# 内存泄漏和性能优化

借鉴文章：https://www.jianshu.com/p/a0e242d57360

##### 3.1 内存泄漏如何查找

3.1.0 什么是内存泄漏

当一个对象已经不需要再使用了，本该被回收时，而有另外一个正在使用的对象持有它的引用，从而就导致对象不能被回收。这种导致了本该被回收的对象不能被回收而停留在堆内存中，就产生了内存泄漏。

3.1.1 产生的内存泄露

* 资源对象没关闭造成的内存泄漏

资源性对象如Cursor、Stream、Socket，Bitmap，应该在使用后及时关闭。未在finally中关闭，会导致异常情况下资源对象未被释放的隐患。

* 构造Adapter时，没有使用缓存的convertView
* Bitmap对象不在使用时调用recycle()释放内存
* 试着使用关于application的context来替代和activity相关的context
* 注册没取消造成的内存泄漏

我们常常写很多的Listener,未反注册会导致观察者列表里维持着对象的引用，阻止垃圾回收。

* 集合中对象没清理造成的内存泄漏
* 非静态内部类的静态实例。

由于内部类默认持有外部类的引用，而静态实例属于类。所以，当外部类被销毁时，内部类仍然持有外部类的引用，致使外部类无法被GC回收。因此造成内存泄露。

* 类的静态变量持有大数据对象

静态变量长期维持到大数据对象的引用，阻止垃圾回收。

* Context泄露  
  这个太多了，不细说，单利模式写的不恰当就属于这种。
* Handler临时性内存泄露

Handler通过发送Message与主线程交互，Message发出之后是存储在MessageQueue中的，有些Message也不是马上就被处理的。

...

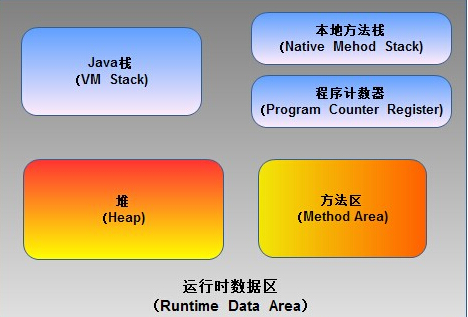
查找内存泄漏

查找内存泄漏可以使用Android Stdio3.0 自带的Android Profiler工具(替代了android monitor),也可以使用Square产品的LeadCanary

3.1.2 Java内存区域如何划分

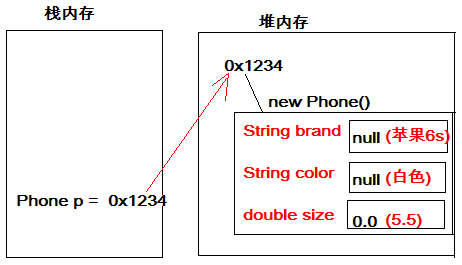
有两种说法：  
 一种说法是分为: 堆（Heap），栈（Stacks）方法区（MethodArea），运行时常量池（RuntimeConstant Pool），本地方法栈（NativeMethod Stacks），PC Register(PC寄存器)。是从抽象的JVM的角度去看的。  
 另一种说法是分为：堆（Heap），栈（Stacks），数据段（data segment），代码段（code segment）。则是从操作系统上的进程的角度去看的。

看看第一种方法的内存图



在上图中，我们主要区分堆/栈

* Heap内存的分配也叫做动态内存分配，java中运行环境用来分配给对象和JRE类的内存都在堆内存，C/C++有时候可以用malloc或者new来申请分配一个内存。在C/C++可能需要自己负责释放（java里面直接依赖GC机制）。
* Stack内存是相对于线程Thread而言的, 在执行函数(方法)时，函数一些内部变量的存储都可以放在栈上面创建，函数执行结束的时候这些存储单元就会自动被释放掉。栈内存包括分配的运算速度很快，因为内置在处理器的里面的。当然容量有限。它保存线程中方法中短期存在的变量值和对Heap中对象的引用等.



区别：

堆是不连续的内存区域，堆空间比较灵活也特别大。 栈式一块连续的内存区域，大小是有操作系统觉决定的。堆管理很麻烦，频繁地new/remove会造成大量的内存碎片，这样就会慢慢导致效率低下。对于栈的话，他先进后出，进出完全不会产生碎片，运行效率高且稳定。

我们通常说的内存泄露，GC，是针对Heap内存的. 因为Stack内存在函数出栈的时候就销毁了。

public class People{

int a = 1;

Student s1 = new Student();

public void XXX(){

int b = 1;

Student s2 = new Student();

}

}

请问a的内存在哪里，b的内存在哪里，s1，s2的内存在哪里？记住下面两句话。

* 成员变量全部存储在堆中(包括基本数据类型，引用及引用的对象实体)，因为他们属于类，类对象最终还是要被new出来的。
* 局部变量的基本数据类型和引用存储于栈当中，引用的对象实体存储在堆中。因为他们属于方法当中的变量，生命周期会随着方法一起结束。

所以答案就是a，s1，s2对象在堆中，b和s2对象引用在栈中。

3.1.3 java中的引用

把对象的引用分为四种级别，从而使程序能更加灵活的控制对象的生命周期

* 强引用

我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于必不可少的生活用品，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问题。

* 软引用

如果内存空间足够，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。

* 弱引用

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。

* 虚引用

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。虚 引用主要用来跟踪对象被垃圾回收的活动

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。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

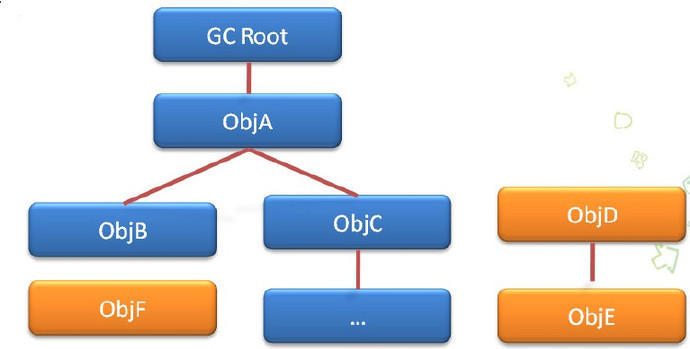
3.1.4 垃圾回收

GC 是 garbage collection 的缩写, 垃圾回收的意思. 也可以是 Garbage Collector, 也就是垃圾回收器.

垃圾回收机制有好几套算法，java语言规范没有明确的说明JVM 使用哪种垃圾回收算法，但是任何一种垃圾回收算法一般要做两件基本事情：

1. 发现无用的信息对象；
2. 回收将无用对象占用的内存空间。使该空间可被程序再次使用。

有一种算法是根搜索算法



根搜索算法是从离散数学中的图论引入的，程序把所有的引用关系看作一张图，从一个节点GC ROOT开始，寻找对应的引用节点，找到这个节点以后，继续寻找这个节点的引用节点，当所有的引用节点寻找完毕之后，剩余的节点则被认为是没有被引用到的节点，即无用的节点。如果这个对象是引用可达的, 则称之为活的(live), 反之, 如果这个对象引用不可达, 则称之为死的(dead), 也可以称之为垃圾(garbage).这个引用可达与不可达就是相对于GC Root来说的，在上图中，左边4个对象就是活的，右边两个就是死的，也就是我们说的可以被回收的垃圾。

##### 3.2 性能优化手段

Android性能优化一般从以下几个方面

* APP启动速度优化
* UI流畅度优化
* 内存优化
* APK瘦身
* 电量优化

还有一些网络的优化、特定APP的优化(如播放器的优化)

##### 3.3 App启动速度优化

借鉴文档：<https://www.jianshu.com/p/f5514b1a826c>

**3.3.1 启动方式**

* 冷启动：当启动应用时，后台没有该应用的进程，这时系统会重新创建一个新的进程分配给该应用，这个启动方式就是冷启动，启动最慢，挑战最大。
* 热启动：当启动应用时，后台已有该应用的进程（例：按back键、home键，应用虽然会退出，但是该应用的进程是依然会保留在后台，可进入任务列表查看），所以在已有进程的情况下，这种启动会从已有的进程中来启动应用，这个方式叫热启动。
* 温启动

**3.3.2 启动特点**

* 冷启动：冷启动因为系统会重新创建一个新的进程分配给它，所以会先创建和初始化Application类，再创建和初始化MainActivity类（包括一系列的测量、布局、绘制），最后显示在界面上。
* 热启动：在热启动里，系统所做的就是把你的应用带到前台。如果你的应用的所有Activity在内存里都是存活的，应用可以避免重复的对象初始化，布局加载和填充。但是，如果一些内存由于像 [onTrimMemory()](https://developer.android.com/reference/android/content/ComponentCallbacks2.html" \l "onTrimMemory(int)" \t "http://blog.csdn.net/y505772146/article/details/_blank)这样的内存回收事件而被回收，这些对象将由于热启动事件而被重建。热启动展示了与冷启动相同的屏幕显示：系统进程展示了一个空白的屏幕，直到应用已经结束渲染当前Activity。
* 温启动：微温启动包含了一些发生在冷启动的操作；同时，它比热启动的开销要小。有许多可能的状态可以被称为温启动状态

例如：

用户退出你的应用，但是随后重新启动它。进程有可能会继续运行，但是应用将会通过onCreate方法重头重新创建Activity。

系统把你的应用从内存中清除，然后用户重新启动它。进程和Activity将会被重新启动，但是启动速度将会从传递进onCreate方法的已保存实例的状态集中得到加快。

**3.3.3 启动流程**

在冷启动之前要执行三个任务

* 加载启动App；
* App启动之后立即展示出一个空白的Window；
* 创建App的进程；

这三个任务执行完成后会执行下面过程

* 创建App对象；
* 启动Main Thread；
* 创建启动的Activity对象；
* 加载View；
* 布置屏幕；
* 进行第一次绘制；

冷启动：Application的构造器方法——>attachBaseContext()——>onCreate()——>Activity的构造方法——>onCreate()——>配置主题中背景等属性——>onStart()——>onResume()——>测量布局绘制显示在界面上

**3.3.4 优化方向**

作为普通应用，App进程的创建等环节我们是无法主动控制的，可以优化的也就是Application、Activity创建以及回调等过程。

同样，Google也给出了启动加速的方向：

* 利用提前展示出来的Window，快速展示出来一个界面，给用户快速反馈的体验；
* 避免在启动时做密集沉重的初始化（Heavy app initialization）；
* 定位问题：避免I/O操作、反序列化、网络操作、布局嵌套等。

上面的方法一是治标不治本，只是表面上快，方法二和三才是真实的可以加速启动过程

**3.3.5 启动加速主题优化**

按照官方文档的说明：使用Activity的windowBackground主题属性来为启动的Activity提供一个简单的drawable。

主题步骤：

1. 创建一个drawable作为启动界面

<?xml version="1.0" encoding="utf-8"?>  
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">  
  
 <item android:drawable="@color/color\_bg\_white" />  
 <item>  
 <bitmap  
 android:gravity="center"  
 android:src="@drawable/img\_prelauncher" />  
 </item>  
</layer-list>

2，把这个drawable设置到theme主题里

<!--设置冷启动预界面-->  
<style name="AppTheme.Launcher" parent="Theme.AppCompat.Light.NoActionBar">  
 <item name="android:windowBackground">@drawable/drawable\_prelaunch</item>  
 <item name="android:windowFullscreen">true</item>  
</style>

3，把这个主题设置到启动activity里

<activity android:name="com.example.pver.newcatser.SplashActivity"

android:theme="@style/AppTheme.Launcher">  
 <intent-filter>  
 <action android:name="android.intent.action.MAIN" />  
  
 <category android:name="android.intent.category.LAUNCHER" />  
 </intent-filter>  
</activity>

4，在启动的过程中会先展示这个界面，等activity加载完毕后，再去加载activity里的界面，而在activity里，我们将主题重新加载正常的主题

@Override  
public void setRootView() {  
 setTheme(R.style.WelcomeTheme);  
 setContentView(R.layout.activity\_splash);

}

**3.3.6 启动加速之avoid heavy app initialization**

在application和主屏activity里经常做一些初始化操作或者一些第三方的初始化操作，我们可以把一些耗时放在异步线程，一些操作可以延时几秒再初始化

注意：闪屏页的2秒停留可以利用，把耗时操作延迟到这个时间间隔里。

应用启动优化核心思想是在启动过程中少做事情，具体实践无非是下面几种

* 异步加载
* 延时加载
* 懒加载

对于延时加载，推荐方法为：

getWindow().getDecorView().post(new Runnable() {

@Override

public void run() {

myHandler.post(mLoadingRunnable);

}

});

这种方法保证了App第一帧图片在屏幕显示，立马执行延时的run方法，即起到了延时效果，也能正确的显示信息

通过上面我们知道一种懒加载机制，所以我们可以将Application中和首页的onCreate中的有些耗时任务，放到首页渲染完毕后加载

**3.3.7 启动加速之Diagnosing the problem**

本节我们实际定位耗时的操作，在开发阶段我们一般使用BlockCanary或者ANRWatchDog找耗时操作，简单明了，但是无法得到每一个方法的执行时间以及更详细的对比信息。我们可以通过Method Tracing或者DDMS来获得更全面详细的信息。  
启动应用，点击 Start Method Tracing，应用启动后再次点击，会自动打开刚才操作所记录下的.trace文件，建议使用DDMS来查看，功能更加方便全面。

分析到部分耗时操作发生在主线程，那我们把耗时操作都改到子线程是不是就万事大吉了？非也！！

* 卡顿不能都靠异步来解决，错误的使用工程线程不仅不能改善卡顿，反而可能加剧卡顿。是否需要开启工作线程需要根据具体的性能瓶颈根源具体分析，对症下药，不可一概而论；
* 而如何开启线程同样也有学问：Thread、ThreadPoolExecutor、AsyncTask、HandlerThread、IntentService等都各有利弊；例如通常情况下ThreadPoolExecutor比Thread更加高效、优势明显，但是特定场景下单个时间点的表现Thread会比ThreadPoolExecutor好：同样的创建对象，ThreadPoolExecutor的开销明显比Thread大；
* 正确的开启线程也不能包治百病，例如执行网络请求会创建线程池，而在Application中正确的创建线程池势必也会降低启动速度；因此延迟操作也必不可少。

通过对traceview的详细跟踪以及代码的详细比对，我发现卡顿发生在：

* 部分数据库及IO的操作发生在首屏Activity主线程；
* Application中创建了线程池；
* 首屏Activity网络请求密集；
* 工作线程使用未设置优先级；
* 信息未缓存，重复获取同样信息；
* 流程问题：例如闪屏图每次下载，当次使用；

以及其它细节问题：

* 执行无用老代码；
* 执行开发阶段使用的代码；
* 执行重复逻辑；
* 调用三方SDK里或者Demo里的多余代码；

优化思路总结  
1、UI渲染优化，去除重复绘制，减少UI重复绘制时间，打开设置中的GPU过度绘制开关，各界面过度绘制不应超过2.5x；也就是打开此调试开关后，界面整体呈现浅色，特别复杂的界面，红色区域也不应该超过全屏幕的四分之一；  
2、根据优先级的划分，KoMobileApplication的一些初始化工作能否将任务优先级划分成3,在首页渲染完成后进行加载，比如：PaySDKManager。  
3、主线程中的所有SharedPreference能否在非UI线程中进行，SharedPreferences的apply函数需要注意，因为Commit函数会阻塞IO，这个函数虽然执行很快，但是系统会有另外一个线程来负责写操作，当apply频率高的时候，该线程就会比较占用CPU资源。类似的还有统计埋点等，在主线程埋点但异步线程提交，频率高的情况也会出现这样的问题。  
4、检查BaseActivity,不恰当的操作会影响所有子Activity的启动。  
5、对于首次启动的黑屏问题，对于“黑屏”是否可以设计一个.9图片替换掉，间接减少用户等待时间。  
6、对于网络错误界面，友好提示界面，使用ViewStub的方式，减少UI一次性绘制的压力。  
7、任务优先级为2，3的，通过下面这种方式进行懒加载的方式

8、Multidex的使用，也是拖慢启动速度的元凶，必须要做优化。后面有空专门写一篇Multidex。

项目修改：

1. 数据库及IO操作都移到工作线程，并且设置线程优先级为THREAD\_PRIORITY\_BACKGROUND，这样工作线程最多能获取到10%的时间片，优先保证主线程执行。

2. 流程梳理，延后执行；  
实际上，这一步对项目启动加速最有效果。通过流程梳理发现部分流程调用时机偏早、失误等，例如：

更新等操作无需在首屏尚未展示就调用，造成资源竞争；

调用了IOS为了规避审核而做的开关，造成网络请求密集；

自有统计在Application的调用里创建数量固定为5的线程池，造成资源竞争，在上图traceview功能说明图中最后一行可以看到编号12执行5次，耗时排名前列；此处线程池的创建是必要但可以延后的。

修改广告闪屏逻辑为下次生效。

1. 其它优化；

去掉无用但被执行的老代码；

去掉开发阶段使用但线上被执行的代码；

去掉重复逻辑执行代码；

去掉调用三方SDK里或者Demo里的多余代码；

信息缓存，常用信息只在第一次获取，之后从缓存中取；

项目是多进程架构，只在主进程执行Application的onCreate()；

##### 通过ADB命令统计应用的启动时间：adb shell am start -W 首屏Activity。

##### TransView的重要性

执行成功后将返回三个测量到的时间：

* ThisTime：调用过程中最后一个 Activity 启动时间到这个 Activity 的 startActivityAndWait 调用结束
* TotleTime：调用过程中第一个 Activity 的启动时间到最后一个 Activity 的 startActivityAndWait 结束。如果过程中只有一个 Activity ，则 TotalTime 等于 ThisTime
* WaitTime：startActivityAndWait 这个方法的调用耗时

总结：如果只关心某个应用自身启动耗时，参考TotalTime；如果关心系统启动应用耗时，参考WaitTime；如果关心应用有界面Activity启动耗时，参考ThisTime。

**3.3.8 TraceView的使用**

借鉴文章：<http://www.ituring.com.cn/article/273967>

https://www.cnblogs.com/sunzn/p/3192231.html

TraceView是安卓平台特有的数据采集和分析工具，它主要用于分析安卓中应用程序的hotspot，TraceView 本身只是一个数据分析工具，而数据的采集则需要使用 Android SDK 中的 Debug 类或者利用 DDMS 工具

使用TraceView主要有两种方式：

最简单的方式就是直接打开DDMS，选择一个进程，然后按上面的“Start Method Profiling”按钮，等红色小点变成黑色以后就表示TraceView已经开始工作了。然后我就可以滑动一下列表（现在手机上的操作肯定会很卡，因为Android系统在检测Dalvik虚拟机中每个Java方法的调用，这是我猜测的）。操作最好不要超过5s，因为最好是进行小范围的性能测试。然后再按一下刚才按的按钮，等一会就会出现上面这幅图，然后就可以开始分析了。

第2种方式就是使用android.os.Debug.startMethodTracing();和android.os.Debug.stopMethodTracing();方法，当运行了这段代码的时候，就会有一个trace文件在/sdcard目录中生成，也可以调用startMethodTracing(String traceName) 设置trace文件的文件名，最后你可以使用adb pull /sdcard/test.trace /tmp 命令将trace文件复制到你的电脑中，然后用DDMS工具打开就会出现第一幅图了

第一种方式相对来说是一种简单，但是测试的范围很宽泛，第二中方式相对来说精确一点，不过我个人喜欢使用第一种，因为简单，而且它是检测你的某一个操作。因为第二中更适合检测某一个方法的性能，其实也没有那种好，看使用的场景和喜好了

几个比较重要的时间

* Cpu Time/Call ：该方法平均占用 CPU 的时间
* Real Time/Call ：平均执行时间，包括切换、阻塞的时间，>= Cpu Time
* Calls + Recur Calls/Total ：调用、递归次数

定位问题时 TraceView 的使用方式：

* 从上半部分查看哪些线程执行时间长？什么时候开始执行？与主线程交错时间？
* 哪些方法的执行需要花费很长时间

点击 TraceView 中的 Cpu Time/Call，按照占用 CPU 时间从高到低排序

* 哪些方法调用次数非常频繁

点击 TraceView 中的 Calls + Recur Calls/Total ，按照调用次数从高到底排序

##### 3.4 布局优化

借鉴文章：<https://www.jianshu.com/p/4f44a178c547>

根据Google官方出品的[Android性能优化典范](https://link.jianshu.com/?t=https://www.youtube.com/watch?v=CaMTIgxCSqU&index=62&list=PLWz5rJ2EKKc9CBxr3BVjPTPoDPLdPIFCE" \t "https://www.jianshu.com/p/_blank)，60帧每秒是目前最合适的图像显示速度，事实上绝大多数的Android设备也是按照每秒60帧来刷新的。为了让屏幕的刷新帧率达到60fps，我们需要确保在时间16ms（1000/60Hz）内完成单次刷新的操作（包括measure、layout以及draw），这也是Android系统每隔16ms就会发出一次VSYNC信号触发对UI进行渲染的原因。

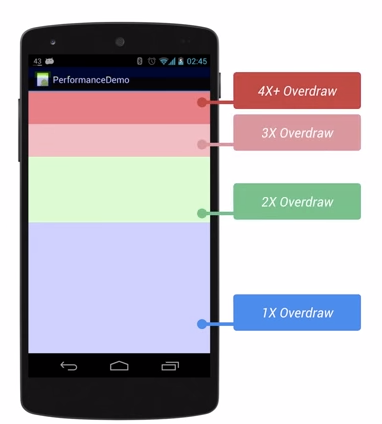
如果整个过程在16ms内顺利完成则可以展示出流畅的画面；然而由于任何原因导致接收到VSYNC信号的时候无法完成本次刷新操作，就会产生掉帧的现象，刷新帧率自然也就跟着下降(假定刷新帧率由正常的60fps降到30fps，用户就会明显感知到卡顿)。

作为开发人员，我们的目标只有一个：保证稳定的帧率来避免卡顿。

**3.4.1 Avoid Overdraw**

理论上一个像素每次只绘制一次是最优的，但是由于重叠的布局导致一些像素会被多次绘制，Overdraw由此产生。

我们可以通过调试工具来检测Overdraw：设置——开发者选项——调试GPU过度绘制——显示过度绘制区域。



原色 – 没有过度绘制 – 这部分的像素点只在屏幕上绘制了一次。  
蓝色 – 1次过度绘制– 这部分的像素点只在屏幕上绘制了两次。  
绿色 – 2次过度绘制 – 这部分的像素点只在屏幕上绘制了三次。  
粉色 – 3次过度绘制 – 这部分的像素点只在屏幕上绘制了四次。  
红色 – 4次过度绘制 – 这部分的像素点只在屏幕上绘制了五次。

在实际项目中，一般认为蓝色即是可以接受的颜色。

* 分析布局可知：多层布局重复设置了背景色导致Overdraw。

那么我们结合产品的需求(任何不结合具体场景优化都是耍流氓):

* 去掉每行RelativeLayout的背景色；
* 去掉每行TextView的背景色；

备注：一个容易忽略的点是我们的Activity使用的[Theme](https://link.jianshu.com/?t=https://android.googlesource.com/platform/frameworks/base/+/android-5.0.0_r2/core/res/res/values/themes.xml" \t "https://www.jianshu.com/p/_blank)可能会默认的加上背景色，不需要的情况下可以去掉。

有些过度绘制都是不可避免的，需要结合具体的布局场景具体分析。

**3.4 .2 减少嵌套层次及控件个数**

Android的布局文件的加载是LayoutInflater利用pull解析方式来解析，然后根据节点名通过反射的方式创建出View对象实例；

同时嵌套子View的位置受父View的影响，类如RelativeLayout、LinearLayout等经常需要measure两次才能完成，而嵌套、相互嵌套、深层嵌套等的发生会使measure次数呈指数级增长，所费时间呈线性增长；

由此得到结论：那么随着控件数量越多、布局嵌套层次越深，展开布局花费的时间几乎是线性增长，性能也就越差。

幸运的是，我们有Hierarchy Viewer这个方便可视化的工具，可以得到：树形结构总览、布局view、每一个View（包含子View）绘制所花费的时间及View总个数。

备注： Hierarchy Viewer不能连接真机的问题可以通过ViewServer这个库解决；

使用Hierarchy Viewer来看查看一下设置界面，可以从下图中得到设置界面的一些数据及存在的问题：

* 嵌套共计7层（仅setContentView设置的布局），布局嵌套过深；
* measure时间1.569ms，layout时间0.120ms，draw时间16.128ms，合计共计耗时17.871ms；
* 共绘制85个View，5个多余定位，以及若干个无用布局。

优化方案：

将之前使用RelativeLayout来做的可以替换的行换为TextView；

去掉之前多余的无用布局；

**3.4.3 Profiling GPU Rendering**

根据[Android性能优化典范](https://link.jianshu.com/?t=https://www.youtube.com/watch?v=we6poP0kw6E&index=64&list=PLWz5rJ2EKKc9CBxr3BVjPTPoDPLdPIFCE" \t "https://www.jianshu.com/p/_blank)，打开设备的GPU配置渲染工具——》在屏幕上显示为条形图，可以协助我们定位UI渲染问题。

根据[Android性能优化典范](https://link.jianshu.com/?t=https://www.youtube.com/watch?v=we6poP0kw6E&index=64&list=PLWz5rJ2EKKc9CBxr3BVjPTPoDPLdPIFCE" \t "https://www.jianshu.com/p/_blank)，打开设备的GPU配置渲染工具——》在屏幕上显示为条形图，可以协助我们定位UI渲染问题。

根据[Android性能优化典范](https://link.jianshu.com/?t=https://www.youtube.com/watch?v=we6poP0kw6E&index=64&list=PLWz5rJ2EKKc9CBxr3BVjPTPoDPLdPIFCE" \t "https://www.jianshu.com/p/_blank)，打开设备的GPU配置渲染工具——》在屏幕上显示为条形图，可以协助我们定位UI渲染问题。

### IMG_256

从Android M版本开始，GPU Profiling工具把渲染操作拆解成如下8个详细的步骤进行显示。

### IMG_256

1. Swap Buffers：表示处理任务的时间，也可以说是CPU等待GPU完成任务的时间，线条越高，表示GPU做的事情越多；
2. Command Issue：表示执行任务的时间，这部分主要是Android进行2D渲染显示列表的时间，为了将内容绘制到屏幕上，Android需要使用Open GL ES的API接口来绘制显示列表，红色线条越高表示需要绘制的视图更多；
3. Sync & Upload：表示的是准备当前界面上有待绘制的图片所耗费的时间，为了减少该段区域的执行时间，我们可以减少屏幕上的图片数量或者是缩小图片的大小；
4. Draw：表示测量和绘制视图列表所需要的时间，蓝色线条越高表示每一帧需要更新很多视图，或者View的onDraw方法中做了耗时操作；
5. Measure/Layout：表示布局的onMeasure与onLayout所花费的时间，一旦时间过长，就需要仔细检查自己的布局是不是存在严重的性能问题；
6. Animation：表示计算执行动画所需要花费的时间，包含的动画有ObjectAnimator，ViewPropertyAnimator，Transition等等。一旦这里的执行时间过长，就需要检查是不是使用了非官方的动画工具或者是检查动画执行的过程中是不是触发了读写操作等等；
7. Input Handling：表示系统处理输入事件所耗费的时间，粗略等于对事件处理方法所执行的时间。一旦执行时间过长，意味着在处理用户的输入事件的地方执行了复杂的操作；
8. Misc Time/Vsync Delay：表示在主线程执行了太多的任务，导致UI渲染跟不上vSync的信号而出现掉帧的情况；出现该线条的时候，可以在Log中看到这样的日志：

**备注：GPU配置渲染工具虽然可以定位出问题发生在某个步骤，但是并不能定位到具体的某一行；当我们定位到某个步骤之后可以使用工具TraceView进行更加详细的定位。**

**3.4.4 Use Tags**

#### merge标签

merge可以用来合并布局，减少布局的层级。merge多用于替换顶层FrameLayout或者include布局时,用于消除因为引用布局导致的多余嵌套。  
例如：需要显示一个Button，布局如下；

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout

xmlns:android="http://schemas.android.com/apk/res/android

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical">

<Button

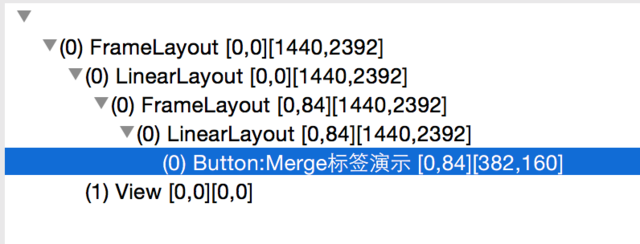
android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Merge标签演示" />

</LinearLayout>

我们通过UiAutoMatorViewer（无需root，相比Hierarchy Viewer只能查看布局层次，不能得到绘制时间）看一下布局的层次



我们使用Merge标签对代码进行修改；

<?xml version="1.0" encoding="utf-8"?>

<merge

xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<Button

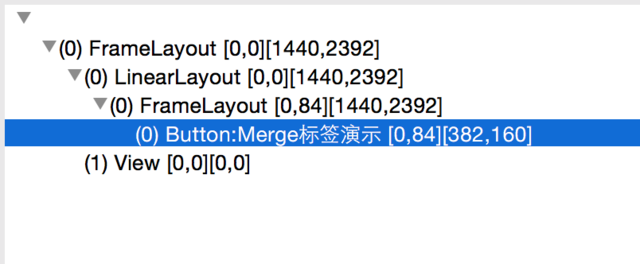
android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Merge标签演示" />

</merge>

再看下布局的层次：



可以看到使用Merge标签进行优化之后布局嵌套就少了一层，Button作为父视图第三层FrameLayout的直接子视图。

**注意：merge标签常用于减少布局嵌套层次，但是只能用于根布局。**

#### ViewStub标签

推迟创建对象、延迟初始化，不仅可以提高性能，也可以节省内存（初始化对象不被创建）。Android定义了ViewStub类，ViewStub是轻量级且不可见的视图，它没有大小，没有绘制功能，也不参与measure和layout，资源消耗非常低。

<ViewStub

android:id="@+id/mask"

android:layout="@layout/b\_me\_mask"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent" />

使用：

ViewStub viewStub = (ViewStub)view.findViewById(R.id.mask);

viewStub.inflate();

App里常见的视图如蒙层、小红点，以及网络错误、没有数据等公共视图，使用频率并不高，如果每一次都参与绘制其实是浪费资源的，都可以借助ViewStub标签进行延迟初始化，仅当使用时才去初始化。

### 七、其它

1. 自定义控件时，注意在onDraw不能进行复杂运算；以及对待三方UI库选择高性能；
2. 内存对布局的影响：如同Misc Time/Vsync Delay步骤产生的影响，在之后内存优化的篇章详细讲。

### 八、总结

#### 布局优化的通用套路

1. 调试GPU过度绘制，将Overdraw降低到合理范围内；
2. 减少嵌套层次及控件个数，保持view的树形结构尽量扁平（使用Hierarchy Viewer可以方便的查看），同时移除所有不需要渲染的view；
3. 使用GPU配置渲染工具，定位出问题发生在具体哪个步骤，使用TraceView精准定位代码；
4. 使用标签，Merge减少嵌套层次、ViewStub延迟初始化。

经过这几步的优化之后，一般就不会再有布局的性能问题，同时还是要强调：优化是一个长期的工作，同时也必须结合具体场景：有取有舍！

当然了对于UI卡顿，不可避免的要引入检测的方案：

* 一般有监听Looper的日志
* 利用Choreographer

##### 3.3 内存优化

##### 3.4 APK瘦身

* 关于Apk瘦身，主要由以下几个方式：
* 利用ProGuard压缩代码去除无用资源
* andresguard进一步压缩与混淆资源
* 第三方开源库的瘦身，仅保留自己需要的部分
* 极致的图片压缩与webp的使用
* 合理配置去除不必要的配置，仅保留中文配置等...
* so的优化与配置，只保留一类so
* 动态下发一些资源:字库、so、换肤包等；

以上仅有7比较麻烦，需要服务端的配合，此外对于动态下发So，可以参考tinker对So热修复部分代码。

##### 3.5 电量优化

### 4 . Android优化

性能优化

1).节制的使用Service 如果应用程序需要使用Service来执行后台任务的话，只有当任务正在执行的时候才应该让Service运行起来。当启动一个Service时，系统会倾向于将这个Service所依赖的进程进行保留，系统可以在LRUcache当中缓存的进程数量也会减少，导致切换程序的时候耗费更多性能。我们可以使用IntentService，当后台任务执行结束后会自动停止，避免了Service的内存泄漏。

2).当界面不可见时释放内存 当用户打开了另外一个程序，我们的程序界面已经不可见的时候，我们应当将所有和界面相关的资源进行释放。重写Activity的onTrimMemory()方法，然后在这个方法中监听TRIM\_MEMORY\_UI\_HIDDEN这个级别，一旦触发说明用户离开了程序，此时就可以进行资源释放操作了。

3).当内存紧张时释放内存 onTrimMemory()方法还有很多种其他类型的回调，可以在手机内存降低的时候及时通知我们，我们应该根据回调中传入的级别来去决定如何释放应用程序的资源。

4).避免在Bitmap上浪费内存 读取一个Bitmap图片的时候，千万不要去加载不需要的分辨率。可以压缩图片等操作。

5).使用优化过的数据集合 Android提供了一系列优化过后的数据集合工具类，如SparseArray、SparseBooleanArray、LongSparseArray，使用这些API可以让我们的程序更加高效。HashMap工具类会相对比较低效，因为它需要为每一个键值对都提供一个对象入口，而SparseArray就避免掉了基本数据类型转换成对象数据类型的时间

布局优化

1).重用布局文件

标签可以允许在一个布局当中引入另一个布局，那么比如说我们程序的所有界面都有一个公共的部分，这个时候最好的做法就是将这个公共的部分提取到一个独立的布局中，然后每个界面的布局文件当中来引用这个公共的布局。

Tips:如果我们要在标签中覆写layout属性，必须要将layout\_width和layout\_height这两个属性也进行覆写，否则覆写效果将不会生效。

标签是作为标签的一种辅助扩展来使用的，它的主要作用是为了防止在引用布局文件时引用文件时产生多余的布局嵌套。布局嵌套越多，解析起来就越耗时，性能就越差。因此编写布局文件时应该让嵌套的层数越少越好。

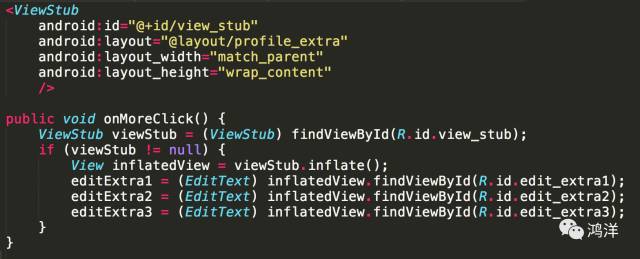
举例：比如在LinearLayout里边使用一个布局。里边又有一个LinearLayout，那么其实就存在了多余的布局嵌套，使用merge可以解决这个问题。

2).仅在需要时才加载布局

某个布局当中的元素不是一起显示出来的，普通情况下只显示部分常用的元素，而那些不常用的元素只有在用户进行特定操作时才会显示出来。

举例：填信息时不是需要全部填的，有一个添加更多字段的选项，当用户需要添加其他信息的时候，才将另外的元素显示到界面上。用VISIBLE性能表现一般，可以用ViewStub。

ViewStub也是View的一种，但是没有大小，没有绘制功能，也不参与布局，资源消耗非常低，可以认为完全不影响性能。



tips：ViewStub所加载的布局是不可以使用标签的，因此这有可能导致加载出来出来的布局存在着多余的嵌套结构。

高性能编码优化

都是一些微优化，在性能方面看不出有什么显著的提升的。使用合适的算法和数据结构是优化程序性能的最主要手段。

1).避免创建不必要的对象 不必要的对象我们应该避免创建：

如果有需要拼接的字符串，那么可以优先考虑使用StringBuffer或者StringBuilder来进行拼接，而不是加号连接符，因为使用加号连接符会创建多余的对象，拼接的字符串越长，加号连接符的性能越低。

当一个方法的返回值是String的时候，通常需要去判断一下这个String的作用是什么，如果明确知道调用方会将返回的String再进行拼接操作的话，可以考虑返回一个StringBuffer对象来代替，因为这样可以将一个对象的引用进行返回，而返回String的话就是创建了一个短生命周期的临时对象。

尽可能地少创建临时对象，越少的对象意味着越少的GC操作。

2).在没有特殊原因的情况下，尽量使用基本数据类型来代替封装数据类型，int比Integer要更加有效，其它数据类型也是一样。

基本数据类型的数组也要优于对象数据类型的数组。另外两个平行的数组要比一个封装好的对象数组更加高效，举个例子，Foo[]和Bar[]这样的数组，使用起来要比Custom(Foo,Bar)[]这样的一个数组高效的多。

3).静态优于抽象

如果你并不需要访问一个对系那个中的某些字段，只是想调用它的某些方法来去完成一项通用的功能，那么可以将这个方法设置成静态方法，调用速度提升15%-20%，同时也不用为了调用这个方法去专门创建对象了，也不用担心调用这个方法后是否会改变对象的状态(静态方法无法访问非静态字段)。

4).对常量使用static final修饰符

Static int intval = 42

编译器会为上面的代码生成一个初始方法，称为方法，该方法会在定义类第一次被使用的时候调用。这个方法会将42的值赋值到intVal当中，从字符串常量表中提取一个引用赋值到strVal上。当赋值完成后，我们就可以通过字段搜寻的方式去访问具体的值了。

Static final int intval = 42

这样，定义类就不需要方法了，因为所有的常量都会在dex文件的初始化器当中进行初始化。当我们调用intVal时可以直接指向42的值，而调用strVal会用一种相对轻量级的字符串常量方式，而不是字段搜寻的方式。

这种优化方式只对基本数据类型以及String类型的常量有效，对于其他数据类型的常量是无效的。

5).使用增强型for循环语法

6).多使用系统封装好的API

7).避免在内部调用Getters/Setters方法

面向对象中封装的思想是不要把类内部的字段暴露给外部，而是提供特定的方法来允许外部操作相应类的内部字段。但在Android中，字段搜寻比方法调用效率高得多，我们直接访问某个字段可能要比通过getters方法来去访问这个字段快3到7倍。

但是编写代码还是要按照面向对象思维的，我们应该在能优化的地方进行优化，比如避免在内部调用getters/setters方法。