链滴

Android APP 性能优化的四个方面的总结

作者：lzlyy
原文链接：https://ld246.com/article/1587695728023
来源网站：链滴

说到Android系统手机，大部分人的印象是用了一段时间就变得有点卡顿，有些程序在运行期间莫名妙的出现崩溃，打开系统文件夹一看，发现多了很多文件，然后用手机管家 APP 不断地进行清理优化，才感觉运行速度稍微提高了点，就算手机在各种性能跑分软件面前分数遥遥领先，还是感觉无论有大的内存空间都远远不够用。相信每个使用Android系统的用户都有过以上类似经历，确实，Androi系统在流畅性方面不如iOS系统，为何呢，明明在看手机硬件配置上时，Android设备都不会输于iOS备，甚至都强于它，关键是在于软件上。造成这种现象的原因是多方面的，简单罗列几点如下：

● 其实近年来，随着Android版本不断迭代，Google提供的Android系统已经越来越流畅，目前最新布的版本是Android Oreo（8.0）。但是在国内大部分用户用的Android手机系是各大厂商定制过的本，往往不是最新的原生系统内核，可能绝大多数还停留在Android 5.0系统上，甚至 Android 6.0上所占比例还偏小，更新存在延迟性。

● 由于Android系统源码是开放的，每个人只要遵从相应的协议，就可以对源码进行修改，那么国内个厂商就把基于Android源码改造成自己对外发布的系统，比如我们熟悉的小米手机MIUI系统、华为机EMUI系统、Oppo手机ColorOS系统等。由于每个厂商都修改过Android原生系统源码，这里面就引发一个问题，那就是著名的Android碎片化问题，本质就是不同Android系统的应用兼容性不同，不到一致性。

● 由于存在着各种Android碎片化和兼容性问题，导致Android开发者在开发应用时需要对不同系统行适配，同时每个Android开发者的开发水平参差不齐，写出来的应用性能也都存在不同类型的问题

原文链接：Android APP 性能优化的四个方面的总结

导致用户在使用过程中用户体验感受不同，那么有些问题用户就会转化为Android系统问题，进而影对Android手机的评价。

性能优化

今天想说的重点是Android APP性能优化，也就是在开发应用程序时应该注意的点有哪些，如何更好提高用户体验。一个好的应用，除了要有吸引人的功能和交互之外，在性能上也应该有高的要求，即应用非常具有特色，在产品前期可能吸引了部分用户，但是用户体验不好的话，也会给产品带来不好口碑。那么一个好的应用应该如何定义呢？主要有以下三方面：

- 业务/功能
- 符合逻辑的交互
- 优秀的性能

众所周知，Android系统作为以移动设备为主的操作系统，硬件配置是有一定的限制的，虽然配置现越来越高级，但仍然无法与PC相比，在CPU和内存上使用不合理或者耗费资源多时，就会碰到内存不导致的稳定性问题、CPU 消耗太多导致的卡顿问题等。

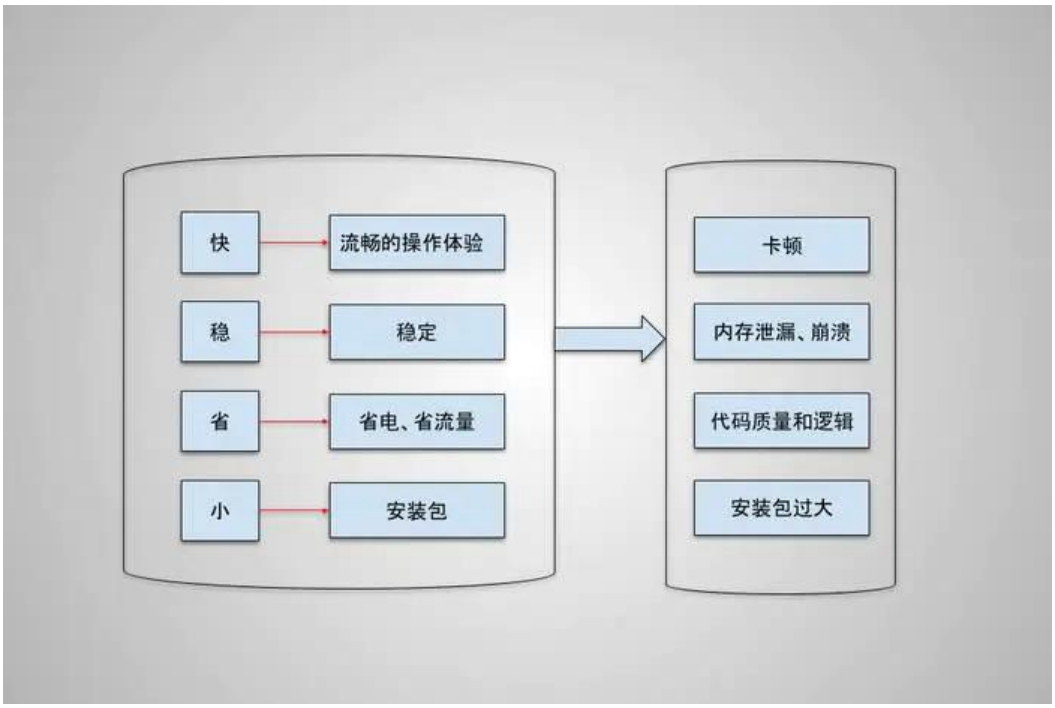
面对问题时，大家想到的都是联系用户，然后查看日志，但殊不知有关性能类问题的反馈，原因也非难找，日志大多用处不大，为何呢？因为性能问题大部分是非必现的问题，问题定位很难复现，而又有关键的日志，当然就无法找到原因了。这些问题非常影响用户体验和功能使用，所以了解一些性能化的一些解决方案就显得很重要了，并在实际的项目中优化我们的应用，进而提高用户体验。

四个方面

可以把用户体验的性能问题主要总结为4个类别：

- 流畅
- 稳定
- 省电、省流量
- 安装包小

性能问题的主要原因是什么，原因有相同的，也有不同的，但归根到底，不外乎内存使用、代码效率合适的策略逻辑、代码质量、安装包体积这一类问题，整理归类如下：



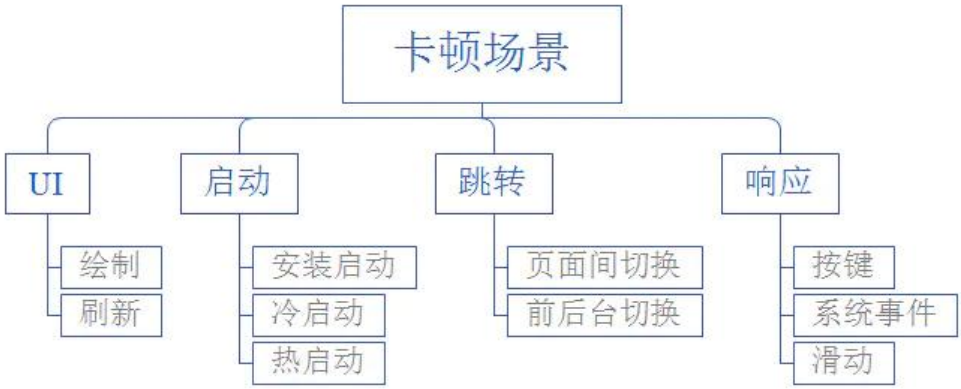
从图中可以看到，打造一个高质量的应用应该以4个方向为目标：快、稳、省、小。

- 快：使用时避免出现卡顿，响应速度快，减少用户等待的时间，满足用户期望。
- 稳：减低 crash 率和 ANR 率，不要在用户使用过程中崩溃和无响应。
- 省：节省流量和耗电，减少用户使用成本，避免使用时导致手机发烫。
- 小：安装包小可以降低用户的安装成本。

要想达到这4个目标，具体实现是在右边框里的问题：卡顿、内存使用不合理、代码质量差、代码逻乱、安装包过大，这些问题也是在开发过程中碰到最多的问题，在实现业务需求同时，也需要考虑到点，多花时间去思考，如何避免功能完成后再来做优化，不然的话等功能实现后带来的维护成本会增。

卡顿优化

Android 应用启动慢，使用时经常卡顿，是非常影响用户体验的，应该尽量避免出现。卡顿的场景有多，按场景可以分为4类：UI 绘制、应用启动、页面跳转、事件响应，如图：



这4种卡顿场景的根本原因可以分为两大类：

- 界面绘制。主要原因是绘制的层级深、页面复杂、刷新不合理，由于这些原因导致卡顿的场景更多现在UI和启动后的初始界面以及跳转到页面的绘制上。
- 数据处理。导致这种卡顿场景的原因是数据处理量太大，一般分为三种情况，一是数据在处理UI线，二是数据处理占用CPU高，导致主线程拿不到时间片，三是内存增加导致GC频繁，从而引起卡顿。

通常引起卡顿的原因很多，但不管怎么样的原因和场景，最终都是通过设备屏幕上显示来达到用户，根到底就是显示有问题，所以，要解决卡顿，就要先了解Android系统的显示原理。

Android系统显示原理

Android显示过程可以简单概括为：Android应用程序把经过测量、布局、绘制后的Surface缓存数据通过SurfaceFlinger把数据渲染到显示屏幕上， 通过Android的刷新机制来刷新数据。也就是说应用负责绘制，系统层负责渲染，通过进程间通信把应用层需要绘制的数据传递到系统层服务，系统层服通过刷新机制把数据更新到屏幕上。

我们都知道在Android的每个View绘制中有三个核心步骤：Measure、Layout、Draw。具体实现是从ViewRootImp类的performTraversals() 方法开始执行，Measure和Layout都是通过递归来获取Vie的大小和位置，并且以深度作为优先级，可以看出层级越深、元素越多、耗时也就越长。

真正把需要显示的数据渲染到屏幕上，是通过系统级进程中的SurfaceFlinger服务来实现的，那么这SurfaceFlinger服务主要做了哪些工作呢？如下：

- 响应客户端事件，创建Layer与客户端的Surface建立连接
- 接收客户端数据及属性，修改Layer属性，如尺寸、颜色、透明度等。
- 将创建的Layer内容刷新到屏幕上。
- 维持Layer的序列，并对Layer最终输出做出裁剪计算。

既然是两个不同的进程，那么肯定是需要一个跨进程的通信机制来实现数据传递，在Android显示系中，使用了Android的匿名共享内存：SharedClient，每一个应用和SurfaceFlinger之间都会创建一个haredClient ，然后在每个SharedClient中，最多可以创建31个 SharedBufferStack，每个Surface对应一个SharedBufferStack，也就是一个Window。

一个SharedClient对应一个Android应用程序，而一个Android应用程序可能包含多个窗口，即Surfae。也就是说SharedClient包含的是SharedBufferStack的集合，其中在显示刷新机制中用到了双缓和三重缓冲技术。最后总结起来显示整体流程分为三个模块：应用层绘制到缓存区，SurfaceFlinger缓存区数据渲染到屏幕，由于是不同的进程，所以使用Android的匿名共享内存SharedClient缓存需显示的数据来达到目的。

除此之外，我们还需要一个名词：FPS。FPS表示每秒传递的帧数。在理想情况下，60FPS就感觉不到，这意味着每个绘制时长应该在16ms以内。但是 Android系统很有可能无法及时完成那些复杂的页渲染操作。Android系统每隔16ms发出VSYNC信号，触发对UI进行渲染，如果每次渲染都成功，这就能够达到流畅的画面所需的60FPS。如果某个操作花费的时间是24ms ，系统在得到VSYNC信号时无法正常进行正常渲染，这样就发生了丢帧现象。那么用户在32ms内看到的会是同一帧画面，这种象在执行动画或滑动列表比较常见，还有可能是你的Layout太过复杂，层叠太多的绘制单元，无法在6ms完成渲染，最终引起刷新不及时。

卡顿根本原因

根据Android系统显示原理可以看到，影响绘制的根本原因有以下两个方面：

- 绘制任务太重，绘制一帧内容耗时太长。

- 主线程太忙，根据系统传递过来的VSYNC信号来时还没准备好数据导致丢帧。

绘制耗时太长，有一些工具可以帮助我们定位问题。主线程太忙则需要注意了，主线程关键职责是处用户交互，在屏幕上绘制像素，并进行加载显示相关的数据，所以特别需要避免任何主线程的事情，样应用程序才能保持对用户操作的即时响应。总结起来，主线程主要做以下几个方面工作：

- UI生命周期控制
- 系统事件处理
- 消息处理
- 界面布局
- 界面绘制
- 界面刷新

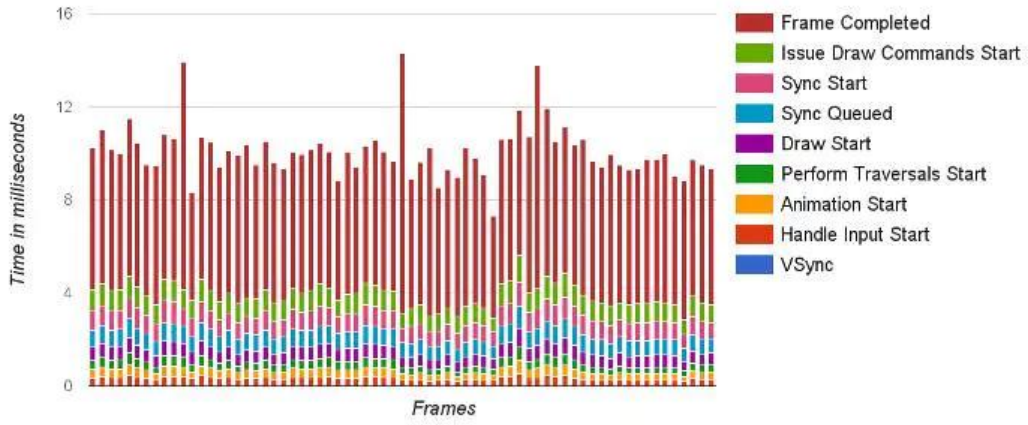
除此之外，应该尽量避免将其他处理放在主线程中，特别复杂的数据计算和网络请求等。

性能分析工具

性能问题并不容易复现，也不好定位，但是真的碰到问题还是需要去解决的，那么分析问题和确认问是否解决，就需要借助相应的的调试工具，比如查看Layout层次的Hierarchy View、Android系统上的GPU Profile工具和静态代码检查工具Lint等，这些工具对性能优化起到非常重要的作用，所以要熟，知道在什么场景用什么工具来分析。

1，Profile GPU Rendering

在手机开发者模式下，有一个卡顿检测工具叫做：Profile GPU Rendering，如图：



它的功能特点如下：

- 一个图形监测工具，能实时反应当前绘制的耗时
- 横轴表示时间，纵轴表示每一帧的耗时
- 随着时间推移，从左到右的刷新呈现
- 提供一个标准的耗时，如果高于标准耗时，就表示当前这一帧丢失

2，TraceView

TraceView是Android SDK自带的工具，用来分析函数调用过程，可以对Android的应用程序以及Fraework层的代码进行性能分析。它是一个图形化的工具，最终会产生一个图表，用于对性能分析进行明，可以分析到每一个方法的执行时间，其中可以统计出该方法调用次数和递归次数，实际时长等参维度，使用非常直观，分析性能非常方便。

3，Systrace UI 性能分析

Systrace是Android 4.1及以上版本提供的性能数据采样和分析工具，它是通过系统的角度来返回一信息。它可以帮助开发者收集Android关键子系统，如Surfaceflinger、WindowManagerService等Famework部分关键模块、服务、View系统等运行信息，从而帮助开发者更直观地分析系统瓶颈，改性能。Systrace的功能包括跟踪系统的I/O操作、内核工作队列、CPU负载等，在UI显示性能分析上供很好的数据，特别是在动画播放不流畅、渲染卡等问题上。

优化建议

1，布局优化

布局是否合理主要影响的是页面测量时间的多少，我们知道一个页面的显示测量和绘制过程都是通过归来完成的，多叉树遍历的时间与树的高度h有关，其时间复杂度O(h)，如果层级太深，每增加一层会增加更多的页面显示时间，所以布局的合理性就显得很重要。

那布局优化有哪些方法呢，主要通过减少层级、减少测量和绘制时间、提高复用性三个方面入手。总如下：

- 减少层级。合理使用RelativeLayout和LinerLayout，合理使用Merge。
- 提高显示速度。使用ViewStub，它是一个看不见的、不占布局位置、占用资源非常小的视图对象。
- 布局复用,可以通过标签来提高复用。
- 尽可能少用wrap_content。wrap_content 会增加布局measure时计算成本，在已知宽高为固定值，不用wrap_content 。
- 删除控件中无用的属性。

2，避免过度绘制

过度绘制是指在屏幕上的某个像素在同一帧的时间内被绘制了多次。在多层次重叠的UI结构中，如果可见的UI也在做绘制的操作，就会导致某些像素区域被绘制了多次，从而浪费了多余的CPU以及GPU。

如何避免过度绘制呢，如下：

布局上的优化。移除XML中非必须的背景，移除Window默认的背景、按需显示占位背景图片

自定义View优化。使用 canvas.clipRect()来帮助系统识别那些可见的区域，只有在这个区域内才会绘制。

3，启动优化

通过对启动速度的监控，发现影响启动速度的问题所在，优化启动逻辑，提高应用的启动速度。启动要完成三件事：UI布局、绘制和数据准备。因此启动速度优化就是需要优化这三个过程：

- UI布局。应用一般都有闪屏页，优化闪屏页的UI布局，可以通过Profile GPU Rendering检测丢帧况。
- 启动加载逻辑优化。可以采用分布加载、异步加载、延期加载策略来提高应用启动速度。

- 数据准备。数据初始化分析，加载数据可以考虑用线程初始化等策略。

4，合理的刷新机制

在应用开发过程中，因为数据的变化，需要刷新页面来展示新的数据，但频繁刷新会增加资源开销，且可能导致卡顿发生，因此，需要一个合理的刷新机制来提高整体的UI流畅度。合理的刷新需要注意下几点：

- 尽量减少刷新次数。
- 尽量避免后台有高的CPU线程运行。
- 缩小刷新区域。

5，其他

在实现动画效果时，需要根据不同场景选择合适的动画框架来实现。有些情况下，可以用硬件加速方来提供流畅度。

内存优化

在Android系统中有个垃圾内存回收机制，在虚拟机层自动分配和释放内存，因此不需要在代码中分和释放某一块内存，从应用层面上不容易出现内存泄漏和内存溢出等问题，但是需要内存管理。Andro d系统在内存管理上有一个Generational Heap Memory模型，内存回收的大部分压力不需要应用层心，Generational Heap Memory有自己一套管理机制，当内存达到一个阈值时，系统会根据不同的则自动释放系统认为可以释放的内存，也正是因为Android程序把内存控制的权力交给了Generational Heap Memory，一旦出现内存泄漏和溢出方面的问题，排查错误将会成为一项异常艰难的工作。除之外，部分Android应用开发人员在开发过程中并没有特别关注内存的合理使用，也没有在内存方面太多的优化，当应用程序同时运行越来越多的任务，加上越来越复杂的业务需求时，完全依赖Androi的内存管理机制就会导致一系列性能问题逐渐呈现，对应用的稳定性和性能带来不可忽视的影响，因，解决内存问题和合理优化内存是非常有必要的。

Android内存管理机制

Android应用都是在 Android的虚拟机上运行，应用 程序的内存分配与垃圾回收都是由虚拟机完成的在Android系统，虚拟机有两种运行模式：Dalvik和ART。

1，Java对象生命周期

一般Java对象在虚拟机上有7个运行阶段：

创建阶段->应用阶段->不可见阶段->不可达阶段->收集阶段->终结阶段->对象空间重新分配阶段

2，内存分配

在Android系统中，内存分配实际上是对堆的分配和释放。当一个Android程序启动，应用进程都是一个叫做Zygote的进程衍生出来，系统启动 Zygote 进程后，为了启动一个新的应用程序进程，系统衍生Zygote进程生成一个新的进程，然后在新的进程中加载并运行应用程序的代码。其中，大多数的AM pages被用来分配给Framework代码，同时促使RAM资源能够在应用所有进程之间共享。

但是为了整个系统的内存控制需要，Android系统会为每一个应用程序都设置一个硬性的Dalvik Heap Size最大限制阈值，整个阈值在不同设备上会因为RAM大小不同而有所差异。如果应用占用内存空间经接近整个阈值时，再尝试分配内存的话，就很容易引起内存溢出的错误。

3，内存回收机制

我们需要知道的是，在Java中内存被分为三个区域：Young Generation(年轻代)、Old Generation(老代)、Permanent Generation(持久代)。最近分配的对象会存放在Young Generation区域。对象某个时机触发GC回收垃圾，而没有回收的就根据不同规则，有可能被移动到Old Generation，最后积一定时间在移动到Permanent Generation 区域。系统会根据内存中不同的内存数据类型分别执行同的GC操作。GC通过确定对象是否被活动对象引用来确定是否收集对象，进而动态回收无任何引用对象占据的内存空间。但需要注意的是频繁的GC会增加应用的卡顿情况，影响应用的流畅性，因此要尽量减少系统GC行为，以便提高应用的流畅度，减小卡顿发生的概率。

内存分析工具

做内存优化前，需要了解当前应用的内存使用现状，通过现状去分析哪些数据类型有问题，各种类型分布情况如何，以及在发现问题后如何发现是哪些具体对象导致的，这就需要相关工具来帮助我们。

1，Memory Monitor

Memory Monitor是一款使用非常简单的图形化工具，可以很好地监控系统或应用的内存使用情况，要有以下功能：

- 显示可用和已用内存，并且以时间为维度实时反应内存分配和回收情况。
- 快速判断应用程序的运行缓慢是否由于过度的内存回收导致。
- 快速判断应用是否由于内存不足导致程序崩溃。

2，Heap Viewer

Heap Viewer的主要功能是查看不同数据类型在内存中的使用情况，可以看到当前进程中的Heap Siz的情况，分别有哪些类型的数据，以及各种类型数据占比情况。通过分析这些数据来找到大的内存对，再进一步分析这些大对象，进而通过优化减少内存开销，也可以通过数据的变化发现内存泄漏。

3，Allocation Tracker

Memory Monitor和Heap Viewer都可以很直观且实时地监控内存使用情况，还能发现内存问题，但现内存问题后不能再进一步找到原因，或者发现一块异常内存，但不能区别是否正常，同时在发现问后，也不能定位到具体的类和方法。这时就需要使用另一个内存分析工具Allocation Tracker，进行详细的分析，Allocation Tracker可以分配跟踪记录应用程序的内存分配，并列出了它们的调用堆栈可以查看所有对象内存分配的周期。

4，Memory Analyzer Tool(MAT)

MAT是一个快速，功能丰富的Java Heap分析工具，通过分析Java进程的内存快照HPROF分析，从多的对象中分析，快速计算出在内存中对象占用的大小，查看哪些对象不能被垃圾收集器回收，并可

通过视图直观地查看可能造成这种结果的对象。

常见内存泄漏场景

如果在内存泄漏发生后再去找原因并修复会增加开发的成本，最好在编写代码时就能够很好地考虑内问题，写出更高质量的代码，这里列出一些常见的内存泄漏场景，在以后的开发过程中需要避免这类题。

- 资源性对象未关闭。比如Cursor、File文件等，往往都用了一些缓冲，在不使用时，应该及时关闭们。
- 注册对象未注销。比如事件注册后未注销，会导致观察者列表中维持着对象的引用
- 类的静态变量持有大数据对象。
- 非静态内部类的静态实例。
- Handler临时性内存泄漏。如果Handler是非静态的，容易导致Activity或Service不会被回收。
- 容器中的对象没清理造成的内存泄漏。
- WebView。WebView存在着内存泄漏的问题，在应用中只要使用一次WebView，内存就不会被放掉。

除此之外，内存泄漏可监控，常见的就是用LeakCanary第三方库，这是一个检测内存泄漏的开源库使用非常简单，可以在发生内存泄漏时告警，并且生成leak tarce分析泄漏位置，同时可以提供Dum文件进行分析。

优化内存空间

没有内存泄漏，并不意味着内存就不需要优化，在移动设备上，由于物理设备的存储空间有限，Andro d 系统对每个应用进程也都分配了有限的堆内存，因此使用最小内存对象或者资源可以减小内存开销同时让GC 能更高效地回收不再需要使用的对象，让应用堆内存保持充足的可用内存，使应用更稳定效地运行。常见做法如下：

- 对象引用。强引用、软引用、弱引用、虚引用四种引用类型，根据业务需求合理使用不同，选择不的引用类型。
- 减少不必要的内存开销。注意自动装箱，增加内存复用，比如有效利用系统自带的资源、视图复用对象池、Bitmap对象的复用。
- 使用最优的数据类型。比如针对数据类容器结构，可以使用ArrayMap数据结构，避免使用枚举类，使用缓存Lrucache等等。
- 图片内存优化。可以设置位图规格，根据采样因子做压缩，用一些图片缓存方式对图片进行管理等。

稳定性优化

Android应用的稳定性定义很宽泛，影响稳定性的原因很多，比如内存使用不合理、代码异常场景考不周全、代码逻辑不合理等，这些都会对应用的稳定性造成影响。其中最常见的两个场景是：Crash ANR，这两个错误将会使得程序无法使用，比较常用的解决方式如下：

- 提高代码质量。比如开发期间的代码审核，看些代码设计逻辑，业务合理性等。
- 代码静态扫描工具。常见工具有Android Lint、Findbugs、Checkstyle、PMD等等。
- Crash监控。把一些崩溃的信息，异常信息及时地记录下来，以便后续分析解决。

● Crash上传机制。在Crash后，尽量先保存日志到本地，然后等下一次网络正常时再上传日志信息。

耗电优化

在移动设备中，电池的重要性不言而喻，没有电什么都干不成。对于操作系统和设备开发商来说，耗优化一致没有停止，去追求更长的待机时间，而对于一款应用来说，并不是可以忽略电量使用问题，别是那些被归为“电池杀手”的应用，最终的结果是被卸载。因此，应用开发者在实现需求的同时，要尽量减少电量的消耗。

在Android5.0以前，在应用中测试电量消耗比较麻烦，也不准确，5.0之后专门引入了一个获取设备电量消耗信息的API:Battery Historian。Battery Historian是一款由Google提供的Android系统电量析工具，和Systrace一样，是一款图形化数据分析工具，直观地展示出手机的电量消耗过程，通过输电量分析文件，显示消耗情况，最后提供一些可供参考电量优化的方法。

除此之外，还有一些常用方案可提供：

● 计算优化，避开浮点运算等。

● 避免WaleLock使用不当。

● 使用Job Scheduler。

安装包大小优化

应用安装包大小对应用使用没有影响，但应用的安装包越大，用户下载的门槛越高，特别是在移动网情况下，用户在下载应用时，对安装包大小的要求更高，因此，减小安装包大小可以让更多用户愿意载和体验产品。

常用应用安装包的构成，如图所示：

名称	大小	压缩后大小	类型	修改时间	CRC32
..			文件夹		
assets			文件夹		
META-INF			文件夹		
res			文件夹		
AndroidManifest.xml	3,500	1,065	XML 文档		FCDB16...
classes.dex	3,266,140	1,107,505	DEX 文件		E748468B
resources.arsc	223,808	223,808	ARSC 文件		ADBFC0...

从图中我们可以看到：

● assets文件夹。存放一些配置文件、资源文件，assets不会自动生成对应的 ID，而是通过AssetManger类的接口获取。

● res。res是resource的缩写，这个目录存放资源文件，会自动生成对应的ID并映射到 .R文件中，访直接使用资源ID。

● META-INF。保存应用的签名信息，签名信息可以验证APK文件的完整性。

● AndroidManifest.xml。这个文件用来描述Android应用的配置信息，一些组件的注册信息、可使权限等。

● classes.dex。Dalvik字节码程序，让Dalvik虚拟机可执行，一般情况下，Android应用在打包时通Android SDK中的dx工具将Java字节码转换为Dalvik字节码。

● resources.arsc。记录着资源文件和资源ID之间的映射关系，用来根据资源ID寻找资源。

减少安装包大小的常用方案:

- 代码混淆。使用ProGuard代码混淆器工具，它包括压缩、优化、混淆等功能。
- 资源优化。比如使用Android Lint删除冗余资源，资源文件最少化等。
- 图片优化。比如利用AAPT工具对PNG格式的图片做压缩处理，降低图片色彩位数等。
- 避免重复功能的库，使用WebP图片格式等。
- 插件化。比如功能模块放在服务器上，按需下载，可以减少安装包大小。

总结一下：

性能优化不是更新一两个版本就可以解决的，是持续性的需求，持续集成迭代反馈。在实际的项目中在项目刚开始的时候，由于人力和项目完成时间限制，性能优化的优先级比较低，等进入项目投入使阶段，就需要把优先级提高，但在项目初期，在设计架构方案时，性能优化的点也需要提早考虑进去这就体现出一个程序员的技术功底了。

什么时候开始有性能优化的需求，往往都是从发现问题开始，然后分析问题原因及背景，进而寻找最解决方案，最终解决问题，这也是日常工作中常会用到的处理方式。