Android绘制优化

转载自：Android性能优化之绘制优化

前言

成为一名优秀的Android开发，需要一份完备的知识体系，在这里，让我们一起成长为自己所想的那样~。

我们都知道，造成绘制不流畅最大的罪魁祸首就是卡顿，而卡顿的主要场景有很多，按场景可以分成4类：UI绘制、应用启动、页面跳转、事件响应，其中又可细分为如下：

UI

- 绘制
- 刷新

启动

- 安装启动
- 冷启动
- 热启动

跳转

- 页面间跳转
- 前后台切换

响应

- 按键
- 系统事件
- 滑动

造成其根本原因可以分为两大类：

界面绘制

- 绘制层级深
- 页面复杂
- 刷新不合理

数据处理

- 数据处理在UI线程
- 占用CPU高，导致主线程拿不到时间片
- 内存增加导致GC频繁，从而引起卡顿

一、Android系统显示原理

Android的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。也就是说应用层负责绘制，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，系统层服务通过刷新机制把数据更新到屏幕。

1.1、绘制原理

1、应用层

在Android的每个View都会经过Measure和Layout来确定当前需要绘制的View所在的大小和位置，通过绘制到surface，在Android系统中整体的绘制源码是在ViewRootImpl类的performTraversals()方法，通过这个方法可以看出Measure和Layout都是递归来获取View的大小和位置，并且以深度作为优先级。显然，层级越深，元素越多，耗时就越长。

对于绘制，Android支持两种绘制方式：

- 软件绘制（CPU）
- 硬件绘制（GPU）

硬件加速从Android 3.0开始支持，它在UI显示和绘制效率方面远高于软件绘制。但它的局限如下：

- 耗电：GPU功耗高于CPU。
- 兼容性：不兼容某些接口和函数。
- 内存大：使用OpenGL的接口需要占用内存8MB。

2、系统层

将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，它的主要工作流程如下：

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

其中，SurfaceFlinger系统进程和应用进程使用了匿名共享内存SharedClient，并且，每一个应用和SurfaceFlinger之间都会创建一个SharedClient，并且，每个SharedClient中，最多可以创建31个SharedBufferStack，每一个SharedBufferStack对应一个Surface，即一个window。（其中包含了两个（小于4.1版本）或者三个（4.1及以上版本）缓冲区）

因此，从上可知，一个Android应用程序最多可以包含31个窗口。最后，显示的整体流程如下：

- 应用层绘制到缓冲区
- SurfaceFlinger把缓冲区数据渲染到屏幕，其中使用了Android匿名共享内存SharedClient缓存需要显示的数据来达到目的。

绘制的过程首先是CPU准备数据，通过Driver层把数据交给CPU渲染，其中CPU主要负责Measure、Layout、Record、Execute的数据计算工作，GPU负责Rasterization（栅格化）、渲染。因为图形API不允许CPU直接和GPU通信，所以要通过一个图形驱动的中间层来进行连接。图形驱动里面维护了一个队列，CPU把display list（待显示的数据列表）添加到队列中，GPU从这个队列中取出数据进行绘制，最终才在显示屏上显示出来。

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，这样就能够达到流畅的画面所需的60FPS。

1.2 刷新机制

4.1版本的Project Butter对Android Display系统进行了重构，引入了三个核心元素：VSYNC（Vertical Synchronization）、Triple Buffer、Choreographer。其中作为Project Buffer核心的VSYNC，即垂直

同步可认为是一种定时中断。而Choreographer起调度的作用，将绘制工作统一到VSYNC的某个时间点上，使应用的绘制工作有序。

为什么要推出Project Butter?

解决刷新不同步的问题。

为什么要使用双缓冲技术?

在Linux上通常使用Framebuffer来做显示输出，当用户进程更新Framebuffer中的数据后，显示驱动会把FrameBuffer中每个像素点的值更新到屏幕，但是如果上一帧数据还没显示完，Framebuffer中的数据又更新了，就会带来残影的问题，用户会觉得有闪烁感，所以采用了双缓冲技术。

双缓冲的含义?

双缓冲意味着要使用两个缓冲区（在上文提及的SharedBufferStack中），其中一个称为Front Buffer，另一个称为Back Buffer。UI总是先在Back Buffer中绘制，然后再和Front Buffer交换，渲染到显示设备中。即只有当另一个buffer的数据准备好后，通过io_ctl来通知显示设备切换Buffer。

Choreographer的作用是什么?

当收到VSYNC信号时，调用用户设置的回调函数。回调类型的优先级从高到低为CALLBACK_INPUT、CALLBACK_ANIMATION、CALLBACK_TRAVERSAL。

当第一帧数据没有及时处理时，为什么CPU不能在第二个16ms处即VSync到来就开始工作呢?

因为只有两个Buffer；所以4.1版本后，出现了第三个缓冲区：Triple Buffer。它利用CPU/GPU的空闲等待时间提前准备好数据，并不一定会使用。
注意：除非必要，大部分情况下只是用到双缓冲。而且，缓冲区并不是越多越好，要做到平衡到最佳效果。

Google做了这么多的优化，为什么实际开发中应用还存在卡顿现象?

因为VSync 中断处理的线程优先级一定要最高，否则即使接收到VSync中断，不能及时处理，也是徒劳无功。

1.3、卡顿的根本原因

- 绘制任务太重、绘制一帧内容耗时太长。
- 主线程太忙，导致VSync信号来时还没有准备好数据导致丢帧。

二、性能分析工具

Android常用的性能优化工具一般有如下几种：

- Hierarchy View：查看Layout层次
- Android Studio自带的Profile工具
- 静态代码检查工具Lint
- TraceView
- Systrace

2.1、卡顿检测工具Profile GPU Rendering

它是Android手机上自带的一个辅助工具，打开Profile GPU Rendering后可以看到实时刷新的彩色图，其中每一根竖线表示一帧，由多个颜色组成，不同颜色的解释如下：

- 每一条柱状图都由红、黄、蓝、紫组成，分别对应每一帧在不同阶段的实际耗时。
- 蓝色：测量绘制的时间，需要多长时间去创建和更新DisplayList。在蓝色的线很高时，有可能是因为需要重新绘制，或者自定义视图的onDraw函数处理事情太多。
- 红色：Android进行2D渲染Display List的执行的时间。当红色的线非常高时，可能是由于重新提交了视图导致的。
- 橙色：处理时间或CPU告诉GPU渲染一帧的地方，如果柱状图很高，就意味着GPU太繁忙了。
- 紫色：将资源转移到渲染线程的时间。（4.0版本以上提供）

并且，从Android M开始变成了渲染八步骤：

1、橙色-Swap Buffers

表示GPU处理任务的时间。

2、红色-Command Issue

进行2D渲染显示列表的时间，越高表示需要绘制的视图越多。

3、浅蓝-Sync&Upload

准备有待绘制的图片所耗费的时间，越高表示图片数量越多或图片越大。

4、深蓝-Draw

测量和绘制视图所需的时间，越高表示视图越多或onDraw方法有耗时操作。

5、一级绿-Measure/Layout

onMeasure与onLayout所花费的时间。

6、二级绿-Animation

执行动画所需要花费的时间。越高表示使用了非官方动画工具或执行中有读写操作。

7、三级绿-Input Handling

系统处理输入事件所耗费的时间。

8、四级绿-Misc Time/Vsync Delay

主线程执行了太多任务，导致UI渲染跟不上vSync的信号而出现掉帧。
此外，可通过如下adb命令将具体的耗时输出到日志中来分析：

```
1 adb shell dumpsys gfxinfo com.**.**
```

2.2、TraceView

它主要用来分析函数的调用过程，可以对Android的应用程序以及Framework层代码进行性能分析。
使用TraceView查看耗时，主要关注Calls + Recur Calls / Total和（该方法调用次数+递归次数）和Cpu Time / Call（该方法耗时）这两个值，然后优化这些方法的逻辑和调用次数，减少耗时。

注意：RealTime（实际时长）的实际执行时间要比CPU Time要长，因为它包括了CPU的上下文切换、阻塞、GC等。

2.3、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，它的作用有：

- 收集Android关键子系统（如surfaceflinger、WindowManagerService等Framework部分关键模块、服务、View系统等）的运行信息，这样可以更直观地分析系统瓶颈，改进性能。
- 跟踪系统的I/O操作、内核工作队列、CPU负载等，在UI显示性能分析上提供很好的数据，特别是在动画播放不流畅、渲染卡等问题上。

注意：Systrace是以系统的角度返回一些信息，并不能定位到具体耗时的方法，建议使用TraceView。

1、Systrace使用方法

使用事项如下：

- 支持4.1版本及以上。
- 4.3以前的系统版本需要打开Setting>Developer options>Monitoring>Enable traces。

一般我们使用命令行来得到输出的html表单，在4.3版本及以上可以省略设置跟踪类别标签来获取默认值。
命令如下：

```
1 cd android-sdk/platform-tools/systrace
2 python systrace.py --time=10 -o mynewtrace.html sched gfx view wm
```

其中，常用的几个参数命令如下：

- -o ： 保存的文件名。
- -t N, 一time=N：多少秒内的数据，默认为5s，以当前时间点往后倒N秒时间。

其余标签用法请参见此处。
此外，我们可以使用代码插桩的方式，在Android 4.3及以上版本可以使用Trace类的Trace.beginSection()与Trace.endSection()方法来进行追踪。其中需要注意：

- 保证beginSection和endSection的调用次数要匹配。
- Trace的begin与end必须在同一线程中执行。

2、分析Systrace报告

使用Chrome打开文件后，其中和UI绘制关系最密切的是Alerts和Frame两个数据：

- Alerts：标记了性能有问题的点，单击该点可以查看详细信息，右侧的Alerts框还可以看到每个类型的Alerts的数量。
- Frame：每个应用都有一行专门显示frame，每一帧就显示为一个绿色的圆圈。当显示为黄色或者红色时，它的渲染时间超过了16.6ms。

这里，列出Systrace有用的快捷键：

- W：放大
- S：缩小
- A：左移
- D：右移

三、布局优化

3.1、布局优化方法

1、减少层级

- 合理使用RelativeLayout和LinearLayout。
- 合理使用Merge。

RelativeLayout也存在性能低的问题，原因是RelativeLayout会对子View做两次测量。但如果在LinearLayout中有weight属性，也需要进行两次测量，因为没有更多的依赖关系，所以仍然会比RelativeLayout的效率高。

注意：由于Android的碎片化程度很高，所以使用RelativeLayout能使构建的布局适应性更强。

merge的原理：在Android布局的源码中，如果是Merge标签，那么直接将其中的子元素添加到Merge标签Parent中。

注意：

- Merge只能用在布局XML文件的根元素。
- 使用merge来加载一个布局时，必须指定一个ViewGroup作为其父元素，并且要设置加载的attachToRoot参数为true。
- 不能在ViewStub中使用Merge标签。原因就是ViewStub的inflate方法中根本没有attachToRoot的设置。

2、提供显示速度

ViewStub是一个轻量级的View，它是一个看不见的，并且不占布局位置，占用资源非常小的视图对象。可以为ViewStub指定一个布局，加载布局时，只有ViewStub会被初始化，然后当ViewStub被设置为可见时，或是调用了ViewStub.inflate()时，ViewStub所指向的布局会被加载和实例化，然后ViewStub的布局属性都会传给它指向的布局。

注意：

- ViewStub只能加载一次，之后ViewStub对象会被置为空。所以它不适用于需要按需显示隐藏的情况。
- ViewStub只能用来加载一个布局文件，而不是某个具体的View。
- ViewStub中不能嵌套Merge标签。

3、布局复用

Android的布局复用可以通过标签来实现。

一些优化总结：

- 使用标签加载一些不常用的布局。
- 尽可能少用wrap_content，wrap_content会增加布局measure时的计算成本，已知宽高为固定值时，不用wrap_content。
- 使用TextView替换RL、LL。
- 使用低端机进行优化，以发现性能瓶颈。
- 使用TextView的行间距替换多行文本：lineSpacingExtra/lineSpacingMultiplier。
- 使用Spannable/Html.fromHtml替换多种不同规格文字。

- 尽可能使用LinearLayout自带的分割线。
- 使用Space添加间距。
- 多利用lint + alibaba规约修复问题点。
- 嵌套层级过多可以考虑使用约束布局。

四、避免过度绘制

导致过度绘制的主要原因是：

- XML布局：控件有重叠且都有设置背景。
- View自绘：View.OnDraw里面同一个区域被绘制多次。

4.1、过度绘制检测工具

打开手机开发者选项中的Show GPU Overdraw选项，会有不同的颜色来表示过度绘制次数，依次是无、蓝、绿、淡红、深红，分别对应0–4次过度绘制。

4.2、如何避免过度绘制

1、布局上的优化

- 移除XML中非必需的背景，或根据条件设置。
- 有选择性地移除窗口背景：getWindow().setBackgroundDrawable(null)。
- 按需显示占位背景图片。

比如：在获取Avatar的图像之后，把ImageView的Background设置为Transparent，只有当图像没有获取到时，才设置对应的Background占位图片。

2、自定义View优化

通过canvas.clipRect()来帮助系统识别那些可见的区域。这个方法可以指定一块矩形区域，只有在这个区域内才会被绘制。并且，它还可以节约CPU和GPU资源，在clipRect区域之外的绘制指令都不会被执行。在绘制一个单元之前，首先判断该单元的区域是否在Canvas的剪切域内。若不在，直接返回，避免CPU和GPU的计算和渲染工作。

五、合理的刷新机制

5.1 减少刷新次数

- 控制刷新频率
- 避免没有必要的刷新

5.2 避免后台线程的影响

如通过监听ListView的onScrollStateChanged事件，在滚动时暂停图片下载线程工作，结束后再开始，可以提高ListView的滚动平滑度。

5.3 缩小刷新区域

如自定义View一般采用invalidate方法刷新，可以用以下重载方法刷新要刷新的区域：

- invalidate(Rect dirty);
- invalidate(int left, int top, int right, int bottom);

六、提升动画性能

提升动画性能主要从以下三个纬度着手：

- 流畅度：控制每一帧动画在16m内完成。
- 内存：避免内存泄漏，减小内存开销。
- 耗电：减小运算量，优化算法，减小CPU占用。

6.1 帧动画

消耗资源最多，效果最差，能不用就不用。

6.2 补间动画

使用补间动画实现导致View重绘非常频繁，更新DisplayList的次数过多，且有以下缺点：

- 只能用于View对象。
- 只有4种动画操作。
- 只是改变View的显示效果，但是不会真正改变View的属性。

6.3 属性动画

相比于补间动画，属性动画重绘明显会少很多，应优先使用。

6.4 硬件加速

1、硬件加速原理

核心类：DisplayList，每一个View对应一个。
在打开硬件渲染后绘制View时，其中执行绘制的draw()方法会把所有绘制命令记录到一个新的显示列表（DisplayList），这个显示列表包含了输出的View层级的绘制代码，但并不是加入到显示列表就立刻执行，当这个ViewTree的DisplayList全都记录完毕后，由OpenGLRender负责将Root View中的DisplayList渲染到屏幕上。而invalidate()方法只是在显示列表中记录和更新显示层级，去标记不需要绘制的View。

2、硬件加速控制级别

如果应用程序中只使用了标准View或者Drawable，就可以为整个系统打开硬件加速的全局设置。

3、在动画上使用硬件加速

硬件纹理操作对一个View进行动画绘制，如果不调用invalidate()方法，就可以减少对View自身频繁的重绘。同时Android 3.0的属性动画也减小了重绘，当View通过硬件层返回时，最终所有的层叠画面显示到屏幕，View的属性同时被处理好，因此只要设置这些属性，就可以明显提高绘制的效率，它们不需要View重绘，设置属性后，View会自动刷新。因此，属性动画中绘制的递归次数比补间动画少很多。
在Android 3.0前，使用View的绘制缓冲或Canvas.saveLayer()函数对离屏缓冲进行渲染。Android 3.0后则使用View,.setLayerType(type, paint)方法代替，type可以为以下三种Layer类型之一：

- LAYER_TYPE_NONE：普通渲染方式，不会返回一个离屏的缓冲，默认值。
- LAYER_TYPE_HARDWARE：如果这个应用使用了硬件加速，这个View将会在硬件中渲染为硬件纹理。
- LAYER_TYPE_SOFTWARE：此View通过软件渲染为一个Bitmap。

设计一个动画的流程如下：

1、将要执行动画的View的LayerType设置为LAYER_TYPE_HARDWARE。
2、计算动画View的属性等信息，更新View的属性。
3、若动画结束，将LayerType设置为NONE。

硬件加速需要注意的问题：

- 在软件渲染时，可以使用重用Bitmap的方法来节省内存，但是如果开起来硬件加速，这个方案就不起作用。
- 开启硬件加速的View在前台运行时，需要耗费额外的内存，加速的UI切换到后台时，产生的额外内存有可能不释放。
- 当UI中存在过渡绘制时，硬件加速会比较容易发问题。

七、卡顿监控方案与实现

目前比较流行的方案都是利用了Looper中的Printer来实现监控。

7.1 监控原理

利用主线程的消息队列处理机制，通过自定义Printer，然后在Printer中获取到两次被调用的时间差，这个时间差就是执行时间。如果该时间超过阈值（如1000ms）时，主线程卡顿发生，并抛出各种有用信息，供开发者分析。（此外，也可以在UI线程以外开启一个异步线程，定时向UI线程发送一个任务，并记下发送时间。任务的内容是将执行时间同步到发送线程，如果UI线程被阻塞，那么发送过去的任务不能被准时执行。但此方法会增加系统开销，不可取）

可抓取的信息

- 基础信息：系统版本、机型、进程名、应用版本号、磁盘空间、UID等。
- 耗时信息：卡顿开始和结束时间。
- CPU信息：CPU的信息、整体CPU使用率和本进程CPU使用率（可粗略判断是当前应用消耗CPU资源太多导致的卡顿，还是其他原因）等。
- 堆栈信息。

注意：这里的信息建议抽样上报或者可以先将其保存到本地，在合适的时机以及达到一定的量时，再压缩上报到服务器，供开发者分析。具体监控代码实现可以参考BlockCanary开源项目的代码。

八、总结

至此，这里我们分析一下绘制优化应经历的几个过程：

- 1、发现问题：除使用时感知的卡顿外，还应通过卡段监控工具来发现整体的耗时情况，或打开开发者选项的一些辅助工具来发现问题。
- 2、分析问题：可以使用Systrace和TraceView来分析耗时，使用Hierarhy Viewer来分析页面层级。
- 3、寻求原因：深入探索导致问题的根本原因。
- 4、解决问题。

应用之所以会出现卡顿，除了绘制方面的问题，还有一个影响因素就是内存，不合理地使用内存不仅会导致卡顿，还会对耗电和应用的稳定性造成很大影响，下一篇文章，笔者将对Android中的内存优化进行全面

的讲解，若读者觉得哪里有写的不好的地方或有误的地方希望多多进行批评指正，愿我们共同进步和成长！

参考链接：

1、Android应用性能优化最佳实践
2、必知必会 | Android 性能优化的方面方面都在这儿