## 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。

[Android系统Surface机制的SurfaceFlinger服务简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/8010977)