Android 应用性能优化

前言

性能优化在一款产品的迭代过程中非常重要；程序实现了功能、还原产品原型只能保证程序能用，但如果要让用户更愿意使用，产品得好用。试想一下如果你开发的产品启动慢、页面显示需要长时间转圈加载、页面切换卡顿、黑白屏、用一会机器就发烫、耗内存、OOM、程序切换到后台后占用内存无法释放……这些问题就像正在玩游戏时弹出提示框这类糟糕的用户体验一样让用户恼火，如果用户不得不使用你的产品，可能还会一直忍受；但如果有很多同类竞品，糟糕的用户体验会大大影响留存率。有时候产品在市场上的表现差，真不能全怪产品和运营，程序体验问题也是很大一部分原因。

但大部分产品并没有足够重视性能问题，随便打开一个应用，即使是大厂出品，也极大可能存在过渡绘制和内存泄露的问题；也有可能是开发人员意识到了程序存在性能问题，但完成迭代就够忙的了，哪有时间去做这类不能体现绩效的事情。其实在越来越重视体验，同类产品竞争越来越激烈的环境下，对于开发人员来讲，只完成迭代，把功能做完远远不够，最重要的是把产品做好，让更多人愿意使用。重视性能问题，优化产品的体验，比改几个无关痛痒的 bug 会有价值得多。

网上能够找到很多关于性能优化很有价值的参考资料（详见文末），包括腾讯、阿里、魅族、豌豆荚、小米、UC 等知名互联网公司都做过关于 APP 性能优化的分享，如果你专注于应用开发，并且想做一款备受欢迎的产品，性能优化是你进阶路上必须去学习和实践的。

一、性能问题分类

除了交互、视觉、内容方面的问题外，在用户使用过程中，给用户造成烦恼的问题都可以归结为是性能问题，比如上文中列出的这些都属于性能问题，按照影响的方面不同，可以分为如下几大类：

内存问题：耗内存、OOM、程序切换到后台后占用内存无法释放（OOM 会影响产品的稳定性；耗内存、内存泄露会影响整机的性能；占用内存多预示着留给其它应用的剩余内存空间小）

功耗问题：发烫（耗电）

流畅度问题：启动慢、页面显示需要长时间转圈加载、页面切换卡顿、黑白屏（卡慢崩会让人烦躁）

针对上面一系列的性能问题，谷歌官方提供了各种各样的工具来针对性的解决各个方面的问题，也有很多不错的第三方工具值得尝试：

内存问题：提供了 Android Studio 的静态代码检测功能、Android Monitor；第三方内存泄

露分析工具 Leakcanary、MAT

功耗问题：提供了 GPU 呈现模式、battery-historian、Android Monitor

流畅度问题: 提供了 Android Studio 的静态代码检测功能、Android Monitor、HierarchyViewer、StrictMode、过渡绘制检测工具、TraceView 等

除了上面提到的这些性能优化工具外，谷歌还在 Youtube 上提供了一系列关于 Android 应用性能优化的短视频 Android Performance Patterns，介绍如何优化 Android 各个方面的性能问题。

二、性能优化指标

性能优化的效果仅凭感觉很难衡量，一切应该看数据说话，比如流畅度优化，刷新频率每秒越接近 60 帧越理想，但只要每秒钟超过 24 帧人眼就无法辨别了，所以仅凭感觉是无法区分优化前的 30 帧和优化后的 40 帧的区别的。为了说明做性能优化有足够的价值，就有必要通过一系列指标来说明优化前后的区别。

性能指标的定义应该具有可衡量、可比较的特点，所以每项性能指标可以是数值，也可以是一份报告，比如：

流畅度：FPS，即 Frams per Second，一秒内的刷新帧数，越接近 60 帧越好

启动时间：时间，越短越好

内存泄露：AS 静态代码检测结果、MAT 检测结果，内存泄露很难用数值定义，但可以通过将优化前后工具检测的结果对比得出结论。没有内存泄露最好

内存大小：峰值，峰值越低越好

功耗：单位时间内的掉电量，掉电量越少越好

从上面各项性能指标的定义可以看出，性能优化效果的评估主要是通过对比得出来的，性能如何只是相对的。只要针对同一个应用的同一项指标，优化后比优化前更优，就说明优化是有效果的。

三、性能优化原则和方法

1、优化原则

解决性能问题的过程中，遵循以下几个原则，有助于提高解决问题的效率：

足够多的测量: 不要凭感觉去检测性能问题、评估性能优化的效果, 应该保持足够多的测量, 数据不会说谎。使用各种性能工具有助于快速定位问题，这比凭感觉要靠谱得多

使用低配置的设备：同样的程序，在低端配置的设备中，相同的问题会暴露得更为明显；高配的设备很多时候会让你忽略掉性能问题

权衡利弊：在能够保证产品稳定、按时交付的前提下去做优化，不能顾此失彼，为了性能优化导致产品迟迟不能交付

2、优化方法

性能优化的指标很多，乍看上去无从下手，但和解 bug 一样，只要讲方法，事情就变得迎刃而解。对于大多数问题来讲，只要遵循了解问题→定位问题→分析问题→解决问题→验证问题的思路，基本上都可以解决：

了解问题：对于性能问题来讲，这个步骤只适用于某些明显的性能问题，很多无法感知的性能问题需要通过工具定位

定位问题：通过工具检测、分析，定位在什么地方存在性能问题；如果很难定位，可以采用排除法（屏蔽部分代码，看现象是否仍然存在，如果还存在，则说明被屏蔽的代码没有问题，这样逐渐缩小问题的范围）

分析问题：找到问题后，分析针对这个问题该如何解决，确定解决方案

解决问题：这个没什么可说的，如果是自己无法解决的问题，借助搜索引擎，你遇到过的问题很多人都遇到过，并且极有可能已经被解决了

验证问题：保证每一次优化都有效，没有产生新问题，保证产品的稳定

四、性能优化工具

本文重点介绍谷歌官方提供的一系列应用性能优化工具以及值得推荐的第三方性能优化工具，这些工具主要集中在如下几个地方：

开发者选项：GPU 呈现模式分析、GPU 过渡绘制、严格模式、应用无响应 ANR 等

IDE 中：Android Studio，比如静态代码检测工具、Memory Monitor、CPU Monitor、NetWork Monitor、GPU Monitor、Layout Inspector、Analyze APK 等

SDK 中：sdk\tools，比如 DDMS、HierarchyViewer、TraceView 等

第三方性能优化工具：MAT、Leakcanary 等

调试 GPU 过渡绘制（Visualize GPU Overdraw）

过渡绘制用于检测你的程序是否存在不必要的绘制(举个栗子：同一个区域存在多个视图，

刷新的时候被遮挡的视图也在绘制)，导致显示时的性能问题，显示过渡绘制区域的步骤如下：“开发者选项”→点击“调试 GPU 过渡绘制”→点击“显示过渡绘制区域”，一旦使能，对设备中的任何应用都有效。
过渡绘制的检测和解决方案：通过“开发者选项”中的“显示过渡绘制”和 Android 提供的工具“HierarchyViewer”，以每个界面为单位，可以完全检测出每个界面的过渡绘制问题；因为导致过渡绘制的原因不一，所以也会有多种对应的解决方案：

1、merge 标签可以解决相同布局嵌套导致的过渡绘制问题：merge 标签就是为减少布局层次而生的，它通过减少 View 树的层级来优化布局，merge 只能作为 xml 布局的根标签使用（因为 Activity 的根布局是 FrameLayout，所以只有 Activity 对应的布局文件根标签为 FrameLayout 时才适合使用 merge 标签)，如果在代码中 Inflate 带 merge 标签的布局时，必须为这个自定义 View 指定一个父 ViewGroup，并且设置 attachToRoot 为 true。merge 只能够在 xml 布局文件中使用，没有对应的 java 类。下面的实例演示了 merge 标签的用法，通过“GPU 过渡绘制”查看优化前后的效果，可以明显看到通过 merge 标签解决了过渡绘制的问题；通过 Hierarch View 观察优化前后的视图树，可以明显看到使用 merge 标签后的视图层级减少了。

2、ViewStub 标签可以解决动态加载页面布局，避免默认加载不必要布局的问题：在开发应用的时候，经常会遇到这样的情况，在程序运行时根据条件来决定显示/隐藏哪个视图；通常会在布局文件中将其写上去，默认隐藏，然后在代码中根据条件去判断是否显示。这样做的优点是逻辑清晰，但缺点是耗费资源，在布局文件中将某个视图默认设置为 invisable 或者 gone，在 Inflate 布局文件的时候仍然会被 infalte，同样会被实例化、设置属性，但有可能默认被隐藏的视图用户在某一次操作中很可能不会去触发它。为了提高布局文件加载效率和减少额外的资源消耗，强烈建议使用 ViewStub 标签，ViewStub 是一个用于在运行时加载布局资源、不可见、宽高为 0 的 View，在布局文件中使用它只是用于占位，在代码中没有手动加载它时，并不会影响页面的测量、绘制、显示效率，在代码中通过 inflate 加载 ViewStub 时，ViewStub 会用在布局文件中为其指定的布局文件来代替它自身，通过前面的解释可想而知，ViewStub 只能够被 inflate 一次，一旦加载后 ViewStub 对象就会被置为空；ViewStub 标签有对应的 java 类 ViewStub.java，通过阅读源码可以发现，确实在初始化的时候设置为隐藏、不绘制、宽高为 0，并且它复写了 View 的 dispatchDraw 和 draw 方法，这俩方法是空方法，没有调用 super 的方法，也没有执行自己的代码。

3、Space 标签可以解决只占位、不刷新的视图问题：过渡绘制问题是因为绘制引起的，space 标签可以只在布局文件中占位，不绘制，Space 标签有对应的 java 类 Space.java，通过阅读源码可以发现，它继承至 View.java，并且复写了 draw 方法，该方法为空，既没有调用父类的 draw 方法，也没有执行自己的代码，表示该类是没有绘制操作的，但 onMeasure 方法正常调用，说明是有宽高的。

4、去掉 Window 背景可以解决所有界面的过渡绘制问题：每个 Activity 都会在 AndroidManifest.xml 中设置主题，主题的目的是设置界面的显示风格，但在设置主题的时候通常情况下默认给 Window 设置了背景，注意是 Window 而不是 Activity，Activity 是依附在 Window 上的，Android 系统在刷新整个界面时不仅仅是刷新 Activity，还会刷新 Window。如果默认没有去掉 window 的背景，并且在布局文件中给 Activity 设置了背景，就会存在过渡绘制的问题。

下面是去掉 window 背景后的效果（布局文件不变，主题变动如下）：

```
<style name="AppTheme" parent="@android:style/Theme.Light.NoTitleBar">
    <item name="android:windowBackground">@null</item>
</style>
```

说明：

1、在主题中去掉 Window 的背景时要注意，去掉之后必须重新运行程序检查一下，避免有些 Activity 并没有设置背景导致界面背景为黑色。

2、有的程序为了避免冷启动时界面黑屏/白屏的问题，在主题中为 window 设置了一张图片，然后在布局文件中为 Activity 也设置了背景，这样既会导致过渡绘制问题，还会导致内存问题（同一个页面两张全屏的图片，双倍内存）；所以这种解决方式并不妥，如果是启动速度问题，直接优化启动速度比这种方式靠谱。

5、clipRect 可以解决只刷新固定区域的问题：通过 Canvas 的 clipRect 方法控制每个视图每次刷新的区域，这样可以避免刷新不必要的区域，从而规避过渡绘制的问题

6、不必要的 alpha 值设置可以解决同一视图被多次绘制的问题：如对一个 View 做 Alpha 转化，需要先将 View 绘制出来，然后做 Alpha 转化，最后将转换后的效果绘制在界面上。通俗点说，做 Alpha 转化就需要对当前 View 绘制两遍，可想而知，绘制效率会大打折扣，耗时会翻倍。

启用严格模式（Strict mode enabled）

当当前界面在主线程中存在耗时操作时，会闪烁屏幕，但只会提示你存在耗时操作，不会告诉你具体的地方；如果要精确定位具体哪里耗时，应该在代码中添加 StrictMode 检查，在 log 中会报详细的耗时信息：

```
StrictMode.setThreadPolicy(new
StrictMode.ThreadPolicy.Builder().detectDiskReads().detectDiskWrites()
        .detectNetwork()
        .penaltyLog().build());
StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder().detectLeakedSqlLiteObjects()
        .detectLeakedClosableObjects().penaltyLog().penaltyDeath().build());
```

显示所有“应用无响应”（Show all ANRs）

当任何一个应用（包括后台应用）无响应时会弹出“App Not Responding”对话框，主要用于识别程序之间是否存在干扰

GPU 呈现模式分析（Profiling GPU Rendering）

通过如下方式可以打开 GPU 呈现模式分析：“系统设置”→“开发者选项”→“GPU 呈现模式分析”→在弹出的窗口中选择“在屏幕上显示成条形图（On screen as bars）”。

从图中可以看出，每一条柱状线包含三种颜色，但从 Android 6.0 开始，你看到的每条柱状线已不止三种颜色：

每种颜色代表每一帧渲染过程中需要完成的某一件事情，因为 6.0 之前的三种颜色不大能够清晰地帮助我们定位性能问题的具体原因，所以从 6.0 开始，将每一帧的渲染过程拆分成了 8 个步骤，每个步骤一种颜色，每种颜色的意义如下：

(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：
I/Choreographer(*): Skipped XXX frames! The application may be doing too much work on its main thread

通过 GPU 呈现模式可以清晰地检测出导致渲染问题的具体原因，但不能定位是哪一行代码出了问题，从上面的描述可知，减少过渡绘制可以很好地提升 GPU 呈现模式的表现力；如果要跟踪具体哪一行代码导致了渲染的性能问题，需要借助各种性能检测工具。比如通过 TraceView 跟踪是否存在耗时操作；通过“显示过渡绘制”跟踪是否存在过渡绘制等。

IDE Android Studio

AS 不仅提供了程序开发、构建、调试的环境，还提供了一系列优化应用质量的工具，这些工具包括静态代码检测工具 Inspect Code、Android Monitor、Analyze APK...同时还集成了 Android Device Monitor。

应用内存泄露分析

1、非静态内部类导致的内存泄露，比如 Handler，解决方法是将内部类写成静态内部类，在静态内部类中使用软引用/弱引用持有外部类的实例。

2、IO 操作后，没有关闭文件导致的内存泄露，比如 Cursor、FileInputStream、FileOutputStream 使用完后没有关闭，这种问题在 Android Studio 2.0 中能够通过静态代码分析检查出来，直接改善就可以了。

3、自定义 View 中使用 TypedArray 后，没有 recycle，这种问题也可以在 Android Studio 2.0 中能够通过静态代码分析检查出来，直接改善就可以了。

4、某些地方使用了四大组件的 context，在离开这些组件后仍然持有其 context 导致的内存泄露，这种问题属于共识，在编写代码的过程中就应该按照规则来，使用 Application 的 Context 就可以解决这类内存泄露的问题，至于什么情况下应该使用四大组件的 Context，什么时候应该使用 Application 的 context 可以参见下表

	Application	Activity	Service	ContentProvider	BroadcastReceiver
Show a Dialog	NO	YES	NO	NO	NO
Start an Activity	NO[1]	YES	NO[1]	NO[1]	NO[1]
Layout Inflation	NO[2]	YES	NO[2]	NO[2]	NO[2]
Start a Service	YES	YES	YES	YES	YES
Bind to a Service	YES	YES	YES	YES	NO
Send a Broadcast	YES	YES	YES	YES	YES
Register BroadcastReceiver	YES	YES	YES	YES	NO[3]
Load Resource Values	YES	YES	YES	YES	YES

备注：大家注意看到有一些 NO 上添加了一些数字，其实这些从能力上来说是 YES，但是为什么说是 NO 呢？下面一个一个解释：

1、数字 1：启动 Activity 在这些类中是可以的，但是需要创建一个新的 task，一般情况不推荐。

2、数字 2：在这些类中去 layout inflate 是合法的，但是会使用系统默认的主题样式，如果你自定义了某些样式可能不会被使用。

3、数字 3：在 Receiver 为 null 时允许，在 4.2 或以上的版本中，用于获取黏性广播的当前值。（可以无视）。

4、ContentProvider、BroadcastReceiver 之所以在上述表格中，是因为在其内部方法中都有一个 context 用于使用。

还有一种不属于内存泄露，但在分析内存泄露的问题时应该一并解决：同一个 APP，将图片放在不同的 drawable 文件夹下，在相同的设备上占用的内存情况不一样，具体可以参见：关于 Android 中图片大小、内存占用与 drawable 文件夹关系的研究与分析。解决这个问题遵循以下原则就可以了：
1、UI 只提供一套高分辨率的图，图片建议放在 drawable-xxhdpi 文件夹下（放在 xxxhdpi 或者更高分辨率的文件夹下没有必要，权衡利弊，照顾主流设备即可），这样在低分辨率设备中图片的大小只是压缩，不会存在内存增大的情况。

2、涉及到桌面插件或者不需要缩放的图片，放在 drawable-nodpi 文件夹下，这个文件夹下的图片在任何设备上都是不会缩放的。

通过工具检查程序运行后的内存泄露

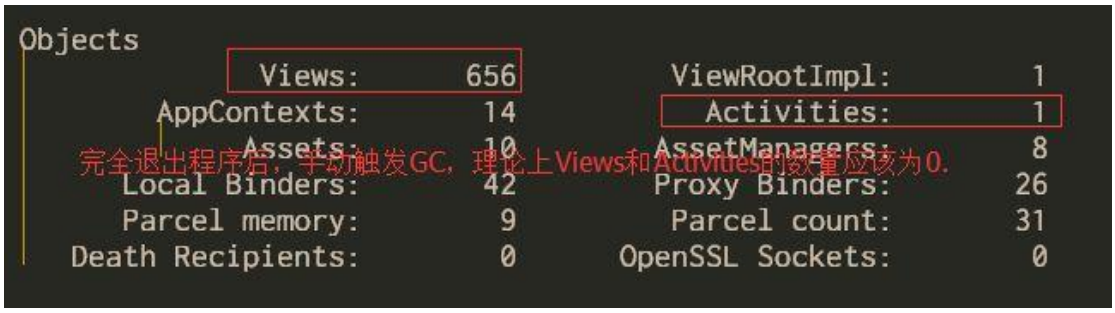
通过上面的步骤，应用中的大部分内存泄露问题都能够得到解决，还有一些内存泄露，需要

运行程序，分析运行后的内存快照来解决，比如注册之后没有反注册、类中的静态成员变量导致的内存泄露、SDK 中的内存泄露等。解决这类问题可以分两步进行：

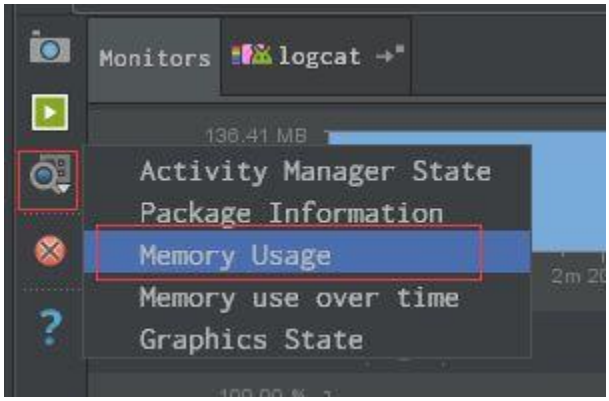
通过内存泄露检测工具先定位是哪有问题，内存泄露的检测有两种比较便捷的方式：

1、一种是使用开源项目 Leakcanary，需要添加到代码中，运行后生成分析结果。

2、另一种方式是使用 adb shell dumpsys meminfo packagename -d 命令，在进入一个界面之前查看一遍 Activity 和 View 的数量，在退出这个界面之后再查看一遍 Activity 和 View 的数量，对比进入前和进入后 Activity 和 View 数量的变化情况，如果有差异，则说明存在内存泄露（在使用命令查看 Activity 和 View 的数量之前，记得手动触发 GC）。



备注：在 Android Studio 中，可以通过如下方式获取当前选中进程的内存信息：



然后通过 MAT 取程序运行时的内存快照做详细分析，对于 MAT 的使用，网上有很多优质的文章，比如：Android 性能优化之使用 MAT 分析内存泄露问题，在使用 MAT 前，有必要知道这几点：

1、 不要指望 MAT 明确告诉你哪里存在内存泄露，这需要你根据上一步骤首先定位到可能存在内存泄露的类，然后借助 MAT 确认是否真的存在内存泄露，具体哪个地方存在内存泄露。

2、借助 Retained Size 分析某一个类及与之相关的实例所消耗的内存，如果这个类的 Retained Size 比较大，优先分析。

3、检查某个类是否存在内存泄露时，排除其软/弱/虚引用，右键某个类→Merge Shortest Paths to GC Roots→exclude all phantom/weak/soft etc.references。