【安卓性能优化总结】

【八年工作经验精华积累】

目录

[最全的性能优化点总结： 4](#_Toc82304108)

[零、 启动优化 4](#_Toc82304109)

[1、项目背景 4](#_Toc82304110)

[2、 检测启动时间 4](#_Toc82304111)

[3、打印启动时间 4](#_Toc82304112)

[4、优化理念： 4](#_Toc82304113)

[5、启动时透明页优化： 5](#_Toc82304114)

[6、MultiDex优化 5](#_Toc82304115)

[7、多进程时，防止sdk多次初始化 5](#_Toc82304116)

[8、最终结果： 5](#_Toc82304117)

[一、 内存优化 6](#_Toc82304118)

[1、 项目背景 6](#_Toc82304119)

[2、性能优化的理念 6](#_Toc82304120)

[3、了解对象之间的引用关系和对象大小的占用 6](#_Toc82304121)

[4、了解Android中经常造成内存泄漏的点 6](#_Toc82304122)

[(1)、耗时任务：网络请求、属性动画、Timer 6](#_Toc82304123)

[(2)、handler 6](#_Toc82304124)

[(3)、匿名/非静态内部类 7](#_Toc82304125)

[(4)、单例、applicationContext 、ThreadLocal等 7](#_Toc82304126)

[(5)、WebView内存泄露 7](#_Toc82304127)

[(6)、资源未及时关闭 7](#_Toc82304128)

[5、内存溢出问题 7](#_Toc82304129)

[6、内存抖动的问题 7](#_Toc82304130)

[7、体验优化 7](#_Toc82304131)

[8、内存视图查看； 8](#_Toc82304132)

[9、了解对象产生和分配过程； 9](#_Toc82304133)

[10、对象创建的过程中代码执行顺序 9](#_Toc82304134)

[11、SP文件的优化 9](#_Toc82304135)

[12、详细的优化过程 9](#_Toc82304136)

[13、优化结果 10](#_Toc82304137)

[二、 安装包瘦身 10](#_Toc82304138)

[三、 网络优化 13](#_Toc82304139)

[四、 第三方库的一些简单替代方案 14](#_Toc82304140)

[五、 图片加载库的问题以及性能比较 16](#_Toc82304141)

[1、问题背景： 16](#_Toc82304142)

[2、图片库测试结果： 17](#_Toc82304143)

[3、测试过程 17](#_Toc82304144)

[4、使用建议： 17](#_Toc82304145)

[六、 锁的优化synchronized和lock 17](#_Toc82304146)

[锁的优化方向： 17](#_Toc82304147)

[延伸单例写法推荐 18](#_Toc82304148)

[七、 MVP设计模式的弊端及解决方案(重点) 19](#_Toc82304149)

[Mvp存在的问题： 19](#_Toc82304150)

[(1) 首先是设计思路问题 19](#_Toc82304151)

[(2) 过多的present和model的问题 19](#_Toc82304152)

[(3) 内存泄露的问题 19](#_Toc82304153)

[(4) 解决办法 19](#_Toc82304154)

[(5) 此mvp设计总结： 25](#_Toc82304155)

[九、View绘制方面的优化 25](#_Toc82304156)

[（1）多嵌套问题： 25](#_Toc82304157)

[（2）各种控件性能比较： 25](#_Toc82304158)

[（3）使用占位符ViewStub： 25](#_Toc82304159)

[（4）结语： 25](#_Toc82304160)

[十、一些知名的内存管理监测软件 26](#_Toc82304161)

[(1) LeakCanary 26](#_Toc82304162)

[(2) koom 26](#_Toc82304163)

[十一、数据加密优化 26](#_Toc82304164)

[项目背景： 26](#_Toc82304165)

[优化方案： 26](#_Toc82304166)

[对称加密 26](#_Toc82304167)

[非对称加密 27](#_Toc82304168)

[散列算法： 27](#_Toc82304169)

[常用的加密方案： 27](#_Toc82304170)

[总结： 27](#_Toc82304171)

[十二、修正一下编程思想！面向对象 or 面向过程 27](#_Toc82304172)

[（1）面向对象开发和面向过程开发 27](#_Toc82304173)

[十三、架构方面：对app代码业务逻辑的一些设计思考 30](#_Toc82304174)

[1、项目背景： 30](#_Toc82304175)

[2、优化思路： 30](#_Toc82304176)

[3、详细内容，举例说明 30](#_Toc82304177)

[十四、数据库和高并发优化 33](#_Toc82304178)

[1、问题背景 33](#_Toc82304179)

[2、 高并发优化 33](#_Toc82304180)

[3、数据库优化 33](#_Toc82304181)

[十五、android特有的库和一些代码基础写法 33](#_Toc82304182)

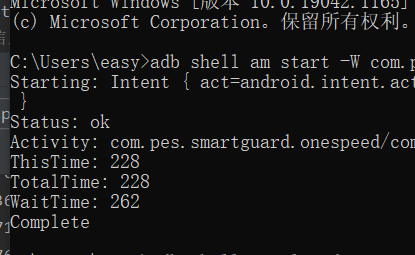
# 最全的性能优化点总结：

# 启动优化

## 1、项目背景

19公司里的一个项目，是一个第三方库特别多的app，会启动10个广告sdk、若干游戏sdk。初期都是在application里面直接初始化的

## 检测启动时间

adb shell am start -W 包名/界面名 

## **3、打印启动时间**

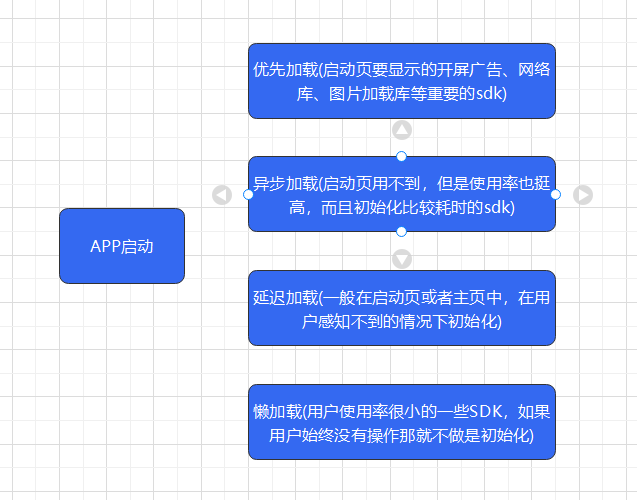
onCreate、onResume等方法的耗时，尤其是初始化许多sdk 的方法(可使用aop方法、或者system时间相减，都可以；aop的包有apply plugin: 'android-aspectjx' implement 'org.aspectj:aspectjrt:1.8.+')

筛选启动时必须加载的sdk，其他的则放在子线程里面或者延时加载

## 4、优化理念：

把十几个要实例化的sdk分成三部分：先加载、延迟加载、异步加载、懒加载

先加载：因为首页有开屏广告，所以开屏广告的sdk必须有先加载



把各种库分成四种情况都是为了尽可能提高启动速度；

## **5、启动时透明页优化**：

设置透明主体或者提早设置背景页面；都比较常规的方法，我选择了设置背景页面，给用户一个第一时间我就打开了app的感觉，否则会有一种卡顿的感觉

## 6、MultiDex优化

这个问题，在我们第三方sdk特别多的项目里尤为明显；体积小的apk一般就一个dex；开启多dex打包之后，就会产生很多dex文件；这个我用的主流的方法；异步加载其他dex；最恶心的过程就是手动分包，因为要保证启主dex体积比较小；能快速的加载出来，然后其他dex就放在子线程里边去加载

## 7、多进程时，防止sdk多次初始化

项目当中有一些音乐播放是夸进程，所以需要防止一些sdk多次初始化，这个比较简单，不贴代码里额

## **8、最终结果：**

启动时间节省了一半以上

# 内存优化

## 项目背景

这个是15年银行的电商app项目，由于当时的项目前后端整体设计不是特别完善；整个的图片的处理都是在客户端完成的；包括加水印、压缩等等，而且图片不管是上传还是下载的都是高清原图，所以对app的性能是极大的考验；也是当时的app前后端设计不是特别合理，所以草导致了这个问题，不过也因此我锻炼了内存优化的能力和知识

## **2、性能优化的理念**

两个强引用对象之间有引用关系，且其中一个对象生命周期较长，在15年优化一个项目的时候，许多人会手动调用System.gc()，这是性能优化一大禁忌；简单说我们创造对象是可回收的环境，让GC需要的时候自己就来处理垃圾收，本身垃圾回收的操作也是消耗性能的；这就像在一个小区里边，有专门收垃圾的车进来，你只需要把垃圾放到垃圾桶中，其他的就不要管了，不要自己频繁的去呼叫垃圾车来收垃圾；一旦来收垃圾的时候你自己也要停止工作(stop the world)，大家都耽误事儿

## **3、了解对象之间的引用关系和对象大小的占用**

按默认开启指针压缩来算，除了基本数据类型之外，其他类型是占用四个字节，也就是多写一个成员变量对象就增大四个字节；和这个成员变量有没有具体引用到那个对象没关系，只要是生命了就会多占用内存

## **4、了解Android中经常造成内存泄漏的点**

还有第三方控件，那个放在后面常见的内存泄漏情况有：

### **(1)、耗时任务**：**网络请求、属性动画、Timer**

解决办法就是及时解除强引用关系、把指针置空、及时停止耗时任务；或者用弱引用设置指针关系

### (2)、handler

Handler一般是因为有延时任务，message里有个Handler类型的target变量，这个就是引用handler的成员变量，所以如果延时任务不结束，那么handler所在对象就无法被回收掉。解决办法就是重写handler或者及时清除延时任务

### (3)、匿名/非静态内部类

这个就是内部类对象会持有外部类的引用，所以如果非静态内部类对象/匿名对象的成生命周期较长就会影响外部类对象的回收。

### (4)、单例、applicationContext 、ThreadLocal、内部类广播等

其实都是强引用关系和生命周期不同造成的的；解决办法就是用弱引用和及时解除他们的关系，因为一般单例、appContext、ThreadLocal不会被干掉，生命周期比较长，所以就只能及时解除关联关系，把变量的引用置空就可以了

### (5)、WebView内存泄露

这个问题存在很久了，也没注意现在有没有被修复，主要原因是webView内核webkit和application有注册关系，所以一定要反注册，解决办法就是activity关闭之前，获取webView的父布局，然后remove掉WebView，然后停止webView的一些方法、清楚历史清除views。最后再执行WebView.destroy()；

### (6)、资源未及时关闭

这个就是IO流、File文件流、Sqlite这些只用完毕时候要及时关闭

### (7)、要反注册

比如：eventBus，广播，以及ContentProvider等等，都需要在onDestory里面反注册；我写的mvp模式的evm也需要反注册；

## **5、内存溢出问题**

这个在Android中，一般就是bitmap或者视频之类的容易造成内存溢出，内存溢出分两种，一种是直接溢出，一种是内存泄露累积起来的内存溢出，都是堆空间里满了造成的oom；bitmap就是压缩图片，bitmap的大小就是和像素的面积和每个像素点的色位决定的；这个就是按照bitmap要显示的实际大小进行压缩；一般就是计算压缩比例，然后抽取bitmap图片；

## **6、内存抖动的问题**

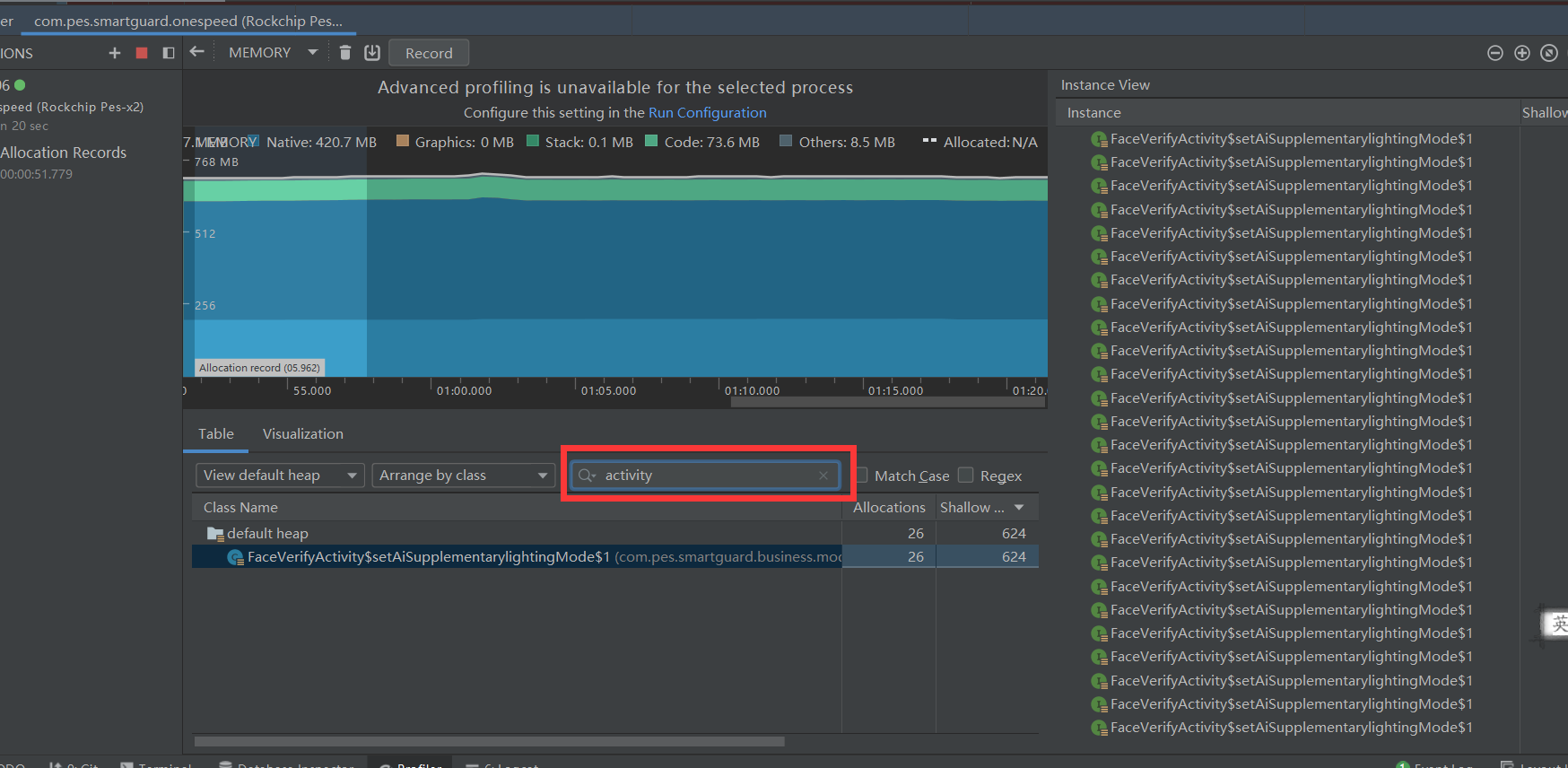
这个就是要优化程序了，现在这个问题出的情况比较少，也是15年的时候优化过，里面有一些比较大的对象进行频发的创建和销毁就会造成内存抖动，其实就是会频繁的出发GC操作；延长这个对象的生命周期或者优化这个对象的大小

## **7、体验优化**

这一般是整体架构的问题，加载一些图片的时候，就算手机内存足够大，但是网速也限制图片的下载速度，尤其是在一些列表图片功能里，体验就会很差，如果图片体积很小，那给人的感觉就不同；现在都比较正规了，这个问题也几乎没有了

## **8、内存视图查看；**

学会使用Android profiler,先手动gc，然后根据类名寻找对象是否存在，就可以查找到Android对象是否内存泄漏了；除了内存外，cpu、网络、能耗都可以查得到

搜索对象是否存在

## 9、了解对象产生和分配过程；

可分配在堆内存、栈内存；堆内存中还包括新生代、老年代，还要根据对象的大小去判断；

## **10、对象创建的过程中代码执行顺序**

静态变量、静态代码块，代码块；构造方法、父类和子类中这几种情况的执行顺序，了解这些会有助于了解代码的执行过程和内存的分配

## **11、SP文件的优化**

其实就是同步还是异步的问题，想要性能好肯定选择异步，然后合并多次commit();

commit（同步）apply（异步）

替代方案：mmkv 腾讯出的，从15年开始一直在微信上使用，可见其性能得到了考验

## 12、详细的优化过程

(1)、选择最优图片加载库：格比较了各种图片库的优缺点和性能问题，选择一个最稳定的，最终确定了使用frsco(选择过程后面，解析第三方图片库有讲解)

(2)、清除项目里手动GC操作

(3)、解除activity里的网络请求操作和主acitvity的匿名回调对象的关系；采用统一的回调方式，这样网络请求的耗时操作就不会影响正常的acitivty对象回收了

(4)、针对项目里的内存抖动问题，尽量延长对象的声明周期、压缩对象的体积，目的旨在减少system.gc出来工作的次数

(5)、bitmap优化，设置bitmap.Config为ARGB\_4444或者565；再就是根据控件实际显示的大小，计算压缩比例，创建像素合适的bitmap；最后就是及时释放资源执行recycle()方法并且置空；因为项目里有bit加水印的操作，所以需要自己处理一下

(6)、写法上的优化，把加载fragment的写法从add改成replace。activity里边减少成员变量，有一些只有个点击事件，并无其他操作，所以不必在activity创建成员变量，少写一个成员变量就节省了4个字节的空间量变引质变。直接再xml文件里写onClick方法，而且效率要远远比先findViewById然后再setOnclickListener高出很多很多；再就是尽量减少布局的嵌套、多用include和megre、viewStub

(7)、测试一些父控件的执行效率，效率最差的RelativeLayout，LinearLayout和Fragmentlayout性能相差无几；那时候ConstraintLayout还未流行起来，所以未做处理，一些简单布局优先使用Linear/FrameLayout；

(8)、用android studio逐个activity检查内存泄漏的情况；检查过程就是先打开此activity页面，然后关闭，然后手动调用几次gc操作；最后查看此activity对象是否还存在，如果还存在就表示有内存泄漏，再逐个排查，多数都是匿名对象造成的  
(9)、检查有和单例、ApplicationContext扯上关系的对象，这些都是内存泄露潜在的点，所以要重点检查

(10)、检查handler的延时任务，那时候还不会弱引用，所以只是手动的在activity关闭的时候，自己手动清除handler的延时任务

(11)、把项目里的ListView替换成RecyclerView，RecyclerView性能比ListView好很多，也是经过测试的；

(12)、app创建线程池，来统一管理一些定时任务，包括轮播、短信验证码等。

(13)、RecyclerView、ListView之类的控件，在滑动的时候要停止图片的加载

## 13、优化结果

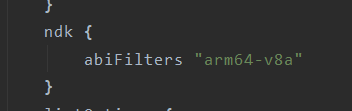
项目从原来的崩溃率特别高，降到非常低，崩溃情况不会出现了

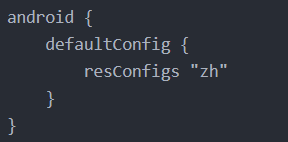
# 安装包瘦身

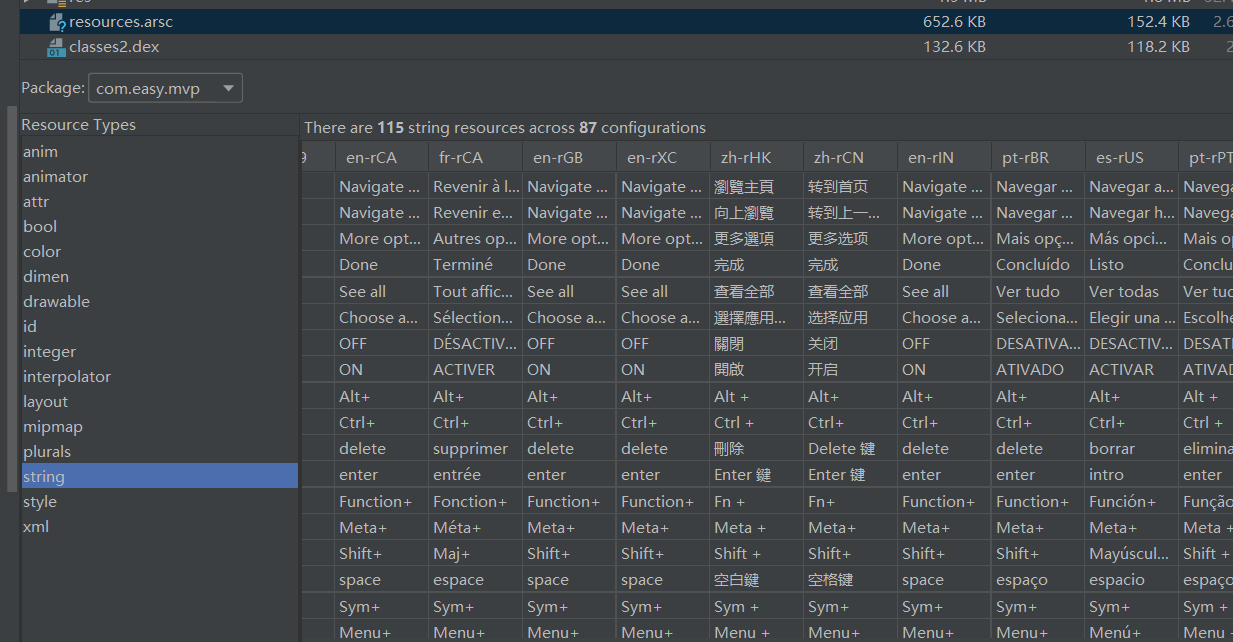
## 项目背景

也是16年银行的直销银行项目，android包体积过大，其中也主要是因为加入人脸识别、地图等各种so库造成的

## 优化方案和过程

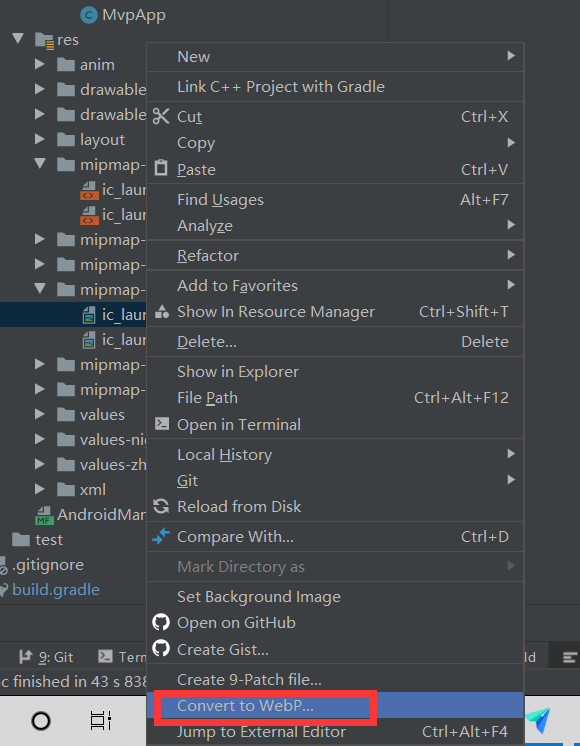
1. so文件过多，按照cpu架构有个多个类型，这个看app的使用潜在人群，如果是手机一般只用v7就可以，如果是pad那就用x64,x86之类的，总之要具体选择当时的项目用的是v7
2. 默认国际化：关掉默认国际化

因为android打包成apk之后，apk里会有一个叫resources.arsc的文件，里边都是res/values文件夹下文件生成的，而且默认会生成很多国家的语言像这样

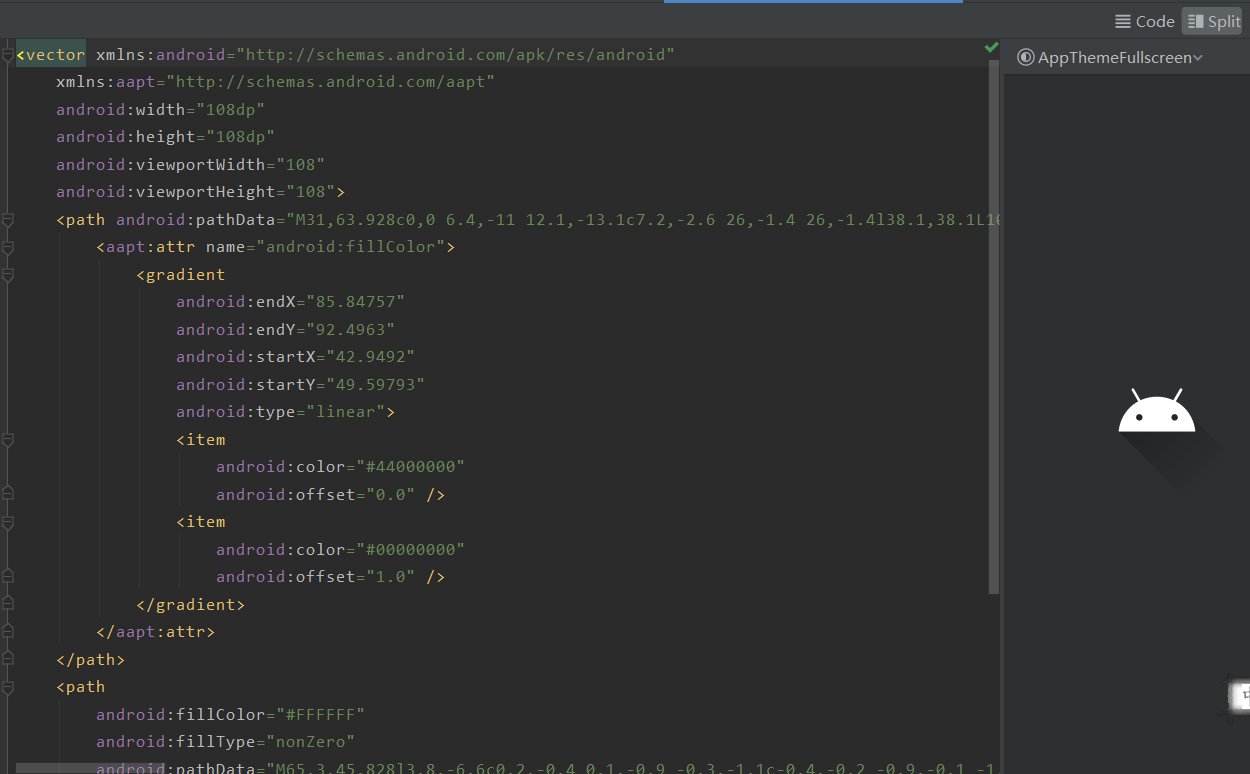


我们只要设置只支持中文或者英文就可以了

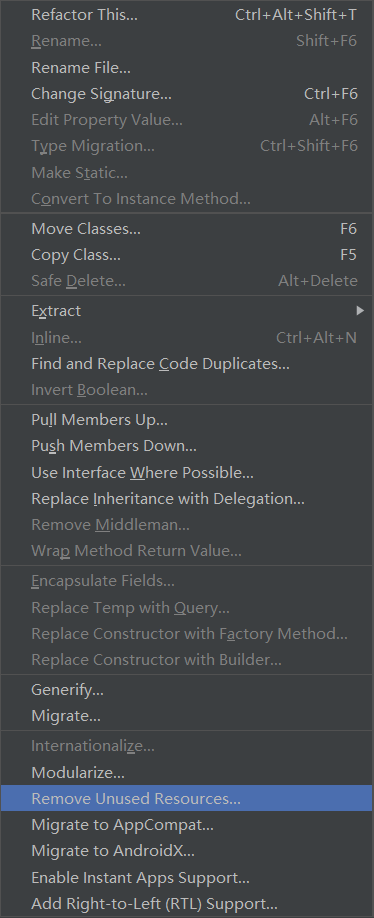
1. 一些图片转换成webp格式，当然了压缩的时候肯定会有是真的情况，这个就看你怎么取舍了，如果失真不明显那么能压缩还是要压缩的；一些小的图标可以使用Vector矢量图，这个体积也是比png小很多，而且不需要进行适配，矢量图可以根据实际需求显示大小，不想使用png还要区分各种屏幕大小



矢量图vector，现在新建个项目android的logo就是vector矢量图



1. 常规手段：混淆(代码混淆、资源文件混淆)、去除无用文件；下图是去处无用文件的，混淆我就不解释了



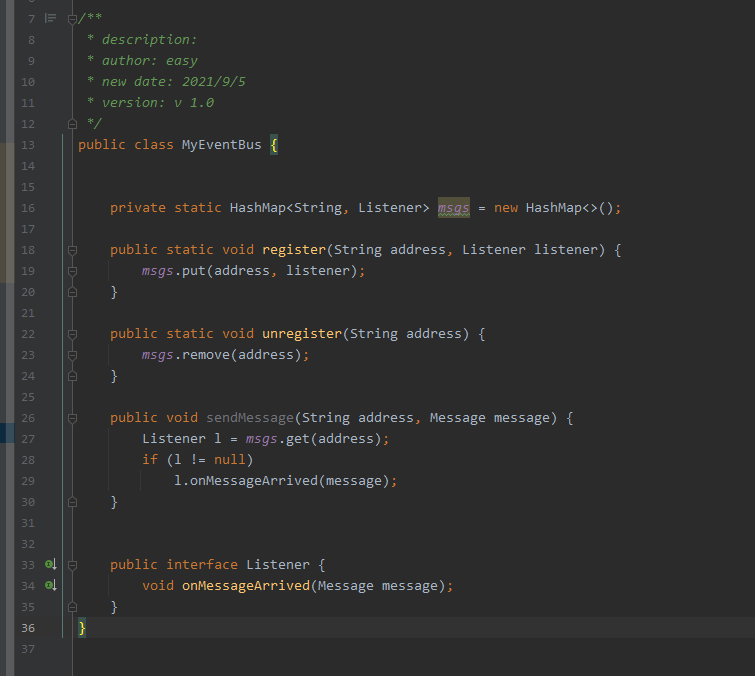
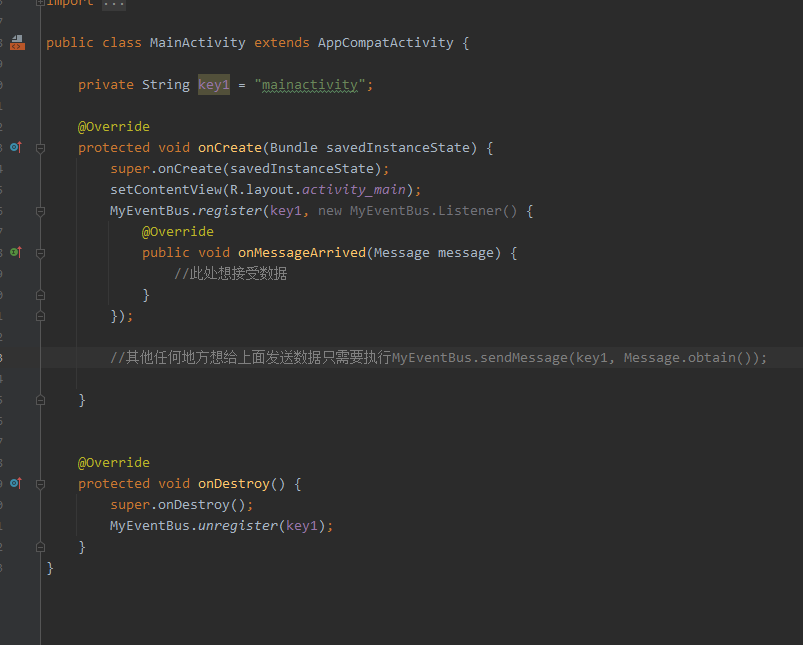
1. 优化结果：项目体积减小了很多，主要是so文件的功劳，so文件的选择对于android体积减小是最客观的，其他的作用都不是太大，但是积少成多效果很明显的；上面的优化方案：资源文件的混淆和Vector矢量图的优化方案在当时的项目是没用上的，那时候还不知道这些，但是借这次机会也总结出来分享给大家

# 网络优化

1. 这个我觉得没什么太好的办法，首先根据当前网络情况来处理数据，如果是网络比较差的时候，可疑更换网络协议，直接使用tcp、mqtt之类的；如果是http那么打开gzip 压缩；使用ip地址免解析等等
2. 自己创建数据解析和压缩的字典；
3. 数据缓存、连接池复用、合并请求

# 第三方库的一些简单替代方案

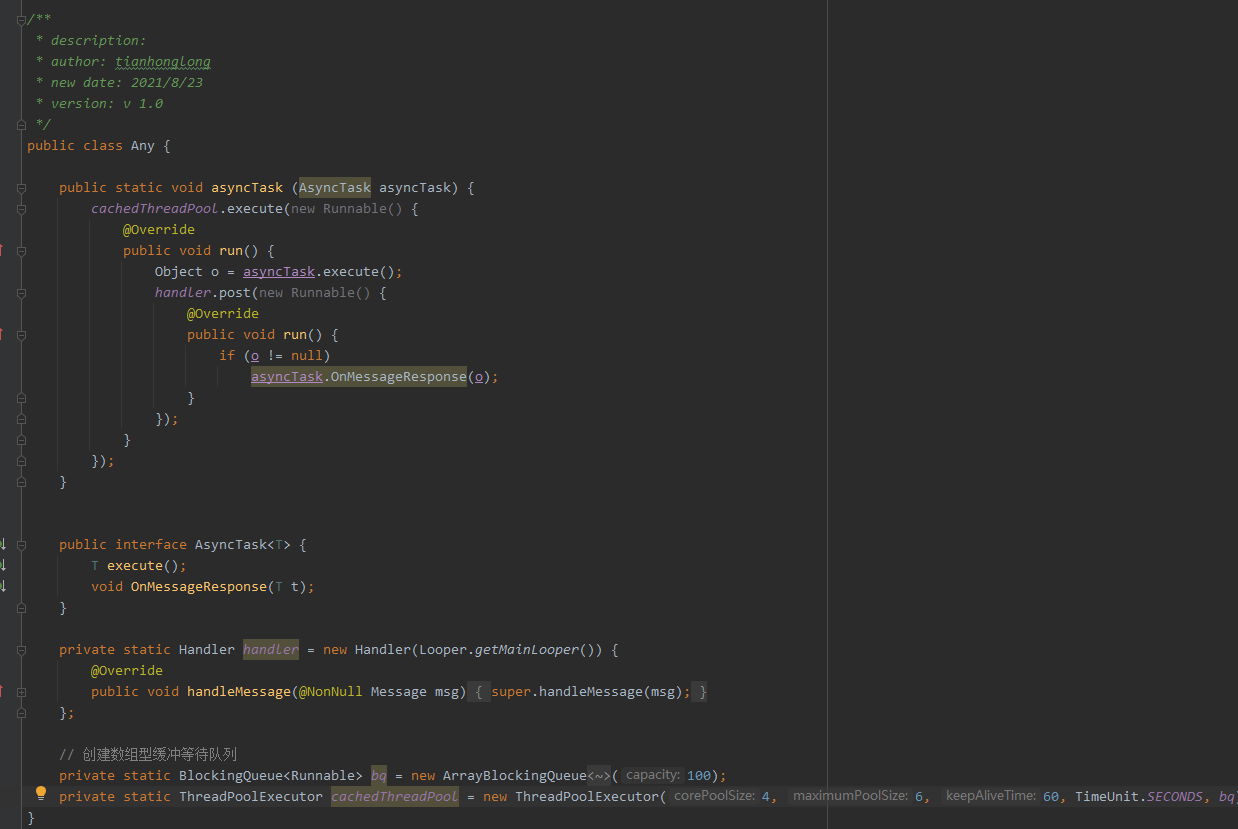
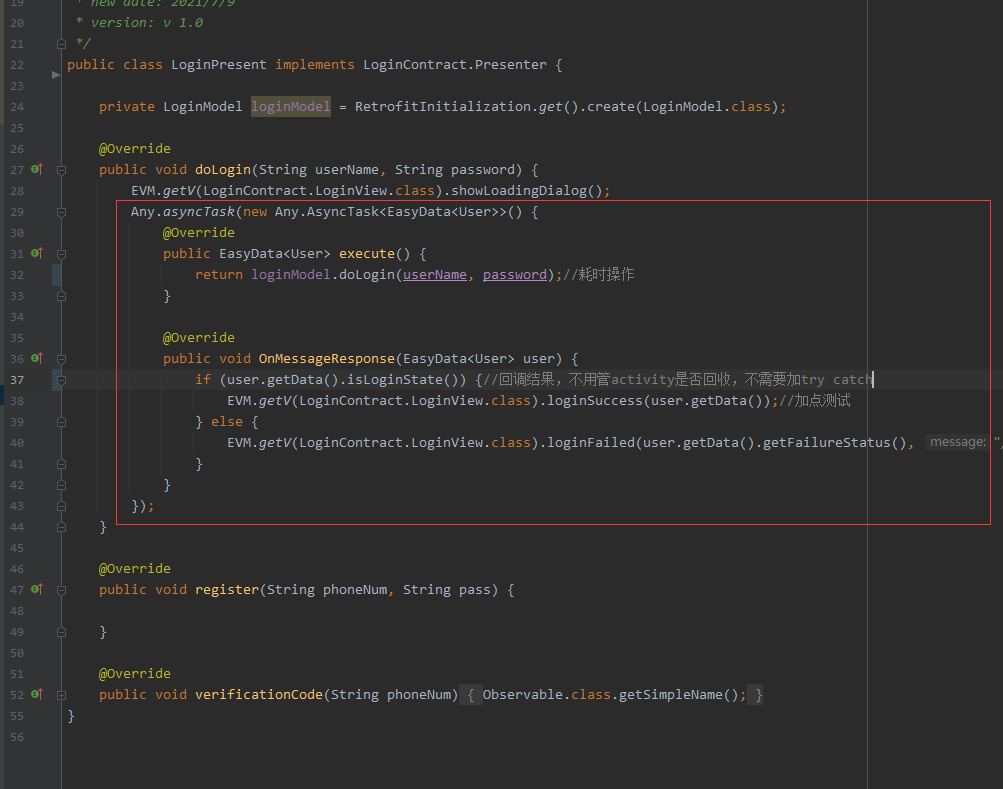
## EventBus

这虽然是一个很好用的库，但是有很严重的性能问题，对于我这种代码洁癖的人来说不可忍受；他提供的库虽然很丰富，但是要遍历一个类里的所有方法，然后识别出需要的，尤其是在activity里边，本身activity的的方法和变量都特别多，无形之中就是消耗的许多性能；解决方案比如可以这样写：想在哪里接受数据，就在哪里执行注册，比如在acitivty中：当然，这里面也有内存泄露的问题，匿名内部类对象；此处只做案例，如果想切换线程，也可在里面创建一个handler，就可以做到eventBus的全部工作，但是性能会比它好很多

## **注解绑定控件库**：

注解去实例化View控件也是存在这个问题；会遍历所有的成员变量；并且会额外产生类，并且包含你所有的控件成员变量，相当于acitivty站用内存相当于翻倍了，多声明一个成员变量就多占用四个字节的空间，而且还会多创建一个对象，16个字节；如果页面比较多累计起来不仅消耗内存还消耗性能，像Xuitls、buffterknife等等都是相同的原理

## RxJava的替代方案：

可疑自己写一个简单，只要能符合当前app的业务需求，只是不满足，也可以继续改进满足，比如下面的例子：当然，如果你的业务中特别复杂，第三方库了的所有功能基本都能用到，那还是直接用他们的比较好；这个就是开发方便和性能之间的一个平衡。使用方法：

1. 结语：大多数app的业务没那么复杂，第三方库提供能的功能可以说异常的丰富；对于大部分项目来说过于丰富，很多功能是用不上的，所以就造成了额外的性能消耗；当然自己写的时候要注意内存泄露的问题，如果对内存泄漏问题掌握的不够自信，那老老实实用第三方库也是可以的

# 图片加载库的问题以及性能比较

## **1、问题背景：**

这个也是15年银行的电商app项目性能优化的时候，碰到的问题，因为内容比较多，所以单拿出来说一说；比如Imageloader性能最好，但是有很严重的内存泄露的问题；Fresco 会彻底解决内存的问题，但是使用起来却没有imageLoader流畅；picsso 在我看来和ImageLoader差不多；现在最长的是glide，因为glide有一些自动管理图片加载的机制和context的生命周期的管理；在15年的时候，列表类的图片，在滑动的过程中要自己手动停止图片加载功能；这个gilde都帮我们处理了；还有context的声明周期管理，避免的内存泄露的产生

## **2、图片库测试结果**：

Fresco稳定性最好；Glide体验最流畅也没有内存泄露的问题；有自动的生命周期管理，ImageLoader、Xutils、piacsso基本被淘汰了；现如今反编译很多知名厂商app、使用的图片库都是Glide

## **3、测试过程**

当时是写了一个相册的功能，分别使用以上图片加载库去加载相册，然后反复打开关闭相册页面，记录打开次数、流畅度；测的最终上面的结论

## **4、使用建议**：

个人建议：xUitls3、Imageloader、picasso可以抛弃了；Imageloader有很严重的内存泄露问题，而且也不更新了xUtils和picasso估计也有，这是机制的问题；毕竟网络请求是一个耗时的 操作，都需要传入context上下文

建议使用Glide：理由

(1)、绝大多数主流app，反编译这些源码，使用的都是Glide；可见它的受欢迎程度

(2)、它自有的Context生命周期管理，可以在activity/frgment页面关闭的时候Glide可以在第一时间检测到，停止图片的下载，这样就防止了内存的泄露和cpu资源的消耗；

(3)、glide支持gif图片

(4)、支持配合滑动列表滑动时候停止加载图片

(5)、最后说Fresco，这是个压箱底的东西，如果Glide都不能满足的时候，再把它拿出来；Fresco用c写的库，把图片数据存储在了ashmem区域，这样就不占用jvm堆内存了，所以说它的终极大招；但是他使用起来却没有前面几个java的体验感更加流畅，所以把它放在最后的选择；

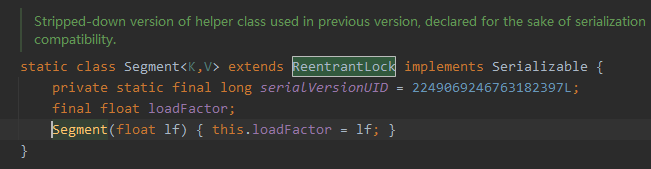
至于其他的一些缓存机制、缓存策略这些，比如缓存的是压缩后的图片还是原图，这个区别不大，而且都可以手动修改设置，所以这些机制就不做参考了

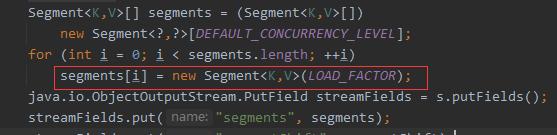
# 锁的优化synchronized和lock

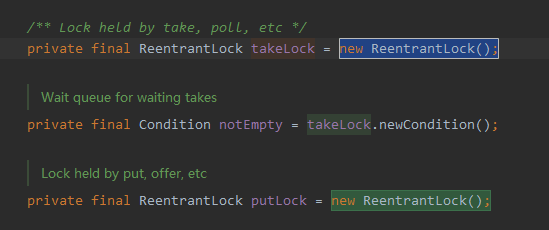
这个就简单一说吧，要了解synchronized的锁升级过程、粗化、消除等等；lock里的抽象队列同步器，cas自旋、用户态、内核态、操作系统互斥量；内容较多，可以自行学习，**我就简单说下结论：并发量小就用sync关键字，并发量大就使用Lock**

## 锁的优化方向：

1. 尽量减小加锁部分代码的执行时间，因为可能有其他线程在等待，等待的线程越多，最后的那个线程能执行到加锁里的内容的时间越长
2. 减小锁的力度或者是范围：比如Map中的ConcurrentHashMap，它是一个线程安全的集合，锁只是锁住了单独的桶，就算是两个线程同时写入数据，只要hash值算的下标不再同一个位置就不会有影响；

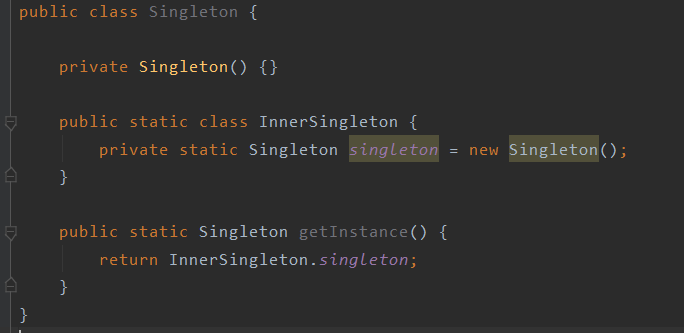


.

1. 锁分离：把两种互不影响的操作，分别加锁，比如linkedBlockQueue

我们看到针对不同的操作，分别用不同的锁；

## **延伸单例写法推荐**

单例推荐静态内部类单例，即使线程安全的也是懒加载，而且不需要枷锁，所以性能上会节省一丢丢；

# 吗MVP设计模式的弊端及解决方案(重点)

## Mvp存在的问题：

从最开始接触mvp模式，不断的思考和改进mvp的写法，以达到最好的要求，代码量小，解耦；业务逻辑清晰，尝试过很多次，今年自己要写开源项目，所以干脆就根据多年积累重新整理了一个mvp的写法，已经放到github上了，欢迎交流

### **首先是设计思路问题**

是按照业务就划分View还按照接口去划分View;如何去划分Persent；present和model还有View之间是不是要一一对应的关系；其实这就牵扯了activity；如果按照接口去划分View那么View直接定义成ResponseData类型的返回值就行了；有几个接口就定义多少个View；另外一种方式就是按照业务去划分，比如登录的业务可能包括登录、验证码、短信登录、忘记密码，这些操作，把这些业务都写到一个LoginContract里边，再分别定义其他各种业务，我是比较推崇后者的

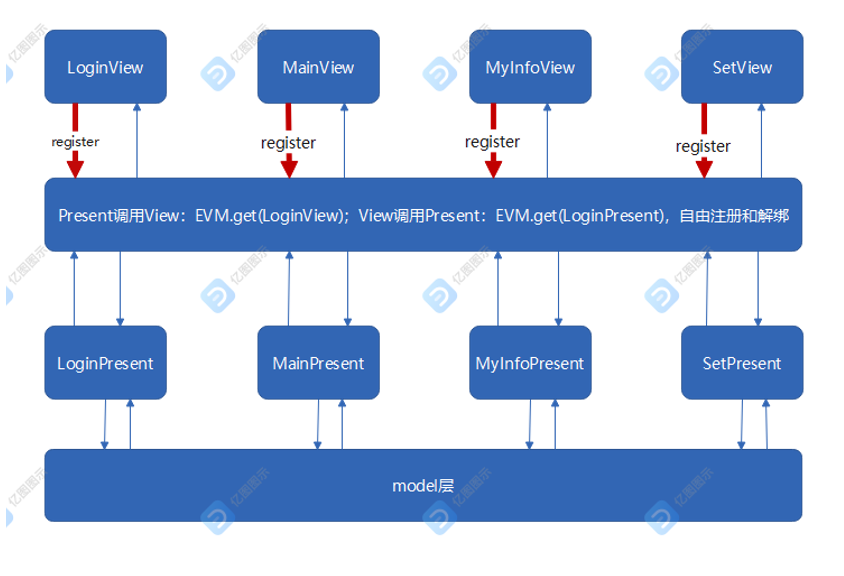
### **过多的present和model的问题**

如果一个页面中，只有一个接口，一个简单的功能，也需要额外写一个present、model、view吗？这样就很冗余；也不利于开发速度；如果这样的activity还特别多，那就是写一个actvity就要写一个view和present，显然这样代码量太大太蠢了

### 内存泄露的问题

我们知道网络请求是耗时操作，一旦网络不好的情况用户又关闭了activity，此时这个activity是无法被回收掉的这就造成了内存泄露问题。Activity一般是内存占用大户，虽然可以用弱引用去处理，但是弱引用也会额外的创建对象，会增大内存的占用

### **解决办法**

创建Evm中间类，让present和View、model中间，不产生强引用关系，所以也就不会产生内存泄露的问题；再就是present和modle都可以自由定义，想定义几个就定义几个，都可以通过EVM中间类进行关联，而且采用的是反射的方法，只获取接口，所以对性能无影响，不像EventBus和ButterKnife那样要遍历所有的方法或者变量，而且如果View已经回收，则会生成一个临时该接口类型的对象，不需要是否为空判断代码是：

/\*\*

\* description:

\* author: tianhonglong

\* new date: 2021/7/9

\* version: v 1.0

\*/

public class EVM {

private EVM() {

}

private Map<String, EasyView> views = new HashMap<>();

private Map<String, EasyPresent> presents = new HashMap<>();

private <T extends EasyView> T getView(Class<T> clazz) {

EasyView easyView = views.get(clazz.getSimpleName());

if (easyView == null) {

try {

return clazz.newInstance();

} catch (IllegalAccessException e) {

e.printStackTrace();

} catch (InstantiationException e) {

e.printStackTrace();

}

}

return (T) easyView;

}

private <T extends EasyPresent> T getPre(Class<T> clazz) {

EasyPresent present = presents.get(clazz.getSimpleName());

if (present == null) {

try {

T t = clazz.newInstance();

presents.put(clazz.getSimpleName(), t);

return t;

} catch (IllegalAccessException e) {

e.printStackTrace();

} catch (InstantiationException e) {

e.printStackTrace();

}

}

return (T) present;

}

private void managerView(EasyView easyView, boolean registerOrNot) {

Class[] classes = easyView.getClass().getInterfaces();

for (Class clazz : classes) {

if (EasyView.class.isAssignableFrom(clazz)) {

if (registerOrNot) {

views.put(clazz.getSimpleName(), easyView);

} else {

views.remove(clazz.getSimpleName());

}

}

}

}

public static void register(EasyView easyView) {

EVM.ins().managerView(easyView, true);

}

public static void unregister(EasyView easyView) {

EVM.ins().managerView(easyView, false);

}

public static <T extends EasyView> T getV(Class<T> clazz) {

return EVM.ins().getView(clazz);

}

public static <T extends EasyPresent> T getP(Class<T> clazz) {

return EVM.ins().getPre(clazz);

}

private static class InnerClass {

private static EVM easyPresent = new EVM();

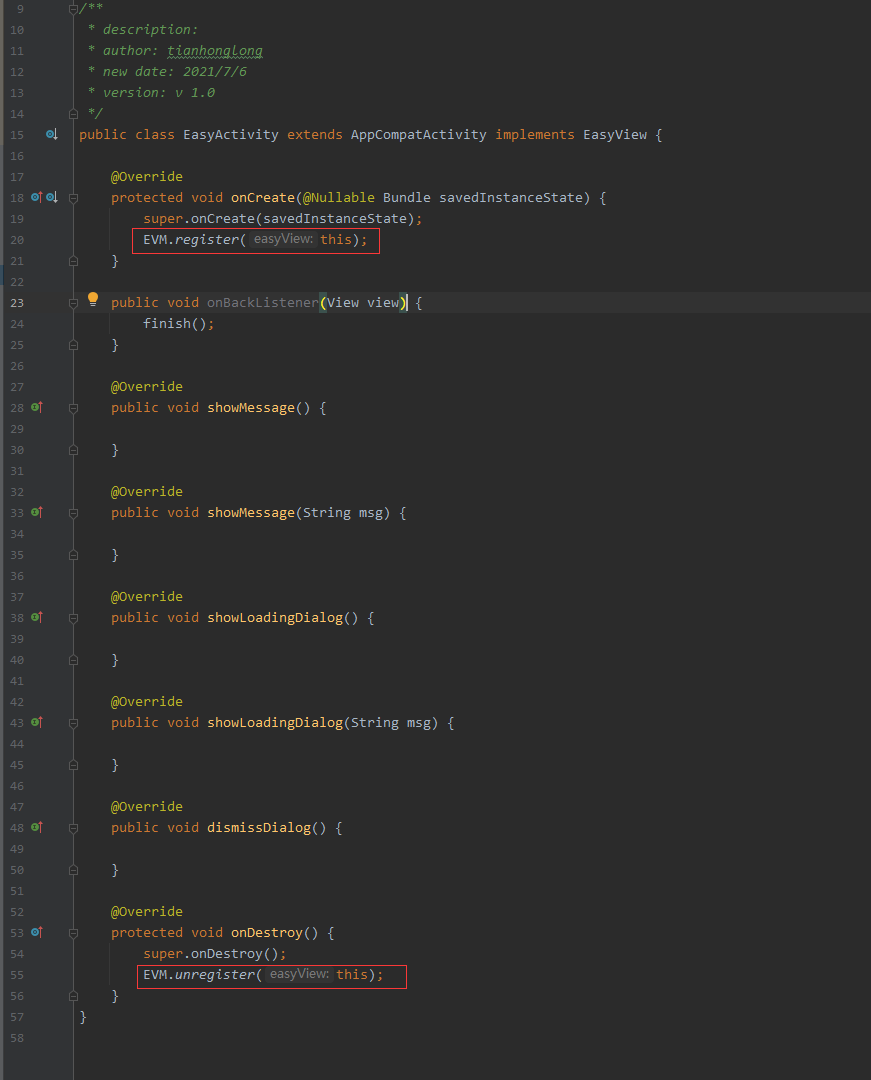
}

private static EVM ins() {

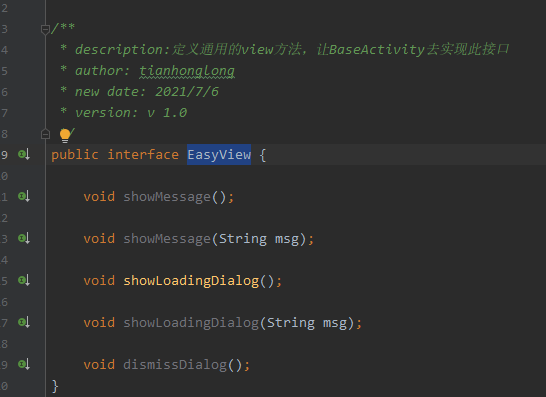
return InnerClass.easyPresent;

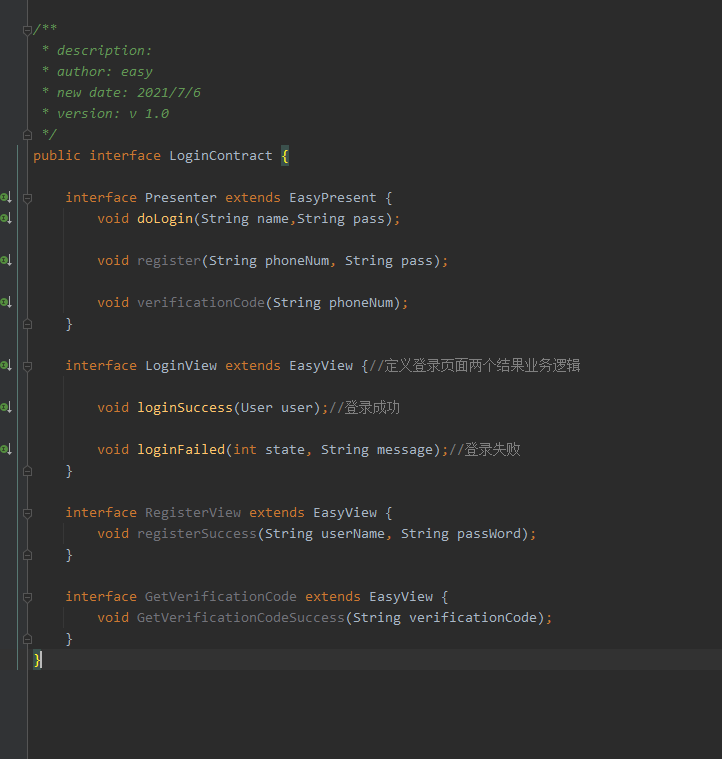
}

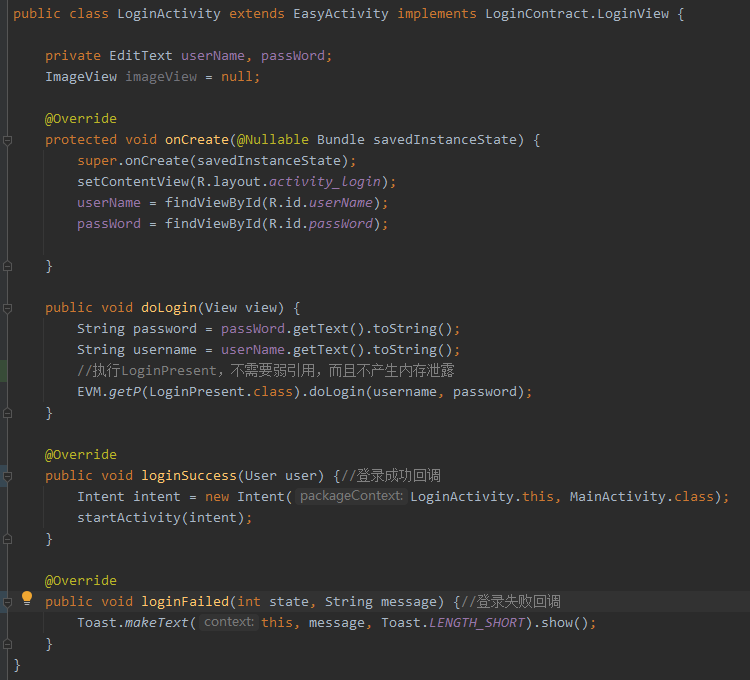
}

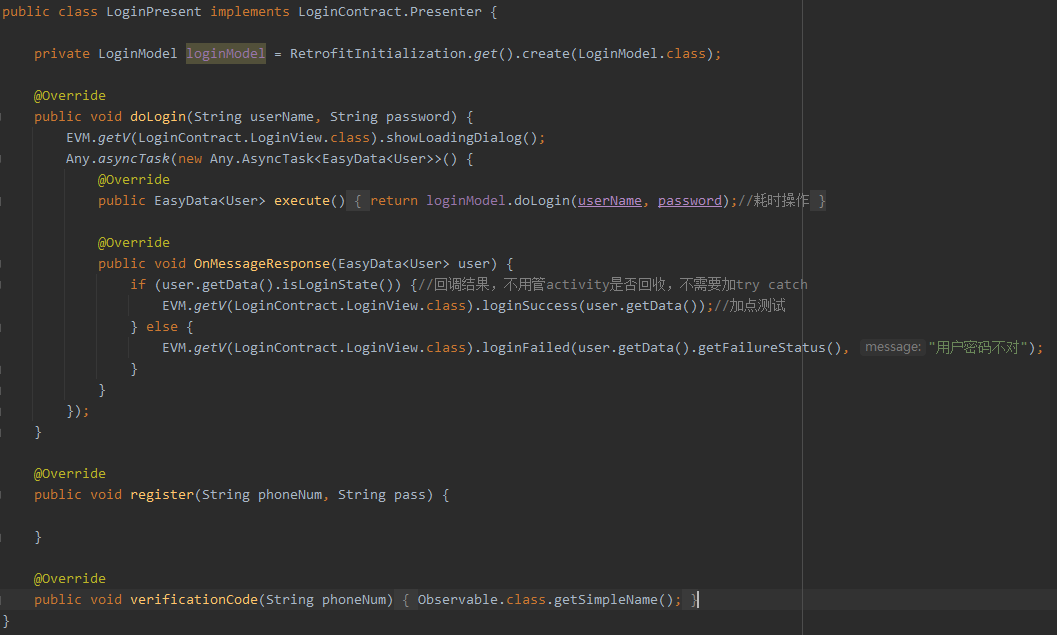
在basectivity中使用：

而且只搜索存在接的接口，一个类的接口不会太多，所以不会影响性能

定义一个基类接口

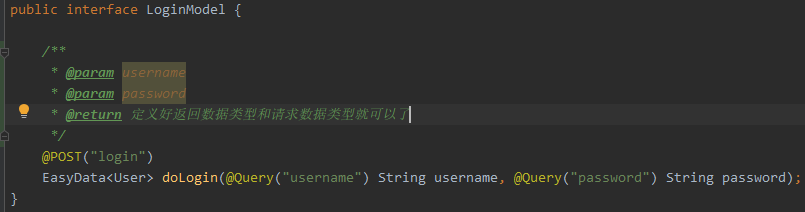
**写一个LoginContract，把和登录有关的View和present都定义在里边**

LoginActivity中，

在看看present代码：优雅的很

### 此mvp设计总结：

1. 彻底解除View、Present、model之间的关联，其中model、View、present各自想写几个就写几个，通过evm中间类关联，都可以互相调用；
2. 解决了View的内存泄露问题，不产生直接强引用就能互相调用，所以不会影响内存回收;
3. 只需要关注业务的开发，针对接口去设计model



1. 只需要在present里面处理业务流程，actvity处理业务的发起和结果接受；modle按照接口文档傻瓜式写上就可以了，也不需要加各种try catch；底层会帮业务处理好，直接回调给当前的View

# 九、View绘制方面的优化

## **（1）多嵌套问题：**

优化别人代码的时候，布局文件中最常见的就是过多嵌套问题，很多子空间就可以实现的，很多人非要加个父控件，比如LinearLayout、RelativeLayout等等，再就是用merge可以减少嵌套，由于空间渲染计算是递归形式，在以前的老旧机型里，只要嵌套7层Layout，就会有明显的卡顿产生，但是如果同级别里，就算是14层也不会有任何问题，所以某些界面使用LinearLayout并不会比RelativeLayout带来更多控件层级时，优先考虑LinearLayout;；

## **（2）各种控件性能比较：**

性能最差的是RelativeLayout；如果LinearLayout和FrameLaout都满足的情况下，优先排除RelativeLayout; 如果再复杂就是使用google新出的ConstraintLayout; 还有RecyclerView的性能都要比ListView好很多，比如加载fragment除了用懒加载外，replace也add要节省内存

## **（3）使用占位符ViewStub：**

在一些界面里的控件，需要最开始状态是View.GONE；但即使是View.GONE，这个控件依然会执行各种实例化方法创建对象占用内存，只是最后没有通过wms渲染在手机屏幕上而已，ViewStub是一个轻量级的View；使用ViewStub后，那么原来的控件就相当于懒加载了，只有用户用手操作让它显示的时候才会去加载，如果一直不要求显示那就永远不会加载；只是会多一个轻量级的ViewStub

## **（4）**结语：

写代码的过程中，有许多不经意的点都是可以节省内存的，只要节省每一处的内存和cpu的使用；累计起来就是一个好的项目；如果对内存不省吃俭用，累积起来的内存泄露造成的oom是最难解决和优化的；

# 十、一些知名的内存管理监测软件

## (1) LeakCanary

就是用WeakReference和ReferenceQueue 这两个类的机制，来检查内存是否泄露的；可疑帮助你快速定位内存泄露的点；但是不能帮你解决内存泄露的问题，只是帮你发现问题

## (2) koom

快手自研OOM解决方案。效率据说比leakCanary好的多，具体还没使用过，推荐给大家；

# 十一、数据加密优化

## 项目背景：

这个是20年优化的一个项目，因为在银行和金融app行业经验丰富一些，所以在一些非银行类app中的数据安全这一块，做的有问题；直接用非对称去加密、解密数据；这个性能是有很大问题的；

## 优化方案：

要把对称加密和非对称加密结合起来使用，通信数据要用对称加密去加密和解密，然后把对称加密的钥匙，用非对称加密进行加解密，然后把加密后的钥匙拼接在隐藏在数据中，这样在安全性不变的情况下，性能会大大的提升

### 对称加密

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 名称 | 密钥长度 | 运算速度 | 安全性 | 资源消耗 |
| DES | 56位 | 较快 | 低 | 中 |
| 3DES | 112位或168位 | 慢 | 中 | 高 |
| AES | 128、192、256位 | 快 | 高 | 低 |

### 非对称加密

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 名称 | 成熟度 | 安全性(取决于密钥长度) | 运算速度 | 资源消耗 |
| RSA | 高 | 高 | 慢 | 高 |
| DSA | 高 | 高 | 慢 | 只能用于数字签名 |
| ECC | 低 | 高 | 快 | 低(计算量小,存储空间占用小,带宽要求低) |

### 散列算法：

|  |  |  |
| --- | --- | --- |
| 名称 | 安全性 | 速度 |
| SHA-1 | 高 | 慢 |
| MD5 | 中 | 快 |

### 常用的加密方案：

MD5：比如用户输入的密码，用MD5进行加密，直接存储在后台的就是MD5数据，再就是校验数据的完整性

Sha-1：一般签名密钥里东西

对称加密：加密基本数据，因为性能比非对称加密好的多

非对称加密：加密对称加密的钥匙，组合使用，这样安全性即达到了非对称级别，性能也上去了；

总结：

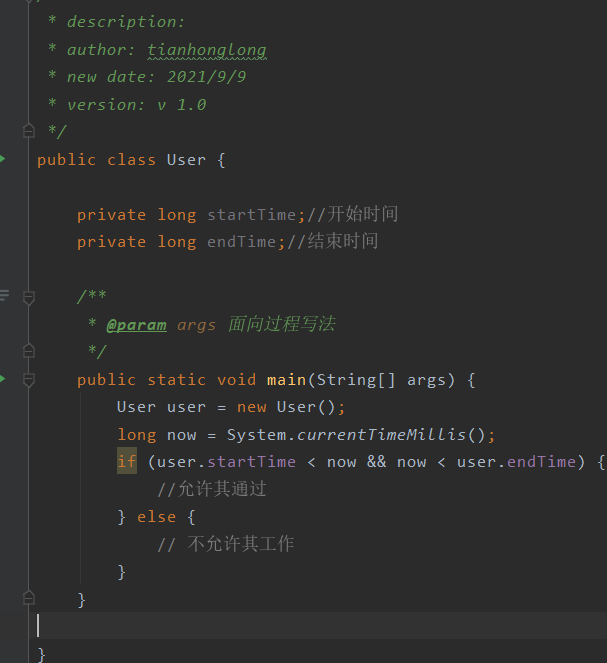
MD5校验完整性+对称加密加密全部数据+非对称加密对称加密的密钥

最常用的就是MD5+RSA+AES

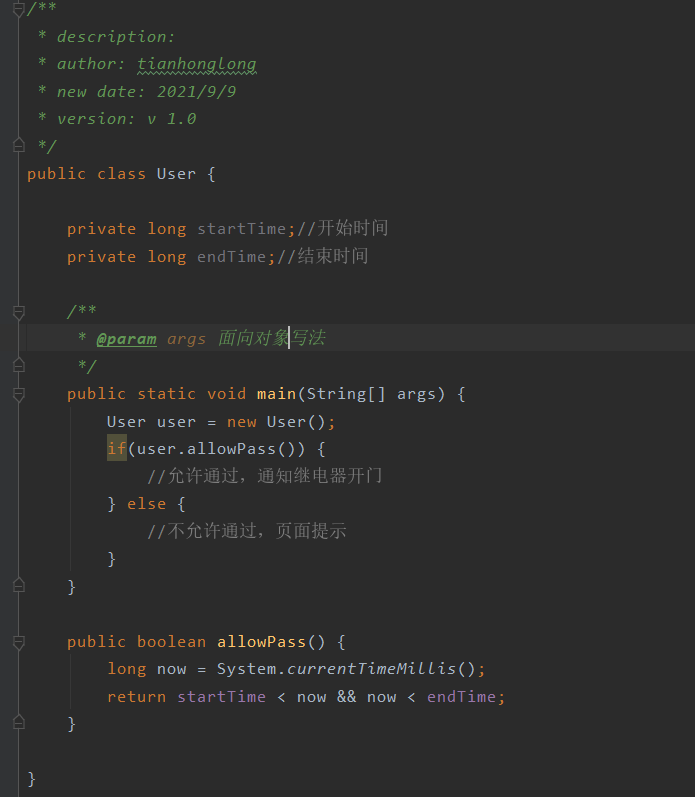
# 十二、修正一下编程思想！面向对象 or 面向过程

## （1）面向对象开发和面向过程开发

这个是优化现在公司里的一个项目；虽然java是面向对象语言，但是在很多项目中的同事，仍然使用面向过程的思维模式开发；举个简单的例子，比如曾经做的人脸识别app中，有这样一个场景，在人脸识别拿到数据之后，产生了一个用户对象，里面包含了用户的起止时间用户身份；先看面向过程写法，极其简单的一个案例：



再看面向对象写法：



可读性和简洁性对比明显；就是在写代码的时候要分清业务流程和业务细节；此处的流程就是此人是否可通行，业务细节就是判断此人是否可通行的过程；这部分代码不要出现在流程里边

上面只是写了个简单的例子，我们项目中实际的是否允许通过的判断要复杂的多，包括多分组、多时段、所以判断过程很复杂代码量也不少

# 十三、架构方面：对app代码业务逻辑的一些设计思考

## 1、项目背景：

这个是优化了代码的业务逻辑，其中最典型的人脸识别页面activity，有7500+行代码，里面包含了各种业务包括：数据的同步、UI的显示(识别结果展示)、人脸认证的过程(包括人脸、温度、口罩、距离识别、硬件接口回调等等)，还包括一些业务细节的处理比如最后两次人脸是否同一个人，wifi状态监听的等等吧；

## **2、优化思路**：

按照业务分类，可分为UI部分、验证流程部分、验证细节处理部分、面向对象部分；拆分成三个activity，原来是一个FaceVerifyActivity，如今拆分成FaceBusinessActivity，FaceUIActivity,其中继承关系：

FaceVerifyActivity 继承FaceBusinessActivity 继承 FaceUIActivity

**FaceUIActivity**的功能应该只包含UI部分，和人脸认证没有任何逻辑关系，只是提供了人脸结果出来的时候，可能需要显示的各种dialog、或者其他UI

**FaceBusinessActivity**里面包含最后两次是否同一个人的判断方法，数据同步的方法等等

**FaceVerifyActivity：**纯粹是业务流程的判断，然后根据每次判断的结果，来调用FaceUIActivity和FaceBusinessActivity的方法，这样整个业务逻辑就清晰很多

## 3、详细内容，举例说明

由于原项目代码量特别大，所以也不可能全部贴出来；所以此处举例说明：

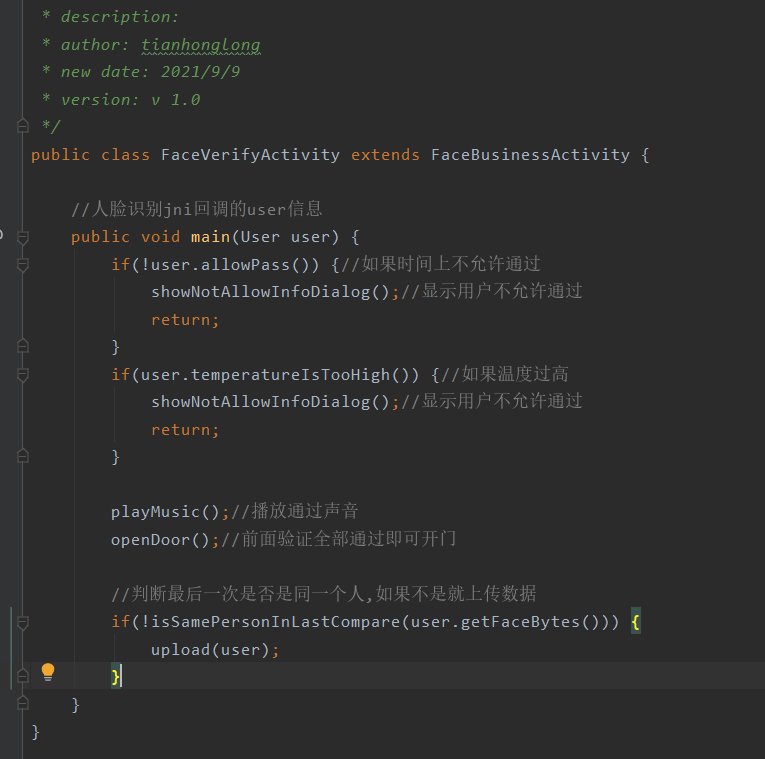
首先是UIActivity，定义了一些需要显示的UI对话框之类的



其次是业务Activity，定义了开门、数据同步，是否同一个人等等



最后是主流程activity：纯粹的流程判断



1. 在案例中每个页面30~40行代码，如果不拆分，不按照面向对象写那么在一个页面中就会有110行的代码，这只是案例，真实项目中把案例代码量扩充70倍，才是我们项目中的实际代码，一个activity有7000多行代码，谁不头疼？拆分后分成三部分，每部分就两千多行，而且除了FaceVerifyActivity外，其他几个类中方法之间没有调用，都是单独的业务方法，以后修改起来会简单容易的多；这其实也是属于模块化思想，按类型分类，就是目的都是为了让代码单一、简洁、易修改、便于扩展；

# 十四、数据库和高并发优化

## 1、问题背景

首先说惭愧；这个问题不是我实际工作的经验；这个是去某家公司面试的时候，面试官问的问题，问的高并发的数据存储和数据优化，回答的不是太好，也确实没这方面经验，本着不会就要学习的态度，还是研究了很多文章和代码，在此也分享出来，在此直说客户端如何解决

## 万人群高并发优化

问题1：消息从未读到已读，这个消息如何发送？ 如果是实时的，那一条已读消息要发送给一万个人，如果一万个人同事读了这条消息，就会发生一万个人同事给一万个人发消息，并发量可想而知；这个解决办法就是降低频率，是否已读，10秒钟才去刷新一次，这样就大大降低了消息的并发数

问题2：一个人接受到不同人的多条信息时，建立缓冲区。，合并多条消息给一人，甲乙丙丁同时给A发送消息，那么就建立缓冲机制，把甲乙丙丁合成一条消息，这样最后技术层面A只收到一套消息，总之目的就是降低并发数量；

## 数据库优化

数据库的并发优化和IM通信类似，数据里边有事务，其实就相当于建立缓冲区合并多条数据，然后开启专门的现成去执行数据库的操作

### (1)、开启事务

Android中的sqlite是默认开启事务的，就算只是只有一条数据的插入更新也会帮你开启事务，所以当并发量大的时候，把多条执行语句放在一个事务里，这样就提高了性能，不用每次都打开关闭事务；

### (2)、建立索引

索引就是把数据库里的数据，建立一个目录，在查找的时候不用一页一页去查了，索引一般是平衡树结构；简单说就是利用算法和数据结构提高效率，但是貌似在数据量特别大的时候，维护索引也会产生不小的开销；

### (3)、耗时的话进行异步操作

比如有些数据的存储和同步不需要知道返回结果，这样建立一个线程池去执行这些任务，这样就不会影响主线程操作

# 十五、常见的ANR操作

原理也很简单，系统服务ams和wms会检测app响应时间，也就是本地的applicationThread是否会及时的给ams和wsm发送binder信息，如果超过一定的时间未发送，系统就会认为你卡住了；就会提示无响应，因为通过applicationThread给系统服务发送信息都是通过主线程来执行的，所以一旦在主线程中执行耗时操作就会引起ANR

# 十六、android特有的库和一些代码基础写法

1. 对象的序列化android 特有的Parcelable比Serializable的性能好
2. Android特有的集合：在数据量小的情况下使用SparseArray、ArrayMap代替java集合
3. Map的多种遍历方式，那种效率最快
4. 在写基础库的时候，尤其是处理数据，线程安全的情况下使用StringBuilder、线程不全的情况下使用StringBuffer
5. 根据数据存储和使用情况来判断使用链表集合还是数组集合
6. 尽量使用基本数据类型，比如int类型的成员变量一共就占用四个字节的堆空间。如果用Integer除了所在类中成员变量的四个字节外，还会有Integer对象的16个字节；
7. 循环中减少对变量的重新计算

比如：for(int I = 0; i < list.size(); i++) 改为for(int I = 0, len = list.size(); i = len; i++)

1. 避免使用二位数组，数据比较特殊，不管你是否存入对象，数组创建的那一刻，内存已经消耗掉了，数组里每个指针占用四个字节；不算对象头和数组长度，一个长度为10的二维空数组创建的那一刻就是占用10\*10\*4 = 400个字节；还不算对象头类指针数组长度；像ArrayList也一样，因为都是数组
2. Json序列化性能对比：数据量小就用gson，数据量大就用阿里巴巴的fastjson