|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 分类号 |  | |  | | | 密 级 | |  |
| U D C |  | |  | | | 编 号 | | 0003 |
|  | | | | | | | | |
| **读 书 笔 记** | | | | | | | | |
|  | | | | | | | | |
| Android应用性能优化最佳实践 | | | | | | | | |
|  | |  | |  |  | |  | |
|  | |  |  | |
|  | |  |  | |
|  | |  |  | |
|  | |  |  | |
|  | |  |  | |
|  | | | | | | | | |
| By key.guan  2017.5~ | | | | | | | | |

# 前言

## 为什么写这本书

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，即使应用非常具有特色，或者功能和业务具有唯一性，在产品前期可能吸引了部分用户，但用户体验不好的话，也会给产品带来很差的口碑，如果有在体验上更好的竞品，用户也会很快转移。那么一个好的应用应该如何定义呢？主要有三方面：

* 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一款操作系统，硬件配置有一定的限制，虽然配置现在越来越高级，但仍然无法和PC相比，在CPU和内存上的使用不合理或者耗费资源多时，就会碰到内存不足导致的稳定性问题、CPU消耗太多导致的卡顿问题等。例如，我们发布一款产品后会收到很多的反馈，这些反馈来自很多渠道，有用户反馈，有应用发布平台的反馈通道等。

面对这些问题时，大家想到的都是联系用户，然后看日志，特别是有关性能类问题的反馈，原因也非常难找，日志大多用处不大，为什么呢？因为性能问题大部分是非必现的问题，定位时很难复现，而又没有关键的日志，当然就无法找到原因了。这些问题非常影响用户的体验和功能的使用，所以解决这些问题是非常重要的。当前市场上讲解性能优化的书太少，即使有些书讲到，很多也是一笔带过，没有深入分析和寻找解决方案，所以有必要用一本书来从多个维度讲解在性能上我们面临了什么问题，如何解决这些问题，并在实际的项目中来优化我们的应用，以提高用户体验。

## 本书面向的读者

本书适合所有Android应用开发从业人员及在校学生，特别是有一定Android应用开发经验的开发人员，高级开发人员也可以通过本书了解更多的性能调优知识。

## 本书特色

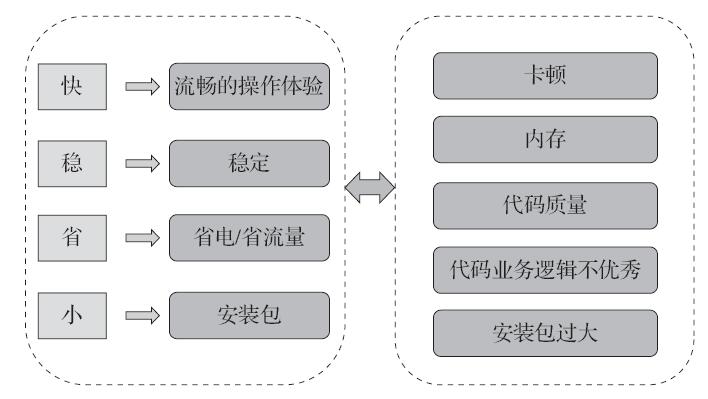
本书为进阶类图书，对于一些基础技术和基础理论知识不会做过多的阐述，特别是入门类的知识点，大家可以从其他书籍获取相关的知识。书中以性能优化为核心，深入剖析性能优化具体涉及的技术背景与优化方案，同时提供典型案例，帮助读者更深入地掌握Android应用开发技术，理解Android的运行机制和原理，掌握Android性能优化的思想，让开发者快速成长，打造高质量的Android应用。

## 本书的主要内容

可以把用户能体验到的性能问题主要总结为4个类别：

* 流畅
* 稳定
* 省电
* 省流量

性能问题的主要原因是什么，原因有相同的，也有不同的，但归根结底，不外乎内存使用、代码效率、合适的策略逻辑、代码质量这一类问题。本书讲解内容的目标和方向如下图所示。



从上图可以看到，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，减少用户的等待时间，满足用户预期。
* 稳：降低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，减小用户使用成本，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。

这4类问题需要从根源上解决，也就是要解决图中第二个框里的问题：卡顿、内存使用不合理、代码质量差、代码逻辑不优秀、安装包过大。这些问题也是在开发过程中碰到最多的问题，在实现业务需求的同时，也需要考虑到这些点，多花时间去思考，避免功能完成后再来做优化和修复Bug，这个时候带来的成本会增加。如果是维护之前的代码，就需要使用一系列工具来发现问题点。

性能优化不是更新一两个版本可以解决的，是持续性的需求，结合到实际中，在一个新产品/项目开始时，由于人力和上线时间的限制，可以把优先级放低，但有些点是在写代码时就要考虑的，这就体现出程序员的技术功底。

本书强调性能调优的核心思想和方向如下：

发现问题→分析问题原因及背景→寻找最优解决方案→解决问题。

本书一共7章，在简单介绍了Android Studio的使用指南后，分别从绘制（UI）、内存、存储、稳定性、耗电以及安装包6个方面进行优化，从系统上深入分析绘制和内存的原理，一步步深入了解导致性能问题的本质原因，同时讲述了多种性能优化工具的使用，通过分析典型案例，得到有效的优化方案，从而实现更高质量的应用。书中所讲述的内容均基于Android 6.0系统。

## 勘误和资源下载

本书代码的下载地址：<https://github.com/lyc7898/AndroidTech>。

Android Studio的使用不再这里展开了。

# 绘制优化

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，如图2-1所示。在这四种场景下又有多个小分类，基本上覆盖了卡顿的各个场景。

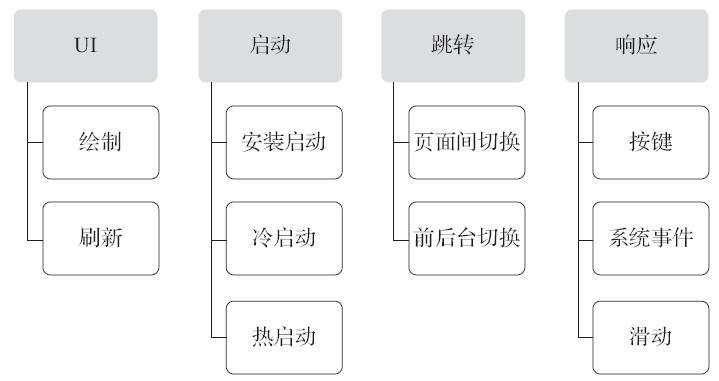


图2-1　卡顿主要场景

这4种卡顿场景的根本原因又可以分成两大类。

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出现在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，一是数据处理在UI线程（这种应该避免），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。

本章主要通过优化UI界面编程来减少卡顿，以场景为纬度，通过工具深入分析症结所在，找到导致问题的根本原因，利用涉及的相关技术背景，以及了解当前业内主流解决方案，然后结合实例来找到最终的优化方案，使应用流畅。

引起卡顿的原因有很多，但不管怎么样的原因和场景，最终都是通过设备屏幕上的显示来到达用户，归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

说到显示原理，相信大家从网上或其他书籍上看过相关的知识，但大部分人看得云里雾里，是因为整个显示系统很复杂吗？确实很复杂，但我们只需要了解整体流程，抓住关键知识，从应用角度上来讲，需要掌握的不多，如果自己有兴趣，可以阅读专门介绍系统框架的书籍，结合源码来分析，这里就不过多地介绍系统层的知识了。下面我们首先介绍在应用开发上需要涉及的知识点和整体流程。

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。**

通过阅读Android系统的源码可以了解显示的流程，Android的图形显示系统采用的是Client/Server架构。SurfaceFlinger（Server）由C++代码编写。Client端代码分为两部分，一部分是由Java提供给应用使用的API，另一部分则是由C++写成的底层具体实现。下面通过介绍绘制原理和刷新机制来学习整个显示过程。

### 绘制原理

绘制任务是由应用发起的，最终通过系统层绘制到硬件屏幕上，也就是说，应用进程绘制好后，通过跨进程通信机制把需要显示的数据传到系统层，由系统层中的SurfaceFlinger服务绘制到屏幕上。那么应用层和系统层中的流程是什么样的呢？接下来将进行具体介绍。

#### 应用层

先来看一个UI界面的典型构成框架，也可以是一个Activity的构成。如图2-2所示，有很多不同层次的基本元素——View，整体是一个树型结构，有不同的嵌套，存在着父子关系，子View在父View中，这些View都经过一个相同的流程最终显示到屏幕上，这也意味着要完整地显示所有数据，就要对其中的View都进行一次绘制工作，并且针对每个View的操作都是一个递归过程。

在Android的每个View绘制中有三个核心步骤（见图2-3），通过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（Draw）到surface，在Android系统中整体的绘图源码是在ViewRootImp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。可以看出，层级越深，元素越多，耗时也就越长。



图2-2　页面构成框架



图2-3　View绘制流程

（1）Measure

用**深度优先**原则递归得到所有视图（View）的宽、高；获取当前View的正确宽度**childWidthMeasureSpec**和高度childHeightMeasureSpec之后，可以调用它的成员函数Measure来设置它的大小。如果当前正在测量的子视图child是一个视图容器，那么它又会重复执行操作，直到它的所有子孙视图的大小都测量完成为止。

（2）Layout

用深度优先原则递归得到所有视图（View）的位置；当一个子View在应用程序窗口左上角的位置确定之后，再结合它在前面测量过程中确定的宽度和高度，就可以**完全确定它在应用程序窗口中的布局。**

（3）Draw

目前Android支持了两种绘制方式：软件绘制（CPU）和硬件加速（GPU），其中硬件加速在Android 3.0开始已经全面支持，很明显，硬件加速在UI的显示和绘制的效率远远高于CPU绘制，但硬件加速并非如大家所想的那么完善，它也存在明显的缺点：

·耗电：GPU的功耗比CPU高。

·兼容问题：某些接口和函数不支持硬件加速。

·内存大：使用OpenGL的接口至少需要8MB内存。

所以是否使用硬件加速，需要考虑一些接口是否支持硬件加速，同时结合产品的形态和平台，比如TV版本就不需要考虑功耗的问题，而且TV屏幕大，使用硬件加速容易实现更好的显示效果。

#### 系统层

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，SurfaceFlinger的具体实现和工作原理因为和应用层关系不大，所以这里不做过多介绍，只需要了解它主要是做些什么工作。

* ·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。

既然是两个不同进程，那么肯定需要一个跨进程的通信机制来实现数据传输，在Android的显示系统，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SharedClient，如图2-4所示。从图2-4中可以看出，在每个SharedClient中，最多可以创建31个SharedBufferStack，每个Surface都对应一个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Surface。也就是说SharedClient包含的是SharedBufferStack的集合。因为最多可以创建31个SharedBufferStack，这也意味着一个Android应用程序最多可以包含31个窗口，同时每个SharedBufferStack中又包含了两个（低于4.1版本）或者三个（4.1及以上版本）缓冲区，即在后面的显示刷新机制中会提到的双缓冲和三重缓冲技术。



图2-4　Android显示框架

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把缓存区数据渲染到屏幕，由于是两个不同的进程，所以使用Android的匿名共享内存SharedClient缓存需要显示的数据来达到目的。

SurfaceFlinger把缓存区数据渲染到屏幕（流程如图2-5所示），主要是驱动层的事情，这里不做太多解释。

从图2-5中可以看出，绘制过程首先是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。由于图形API不允许CPU直接与GPU通信，而是通过中间的一个图形驱动层（Graphics Driver）来连接这两部分。图形驱动维护了一个队列，CPU把display list添加到队列中，GPU从这个队列取出数据进行绘制，最终才在显示屏上显示出来。



图2-5　渲染数据流程图

知道了绘制的原理后，那么到底绘制一个单元多长时间才是合理的，首先需要了解一个名词：FPS。FPS（Frames Per Second）表示每秒传递的帧数。在理想情况下，**60 FPS就感觉不到卡**，这意味着每个绘制时长应该在**16ms**以内，如图2-6所示。



图2-6　理想状态下的绘制操作

但是Android系统很有可能无法及时完成那些复杂的界面渲染操作。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，这样就能够达到流畅的画面所需的60FPS。即为了实现60FPS，就意味着程序的大多数绘制操作都必须在16ms内完成。

如果某个操作花费的时间是24ms，系统在得到VSYNC信号时就无法进行正常渲染，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。主要场景在执行动画或者滑动ListView时更容易感知到卡顿不流畅，是因为这里的操作相对复杂，容易发生丢帧的现象，从而感觉卡顿。有很多原因可以导致CPU或者GPU负载过重从而出现丢帧现象：可能是你的Layout太过复杂，无法在16ms内完成渲染；可能是UI上有层叠太多的绘制单元；还有可能是动画执行的次数过多。

最终的数据是刷新机制通过系统去刷新数据，刷新不及时也是引起卡顿的一个主要原因。接下来将详细介绍系统是怎么刷新的以及在什么情况下会导致卡顿发生。

### 刷新机制

Google发布Android操作系统后，Android OS系统一直在不断优化、更新。但直到Android 4.0版本发布，有关UI显示不流畅的问题仍未得到根本解决。在整个Android版本升级过程中，Android在显示系统方面做了不少优化和改进，比如支持硬件加速等技术，但本质原因似乎和硬件关系并不大，也没有得到太多改善。而与高端硬件配置的Android机器价格相近的iPhone，其UI的流畅性强却是有目共睹的。

从Android 4.1（Jelly Bean）开始，Android OS开发团队便力图在每个版本中解决一个重要问题。作为严重影响Android口碑问题之一的UI流畅性差的问题，首先在Android 4.1版本中得到了有效处理。其解决方法即在4.1版本推出的Project **Butter**。Project Butter对Android Display系统进行了重构，引入三个核心元素：**VSYNC、Triple Buffer和Choreographer**。其中，VSYNC是理解Project Buffer的核心。VSYNC是Vertical Synchronization（垂直同步）的缩写，是一种在PC上已经**很早就广泛使用的技术**，读者可简单地把它认为是一种定时中断。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。接下来，本文将围绕VSYNC来介绍Android Display系统的工作方式。

在讲解刷新机制之前，先介绍几个名词以及VSYNC和Choreographer主要功能及工作方式。

·双缓冲：显示内容的数据内存，为什么要用双缓冲，我们知道在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，但这样会带来一个问题，如果上一帧的数据还没有显示完，Framebuffer中的数据又更新了，就会带来**残影的问题，给用户直观的感觉就会有闪烁感**，所以普遍采用了双缓冲技术。双缓冲意味着要使用两个缓冲区（在SharedBufferStack中），其中一个称为Front Buffer，另外一个称为Back Buffer。**UI总是先在Back Buffer中绘制**，后台绘制好，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\_ctrl来通知显示设备切换Buffer。

·VSYNC：**从前面的双缓冲介绍中可以了解到，只有当另一个buffer准备好后，才能通知刷新，这就需要CPU以主动查询的方式来保证数据是否准备好，因为这种机制效率很低**，**所以引入了VSYNC**。VSYNC是Vertical Synchronization（垂直同步）的缩写，可以简单地把它认为是一种定时中断，一旦收到VSYNC中断，CPU就开始处理各帧数据。

Choreographer：收到VSYNC信号时，调用用户设置的回调函数。一共有以下三种类型的回调：

·CALLBACK\_INPUT：优先级最高，与输入事件有关。

·CALLBACK\_ANIMATION：第二优先级，与动画有关。

·CALLBACK\_TRAVERSAL：最低优先级，与UI控件绘制有关。

接下来通过时序图来分析刷新的过程，这些时序图是Google在2012 Google I/O讲解新的显示系统提供的，图2-7所示的时序图有三个元素：Display（显示设备），CPU-CPU准备数据，GPU-GPU准备数据。最下面的时间为显示时间，根据理想的60FPS，以16ms为一个显示周期。



图2-7　没有VSync信息的刷新

（1）没有VSync信号同步

我们以16ms为单位来进行分析：

1）从第一个16ms开始看，Display显示第0帧，CPU处理完第一帧后，GPU紧接其后处理继续第一帧。三者都在正常工作。

2）时间进入第二个16ms：因为在上一个16ms时间内，第1帧已经由CPU、GPU处理完毕。所以Display可以正常显示第1帧。显示没有问题，但在本16ms期间，CPU和GPU并未及时绘制第2帧数据（前面的空白区在忙别事情去了），而是在本周期快结束时，CPU/GPU才去处理第2帧数据。

3）时间进入第3个16ms，此时Display应该显示第2帧数据，但由于CPU和GPU还没有处理完第2帧数据，故Display只能继续显示第一帧的数据，结果使得第1帧多画了一次（对应时间段上标注了一个Jank），这就导致错过了显示第二帧。

通过上述分析可知，在第二个16ms时，发生Jank的关键问题在于，为何在第1个16ms段内，CPU/GPU没有及时处理第2帧数据？从第二个16ms开始有一段空白的时间，可以说明原因所在，**那就是CPU可能是在忙别的事情，不知道该到处理UI绘制的时间了**。可CPU一旦想起来要去处理第2帧数据，时间又错过了。为解决这个问题，4.1版本推出了Project Butter，核心目的就是解决刷新不同步的问题。

（2）有VSync信号同步

加入VSync后，从图2-8可以看到，一旦收到VSync中断，CPU就开始处理各帧的数据。大部分的Android显示设备刷新率是60Hz（图2-7的时间轴也是60ms），这也就意味着每一帧最多只能有1/60=16ms左右的准备时间。假如CPU/GPU的FPS高于这个值，显示效果将更好。但是，这时又出现了一个新问题：CPU和GPU处理数据的速度都能在16ms内完成，而且还有时间空余，但必须等到VSYNC信号到来后，才处理下一帧数据，因此CPU/GPU的FPS被拉低到与Display的FPS相同。

从图2-9采用双缓冲区的显示效果来看：在双缓冲下，CPU/GPU FPS大于刷新频率同时采用了双缓冲技术以及VSync，可以看到整个过程还是相当不错的，虽然CPU/GPU处理所用的时间时短时长，但总体来说都在16ms以内，因而不影响显示效果。A和B分别代表两个缓冲区，它们不断交换来正确显示画面。但如果CPU/GPU的FPS小于Display的FPS，情况又不同了，如图2-10所示。



图2-8　有VSync的绘制



图2-9　双缓冲下的时序图（CPU的第三个A->B）



图2-10　双缓冲下CPU/GPU FPS小于刷新频率时序图

从图2-10可以看到，当CPU/GPU的处理时间超过16ms时，第一个VSync就已经到来，但缓冲区B中的数据却还没有准备好，这样就只能继续显示之前A缓冲区中的内容。而后面B完成后，又因为还没有VSync信号，CPU/GPU这个时候只能等待下一个VSync的来临才开始处理下一帧数据。因此在整个过程中，有一大段时间被浪费。总结这段话就是：

1）在第二个16ms时间段内，Display本应显示B帧，但因为GPU还在处理B帧，导致A帧被重复显示。

2）同理，在第二个16ms时间段内，CPU无所事事，因为A Buffer由Display在使用。B Buffer由GPU使用。注意，一旦过了VSYNC时间点，CPU就不能被触发以及处理绘制工作了。

为什么CPU不能在第二个16ms处即VSync到来就开始工作呢？很明显，原因就是只有两个Buffer。如果有第三个Buffer存在，CPU就可以开始工作，而不至于空闲。于是在Andoird 4.1以后，引出了第三个缓冲区：Triple Buffer。Triple Buffer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　在大部分情况下，只使用到双缓存，只有在需要时，才会用三缓冲来增强，这时可以把输入的延迟降到最少，保持画面的流畅。

引入Triple Buffer后的刷新时序如图2-11所示。



图2-11　使用Triple Buffer时序图

在第二个16ms时间段，CPU使用C Buffer绘图。虽然还是会多显示一次A帧，但后续显示就比较顺畅了。是不是Buffer越多越好呢？回答是否定的。由图2-11可知，在第二个时间段内，CPU绘制的第C帧数据要到第四个16ms才能显示，这比双缓存情况多了16ms延迟。所以缓冲区不是越多越好，要做到平衡到最佳效果。

从以上的分析来看，Android系统在显示机制上解决了Android UI不流畅的问题，并且从Google I/O2012给出的视频来看，其效果也达到预期。但实际在应用开发过程中仍然存在卡顿的现象。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。

### 卡顿的根本原因

那卡顿的根本原因是什么呢，从Android系统的显示原理中可以看到，影响绘制的根本原因有以下两方面：

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

耗时太长，需要从UI布局和绘制上来具体分析，详见后面内容。这里我们主要讨论下第二个方面。我们知道所有的绘制工作都是由主线程，也就是UI线程来负责，主线程的关键职责是处理用户交互，在屏幕上绘制像素，并进行加载显示相关的数据。在Android应用开发中，特别需要避免任何阻碍主线程的事情，这样应用程序才能保持对用户操作的即时响应。

在实际的开发过程中，我们需要知道主线程应该做什么，总结起来主线程主要做以下几个方面的工作：

* ·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

除了这些以外，尽量避免将其他处理放到主线程中，特别是复杂的数据计算和网络请求。

## 性能分析工具

从前一节可以看到，Android系统在4.1以后从框架上解决了由于系统问题导致的卡顿现象，但在实际的使用过程中，在用户的感受上，卡顿仍然是应用开发中主要面临的问题，而原因从上一节的分析中也知道本质是VSync信号到来时，不能及时处理绘制事件导致，本节先抛出以下两个问题：

1）应用层做了什么会导致VSync事件不能及时处理？

2）卡顿能监控吗？

性能问题并不容易复现，也不好定位，光从几个场景不能完全覆盖所有的问题，因此在做性能优化时，最直接有效的方法，就是尽量复现存在性能问题的场景，并监控此过程中程序的执行流程，如果能够方便地分析程序中函数的调用关系和执行时间，自然也就很容易找出性能瓶颈。

分析问题和确认问题是否解决，都借助了相应的调试工具，比如查看Layout层次的次的Hierarchy View、Android系统上带的GPU Profile工具和静态代码检查工具Lint等。这些工具对性能优化都起到非常重要的作用。本节将介绍这些工具和另外两个性能优化非常重要的工具：**TraceView**和**Systrace**。这两个工具除了在UI上，对于在后面将要讲到的启动优化、动画优化等上都是很重要的工具，可以说大部分的性能分析都离不开这几个工具，接下来学习几个常用的与流畅度优化相关的工具的使用方法，在后面实际的优化方案中也会介绍其他辅助工具。

### 卡顿检测工具

要做性能优化，就非常有必要借助于一系列辅助工具，Android提供了多个开发辅助工具，在性能调优过程中非常重要，下面介绍几个常用的工具，在后面的具体优化过程中会多次使用到。

我们已经知道，从应用层绘制一个页面（View），主要有三个过程：CPU准备数据→GPU从数据缓存列表获取数据→Display设备绘制，这三个过程的耗时可以通过一个手机开发辅助工具查看：Profile GPU Rendering。Profile GPU Rendering是Android 4.1系统开始提供的一个开发辅助功能，在设置中打开开发者选项，如图2-12所示。

Profile GPU Rendering功能特点如下：

* + - ·它是一个图形监测工具，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（单位为ms）。
    - 随着时间推移，从左到右的刷新呈现。
    - ·提供了一个标准的耗时，如果高于标准耗时，表示当前这一帧丢失。

注意：如果设置中没有开发者选项，可以通过设置页面中的“关于”选项，单击版本号七次即可打开开发者选项，在后面的章节中还会使用到开发者选项中的其他辅助工具。

打开Profile GPU Rendering后可以看到实时刷新的彩色图，如图2-13所示。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：

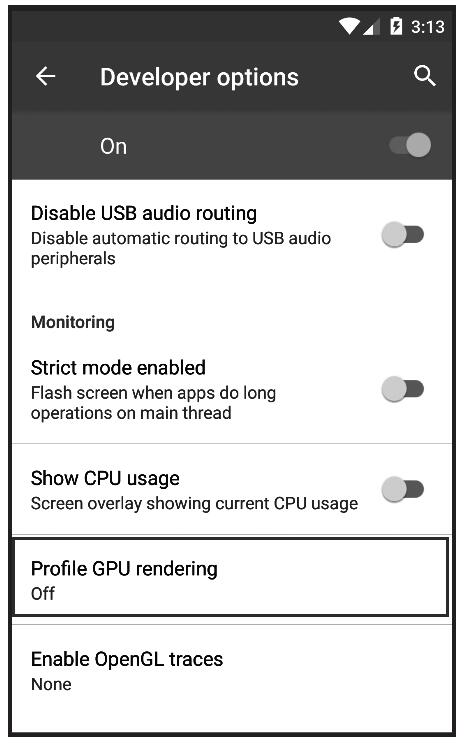


图2-12　打开Profile GPU rendering

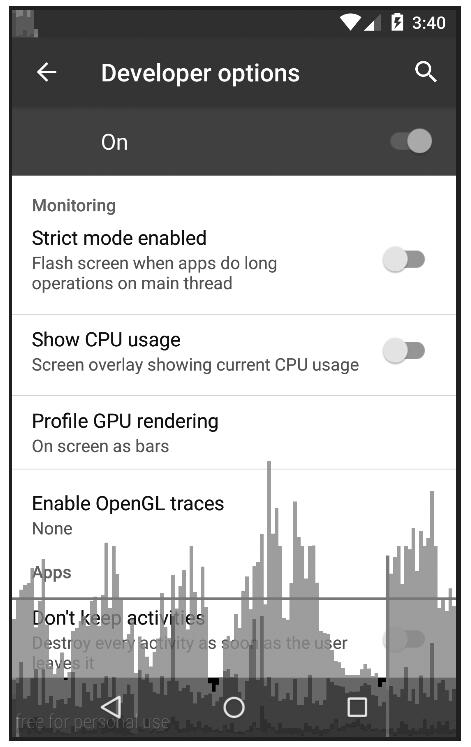


图2-13　Profile GPU rendering

每一条柱状图都由4种颜色组成：红、黄、蓝、紫，这些线对应每一帧在不同阶段的实际耗时。

·蓝色代表测量绘制的时间，它代表需要多长时间去创建和更新DisplayList。在Android中，一个视图在进行渲染之前，它必须被转换成GPU熟悉的格式，简单来说就是几条绘图命令，蓝色就是记录了在屏幕上更新视图需要花费的时间，也可以理解为执行每一个View的onDraw方法，创建或者更新每一个View的Display List对象。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。

红色代表执行的时间，这部分是Android进行2D渲染Display List的时间，为了绘制到屏幕上，Android需要使用OpenGl ES的API接口来绘制Display List，这些API有效地将数据发送到GPU，最终在屏幕上显示出来。当红色的线非常高时，可能是由重新提交了视图而导致的。

橙色部分表示处理时间，或者是CPU告诉GPU渲染一帧的地方，这是一个阻塞调用，因为CPU会一直等待GPU发出接到命令的回复，如果柱状图很高，就意味着GPU太繁忙了。

紫色段表示将资源转移到渲染线程的时间，只有Android 4.0及以上版本才会提供。

任何时候超过绿线（警戒线，对应时长16ms），就有可能丢失一帧的内容，虽然对于大部分应用来说，丢失几帧确实感觉不出卡顿，但保持UI流畅的关键就在于让这些垂直的柱状条尽可能地保持在绿线下面。

GPU Profile工具能够很好地帮助你找到渲染相关的问题，但是要修复这些问题就不是那么简单了。需要结合另一个耗时工具和代码来具体分析，找到性能的瓶颈，并进行优化。在GPU Profile Render发现有问题的页面后，可以通过另外一个工具Hierarchy Viewer来查看页面的布局层次和每个View所花的时间，在后面的布局优化章节，将通过实例来讲解和学习使用方法。

综上，一般关注蓝色、红色和绿线就好。

[图片]技巧　在实际开发中，从图上虽然可以看到绘制的时间，但对不便于进行数据分析，比如进入某一个页面，柱形图虽然实时绘制出来，但不能更好地分析，这里可以通过：adb shell dumpsys gfxinfo com.\*\*.\*\*（包名）把具体的耗时输出到日志中来分析。

|  |
| --- |
| …  Total frames rendered: 1700  Janky frames: 555 (32.65%)  90th percentile: 26ms  95th percentile: 40ms  99th percentile: 81ms  Number Missed Vsync: 86  Number High input latency: 1  Number Slow UI thread: 201  Number Slow bitmap uploads: 18  Number Slow issue draw commands: 372  … |

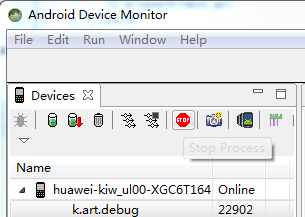
### TraceView

TraceView是AndroidSDK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Framework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说明，可以分析到应用具体每一个方法的执行时间，使用可以非常直观简单，分析性能问题很方便。

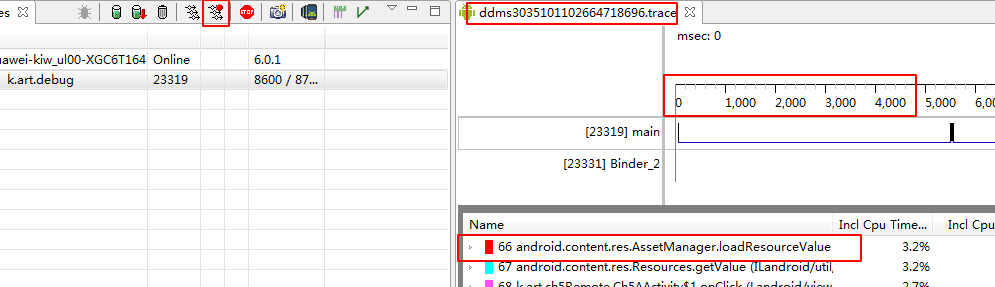
#### 使用方法

在使用TraceVeiw分析问题之前需要得到一个\*.trace的文件，然后通过TraceView来分析trace文件的信息，trace文件的获取有两种方式：

1. 在DDMS中使用
2. 连接设备。
3. 打开应用。
4. 打开DDMS（若在Android Studio中则先打开Android Device Monitor）。
5. 单击Strart Method Profiling按钮，如图2-14所示。



1. 在应用中操作需要监控的点，比如进入一个Activity或者滑动一个列表，完成后单击Stop Method Profiling按钮，如图2-15所示。



6）结束会自动跳转到TraceView视图。

这种方法使用方便，但监控范围不够精确，如果需要精确监控某一个路径，就需要使用下一个方法：在代码中加入调试语句保存Trace文件。

1. **代码中加入调试语句保存trace文件**

有时在开发过程中不好复现的问题，需要在关键的路径上获取TraceView数据，在测试时复现此问题后直接拿到Trace文件查看对应的数据。这时可以在代码中使用TraceView工具并生成对应的trace文件。在android.os.Debug类中提供了相应的方法，过程很简单步骤如下：

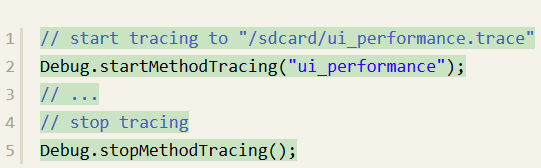
1）在需要开始监控的地方调用startMethodTracing（）。

2）在需要结束监控的地方调用stopMethodTracing（）。

3）系统会在SD卡中创建<trace-name>.trace文件。

4）使用traceveiw打开该文件进行分析。

调用代码如下：



在应用程序中调用startMethodTracing（）时，系统会在指定的路径上创建一个名为<trace\_filename>.trace文件。这个文件包含了方法名跟踪数据，以及与线程和方法名的映射表。然后系统开始缓存应用产生的跟踪数据，直到应用程序调用stopMethodTracing（）结束，此时将其缓冲的数据写入输出文件中。如果系统达到最大缓存大小时，还没有调用stopMethodTracing（），系统会停止跟踪并发送一个通知。

在Android 4.4及更高版本中，可以通过基于采样的方法分析耗时情况，因为减少了分析Trace文件的次数，降低了IO读写，所以可以减少Trace工具在运行时对性能的影响，同时对分析结果也不会有很大的偏差。通过调用startMethodTracing（）方法，就可以指定具体的采样间隔，定期采集样本数据分析。

注意　在代码中使用此方法保存TraceView数据，不要忘记在应用中打开write to external storage权限（WRITE\_EXTERNAL\_STORAGE）。

#### TraceView视图说明

Traceview视图分两部分，上半部分为时间片面板（Timeline Panel），下半部分为分析面板（Profile Panel）。时间片面板如图2-16所示。

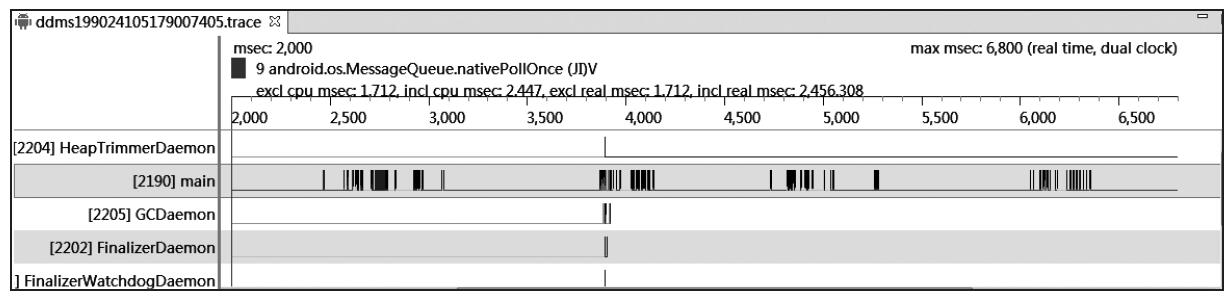
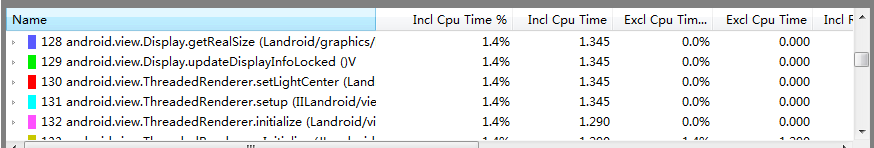


图2-16　时间片面板

X轴表示时间消耗，单位为毫秒（ms），Y轴表示各个线程，每个线程中的不同方法使用了不同的颜色来表示，颜色占用面积越宽，表示该方法占用CPU时间越长。

·时间片面板可以放大/缩小，也可以指定区域放到最大，方便查看具体的过程，一般优先选择放大耗时严重的区域。

分析面板（Profile Panel）如图2-17所示。



分析面板看起来并不复杂，但需要理解各列数据的意义，每一列表示的意义如表2-1所示。

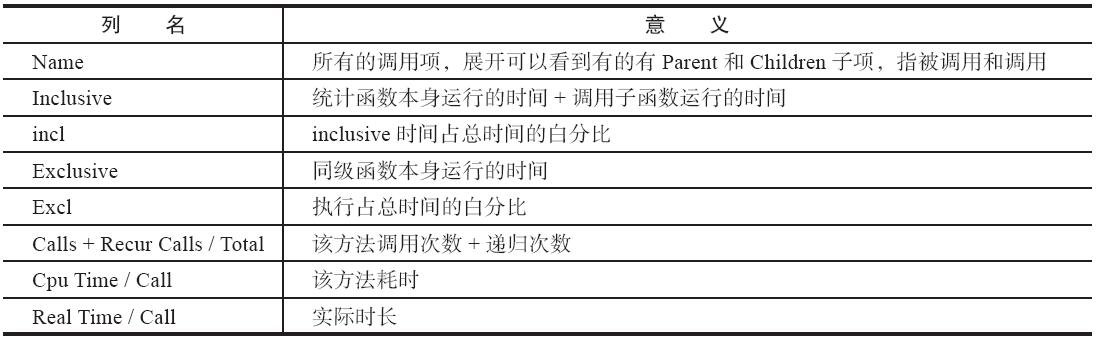


表2-1　分析面板参数意义

使用TraceView查看耗时，主要关注**Calls+Recur Calls/Total**和**Cpu Time/Call**这两个值，也就是关注调用次数多和耗时久的方法，然后优化这些方法的逻辑和调用次数，减少耗时。

[图片]注意　RealTime与cputime区别为：因为RealTime包括了CPU的上下文切换、阻塞、GC等，所以RealTime方法的实际执行时间要比CPU Time稍微长一点。

### Systrace UI性能分析

在应用程序开发过程中，UI（用户界面）的流畅度是体验的核心，特别是在动画、跳转或者列表的滑动过程中，出现卡顿和无响应是非常影响用户体验的，要解决这些问题，首先要找到问题的原因，前面介绍的TraceView是分析性能的一款利器，下面再介绍一个分析应用程序UI性能的工具：Systrace。

Systrace是Android 4.1及以上版本提供的性能数据采样和分析工具。它可以帮助开发者收集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，从而帮助开发者更直观地分析系统瓶颈，改进性能。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。Systrace工具可以跟踪、收集、检查定时信息，可以很直观地查看CPU周期消耗的具体时间，显示每个线程和进程的跟踪信息，使用不同颜色来突出问题的严重性，并提供如何解决这些问题的建议。

[图片]注意　由于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，要进一步获取CPU满负荷运行的原因，就需要使用前面介绍过的工具Traceview。

#### Systrace使用方法

Systrace的使用不复杂。但跟踪的设备必须是Android 4.1（API16）或更高版本。在4.3版本和4.3以前版本的使用上有些区别，后面会讲到。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　4.3以前系统版本的设备需要打开Settings>Developer options>Monitoring>Enable traces。

1. 在DDMS上使用

在Eclipse和Android Studio中都可以在DDMS直接使用Systrace，其他IDE也能支持，且流程都相同，下面以Android Studio为例说明其使用流程。

1）打开Android Device Monitor，连接手机并准备需要抓取的界面。

2）单击Systrace按钮进入抓取前的设置，选择需要跟踪的内容（见图2-18）：

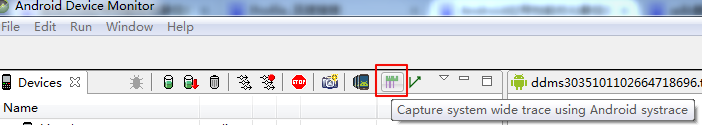


图2-18　DDMS上打开Systrace

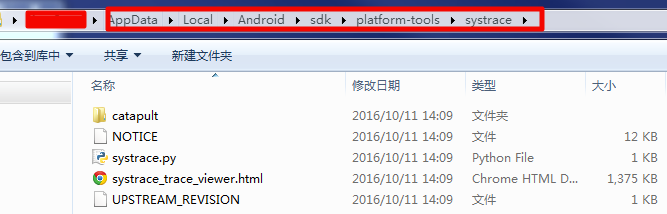
3）手机上开始操作需要跟踪的过程（如滑动列表）。

4）到了设定好的时间后，生成Trace文件。

5）使用Chrome打开文件即可分析。

（2）使用命令行

使用命令行方式更灵活，速度更快，并且配置好后再使用能快速得到结果，在Android 4.3及更高版本的设备上使用Systrace时，可以省略设置跟踪类别标签来获取默认值，或者可以手动列入指定标签。路径和命令如下：



[代码]

# QA

* 阅读系统显示源码SurfaceFlinger
* VSYNC的成功，告诉我们不同技术的融合能取得很好的效果
* 双缓冲技术为甚么还有卡顿情况
* CPU以主动查询的方式为啥低效
* VSYNC的中断机制为何效率就高了？
* TRAVERSAL为何叫与UI控件绘制有关
* 60的帧率是怎么得到的，肉眼生物特性？显存的工作频率？

# 启发

* 硬件加速（GPU）至少需要8MB内存，导致系统奔溃？
* 16ms，如果某个操作花费的时间是24ms，丢帧：可能是你的Layout太过复杂，无法在16ms内完成渲染；可能是UI上有层叠太多的绘制单元；还有可能是动画执行的次数过多
* 绘制任务太重;主线程太忙了
* 优化FPV: 使用TraceView查看耗时，主要关注Calls+Recur Calls/Total和Cpu Time/Call这两个值，也就是关注调用次数多和耗时久的方法，然后优化这些方法的逻辑和调用次数，减少耗时。

# task

实践用命令/代码方式取得trace，并分析