传递消息，这个过程在webRTC里面是没有实现的，需要自己创建。一旦信令服务建立好了,两个客户端之间建立了连接,理论上他们就可以进行点对点通讯了,这样可以减轻信令服务的压力和消息传递的延迟。

因为信令是我们自己定义的,所以安全性问题跟webrtc无关,需要自己处理。一旦黑客掌握了你的信令,那他就是控制会话的开始、结束、重定向等等。最重要的因素在信令安全中还是要靠使用安全协议,如HTTPS,WSS(如TLS),他们能确保未加密的消息不能被截取。为确保信令安全,强烈推荐使用TLS。

二、TURN 和 STUN

元数据是通过信令服务器中转发给另一个客户端,但是对于流媒体数据,一旦会话建立,首先尝试使用点对点连接。

简单一点说就是:

每个客户端都有一个唯一的地址,他能用来和其他客户端进行通讯和数据交换。现实生活中客户端都位于一个或多个NAT之后,或者一些杀毒软件还阻止了某些端口和协议,或者在公司还有防火墙或代理等等,防火墙和NAT或许是同一个设备,如我们家里用的路由器。

webrtc就是通过 **ICE** 这套框架来处理复杂的网络环境的,

如果想启用这个功能,你必须让你的应用程序传 ICE 服务器的URL：

ICE试着找最好的路径来让客户端建立连接,他会尝试所有可能的选项,然后选择最合适的方案,

ICE首先尝试P2P连接,如果失败就会通过**TURN服务器**进行转接。

换一个说法就是:

STUN服务器是用来取外网地址的。

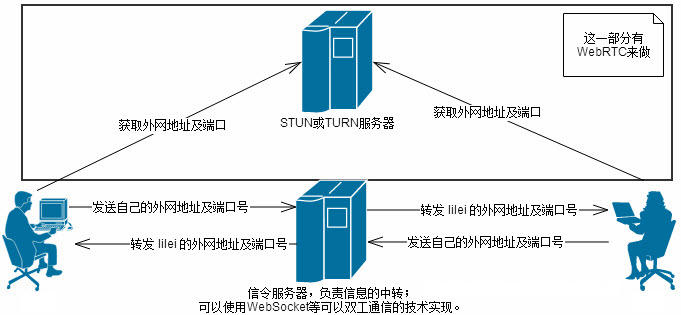
TURN服务器是在P2P失败时进行转发的

STUN和TURN服务的作用主要处理打洞与转发，配合完成ICE协议。首先尝试使用P2P，

如果失败将求助于TCP，使用TURN转发两个端点的音视频数据，TURN转发的是两个端点之间的音视频数据不是信令数据。因为TURN服务器是在公网上，所以他能被各个客户端找到，另外TURN服务器转发的是数据流，很占用带宽和资源。

WebRTC的实现是建立设备之间的直接连接而无需服务器中转，即P2P。因此要求彼此知道对方外网地址，而设备大多位于NAT之后。这就要求一种方式可穿透NAT技术（STUN、TRUN）。

WebRTC使用默认STUN服务器获取当前主机的外网地址和端口，P2P建立过程需依赖服务端中转（信令服务器）



**WebRTC建立连接的流程：**

场景：ClientA向ClientB发起视讯请求

（1）

【ClientA创建PeerConnection对象】→【ClientA打开本地音视频设备】→【ClientA将音视频数据封装成MediaStream添加到PeerConnection中】

（2）

【ClientA调用PeerConnection的CreateOffer方法创建一个用于offer的SDP对象，SDP对象中保存当前音视频的相关参数。】→【ClientA通过PeerConnection的SetLocalDescription方法将该SDP对象保存起来】→【ClientA通过Signal服务器将SDP对象发送给ClientB】

（3）

【ClientB接收到ClientA发送过的offer SDP对象】→【通过PeerConnection的SetRemoteDescription方法将其保存起来】→【调用PeerConnection的CreateAnswer方法创建一个**应答的SDP**对象】→【通过PeerConnection的SetLocalDescription的方法保存该应答SDP对象】→【通过Signal服务器将该应答SDP对象发送给ClientA】

（4）

【ClientA接收到ClientB发送过来的应答SDP对象】→【将应答SDP对象通过PeerConnection的SetRemoteDescription方法保存起来】

（5）

【在SDP信息的offer/answer流程中，ClientA和ClientB已经根据SDP信息创建好相应的音频Channel和视频Channel并开启Candidate数据的收集】【Candidate数据可以简单地理解成Client端的IP地址信息（本地IP地址、公网IP地址、Relay服务端分配的地址）】

（6）

【当ClientA收集到Candidate信息后，PeerConnection会通过OnIceCandidate接口给ClientA发送通知，ClientA将收到的Candidate信息，通过Signal服务器发送给ClientB，ClientB通过PeerConnection的AddIceCandidate方法保存起来。同样的操作ClientB对ClientA再来一次】

（7）

【这样ClientA和ClientB就已经建立了音视频传输的P2P通道，ClientB接收到ClientA传送过来的音视频流，会通过PeerConnection的OnAddStream回调接口返回一个标识ClientA端音视频流的MediaStream对象，在ClientB端渲染出来即可。同样操作也适应ClientB到ClientA的音视频流的传输】

## Smack是什么？

Smack是一个开源，易于使用的XMPP客户端类库。Smack API, 是一个 Java 的XMPP Client Library,也是由Jive Software开发。 优点:编程简单。 缺点:API并非为大量并发用户设计,每个客户要1个线程,占用资源大,1台机器只能模拟有限(数千个)客户。Smack是一个用 java 写的XMPP客户端代码库, 是 spark 的核心。

使用Smack发送消息，大致分以下几步：1、创建一个XMPPConnection对象，并连接服务器；2、注册包（Packet）监听器；3、登录；4、获取Chat对象（通过XMPPConnection对象），发送消息。

XMPP协议是什么？

XMPP即：**可扩展消息处理现场协议**(eXtensible Messaging and Presence Protocol)，它将现场和上下文敏感信息标记嵌入到XML 结构化数据中。

**XMPP消息格式**：

XMPP中定义了3个顶层XML元素: Message、Presence、IQ

**<Message>**

用于在两个jabber用户之间发送信息。Jsm(jabber会话管理器)负责满足全部的消息，无论目标用户的状态怎样。假设用户在线jsm马上提交;否则jsm就存储。

To :标识消息的接收方。

from : 指发送方的名字或标示(id)

Text: 此元素包括了要提交给目标用户的信息。

结构例如以下所看到的:

<message to= ‘lily@jabber.org/contact’ type =’chat’>

<body> 你好，在忙吗</body>

</message>

**<Presence>**

用来表明用户的状态，如：online、away、dnd(请勿打搅)等。当用户离线或改变自己的状态时，就会在stream的上下文中插入一个Presence元素，来表明自身的状态．结构例如以下所看到的：

<presence>From =‘lily @ jabber.com/contact’

To = ‘yaoman @ jabber.com/contact'

<status> Online </status>

</presence>

<presence>元素能够取以下几种值:

Probe :用于向接受消息方法发送特殊的请求

subscribe:当接受方状态改变时，自己主动向发送方发送presence信息。

**< IQ >**

一种请求／响应机制，从一个实体从发送请求，另外一个实体接受请求，并进行响应．比如，client在stream的上下文中插入一个元素，向Server请求得到自己的好友列表，Server返回一个，里面是请求的结果．

**<iq > 基本的属性是type。包含:**

Get :获取当前域值。

Set :设置或替换get查询的值。

Result :说明成功的响应了先前的查询。

Error: 查询和响应中出现的错误。

结构例如以下所看到的:

<iq from =‘lily @ jabber.com/contact’id=’1364564666’ Type=’result’>

## EventBus 工作原理？

## ARouter工作原理？

## Retrofit+OKHttp+RxJava

## LiveData

## DataBinding

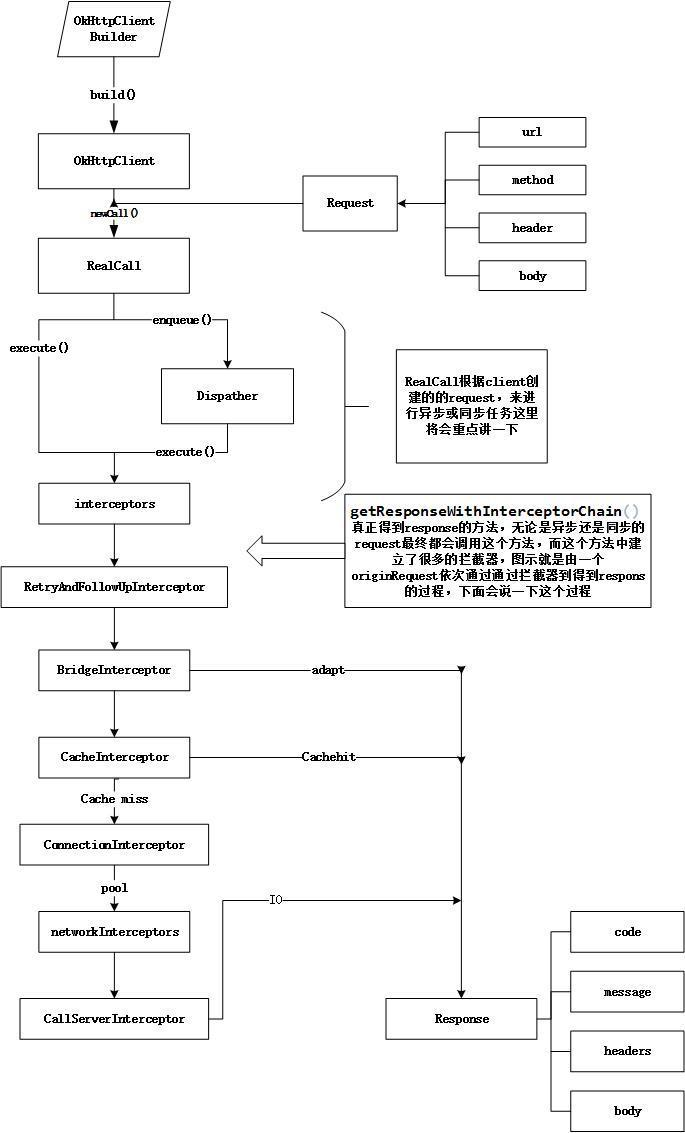
### OkHttp优点？OkHttp的实现原理？

整个流程是:通过OkHttpClient将构建的Request转换为Call，然后在RealCall中进行异步或同步任务，最后通过一些的拦截器interceptor发出网络请求和得到返回的Response对象

**主要优势**：  
1、HTTP/2 支持允许所有访问同一主机的请求**共享一个socket**  
2、利用**连接池**减少请求延迟（如果HTTP/2不可用）  
3、支持**GZIP压缩**  
4、**响应缓存**减少重复请求

5、重试请求如果有不同路线，okhttp将请求不同的路线

6、原理:



## OKHttp是怎么处理https的？

一个SSL/TLS完整的**握手过程**。



SSL/TLS握手过程很复杂的交互过程，但是理解下来就是用非对称加密的手段传递密钥，然后用密钥进行对称加密传递数据。在这个**握手过程中最重要的就是证书校验**，其他就是正常的数据交互过程。如何校验一个证书合法有很大的文章，处理不好就会让你的网络失去了安全性。一个证书的校验，主要包括以下几个方面：

**第一**，校验证书是否是由客户端中“受信任的根证书颁发机构”颁发；

**第二**，校验证书是否在上级证书的吊销列表；

**第三**，校验证书是否过期；

**第四**，校验证书域名是否一致。

**EventBus原理?**