# TODO

节奏大师哦

<https://www.jianshu.com/p/47c866f6fb67>

<https://blog.csdn.net/xlnaan/article/details/87076182>

视频画面帧的展示控件SurfaceView及TextureView对比

<https://blog.csdn.net/prike/article/details/49909665/>

https://www.jianshu.com/p/b9a1e66e95ea

同步190601的sf部分（d2home）

SetText() 原理

OPENGL ES 3.0编程指南(原书第2版).pdf

* Android SurfaceFlinger 学习之路(一)----Android图形显示之HAL层Gralloc模块实现  
  [http://windrunnerlihuan.com/2017/03/12/Android-SurfaceFlinger-%E5%AD%A6%E4%B9%A0%E4%B9%8B%E8%B7%AF-%E4%B8%80-Android%E5%9B%BE%E5%BD%A2%E6%98%BE%E7%A4%BA%E4%B9%8BHAL%E5%B1%82Gralloc%E6%A8%A1%E5%9D%97%E5%AE%9E%E7%8E%B0/](javascript:%20void%200)

需要内核的支持啊  
  
Android SurfaceFlinger 学习之路(二)----SurfaceFlinger概述  
[http://windrunnerlihuan.com/2017/04/27/Android-SurfaceFlinger-%E5%AD%A6%E4%B9%A0%E4%B9%8B%E8%B7%AF-%E4%BA%8C-SurfaceFlinger%E6%A6%82%E8%BF%B0/](javascript:%20void%200)  
  
Android SurfaceFlinger 学习之路(三)----Android开机动画流程简述  
[http://windrunnerlihuan.com/2017/05/02/Android-SurfaceFlinger-%E5%AD%A6%E4%B9%A0%E4%B9%8B%E8%B7%AF-%E4%B8%89-Android%E5%BC%80%E6%9C%BA%E5%8A%A8%E7%94%BB%E6%B5%81%E7%A8%8B%E7%AE%80%E8%BF%B0/](javascript:%20void%200)  
  
Android SurfaceFlinger 学习之路(四)----SurfaceFlinger服务的启动与连接过程  
[http://windrunnerlihuan.com/2017/05/13/Android-SurfaceFlinger-%E5%AD%A6%E4%B9%A0%E4%B9%8B%E8%B7%AF-%E5%9B%9B-SurfaceFlinger%E6%9C%8D%E5%8A%A1%E7%9A%84%E5%90%AF%E5%8A%A8%E4%B8%8E%E8%BF%9E%E6%8E%A5%E8%BF%87%E7%A8%8B/](javascript:%20void%200)  
  
Android SurfaceFlinger 学习之路(五)----VSync 工作原理  
[http://windrunnerlihuan.com/2017/05/25/Android-SurfaceFlinger-%E5%AD%A6%E4%B9%A0%E4%B9%8B%E8%B7%AF-%E4%BA%94-VSync-%E5%B7%A5%E4%BD%9C%E5%8E%9F%E7%90%86/](javascript:%20void%200)  
  
Android SurfaceFlinger 学习之路(六)----SurfaceFlinger创建Surface  
[http://windrunnerlihuan.com/2017/06/17/Android-SurfaceFlinger-%E5%AD%A6%E4%B9%A0%E4%B9%8B%E8%B7%AF-%E5%85%AD-SurfaceFlinger%E5%88%9B%E5%BB%BASurface/](javascript:%20void%200)

* Todd Zhu

Android SurfaceFlinger 学习之路(七)----创建图形缓冲区GraphicBuffer  
[http://windrunnerlihuan.com/2017/06/22/Android-SurfaceFlinger-%E5%AD%A6%E4%B9%A0%E4%B9%8B%E8%B7%AF-%E4%B8%83-%E5%88%9B%E5%BB%BA%E5%9B%BE%E5%BD%A2%E7%BC%93%E5%86%B2%E5%8C%BAGraphicBuffer/](javascript:%20void%200)  
  
Android SurfaceFlinger 学习之路(八)----Surface管理图形缓冲区  
[http://windrunnerlihuan.com/2017/07/03/Android-SurfaceFlinger-%E5%AD%A6%E4%B9%A0%E4%B9%8B%E8%B7%AF-%E5%85%AB-Surface%E7%AE%A1%E7%90%86%E5%9B%BE%E5%BD%A2%E7%BC%93%E5%86%B2%E5%8C%BA/](javascript:%20void%200)  
  
Android SurfaceFlinger 学习之路(九)----SurfaceFlinger事务处理  
[http://windrunnerlihuan.com/2017/10/24/Android-SurfaceFlinger-%E5%AD%A6%E4%B9%A0%E4%B9%8B%E8%B7%AF-%E4%B9%9D-SurfaceFlinger%E4%BA%8B%E5%8A%A1%E5%A4%84%E7%90%86/](javascript:%20void%200)  
  
Android SurfaceFlinger 学习之路(十)----SurfaceFlinger处理Layer更新  
[http://windrunnerlihuan.com/2017/11/09/Android-SurfaceFlinger-%E5%AD%A6%E4%B9%A0%E4%B9%8B%E8%B7%AF-%E5%8D%81-SurfaceFlinger%E5%A4%84%E7%90%86Layer%E6%9B%B4%E6%96%B0/](javascript:%20void%200)  
  
Android SurfaceFlinger 学习之路(十一)----合成Layer之准备合成  
[http://windrunnerlihuan.com/2017/12/01/Android-SurfaceFlinger-%E5%AD%A6%E4%B9%A0%E4%B9%8B%E8%B7%AF-%E5%8D%81%E4%B8%80-%E5%90%88%E6%88%90Layer%E4%B9%8B%E5%87%86%E5%A4%87%E5%90%88%E6%88%90/](javascript:%20void%200)  
  
Android SurfaceFlinger 学习之路(十一)(PostView)--附：硬件HWC选取合成类型（Intel）  
[http://windrunnerlihuan.com/2017/12/19/Android-SurfaceFlinger-%E5%AD%A6%E4%B9%A0%E4%B9%8B%E8%B7%AF-%E5%8D%81%E4%B8%80-PostView-%E9%99%84%EF%BC%9A%E7%A1%AC%E4%BB%B6HWC%E9%80%89%E5%8F%96%E5%90%88%E6%88%90%E7%B1%BB%E5%9E%8B%EF%BC%88Intel%EF%BC%89/](javascript:%20void%200)  
  
Android SurfaceFlinger 学习之路(十二)----合成Layer  
[http://windrunnerlihuan.com/2018/01/09/Android-SurfaceFlinger-%E5%AD%A6%E4%B9%A0%E4%B9%8B%E8%B7%AF-%E5%8D%81%E4%BA%8C-%E5%90%88%E6%88%90Layer/](javascript:%20void%200)

* Android多媒体开发(一)----MediaPlayer框架开始  
  [http://windrunnerlihuan.com/2016/11/28/Android%E5%A4%9A%E5%AA%92%E4%BD%93%E5%BC%80%E5%8F%91-%E4%B8%80-MediaPlayer%E6%A1%86%E6%9E%B6%E5%BC%80%E5%A7%8B/](javascript:%20void%200)  
    
  Android多媒体开发(二)----MediaPlayer的C/S架构以及C++层调用步骤  
  [http://windrunnerlihuan.com/2016/11/30/Android%E5%A4%9A%E5%AA%92%E4%BD%93%E5%BC%80%E5%8F%91-%E4%BA%8C-MediaPlayer%E7%9A%84C-S%E6%9E%B6%E6%9E%84%E4%BB%A5%E5%8F%8AC-%E5%B1%82%E8%B0%83%E7%94%A8%E6%AD%A5%E9%AA%A4/](javascript:%20void%200)  
    
  Android多媒体开发(三)----从StageFright到AwesomePlayer  
  [http://windrunnerlihuan.com/2016/12/11/Android%E5%A4%9A%E5%AA%92%E4%BD%93%E5%BC%80%E5%8F%91-%E4%B8%89-%E4%BB%8EStageFright%E5%88%B0AwesomePlayer/](javascript:%20void%200)  
    
  Android多媒体开发(四)----AwesomePlayer数据源处理  
  [http://windrunnerlihuan.com/2016/12/12/Android%E5%A4%9A%E5%AA%92%E4%BD%93%E5%BC%80%E5%8F%91-%E5%9B%9B-AwesomePlayer%E6%95%B0%E6%8D%AE%E6%BA%90%E5%A4%84%E7%90%86/](javascript:%20void%200)  
    
  Android多媒体开发(五)----OpenMax简介  
  [http://windrunnerlihuan.com/2016/12/15/Android%E5%A4%9A%E5%AA%92%E4%BD%93%E5%BC%80%E5%8F%91-%E4%BA%94-OpenMax%E7%AE%80%E4%BB%8B/](javascript:%20void%200)  
    
  Android多媒体开发(六)----Android中OpenMax的实现(preview)  
  [http://windrunnerlihuan.com/2016/12/26/Android%E5%A4%9A%E5%AA%92%E4%BD%93%E5%BC%80%E5%8F%91-%E5%85%AD-Android%E4%B8%ADOpenMax%E7%9A%84%E5%AE%9E%E7%8E%B0-preview/](javascript:%20void%200)

Android多媒体开发(七)----Android中OpenMax的实现  
[http://windrunnerlihuan.com/2016/12/29/Android%E5%A4%9A%E5%AA%92%E4%BD%93%E5%BC%80%E5%8F%91-%E4%B8%83-Android%E4%B8%ADOpenMax%E7%9A%84%E5%AE%9E%E7%8E%B0/](javascript:%20void%200)  
  
Android多媒体开发(八)----播放流程  
[http://windrunnerlihuan.com/2017/01/06/Android%E5%A4%9A%E5%AA%92%E4%BD%93%E5%BC%80%E5%8F%91-%E5%85%AB-%E6%92%AD%E6%94%BE%E6%B5%81%E7%A8%8B/](javascript:%20void%200)  
  
Android多媒体开发(九)----Video Buffer传输与Audio Playback流程  
[http://windrunnerlihuan.com/2017/01/13/Android%E5%A4%9A%E5%AA%92%E4%BD%93%E5%BC%80%E5%8F%91-%E4%B9%9D-Video-Buffer%E4%BC%A0%E8%BE%93%E4%B8%8EAudio-Playback%E6%B5%81%E7%A8%8B/](javascript:%20void%200)

# 基本原理

## 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（开机动画）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。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。

# SurfaceFlinger

关键字：SurfaceFlinger

## 概述

SurfaceFlinger服务负责绘制Android应用程序的UI，它的实现相当复杂，要从正面分析它的实现不是一件容易的事。既然不能从正面分析，我们就想办法从侧面分析。

说到底，无论SurfaceFlinger服务有多复杂，它都是为android应用程序服务的，因此，我们就从Android应用程序与SurfaceFlinger服务的关系入手，来概述和制定SurfaceFlinger服务的学习计划。

SurfaceFlinger服务运行在Android系统的System进程中，它负责管理Android系统的帧缓冲区（Frame Buffer）。Android系统的帧缓冲区的相关知识，可以参考前面两篇文章Android系统的开机画面显示过程分析和Android帧缓冲区（Frame Buffer）硬件抽象层（HAL）模块Gralloc的实现原理分析。Android应用程序为了能够将自己的UI绘制在系统的帧缓冲区上，它们就必须要与SurfaceFlinger服务进行通信，如图1所示



图1 Android应用程序与SurfaceFlinger服务的关系

注意，Android应用程序与SurfaceFlinger服务是运行在不同的进程中的，因此，它们采用Binder进程间通信机制来进行通信。Android系统的Binder进程间通信机制的相关知识，可以参考Android进程间通信（IPC）机制Binder简要介绍和学习计划这一系列的文章。

在图1中，每一个Android应用程序与SurfaceFlinger服务都有一个连接，这个连接都是通过一个类型为Client的Binder对象来描述的。这些Client对象是Android应用程序连接到SurfaceFlinger服务的时候由SurfaceFlinger服务创建的，而当Android应用程序成功连接到SurfaceFlinger服务之后，就可以获得一个对应的Client对象的Binder代理接口了。有了这些Binder代理接口之后，Android应用程序就可以通知SurfaceFlinger服务来绘制自己的UI了。

Android应用程序在通知SurfaceFlinger服务来绘制自己的UI的时候，需要将UI元数据传递给SurfaceFlinger服务，例如，要绘制UI的区域、位置等信息。一个Android应用程序可能会有很多个窗口，而每一个窗口都有自己的UI元数据，因此，Android应用程序需要传递给SurfaceFlinger服务的UI元数据是相当可观的。在这种情况下，通过Binder进程间通信机制来在Android应用程序与SurfaceFlinger服务之间传递UI元数据是不合适的，这时候Android系统的匿名共享内存机制（Anonymous Shared Memory）就派上用场了。Android系统的匿名共享内存机制的相关知识，可以参考Android系统匿名共享内存Ashmem（Anonymous Shared Memory）简要介绍和学习计划这一系列的文章。

在每一个Android应用程序与SurfaceFlinger服务之间的连接上加上一块用来传递UI元数据的匿名共享内存，我们就得到了图2，如下所示：



图2 用来在Android应用程序与SurfaceFlinger服务之间传递UI元数据的匿名共享内存

在Application和Client这两个高富帅看来，它们之间的原生匿名共享内存块就一个活脱脱的土肥圆。因此，Application和Client是看不上这块原生的匿名共享内存的。于是，这块原生的匿名共享内存当时就怒了，立志要逆袭变成白富美，如图3所示：



图3 结构化后的用来传递UI元数据的匿名共享内存块

土肥圆逆袭后，就变成了一个名字为SharedClient的白富美，从此，它就和Application、Client过上幸福的啪啪啪生活了。 SharedClient到底有多白多富多美？参见图4：



图4 用来描述Android应用程序的UI元数据的SharedClient

在每一个SharedClient里面，有至多31个SharedBufferStack。字面上来看，SharedBufferStack就是共享缓冲区堆栈。怎么理解呢？首先，Shared表明这个堆栈共享的。那么由谁来共享呢？当然就是Android应用程序和SurfaceFlinger服务了。其次，Buffer表明这个堆栈的内容是缓冲区。什么样的缓冲区呢？当然就是用来描述UI元数据的缓冲区了。再者，Stack表明用来描述UI元数据的缓冲区是需要按照一定的规则来访问的。综合起来，我们就可以认为每一个SharedBufferStack就是用来描述一系列需要按照一定规则来访问的缓冲区。

好像还是不能理解SharedBufferStack？好吧，回忆一下，一般我们就绘制UI的时候，都会采用一种称为“双缓冲”的技术。双缓冲意味着要使用两个缓冲区，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，然后再和Front Buffer交换，渲染到显示设备中。这下就可以理解SharedBufferStack的含义了吧？SurfaceFlinger服务只不过是将传统的“双缓冲”技术升华和抽象为了一个SharedBufferStack。可别小看了这个升华和抽象，有了SharedBufferStack之后，SurfaceFlinger服务就可以使用N个缓冲区技术来绘制UI了。N值的取值范围为2到16。例如，在Android 2.3中，N的值等于2，而在Android 4.1中，据说就等于3了。

我们还可以再进一步地理解SharedBufferStack。在SurfaceFlinger服务中，每一个SharedBufferStack都对应一个Surface，即一个窗口。这样，我们就可以知道为什么每一个SharedClient里面包含的是一系列SharedBufferStack而不是单个SharedBufferStack：一个SharedClient对应一个Android应用程序，而一个Android应用程序可能包含有多个窗口，即Surface。从这里也可以看出，一个Android应用程序至多可以包含31个Surface。

SharedBufferStack长什么样子呢？看图5：



图 5 SharedBufferStack的结构示意图

在图5中，为了方便描述，我们假设图中的SharedBufferStack有5个Buffer，其中，Buffer-1和Buffer-2是已经使用了的，而Buffer-3、Buffer-4和Buffer-5是空闲的。指针head和tail分别指向空闲缓冲区列表的头部和尾部，而指针queue\_head指向已经使用了的缓冲区列表的头部。从这里就可以看出，从指针tail到head之间的Buffer即为空闲缓冲区表，而从指针head到queue\_head之间的Buffer即为已经使用了的缓冲区列表。注意，图中的5个Buffer是循环使用的。

空闲缓冲区比较好理解，接下来我们重点解释一下那些已经被使用了的缓冲区，即图5中的Buffer-1和Buffer-2。

前面我们说过，SharedBufferStack中的缓冲区只是用来描述UI元数据的，这意味着它们不包含真正的UI数据。真正的UI数据保存在GraphicBuffer中，后面我们再描述GaphicBuffer。因此，为了完整地描述一个UI，SharedBufferStack中的每一个已经使用了的缓冲区都对应有一个GraphicBuffer，用来描述真正的UI数据。当SurfaceFlinger服务缓制Buffer-1和Buffer-2的时候，就会找到与它们所对应的GraphicBuffer，这样就可以将对应的UI绘制出来了。

当Android应用程序需要更新一个Surface的时候，它就会找到与它所对应的SharedBufferStack，并且从它的空闲缓冲区列表的尾部取出一个空闲的Buffer。我们假设这个取出来的空闲Buffer的编号为index。接下来Android应用程序就请求SurfaceFlinger服务为这个编号为index的Buffer分配一个图形缓冲区GraphicBuffer。SurfaceFlinger服务分配好图形缓冲区GraphicBuffer之后，会将它的编号设置为index，然后再将这个图形缓冲区GraphicBuffer返回给Android应用程序访问。Android应用程序得到了SurfaceFlinger服务返回的图形缓冲区GraphicBuffer之后，就在里面写入UI数据。写完之后，就将与它所对应的缓冲区，即编号为index的Buffer，插入到对应的SharedBufferStack的已经使用了的缓冲区列表的头部去。这一步完成了之后，Android应用程序就通知SurfaceFlinger服务去绘制那些保存在已经使用了的缓冲区所描述的图形缓冲区GraphicBuffer了。用图5的例子来说，SurfaceFlinger服务需要绘制的是编号为1和2的Buffer所对应的图形缓冲区GraphicBuffer。由于SurfaceFlinger服务知道编号为1和2的Buffer所对应的图形缓冲区GraphicBuffer在哪里，因此，Android应用程序只需要告诉SurfaceFlinger服务要绘制的Buffer的编号就OK了。当一个已经被使用了的Buffer被绘制了之后，它就重新变成一个空闲的Buffer了。

上面描述的过程比较复杂，后面我们再用几篇文章来详细描述。

SharedBufferStack是在Android应用程序和SurfaceFlinger服务之间共享的，但是，Android应用程序和SurfaceFlinger服务使用SharedBufferStack的方式是不一样的，具体来说，就是Android应用程序关心的是它里面的空闲缓冲区列表，而SurfaceFlinger服务关心的是它里面的已经使用了的缓冲区列表。从SurfaceFlinger服务的角度来看，保存在SharedBufferStack中的已经使用了的缓冲区其实就是在排队等待渲染。

为了方便SharedBufferStack在Android应用程序和SurfaceFlinger服务中的访问，Android系统分别使用SharedBufferClient和SharedBufferServer来描述SharedBufferStack，其中，SharedBufferClient用来在Android应用程序这一侧访问SharedBufferStack的空闲缓冲区列表，而SharedBufferServer用来在SurfaceFlinger服务这一侧访问SharedBufferStack的排队缓冲区列表。

在SharedBufferClient看来，SharedBufferStack的样子如图6所示：



图6 SharedBufferClient眼中的SharedBufferStack

只要SharedBufferStack中的available的buffer的数量大于0，SharedBufferClient就会将指针tail往前移一步，并且减少available的值，以便可以获得一个空闲的Buffer。当Android应用程序往这个空闲的Buffer写入好数据之后，它就会通过SharedBufferClient来将它添加到SharedBufferStack中的排队缓冲区列表的尾部去，即指针queue\_head的下一个位置上。

在SharedBufferServer看来，SharedBufferStack的样子如图7所示：



图7 SharedBufferServer眼中的SharedBufferStack

当Android应用程序通知SurfaceFlinger服务更新UI的时候，只要对应的SharedBufferStack中的queued的缓冲区的数量大于0，SharedBufferServer就会将指针head的下一个Buffer绘制出来，并且将指针head向前移一步，以及将queued的值减1。

上面我们多次提到了图形缓冲区GraphicBuffer，它是什么东东呢？我们看图8：



图8 图形缓冲区Graphic的结构示意图

每一个GraphicBuffer内部都包含有一块用来保存UI数据的缓冲区，这块缓冲区使用一个buffer\_handle\_t对象来描述。看到buffer\_handle\_t，是不是有点眼熟？在前面Android帧缓冲区（Frame Buffer）硬件抽象层（HAL）模块Gralloc的实现原理分析一文中，我们说过，由HAL层的Gralloc模块分配的图形缓冲区的是使用一个buffer\_handle\_t对象来描述的，而由buffer\_handle\_t对象所描述的图形缓冲区要么是在系统帧缓冲区（Frame Buffer）或者匿名共享内存（Anonymous Shared Memory）中分配的。这样，我们就可以将SurfaceFlinger服务与HAL层中的Gralloc模块关联起来了。

至此，Android应用程序与SurfaceFlinger服务的关系就概述完毕了，但是我们的任务还没有完成，我们还要进一步去具体地学习它，例如：

1. Android应用程序是如何与SurfaceFlinger服务建立连接的？

2. 用来描述Android应用程序的UI元数据的SharedClient是如何创建的？

3. Android应用程序是如何请求SurfaceFlinger服务创建一个Surface的？

4. Android应用程序是如何请求SurfaceFlinger服务渲染一个Surface的？

回答了这4个问题之后，相信我们就可以对SurfaceFlinger服务有一个深刻的认识，进而可以帮助我们从正面去分析SurfaceFlinger服务的实现。后面我们将以Android系统的开机动画为例子，用4篇文章来回答这4个问题，敬请关注！

## App与SurfaceFlinger的连接过程分析

前文在描述Android应用程序和SurfaceFlinger服务的关系时提到，每一个有UI的android应用程序都需要与SurfaceFlinger服务建立一个连接，以便可以通过这个连接来请求SurfaceFlinger服务为它创建和渲染Surface。

在本文中，我们将以Android系统的开机动画应用程序为例，详细描述Android应用程序是如何与SurfaceFlinger服务建立连接的。

就好比在介绍：app要发协议，怎么引用到ProMS昂，没啥好看的。。。

### BootAnimation源码分析

路径: frameworks/base/cmds/bootanimation/

从前面Android系统的开机画面显示过程分析一文可以知道，Android系统的开机动画是主要一个BootAnimation对象来实现，这个BootAnimation对象在构造的时候，会在内部创建一个SurfaceComposerClient对象来负责

BootAnimation类如下所示：

frameworks/native/libs/gui/SurfaceComposerClient.cpp

LibGui

SurfaceComposerClient类内部，有一个类型为sp<ISurfaceComposerClient>的成员变量mClient，如图1所示：



图1 SurfaceComposerClient的结构示意图

SurfaceComposerClient类的成员变量mClient指向的实际上是一个类型为BpSurfaceComposerClient的Binder代理对象，而这个类型为BpSurfaceComposerClient的Binder代理对象引用的是一个类型为Client的Binder本地对象。在前面Android应用程序与SurfaceFlinger服务的关系概述和学习计划一文中提到，类型为Client的Binder本地对象是由SurfaceFlinger服务来负责创建的，并且运行在SurfaceFlinger服务中，用来代表使用SurfaceFlinger服务的一个客户端，即一个与UI相关的Android应用程序。

由于Client类和BpSurfaceComposerClient类分别是一个Binder本地对象类和一个Binder代理对象类，它们都是根据Android系统在应用程序框架层提供的Binder进程间通信库来实现的，它们的实现结构图分别如图2和图3所示：



图2 Client类的实现结构图



图3 BpSurfaceComposerClient类的实现结构图

在图2和图3中，涉及到了比较多的Binder进程间通信库的类，需要读者对Android系统的Binder进程间通信机制有一定的理解和认识。在前面的Android进程间通信（IPC）机制Binder简要介绍和学习计划一系列文章，我们已经学习过Android系统的Binder进程间通信机制了，这里不再详述。

图2和图3给我们最重要的信息是Client类和BpSurfaceComposerClient类均实现了类型为ISurfaceComposerClient的Binder接口。ISurfaceComposerClient接口有两个重要的成员函数getControlBlock和createSurface，它们定义在文件frameworks/base/include/surfaceflinger/ISurfaceComposerClient.h中，如下所示：

1. **class** ISurfaceComposerClient : **public** IInterface
2. {
3. **public**:
4. ......
6. **virtual** sp<IMemoryHeap> getControlBlock() **const** = 0;
7. ......
9. /\*
10. \* Requires ACCESS\_SURFACE\_FLINGER permission
11. \*/
12. **virtual** sp<ISurface> createSurface( surface\_data\_t\* data,
13. **int** pid,
14. **const** String8& name,
15. DisplayID display,
16. uint32\_t w,
17. uint32\_t h,
18. PixelFormat format,
19. uint32\_t flags) = 0;
20. ......
22. };

其中，成员函数getControlBlock用来获得由SurfaceFlinger服务创建的一块用来传递UI元数据的匿名共享内存，而成员函数createSurface用来请求SurfaceFlinger服务创建一个Surface。从前面Android应用程序与SurfaceFlinger服务的关系概述和学习计划一文可以知道，用来传递UI元数据的匿名共享内存最终会被结构化为一个SharedClient对象，这个SharedClient对象在每个应用程序进程中至多存在一个。在接下来的两篇文章中，我们再详细分析ISurfaceComposerClient接口的成员函数getControlBlock和createSurface的实现。

理解了SurfaceComposerClient、Client以及BpSurfaceComposerClient这三个类的关系之后，接下来我们就可以分析Android系统的开机动画应用程序bootanimation是如何与SurfaceFlinger服务建立连接的。

从图1可以知道，SurfaceComposerClient类继承了RefBase类，因此，当BootAnimation类在构造函数创建了一个SurfaceComposerClient对象，并且将这个对象赋值给类型为sp<SurfaceComposerClient>的智能指针mSession时，就会导致SurfaceComposerClient类的成员函数onFirstRef被调用，而SurfaceComposerClient类的成员函数onFirstRef在调用的过程中，就会在应用程序bootanimation与SurfaceFlinger服务建立一个连接，这个过程如图4所示



图4 Android应用程序与SurfaceFlinger服务的连接过程

接下来，我们就详细分析每一个步骤。

### 分解step

#### Step 1. SurfaceComposerClient::onFirstRef

SurfaceComposerClient类的成员函数onFirstRef实现在文件~~frameworks/base/libs/surfaceflinger\_client/SurfaceComposerClient.cpp文件中~~。

frameworks/native/libs/gui/SurfaceComposerClient.cpp

1. **void** SurfaceComposerClient::onFirstRef()
2. {
3. sp<ISurfaceComposer> sm(getComposerService());
4. **if** (sm != 0) {
5. sp<ISurfaceComposerClient> conn = sm->createConnection();
6. **if** (conn != 0) {
7. mClient = conn;
8. ......
9. mStatus = NO\_ERROR;
10. }
11. }
12. }

SurfaceComposerClient类的成员函数getComposerService用来获得SurfaceFlinger服务的一个代理接口，它的实现如下所示：

1. sp<ISurfaceComposer> ComposerService::getComposerService() {
2. **return** ComposerService::getInstance().mComposerService;
3. }

ComposerService类是单例模式，当我们第一次调用它的静态函数getInstance的时候，它就会在构造函数中获得SurfaceFlinger服务的一个代理接口，并且保存在它的成员变量mComposerService中，如下所示：

frameworks/native/libs/gui/SurfaceComposerClient.cpp定义

1. ComposerService::ComposerService()
2. : Singleton<ComposerService>() {
3. **const** String16 name("SurfaceFlinger");
4. **while** (getService(name, &mComposerService) != NO\_ERROR) {
5. usleep(250000);
6. }
7. ~~mServerCblkMemory = mComposerService->getCblk();~~
8. ~~mServerCblk =~~**~~static\_cast~~**~~<surface\_flinger\_cblk\_t~~**~~volatile~~**~~\*>(~~
9. ~~mServerCblkMemory->getBase());~~
10. }

在ComposerService类的构造函数中，除了会获得SurfaceFlinger服务的代理接口之外~~，还会通过这个代理接口的成员函数getCblk来获得一块匿名共享内存mServerCblkMemory。这块匿名共享内存是由SurfaceFlinger服务创建的，用来描述系统显示屏的信息，例如，显示屏的个数、大小、方向、密度等等信息。由于这些信息可以通过一个surface\_flinger\_cblk\_t对象来描述，因此，ComposerService类的构造函数最后就将前面从SurfaceFlinger服务获得的一块匿名共享内存结构化为一个surface\_flinger\_cblk\_t对象，并且保存在ComposerService类的成员变量mServerCblk中。~~

回到SurfaceComposerClient类的成员函数onFirstRef中，由于SurfaceFlinger服务实现了ISurfaceComposer接口，因此，我们可以将前面获得的SurfaceFlinger服务的代理接口赋值给一个类型为ISurfaceComposer的强指针sm，并且调用它的成员函数createConnection来请求SurfaceFlinger服务创建一个连接，即创建一个类型为Client的Binder对象，并且将这个Binder对象的一个代理接口conn返回来。SurfaceComposerClient类获得了SurfaceFlinger服务返回来的Client代理接口conn之后，就将它保存自己的成员变量mClient中，这样开机动画应用程序bootanimation后续就可以通过它来请求SurfaceFlinger创建和渲染Surface了。

至此，ComposerService::getComposerService()==mComposerService==SurfaceFlinger

接下来，我们就继续分析SurfaceFlinger服务的成员函数createConnection的实现，以便可以了解它是如何为Android应用程序创建一个连接的。

#### Step 2. SurfaceFlinger::createConnection

sp<ISurfaceComposerClient> SurfaceFlinger::createConnection()

1. {
2. sp<ISurfaceComposerClient> bclient;
3. sp<Client> client(**new** Client(**this**));
4. status\_t err = client->initCheck();
5. **if** (err == NO\_ERROR) {
6. bclient = client;
7. }
8. **return** bclient;
9. }

SurfaceFlinger类的成员函数createConnection实现在文件frameworks/base/services/surfaceflinger/SurfaceFlinger.cpp中，它的实现很简单，只是创建了一个类型为Client的Binder对象client，并且获得它的一个ISurfaceComposerClient接口，最后将这个ISurfaceComposerClient接口，即一个Client代理对象，返回给开机动画应用程序bootanimation。

接下来，我们再继续分析Client对象的创建过程,，即Client类的构造函数的实现。

#### Step 3. new Client

frameworks/native/services/surfaceflinger/Client.h

1. Client::Client(**const** sp<SurfaceFlinger>& flinger)
2. : mFlinger(flinger)
3. {
4. }

Client类成员变量mFlinger，类型为sp<SurfaceFlinger>，指向SurfaceFlinger服务，~~而后者用来生成SurfaceFlinger服务为Android应用程序所创建的每一个Surface的名称。例如，假设一个Android应用程序请求SurfaceFlinger创建了两个Surface，那么第一个Surface的名称就由数字1来描述，而第二个Surface就由数字2来描述，依次类推。从前面Android应用程序与SurfaceFlinger服务的关系概述和学习计划这篇文章可以知道，一个Android应用程序最多可以创建31个Surface。~~

回到SurfaceFlinger类的成员函数createConnection中，它将一个指向了一个Client对象的ISurfaceComposerClient接口返回到开机动画应用程序bootanimation之后，开机动画应用程序bootanimation就可以将它封装成一个类型为BpSurfaceComposerClient的Binder代理对象。

#### Step 4. return BpSurfaceComposerClient

类型为BpSurfaceComposerClient的Binder代理对象的封装过程实现在SurfaceFlinger服务的Binder代理对象类BpSurfaceComposer的成员函数createConnection中，如下所示：

1. **class** BpSurfaceComposer : **public** BpInterface<ISurfaceComposer>
2. {
3. **public**:
4. ......
6. **virtual** sp<ISurfaceComposerClient> createConnection()
7. {
8. uint32\_t n;
9. Parcel data, reply;
10. data.writeInterfaceToken(ISurfaceComposer::getInterfaceDescriptor());
11. remote()->transact(BnSurfaceComposer::CREATE\_CONNECTION, data, &reply);
12. **return** interface\_cast<ISurfaceComposerClient>(reply.readStrongBinder());
13. }
15. ......
16. }

interface\_cast是一个模板函数，它定义在framework/base/include/binder/IInterface.h文件中：

1. **template**<**typename** INTERFACE>
2. **inline** sp<INTERFACE> interface\_cast(**const** sp<IBinder>& obj)
3. {
4. **return** INTERFACE::asInterface(obj);
5. }

从这里就可以看出，当模板参数为ISurfaceComposerClient的时候，模板函数interface\_cast实际就是通过调用ISurfaceComposerClient类的静态成员函数asInterface来将参数obj所描述的一个Binder代理对象，即一个BpBinder对象，封装成一个BpSurfaceComposerClient对象。

ISurfaceComposerClient类的静态成员函数asInterface是由frameworks/base/libs/surfaceflinger\_client/ISurfaceComposerClient.cpp文件中的IMPLEMENT\_META\_INTERFACE宏来定义的，如下所示：

1. IMPLEMENT\_META\_INTERFACE(SurfaceComposerClient, "android.ui.ISurfaceComposerClient");

IMPLEMENT\_META\_INTERFACE宏展开后，得到ISurfaceComposerClient类的静态成员函数asInterface的实现如下所示：

1. android::sp<ISurfaceComposerClient> ISurfaceComposerClient::asInterface(**const** android::sp<android::IBinder>& obj)       {
2. android::sp<ISurfaceComposerClient> intr;
4. **if** (obj != NULL) {
5. intr = **static\_cast**<ISurfaceComposerClient\*>(
6. obj->queryLocalInterface(ISurfaceComposerClient::descriptor).get());
8. **if** (intr == NULL) {
9. intr = **new** BpSurfaceComposerClient(obj);
10. }
11. ｝
12. **return** intr;
13. }

参数obj是从BpSurfaceComposer类的成员函数createConnection传进来的，它指向的实际上是一个BpBinder对象。当我们调用一个BpBinder对象的成员函数queryLocalInterface时，获得的是一个NULL指针，因此，ISurfaceComposerClient类的静态成员函数asInterface最后就会将参数obj所指向的一个BpBinder对象封装成一个BpSurfaceComposerClient对象，并且返回给调用者。

BpSurfaceComposerClient对象的更具体封装过程可以参考前面浅谈Android系统进程间通信（IPC）机制Binder中的Server和Client获得Service Manager接口之路一文中所描述的BpServiceManager对象的封装过程。

至此，开机动画应用程序bootanimation就通过SurfaceComposerClient类来与SurfaceFlinger服务建立一个连接了。

在前面Android应用程序与SurfaceFlinger服务的关系概述和学习计划一文中提到，一个Android应用程序除了需要与SurfaceFlinger服务建立连接之外，还需要有一块用来传递UI元数据的匿名共享内存，即一个SharedClient对象，因此，在接下来的一篇文章中，我们就继续分析这块匿名共享内存的创建过程，敬请期待！

## 参数解析

此时framework还没有启动取来

获取手机/system/media文件夹下与animation相关的文件

## 参考

[Android应用程序与SurfaceFlinger服务的连接过程分析](http://blog.csdn.net/luoshengyang/article/details/7857163)

[[Android5.1]开机动画显示工作流程分析](http://blog.csdn.net/u010753159/article/details/51325500)

# Android图形系统概述

frameworks/native/services/surfaceflinger/

frameworks/native/libs/gui/

## 概述

Android系统中图形系统是相当复杂的，包括WindowManager，SurfaceFlinger,Open GL,GPU等模块。 其中SurfaceFlinger作为负责绘制应用UI的核心，从名字可以看出其功能是将所有Surface合成工作。不论使用什么渲染API, 所有的东西最终都是渲染到”surface”. surface代表BufferQueue的生产者端, 并且由SurfaceFlinger所消费, 这便是基本的生产者-消费者模式. Android平台所创建的Window都由surface所支持, 所有可见的surface渲染到显示设备都是通过SurfaceFlinger来完成的.

### 图形架构



图解:

1. Image Stream Producers(图形流的生产者): 可产生graphic buffers的生产者. 例如OpenGL ES, Canvas 2D, mediaserver的视频解码器.
2. Image Stream Consumers(图形流的消费者): 最场景的消费者便是SurfaceFlinger,它使用OpenGL和Hardware Composer来组合一组surfaces.
   * OpenGL ES应用能消费图形流, 比如camera app消费camera预览图形流;
   * 非OpenGL ES应用也能消费, 比如ImageReader类
3. Window Manager: 用于管理window, 这是一组view的容器. WM将手机的window元数据(包括屏幕放心,z-order等)信息发送给SurfaceFlinger,因此SurfaceFlinger 能使用这些信息来合成surfaces,并输出到显示设备.
4. Hardware Composer(硬件合成器): 这是显示子系统的硬件抽象层, SurfaceFlinger能将一些合成工作委托给Hardware Composer,从而降低来自OpenGL和GPUd的负载.
5. Gralloc: 全称为graphics memory allocator,图像内存分配器, 用于图形生产这来请求分配内存.

(参考：https://source.android.com/devices/graphics/index.html）

### SurfaceFlinger

SurfaceFlinger进程是由init进程创建的，运行在独立的SurfaceFlinger进程。Android应用程序 必须跟SurfaceFlinger进程交互，才能完成将应用UI绘制到frameBuffer(帧缓冲区)。这个交互便涉及到 进程间的通信，采用的Binder IPC方式，名为”SurfaceFlinger”的Binder服务端运行在[SurfaceFlinger进程](http://gityuan.com/2017/02/11/surface_flinger/)

SurfaceFlinger关系图：点击查看[大图](http://gityuan.com/images/surfaceFlinger/class_surface.jpg)



Client，SurfaceFlinger这两个Binder服务运行在SurfaceFlinger进程. SurfaceComposerClient对象的两个成员变量分别跟着两个Binder服务通信：

* 其成员变量mClient通过Binder调用Client服务，
* 其成员变量mComposer经过Composer(位于SurfaceComposerClient.cpp文件)，ComposerService对象，再通过Binder调用SurfaceFlinger。

也就是说只需要调用new SurfaceComposerClient()便建立应用程序跟SurfaceFlinger服务建立连接， 获取了其中两个Binder的代理类。每一个app在SurfaceFlinger中都有一个Client对象相对应。

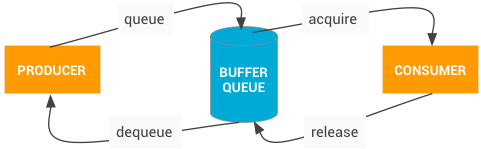
当app来到前台的执行流程：

1. WMS会请求SurfaceFlinger来绘制surface.
2. SurfaceFlinger创建layer;
3. 一个生产者的Binder对象通过WMS传递给app, 因此app可以直接向SurfaceFlinger发送帧信息

对于大多数的app来说都有3个layers: 状态栏,导航栏, 应用UI. 每一个layer都是独立更新的. 状态栏和导航栏是由系统进程负责渲染, app层是由app自己渲染,两者直接并没有协作。

## BufferQueue简析

BufferQueue是Android 中所有图形处理操作的核心。它的作用很简单：将生成图形数据缓冲区的一方（生产方）连接到接受数据以进行显示或进一步处理的一方（消耗方）。几乎所有在系统中移动图形数据缓冲区的内容都依赖于BufferQueue，比如显示、编码等。  
BufferQueue是典型的生产者、消费者模型，通信过程如下所示：

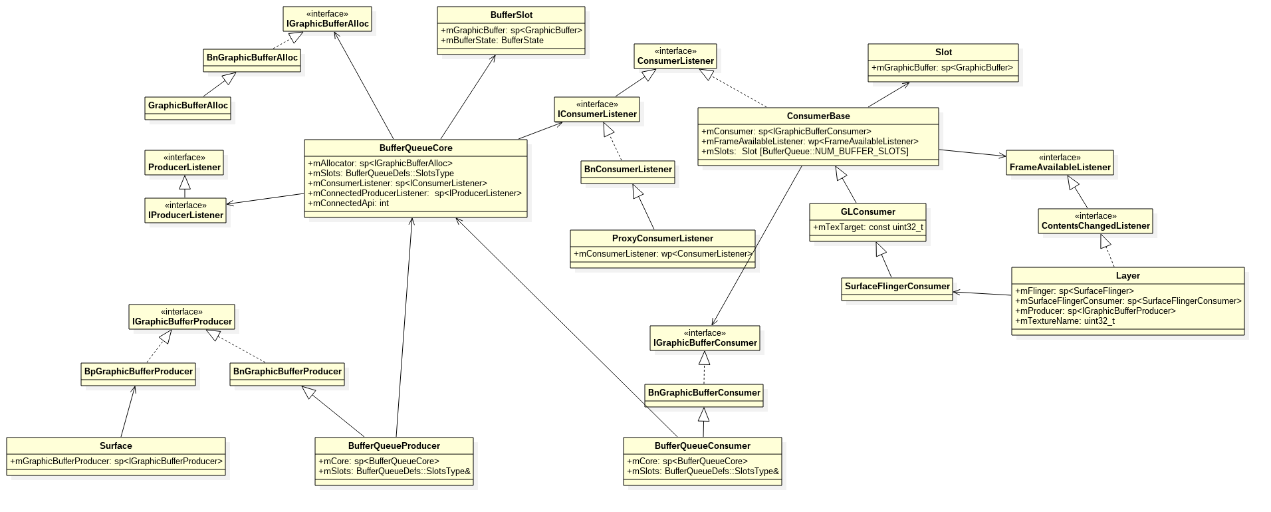


即图形生产者先向BufferQueue申请GraphicBuffer，填充完GraphicBuffer后，将GraphicBuffer移交给BufferQueue，BufferQueue会通知图形消费者（比如SurfaceFlinger、OMX等）有新的图形数据可以使用，图形消费者就可以从BufferQueue取出GraphicBuffer，使用完之后放回到BufferQueue以便循环使用。一般图形生产者和消费者是在不同进程中，BufferQueue采用了binder和共享内存机制，因此可以高效地在进程间传递图形数据。

### 架构

#### 核心类

以显示流程为例，生产者进程一般就是应用程序进程，消费者进程就是SurfaceFlinger进程，应用程序进程的surface对象和SurfaceFlinger进程的layer对象可以看做实际的生产者和消费者，主要类的关系如下所示：

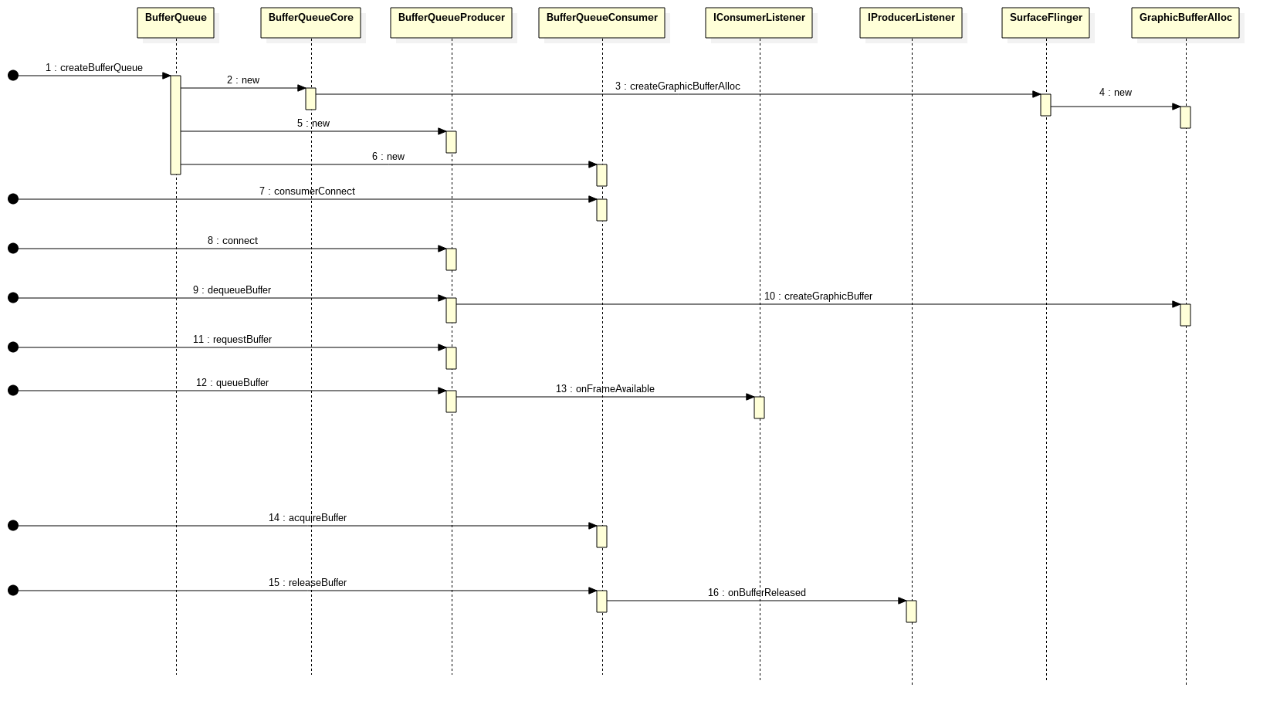


应用程序申请surface时，会在SurfaceFlinger进程创建一个layer对象，接着会创建BufferQueueCore、BufferQueueProducer和BufferQueueConsumer对象，然后创建SurfaceFlingerConsumer和ProxyConsumerListener对象，

而在应用程序进程这端会创建Surface对象和BpGraphicBufferProducer代理对象

应用程序进程通过Surface对象中的BpGraphicBufferProducer向SurfaceFlinger进程中的BufferQueueCore对象申请和提交GraphicBuffer，SurfaceFlinger进程中的BufferQueueCore对象通过ProxyConsumerListener、SurfaceFlingerConsumer、Layer一路通知到SurfaceFlinger有新的GraphicBuffer需要合成，SurfaceFlinger收到通知，通过Layer调用SurfaceFlingerConsumer的updateTexImage，将GraphicBuffer绘制成纹理，然后再合成输出。

#### 时序



一般是在消费者进程调用BufferQueue的createBufferQueue创建BufferQueueCore、BufferQueueProducer和BufferQueueConsumer对象，这三个对象与图一对应，其中BufferQueueProducer和BufferQueueConsumer分别实现了IGraphicBufferProducer和IGraphicBufferConsumer binder接口，然后会在消费者进程调用consumerConnect向BufferQueueCore注册IConsumerListener，这样有新数据的时候就可以通过IConsumerListener通知对应的消费者，收到通知后，消费者通过IGraphicBufferConsumer从BufferQueueCore取出GraphicBuffer，使用完之后将GraphicBuffer放回到BufferQueueCore。

在生产者进程获取到IGraphicBufferProducer接口后，可以通过connect向BufferQueueCore注册IProducerListener，这样在消费者将GraphicBuffer放回BufferQueueCore时可以通知生产者，生产者可以通过IGraphicBufferProducer向BufferQueueCore申请GraphicBuffer，填充完GraphicBuffer后，将GraphicBuffer提交给BufferQueueCore，然后由BufferQueueCore来通知消费者。

### Trace

frameworks/native/include/gui/BufferQueueCore.h

#define ATRACE\_BUFFER\_INDEX(index) \

if (ATRACE\_ENABLED()) { \

char \_\_\_traceBuf[1024]; \

snprintf(\_\_\_traceBuf, 1024, "%s: %d", \

mCore->mConsumerName.string(), (index)); \

android::ScopedTrace \_\_\_bufTracer(ATRACE\_TAG, \_\_\_traceBuf); \

}

ATRACE\_BUFFER打桩

ATRACE\_CALL();

ATRACE\_BUFFER\_INDEX(slot);

BufferQueueConsumer.cpp

acquireBuffer/releaseBuffer

detachBuffer/attachBuffer

----

BufferQueueProducer.cpp

dequeueBuffer/queueBuffer

detachBuffer/attachBuffer

生产者消费者PC模型

优先缓存问题

### 参考

Android BufferQueue简析

https://www.jianshu.com/p/edd7d264be73

## 图形处理

### 图形数据流



图中最左侧是指渲染器,用于生产graphics buffers, 比如状态栏,systemUI等. 再来看看图中BufferQueue的工作

### 生成者消费者模式



图解:

生产者和消费者运行在不同的进程.

* 生产者请求一块空闲的缓存区:dequeueBuffer()
* 生产者填充缓存区并返回给队列: queueBuffer()
* 消费者获取一块缓存区: acquireBuffer()
* 消费者使用完毕,则返回给队列: releaseBuffer()



## REF

[Android图形系统概述](http://gityuan.com/2017/02/05/graphic_arch/)

# SurfaceFlinger启动篇

frameworks/native/services/surfaceflinger/

- main\_surfaceflinger.cpp

- SurfaceFlinger.cpp

- DispSync.cpp

- MessageQueue.cpp

- DisplayHardware/HWComposer.cpp

frameworks/native/libs/gui/

- DisplayEventReceiver.cpp

- BitTube.cpp

## 概述

启动脚本，7.1的如下：

service surfaceflinger /system/bin/surfaceflinger  
 class core  
 user system  
 group graphics drmrpc readproc  
 onrestart restart zygote  
 writepid /dev/stune/foreground/tasks

android 8.1系统中, surfaceflinger进程对应的配置不是在init.rc中,而是在surfaceflinger.rc中,如下

key@ubuntu:~/aosp/android-9.0.0\_r35$ cat frameworks/native/services/surfaceflinger/surfaceflinger.rc

service surfaceflinger /system/bin/surfaceflinger

class core animation

user system

group graphics drmrpc readproc

onrestart restart zygote

writepid /dev/stune/foreground/tasks

socket pdx/system/vr/display/client stream 0666 system graphics u:object\_r:pdx\_display\_client\_endpoint\_socket:s0

socket pdx/system/vr/display/manager stream 0666 system graphics u:object\_r:pdx\_display\_manager\_endpoint\_socket:s0

socket pdx/system/vr/display/vsync stream 0666 system graphics u:object\_r:pdx\_display\_vsync\_endpoint\_socket:s0

service surfaceflinger /system/bin/surfaceflinger

class core

user system

group graphics drmrpc

onrestart restart zygote

writepid /dev/cpuset/system-background/tasks

surfaceflinger服务属于核心类(core class)，另外，当surfaceflinger重启时会触发zygote的重启。 surfaceflinger服务启动的起点便是如下的main()函数。

## 启动过程

### main

[-> main\_surfaceflinger.cpp]

int main(int, char\*\*) {

ProcessState::self()->setThreadPoolMaxThreadCount(4);

sp<ProcessState> ps(ProcessState::self());

ps->startThreadPool();

//实例化surfaceflinger【见小节2.2】

sp<SurfaceFlinger> flinger = new SurfaceFlinger();

setpriority(PRIO\_PROCESS, 0, PRIORITY\_URGENT\_DISPLAY);

set\_sched\_policy(0, SP\_FOREGROUND);

//初始化【见小节2.3】

flinger->init();

//发布surface flinger，注册到Service Manager

sp<IServiceManager> sm(defaultServiceManager());

sm->addService(String16(SurfaceFlinger::getServiceName()), flinger, false);

*// publish GpuService*sp<GpuService> gpuservice = **new** GpuService();  
sm->addService(String16(GpuService::SERVICE\_NAME), gpuservice, **false**);

// 运行在当前线程【见小节2.11】

flinger->run();

return 0;

}

该方法的主要功能：

* 设定surfaceflinger进程的binder线程池个数上限为4，并启动binder线程池；
* 创建SurfaceFlinger对象；【见小节2.1】
* 设置surfaceflinger进程为高优先级以及前台调度策略；
* 初始化SurfaceFlinger；【见小节2.3】
* 将”SurfaceFlinger”服务注册到Service Manager;
* 在当前主线程执行SurfaceFlinger的run方法。【见小节2.11】

### 创建SurfaceFlinger

[-> SurfaceFlinger.cpp]

SurfaceFlinger::SurfaceFlinger()

: BnSurfaceComposer(),

mTransactionFlags(0),

mTransactionPending(false),

mAnimTransactionPending(false),

mLayersRemoved(false),

mRepaintEverything(0),

mRenderEngine(NULL),

mBootTime(systemTime()),

mVisibleRegionsDirty(false),

mHwWorkListDirty(false),

mAnimCompositionPending(false),

mDebugRegion(0),

mDebugDDMS(0),

mDebugDisableHWC(0),

mDebugDisableTransformHint(0),

mDebugInSwapBuffers(0),

mLastSwapBufferTime(0),

mDebugInTransaction(0),

mLastTransactionTime(0),

mBootFinished(false),

mForceFullDamage(false),

mPrimaryHWVsyncEnabled(false),

mHWVsyncAvailable(false),

mDaltonize(false),

mHasColorMatrix(false),

mHasPoweredOff(false),

mFrameBuckets(),

mTotalTime(0),

mLastSwapTime(0)

{

ALOGI("SurfaceFlinger is starting");

char value[PROPERTY\_VALUE\_MAX];

property\_get("ro.bq.gpu\_to\_cpu\_unsupported", value, "0");

mGpuToCpuSupported = !atoi(value);

property\_get("debug.sf.showupdates", value, "0");

mDebugRegion = atoi(value);

property\_get("debug.sf.ddms", value, "0");

mDebugDDMS = atoi(value);

}

SurfaceFlinger继承于BnSurfaceComposer,IBinder::DeathRecipient,HWComposer::EventHandler

flinger的数据类型为**sp强指针类型，当首次被强指针引用时则执行OnFirstRef(**)

#### SF.onFirstRef

void SurfaceFlinger::onFirstRef()

{

mEventQueue.init(this);

}

#### MessageQueue.init

[-> MessageQueue.cpp]

void MessageQueue::init(const sp<SurfaceFlinger>& flinger)

{

mFlinger = flinger;

mLooper = new Looper(true);

mHandler = new Handler(\*this); //【见小节2.2.3】

}

这个Handler是MessageQueue的内部类Handler，此处传递的\*this便是MessageQueue本身

#### MQ.Handler

[-> MessageQueue.cpp]

class MessageQueue {

class Handler : public MessageHandler {

enum {

eventMaskInvalidate = 0x1,

eventMaskRefresh = 0x2,

eventMaskTransaction = 0x4

};

MessageQueue& mQueue;

int32\_t mEventMask;

public:

Handler(MessageQueue& queue) : mQueue(queue), mEventMask(0) { }

virtual void handleMessage(const Message& message);

void dispatchRefresh();

void dispatchInvalidate();

void dispatchTransaction();

};

...

}

### SurfaceFlinger.init

1. void SurfaceFlinger::init() {
2. Mutex::Autolock \_l(mStateLock);
3. //初始化EGL，作为默认的显示
4. mEGLDisplay = eglGetDisplay(EGL\_DEFAULT\_DISPLAY);
5. eglInitialize(mEGLDisplay, NULL, NULL);
6. // 初始化硬件composer对象【见小节2.4】
7. mHwc = new HWComposer(this, \*static\_cast<HWComposer::EventHandler \*>(this));
8. //获取RenderEngine引擎
9. mRenderEngine = RenderEngine::create(mEGLDisplay, mHwc->getVisualID());
10. //检索创建的EGL上下文
11. mEGLContext = mRenderEngine->getEGLContext();
12. //初始化非虚拟显示屏【见小节2.5】
13. for (size\_t i=0 ; i<DisplayDevice::NUM\_BUILTIN\_DISPLAY\_TYPES ; i++) {
14. DisplayDevice::DisplayType type((DisplayDevice::DisplayType)i);
15. //建立已连接的显示设备
16. if (mHwc->isConnected(i) || type==DisplayDevice::DISPLAY\_PRIMARY) {
17. bool isSecure = true;
18. createBuiltinDisplayLocked(type);
19. wp<IBinder> token = mBuiltinDisplays[i];
20. sp<IGraphicBufferProducer> producer;
21. sp<IGraphicBufferConsumer> consumer;
22. //创建BufferQueue的生产者和消费者
23. BufferQueue::createBufferQueue(&producer, &consumer,
24. new GraphicBufferAlloc());
25. sp<FramebufferSurface> fbs = new FramebufferSurface(\*mHwc, i, consumer);
26. int32\_t hwcId = allocateHwcDisplayId(type);
27. //创建显示设备
28. sp<DisplayDevice> hw = new DisplayDevice(this,
29. type, hwcId, mHwc->getFormat(hwcId), isSecure, token,
30. fbs, producer,
31. mRenderEngine->getEGLConfig());
32. if (i > DisplayDevice::DISPLAY\_PRIMARY) {
33. hw->setPowerMode(HWC\_POWER\_MODE\_NORMAL);
34. }
35. mDisplays.add(token, hw);
36. }
37. }
38. getDefaultDisplayDevice()->makeCurrent(mEGLDisplay, mEGLContext);
39. // // start the EventThread
40. //基于不同的vsync信号模型创建的EventThread,mEventThread是app端对vsync信号的处理线程，mSFEventThread是SurfaceFlinger端对vysnc信号的处理线程
41. //当应用和sf的vsync偏移量一致时，则只创建一个EventThread线程
42. if (vsyncPhaseOffsetNs != sfVsyncPhaseOffsetNs) {
43. sp<VSyncSource> vsyncSrc = new DispSyncSource(&mPrimaryDispSync,
44. vsyncPhaseOffsetNs, true, "app");
45. mEventThread = new EventThread(vsyncSrc);
46. sp<VSyncSource> sfVsyncSrc = new DispSyncSource(&mPrimaryDispSync,
47. sfVsyncPhaseOffsetNs, true, "sf");
48. mSFEventThread = new EventThread(sfVsyncSrc);
49. mEventQueue.setEventThread(mSFEventThread);
50. } else {
51. //创建DispSyncSource对象【2.6】
52. sp<VSyncSource> vsyncSrc = new DispSyncSource(&mPrimaryDispSync,
53. vsyncPhaseOffsetNs, true, "sf-app");
54. //创建线程EventThread 【见小节2.7】
55. mEventThread = new EventThread(vsyncSrc);
56. //设置EventThread 【见小节2.8】
57. mEventQueue.setEventThread(mEventThread);
58. }
59. //【见小节2.9】
60. *//mEventControlThread线程用来开关vsync*
61. mEventControlThread = new EventControlThread(this);
62. mEventControlThread->run("EventControl", PRIORITY\_URGENT\_DISPLAY);
63. //当不存在HWComposer时，则设置软件vsync
64. if (mHwc->initCheck() != NO\_ERROR) {
65. mPrimaryDispSync.setPeriod(16666667);
66. }
67. //初始化绘图状态
68. mDrawingState = mCurrentState;
69. //初始化显示设备
70. initializeDisplays();
71. //启动开机动画【2.10】
72. startBootAnim();
73. } ......
74. }

主要功能：

初始化EGL相关；

创建HWComposer对象；

初始化非虚拟显示屏；

启动app和sf两个EventThread线程；

启动开机动画；

另外，当应用和sf的vsync偏移量一致时，则只创建一个EventThread线程；否则会创建两个DispSyncSource对象，分别是用于绘制(app)和合成(SurfaceFlinger)。

vsync信号分为两种：1.硬件产生的vsync；2.软件模拟的vsync

#### android7.0

*// start the EventThread*sp<VSyncSource> vsyncSrc = **new** DispSyncSource(&mPrimaryDispSync,  
 vsyncPhaseOffsetNs, **true**, **"app"**);  
mEventThread = **new** EventThread(vsyncSrc, \***this**);  
sp<VSyncSource> sfVsyncSrc = **new** DispSyncSource(&mPrimaryDispSync,  
 sfVsyncPhaseOffsetNs, **true**, **"sf"**);  
mSFEventThread = **new** EventThread(sfVsyncSrc, \***this**);  
mEventQueue.setEventThread(mSFEventThread);

### 创建HWComposer

[-> HWComposer.cpp]

1. Java
2. HWComposer::HWComposer(
3. const sp<SurfaceFlinger>& flinger,
4. EventHandler& handler)
5. : mFlinger(flinger),
6. mFbDev(0), mHwc(0), mNumDisplays(1),
7. mCBContext(new cb\_context),
8. mEventHandler(handler),
9. mDebugForceFakeVSync(false)
10. {
11. ...
12. bool needVSyncThread = true;
13. int fberr = loadFbHalModule(); //加载framebuffer的HAL层模块
14. loadHwcModule(); //加载HWComposer模块
15. //标记已分配的display ID
16. for (size\_t i=0 ; i<NUM\_BUILTIN\_DISPLAYS ; i++) {
17. mAllocatedDisplayIDs.markBit(i);
18. }
19. if (mHwc) {
20. if (mHwc->registerProcs) {
21. mCBContext->hwc = this;
22. mCBContext->procs.invalidate = &hook\_invalidate;
23. //VSYNC信号的回调方法
24. mCBContext->procs.vsync = &hook\_vsync;
25. if (hwcHasApiVersion(mHwc, HWC\_DEVICE\_API\_VERSION\_1\_1))
26. mCBContext->procs.hotplug = &hook\_hotplug;
27. else
28. mCBContext->procs.hotplug = NULL;
29. memset(mCBContext->procs.zero, 0, sizeof(mCBContext->procs.zero));
30. //注册回调函数
31. mHwc->registerProcs(mHwc, &mCBContext->procs);
32. }
33. //进入此处，说明已成功打开硬件composer设备，则不再需要vsync线程
34. needVSyncThread = false;
35. eventControl(HWC\_DISPLAY\_PRIMARY, HWC\_EVENT\_VSYNC, 0);
36. ...
37. }
38. ...
39. if (needVSyncThread) {
40. //不支持硬件的VSYNC，则会创建线程来模拟定时VSYNC信号
41. mVSyncThread = new VSyncThread(\*this);
42. }
43. }

HWComposer代表着硬件显示设备，注册了VSYNC信号的回调。VSYNC信号本身是由显示驱动产生的， 在不支持硬件的VSYNC，则会创建“VSyncThread”线程来模拟定时VSYNC信号。

### 显示设备

[-> SurfaceFlinger.cpp]

1. Java start
2. void SurfaceFlinger::init() {
3. ...
4. for (size\_t i=0 ; i<DisplayDevice::NUM\_BUILTIN\_DISPLAY\_TYPES ; i++) {
5. DisplayDevice::DisplayType type((DisplayDevice::DisplayType)i);
6. **//建立已连接的显示设备**
7. if (mHwc->isConnected(i) || type==DisplayDevice::DISPLAY\_PRIMARY) {
8. bool isSecure = true;
9. createBuiltinDisplayLocked(type);
10. wp<IBinder> token = mBuiltinDisplays[i];
11. sp<IGraphicBufferProducer> producer;
12. sp<IGraphicBufferConsumer> consumer;
13. **//创建BufferQueue的生产者和消费者**
14. BufferQueue::createBufferQueue(&producer, &consumer,
15. new GraphicBufferAlloc());
16. sp<FramebufferSurface> fbs = new FramebufferSurface(\*mHwc, i, consumer);
17. int32\_t hwcId = allocateHwcDisplayId(type);
18. **//创建显示设备**
19. sp<DisplayDevice> hw = new DisplayDevice(this,
20. type, hwcId, mHwc->getFormat(hwcId), isSecure, token,
21. fbs, producer,
22. mRenderEngine->getEGLConfig());
23. if (i > DisplayDevice::DISPLAY\_PRIMARY) {
24. hw->setPowerMode(HWC\_POWER\_MODE\_NORMAL);
25. }
26. mDisplays.add(token, hw);
27. }
28. }
29. ...
30. }End

创建IGraphicBufferProducer和IGraphicBufferConsumer，以及FramebufferSurface，DisplayDevice对象。另外， 显示设备有3类：主设备，扩展设备，虚拟设备。其中前两个都是内置显示设备，故NUM\_BUILTIN\_DISPLAY\_TYPES=2，

### DispSyncSource

[-> SurfaceFlinger.cpp]

1. Java start
2. class DispSyncSource : public VSyncSource, private DispSync::Callback {
3. DispSyncSource(DispSync\* dispSync, nsecs\_t phaseOffset, bool traceVsync,
4. const char\* label) :
5. mValue(0),
6. mTraceVsync(traceVsync),
7. mVsyncOnLabel(String8::format("VsyncOn-%s", label)),
8. mVsyncEventLabel(String8::format("VSYNC-%s", label)),
9. mDispSync(dispSync),
10. mCallbackMutex(),
11. mCallback(),
12. mVsyncMutex(),
13. mPhaseOffset(phaseOffset),
14. mEnabled(false) {}
15. ... }End

### EventThread线程

[-> EventThread.cpp]

1. Java start
2. EventThread::EventThread(const sp<VSyncSource>& src)
3. : mVSyncSource(src),
4. mUseSoftwareVSync(false),
5. mVsyncEnabled(false),
6. mDebugVsyncEnabled(false),
7. mVsyncHintSent(false) {
8. for (int32\_t i=0 ; i<DisplayDevice::NUM\_BUILTIN\_DISPLAY\_TYPES ; i++) {
9. mVSyncEvent[i].header.type = DisplayEventReceiver::DISPLAY\_EVENT\_VSYNC;
10. mVSyncEvent[i].header.id = 0;
11. mVSyncEvent[i].header.timestamp = 0;
12. mVSyncEvent[i].vsync.count = 0;
13. }
14. struct sigevent se;
15. se.sigev\_notify = SIGEV\_THREAD;
16. se.sigev\_value.sival\_ptr = this;
17. se.sigev\_notify\_function = vsyncOffCallback;
18. se.sigev\_notify\_attributes = NULL;
19. timer\_create(CLOCK\_MONOTONIC, &se, &mTimerId);
20. }
21. End

EventThread继承于Thread和VSyncSource::Callback两个类。

#### ET.onFirstRef

[-> EventThread.cpp]

void EventThread::onFirstRef() {

//运行EventThread线程【见小节2.7.2】

run("EventThread", PRIORITY\_URGENT\_DISPLAY + PRIORITY\_MORE\_FAVORABLE);

}

#### ET.threadLoop

[-> EventThread.cpp]

1. Java start
2. bool EventThread::threadLoop() {
3. DisplayEventReceiver::Event event;
4. Vector< sp<EventThread::Connection> > signalConnections;
5. // 等待事件【见小节2.7.3】
6. signalConnections = waitForEvent(&event);
7. //分发事件给所有的监听者
8. const size\_t count = signalConnections.size();
9. for (size\_t i=0 ; i<count ; i++) {
10. const sp<Connection>& conn(signalConnections[i]);
11. //传递事件【见小节3.10】
12. status\_t err = conn->postEvent(event);
13. if (err == -EAGAIN || err == -EWOULDBLOCK) {
14. //可能此时connection已满，则直接抛弃事件
15. ALOGW("EventThread: dropping event (%08x) for connection %p",
16. event.header.type, conn.get());
17. } else if (err < 0) {
18. //发生致命错误，则清理该连接
19. removeDisplayEventConnection(signalConnections[i]);
20. }
21. }
22. return true;
23. }
24. End

#### waitForEvent

[-> EventThread.cpp]

Vector< sp<EventThread::Connection> > EventThread::waitForEvent(

DisplayEventReceiver::Event\* event)

{

Mutex::Autolock \_l(mLock);

Vector< sp<EventThread::Connection> > signalConnections;

do {

bool eventPending = false;

bool waitForVSync = false;

size\_t vsyncCount = 0;

nsecs\_t timestamp = 0;

for (int32\_t i=0 ; i<DisplayDevice::NUM\_BUILTIN\_DISPLAY\_TYPES ; i++) {

timestamp = mVSyncEvent[i].header.timestamp;

if (timestamp) {

\*event = mVSyncEvent[i];

mVSyncEvent[i].header.timestamp = 0;

vsyncCount = mVSyncEvent[i].vsync.count;

break;

}

}

if (!timestamp) {

//没有vsync事件，则查看其它事件

eventPending = !mPendingEvents.isEmpty();

if (eventPending) {

//存在其它事件可用于分发

\*event = mPendingEvents[0];

mPendingEvents.removeAt(0);

}

}

//查找正在等待事件的连接

size\_t count = mDisplayEventConnections.size();

for (size\_t i=0 ; i<count ; i++) {

sp<Connection> connection(mDisplayEventConnections[i].promote());

if (connection != NULL) {

bool added = false;

if (connection->count >= 0) {

//需要vsync事件，由于至少存在一个连接正在等待vsync

waitForVSync = true;

if (timestamp) {

if (connection->count == 0) {

connection->count = -1;

signalConnections.add(connection);

added = true;

} else if (connection->count == 1 ||

(vsyncCount % connection->count) == 0) {

signalConnections.add(connection);

added = true;

}

}

}

if (eventPending && !timestamp && !added) {

//没有vsync事件需要处理(timestamp==0),但存在pending消息

signalConnections.add(connection);

}

} else {

//该连接已死亡，则直接清理

mDisplayEventConnections.removeAt(i);

--i; --count;

}

}

if (timestamp && !waitForVSync) {

//接收到VSYNC，但没有client需要它，则直接关闭VSYNC

disableVSyncLocked();

} else if (!timestamp && waitForVSync) {

//至少存在一个client，则需要使能VSYNC

enableVSyncLocked();

}

if (!timestamp && !eventPending) {

if (waitForVSync) {

bool softwareSync = mUseSoftwareVSync;

nsecs\_t timeout = softwareSync ? ms2ns(16) : ms2ns(1000);

if (mCondition.waitRelative(mLock, timeout) == TIMED\_OUT) {

mVSyncEvent[0].header.type = DisplayEventReceiver::DISPLAY\_EVENT\_VSYNC;

mVSyncEvent[0].header.id = DisplayDevice::DISPLAY\_PRIMARY;

mVSyncEvent[0].header.timestamp = systemTime(SYSTEM\_TIME\_MONOTONIC);

mVSyncEvent[0].vsync.count++;

}

} else {

//不存在对vsync感兴趣的连接，即将要进入休眠

mCondition.wait(mLock);

}

}

} while (signalConnections.isEmpty());

//到此处，则保证存在timestamp以及连接

return signalConnections;

}

EventThread线程，进入mCondition的wait()方法，等待唤醒。

### setEventThread

df

[-> MessageQueue.cpp]

void MessageQueue::setEventThread(const sp<EventThread>& eventThread)

{

mEventThread = eventThread;

//创建连接

mEvents = eventThread->createEventConnection();

//获取BitTube对象

mEventTube = mEvents->getDataChannel();

//监听BitTube，一旦有数据到来则调用cb\_eventReceiver()

mLooper->addFd(mEventTube->getFd(), 0, Looper::EVENT\_INPUT,

MessageQueue::cb\_eventReceiver, this);

}

此处mEvents的数据类型为sp，mEventTube的数据类型为sp。

### EventControlThread线程

[-> EventControlThread.cpp]

EventControlThread::EventControlThread(const sp<SurfaceFlinger>& flinger):

mFlinger(flinger),

mVsyncEnabled(false) {

}

bool EventControlThread::threadLoop() {

Mutex::Autolock lock(mMutex);

bool vsyncEnabled = mVsyncEnabled;

mFlinger->eventControl(HWC\_DISPLAY\_PRIMARY, SurfaceFlinger::EVENT\_VSYNC,

mVsyncEnabled);

while (true) {

status\_t err = mCond.wait(mMutex);

...

if (vsyncEnabled != mVsyncEnabled) {

mFlinger->eventControl(HWC\_DISPLAY\_PRIMARY,

SurfaceFlinger::EVENT\_VSYNC, mVsyncEnabled);

vsyncEnabled = mVsyncEnabled;

}

}

return false;

}

EventControlThread也是继承于Thread。

### startBootAnim

d

[-> SurfaceFlinger.cpp]

void SurfaceFlinger::startBootAnim() {

property\_set("service.bootanim.exit", "0");

property\_set("ctl.start", "bootanim");

}

通过控制ctl.start属性，设置成bootanim值，则触发init进程来创建开机动画进程bootanim， 到此，则开始显示开机过程的动画。 从小节[2.4 ~2.9]都是介绍SurfaceFlinger的init()过程， 紧接着便执行其run()方法。

### SF.run

Fsd

SurfaceFlinger.cpp]

void SurfaceFlinger::run() {

do {

//不断循环地等待事件【见小节2.12】

waitForEvent();

} while (true);

}

#### SF.waitForEvent

[-> SurfaceFlinger.cpp]

void SurfaceFlinger::waitForEvent() {

mEventQueue.waitMessage(); //【2.13】

}

mEventQueue的数据类型为MessageQueue。

#### MQ.waitMessage

[-> MessageQueue.cpp]

oid MessageQueue::waitMessage() {

do {

IPCThreadState::self()->flushCommands();

int32\_t ret = mLooper->pollOnce(-1);

...

} while (true);

}

可见SurfaceFlinger主线程进入waitMessage来等待消息的到来。

## Vsync信号

HWComposer对象创建过程，会注册一些回调方法，当硬件产生VSYNC信号时，则会回调hook\_vsync()方法。

### HWC.hook\_vsync

[-> HWComposer.cpp]

frameworks/native/services/surfaceflinger/DisplayHardware/HWComposer\_hwc1.cpp

void HWComposer::hook\_vsync(const struct hwc\_procs\* procs, int disp,

int64\_t timestamp) {

cb\_context\* ctx = reinterpret\_cast<cb\_context\*>(

const\_cast<hwc\_procs\_t\*>(procs));

ctx->hwc->vsync(disp, timestamp); //【见小节3.2】

}

### HWC.vsync

[-> HWComposer.cpp]

void HWComposer::vsync(int disp, int64\_t timestamp) {

if (uint32\_t(disp) < HWC\_NUM\_PHYSICAL\_DISPLAY\_TYPES) {

{

Mutex::Autolock \_l(mLock);

if (timestamp == mLastHwVSync[disp]) {

return; //忽略重复的VSYNC信号

}

mLastHwVSync[disp] = timestamp;

}

//【见小节3.3】

mEventHandler.onVSyncReceived(disp, timestamp);

}

}

当收到VSYNC信号则会回调EventHandler的onVSyncReceived()方法，此处mEventHandler是指SurfaceFlinger对象。

**class** SurfaceFlinger : **public** BnSurfaceComposer,  
 **private** IBinder::DeathRecipient,  
 **private** HWComposer::EventHandler

不少厂商会修改这部分代码，同时这个地方笔者发现是8hz？？？为啥

### SF.onVSyncReceived

[-> SurfaceFlinger.cpp]

void SurfaceFlinger::onVSyncReceived(int type, nsecs\_t timestamp) {

bool needsHwVsync = false;

{

Mutex::Autolock \_l(mHWVsyncLock);

if (type == 0 && mPrimaryHWVsyncEnabled) {

// 此处mPrimaryDispSync为DispSync类【见小节3.4】

needsHwVsync = mPrimaryDispSync.addResyncSample(timestamp);

}

}

if (needsHwVsync) {

enableHardwareVsync();

} else {

disableHardwareVsync(false);

}

}

needsHwVsync有时候为0，有时候为1

### DS.addResyncSample

此处调用addResyncSample对象的addResyncSample方法，那么先来看看DispSync对象的初始化过程

#### 创建DispSync

[-> DispSync.cpp]

DispSync::DispSync() :

mRefreshSkipCount(0),

mThread(new DispSyncThread()) {

//【见小节3.4.2】

mThread->run("DispSync", PRIORITY\_URGENT\_DISPLAY + PRIORITY\_MORE\_FAVORABLE);

reset();

beginResync();

...

}

#### DispSyncThread线程

[-> DispSync.cpp]

virtual bool threadLoop() {

status\_t err;

nsecs\_t now = systemTime(SYSTEM\_TIME\_MONOTONIC);

nsecs\_t nextEventTime = 0;

while (true) {

Vector<CallbackInvocation> callbackInvocations;

nsecs\_t targetTime = 0;

{ // Scope for lock

Mutex::Autolock lock(mMutex);

if (mStop) {

return false;

}

if (mPeriod == 0) {

err = mCond.wait(mMutex);

continue;

}

nextEventTime = computeNextEventTimeLocked(now);

targetTime = nextEventTime;

bool isWakeup = false;

if (now < targetTime) {

err = mCond.waitRelative(mMutex, targetTime - now);

if (err == TIMED\_OUT) {

isWakeup = true;

} else if (err != NO\_ERROR) {

return false;

}

}

now = systemTime(SYSTEM\_TIME\_MONOTONIC);

if (isWakeup) {

mWakeupLatency = ((mWakeupLatency \* 63) +

(now - targetTime)) / 64;

if (mWakeupLatency > 500000) {

mWakeupLatency = 500000;

}

}

//收集vsync信号的所有回调方法

callbackInvocations = gatherCallbackInvocationsLocked(now);

}

if (callbackInvocations.size() > 0) {

//回调所有对象的onDispSyncEvent方法

fireCallbackInvocations(callbackInvocations);

}

}

return false;

}

线程”DispSync”停留在mCond的wait()过程，等待被唤醒。

#### addResyncSample

[-> DispSync.cpp]

bool DispSync::addResyncSample(nsecs\_t timestamp) {

Mutex::Autolock lock(mMutex);

size\_t idx = (mFirstResyncSample + mNumResyncSamples) % MAX\_RESYNC\_SAMPLES;

mResyncSamples[idx] = timestamp;

if (mNumResyncSamples < MAX\_RESYNC\_SAMPLES) {

mNumResyncSamples++;

} else {

mFirstResyncSample = (mFirstResyncSample + 1) % MAX\_RESYNC\_SAMPLES;

}

updateModelLocked(); //【见小节3.5】

if (mNumResyncSamplesSincePresent++ > MAX\_RESYNC\_SAMPLES\_WITHOUT\_PRESENT) {

resetErrorLocked();

}

if (kIgnorePresentFences) {

return mThread->hasAnyEventListeners();

}

return mPeriod == 0 || mError > kErrorThreshold;

}

### DS.updateModelLocked

[-> DispSync.cpp]

void DispSync::updateModelLocked() {

...

//【见小节3.6】

mThread->updateModel(mPeriod, mPhase);

}

### DST.updateModel

Gh

[-> DispSyncThread.cpp]

class DispSyncThread: public Thread {

void updateModel(nsecs\_t period, nsecs\_t phase) {

Mutex::Autolock lock(mMutex);

mPeriod = period;

mPhase = phase;

mCond.signal(); //唤醒目标线程

}

}

唤醒DispSyncThread线程，接下里进入DispSyncThread线程。

### 3.7 DispSyncThread线程

[-> DispSync.cpp]

virtual bool threadLoop() {

...

while (true) {

Vector<CallbackInvocation> callbackInvocations;

nsecs\_t targetTime = 0;

{ // Scope for lock

Mutex::Autolock lock(mMutex);

...

if (now < targetTime) {

err = mCond.waitRelative(mMutex, targetTime - now);

...

}

...

//收集vsync信号的所有回调方法

callbackInvocations = gatherCallbackInvocationsLocked(now);

}

if (callbackInvocations.size() > 0) {

//回调所有对象的onDispSyncEvent方法 【见小节3.7.1】

fireCallbackInvocations(callbackInvocations);

}

}

return false;

}

#### 3.7.1 fireCallbackInvocations

void fireCallbackInvocations(const Vector<CallbackInvocation>& callbacks) {

for (size\_t i = 0; i < callbacks.size(); i++) {

//【见小节3.8】

callbacks[i].mCallback->onDispSyncEvent(callbacks[i].mEventTime);

}

}

在前面小节【2.3】SurfaceFlinger调用init()的过程，创建过DispSyncSource对象。接下里便是回调该对象的 onDispSyncEvent。

### 3.8 DSS.onDispSyncEvent

[-> SurfaceFlinger.cpp ::DispSyncSource]

virtual void onDispSyncEvent(nsecs\_t when) {

sp<VSyncSource::Callback> callback;

{

Mutex::Autolock lock(mCallbackMutex);

callback = mCallback;

}

if (callback != NULL) {

callback->onVSyncEvent(when); //【见小节3.9】

}

}

### 3.9 ET.onVSyncEvent

[-> EventThread.java]

void EventThread::onVSyncEvent(nsecs\_t timestamp) {

Mutex::Autolock \_l(mLock);

mVSyncEvent[0].header.type = DisplayEventReceiver::DISPLAY\_EVENT\_VSYNC;

mVSyncEvent[0].header.id = 0;

mVSyncEvent[0].header.timestamp = timestamp;

mVSyncEvent[0].vsync.count++;

mCondition.broadcast(); //唤醒EventThread线程

}

mCondition.broadcast能够唤醒处理waitForEvent()过程的EventThread【见小节2.7.2】，并往下执行conn的postEvent().

### 3.10 ET.postEvent

[-> EventThread.java]

status\_t EventThread::Connection::postEvent(

const DisplayEventReceiver::Event& event) {

ssize\_t size = DisplayEventReceiver::sendEvents(mChannel, &event, 1);

return size < 0 ? status\_t(size) : status\_t(NO\_ERROR);

}

### 3.11 DER.sendEvents

[-> DisplayEventReceiver.cpp]

ssize\_t DisplayEventReceiver::sendEvents(const sp<BitTube>& dataChannel,

Event const\* events, size\_t count)

{

return BitTube::sendObjects(dataChannel, events, count);

}

根据小节【2.8】可知监听BitTube，此处调用BitTube来sendObjects。一旦收到数据，则调用MQ.cb\_eventReceiver()方法。

#### 3.11.1 MQ.cb\_eventReceiver

[-> MessageQueue.cpp]

int MessageQueue::cb\_eventReceiver(int fd, int events, void\* data) {

MessageQueue\* queue = reinterpret\_cast<MessageQueue \*>(data);

return queue->eventReceiver(fd, events);

}

#### 3.11.2 MQ.eventReceiver

[-> MessageQueue.cpp]

int MessageQueue::eventReceiver(int /\*fd\*/, int /\*events\*/) {

ssize\_t n;

DisplayEventReceiver::Event buffer[8];

while ((n = DisplayEventReceiver::getEvents(mEventTube, buffer, 8)) > 0) {

for (int i=0 ; i<n ; i++) {

if (buffer[i].header.type == DisplayEventReceiver::DISPLAY\_EVENT\_VSYNC) {

#if INVALIDATE\_ON\_VSYNC

mHandler->dispatchInvalidate();

#else

mHandler->dispatchRefresh(); //【见小节3.12】

#endif

break;

}

}

}

return 1;

}

### 3.12 MQ.dispatchRefresh

void MessageQueue::Handler::dispatchRefresh() {

if ((android\_atomic\_or(eventMaskRefresh, &mEventMask) & eventMaskRefresh) == 0) {

//发送消息，则进入handleMessage过程【见小节3.13】

mQueue.mLooper->sendMessage(this, Message(MessageQueue::REFRESH));

}

}

### 3.13 MQ.handleMessage

void MessageQueue::Handler::handleMessage(const Message& message) {

switch (message.what) {

case INVALIDATE:

android\_atomic\_and(~eventMaskInvalidate, &mEventMask);

mQueue.mFlinger->onMessageReceived(message.what);

break;

case REFRESH:

android\_atomic\_and(~eventMaskRefresh, &mEventMask);

mQueue.mFlinger->onMessageReceived(message.what);//【见小节3.14】

break;

case TRANSACTION:

android\_atomic\_and(~eventMaskTransaction, &mEventMask);

mQueue.mFlinger->onMessageReceived(message.what);

break;

}

}

对于REFRESH操作，则进入onMessageReceived().

### 3.14 SF.onMessageReceived

[-> SurfaceFlinger.cpp]

void SurfaceFlinger::onMessageReceived(int32\_t what) {

ATRACE\_CALL();

switch (what) {

case MessageQueue::TRANSACTION: {

handleMessageTransaction();

break;

}

case MessageQueue::INVALIDATE: {

bool refreshNeeded = handleMessageTransaction();

refreshNeeded |= handleMessageInvalidate();

refreshNeeded |= mRepaintEverything;

if (refreshNeeded) {

signalRefresh();

}

break;

}

case MessageQueue::REFRESH: {

handleMessageRefresh();

break;

}

}

}

### 3.15 SF.handleMessageRefresh

[-> SurfaceFlinger.cpp]

void SurfaceFlinger::handleMessageRefresh() {

ATRACE\_CALL();

preComposition();

rebuildLayerStacks();

setUpHWComposer();

doDebugFlashRegions();

doComposition();

postComposition();

}

下一篇文章，再来介绍图形输出过程。

## 总结

前面讲述过程中所涉及到的线程情况：

* 主线程“/system/bin/surfaceflinger”: 主线程
* 线程”EventThread”：EventThread
* 线程”EventControl”： EventControlThread
* 线程”DispSync”：DispSyncThread



1. 底层vsync信号发送过来，一路执行到【小节3.6】DispSyncThread.updateModel()方法中调用mCond.signal() 来唤醒DispSyncThread线程；
2. DispSyncThread线程：执行到【小节3.9】EventThread::onVSyncEvent()方法中调用mCondition.broadcast() 唤醒EventThread线程；
3. EventThread线程：执行到【小节3.11】DisplayEventReceiver::sendEvents()方法中调用BitTube::sendObjects()； 由【小节2.8】可知当收到数据则调用MQ.cb\_eventReceiver()，然后再经过handler消息机制，进入SurfaceFlinger主线程； 4.SurfaceFlinger主线程：【小节3.13】进入到MesageQueue的handleMessage()，最终调用SurfaceFlinger的handleMessageRefresh()。

[SurfaceFlinger启动篇](http://gityuan.com/2017/02/11/surface_flinger/)

# SurfaceFlinger绘图篇

frameworks/native/services/surfaceflinger

- Layer.cpp

- Client.cpp

## 图形显示输出

上一篇文章SurfaceFlinger原理(一)，介绍了SurfaceFlinger和VSync的处理流程。当SurfaceFlinger进程收到VSync信号后经层层调用， 最终调用到该对象的handleMessageRefresh()方法。接下来，从该方法说起。

### SF.handleMessageRefresh

[-> SurfaceFlinger.cpp]

void SurfaceFlinger::handleMessageRefresh() {

ATRACE\_CALL();

preComposition(); //【见小节2.1】

rebuildLayerStacks(); //【见小节3.1】

setUpHWComposer(); //【见小节4.1】

doDebugFlashRegions();

doComposition(); //【见小节5.1】

postComposition(); //【见小节6.1】

}

先来看看SurfaceFlinger主线程绘制的systrace图：点击查看

http://gityuan.com/images/surfaceFlinger/systrace\_sf.png

## preComposition

### preComposition

void SurfaceFlinger::preComposition()

{

bool needExtraInvalidate = false;

const LayerVector& layers(mDrawingState.layersSortedByZ);

const size\_t count = layers.size();

for (size\_t i=0 ; i<count ; i++) {

//回调每个图层onPreComposition方法【见小节2.2】

if (layers[i]->onPreComposition()) {

needExtraInvalidate = true;

}

}

//当存在图层有变化，则发送invalidate消息

if (needExtraInvalidate) {

signalLayerUpdate(); //【见小节2.3】

}

}

此处mDrawingState结构体如下

struct State {

//所有参与绘制的Layer图层

LayerVector layersSortedByZ;

//所有输出设备的对象

DefaultKeyedVector< wp<IBinder>, DisplayDeviceState> displays;

};

SurfaceFlinger可以控制某些Layer不参与绘制过程，比如需要将悬浮按钮图层隐藏。

### onPreComposition

g

[-> Layer.cpp]

bool Layer::onPreComposition() {

mRefreshPending = false;

return mQueuedFrames > 0 || mSidebandStreamChanged;

}

mQueuedFrames初始化值为零，当其大于零则代表图层发生变化。那么onFrameAvailable时会对 mQueuedFrames执行加1操作。

#### onFrameAvailable

[-> Layer.cpp]

void Layer::onFrameAvailable(const BufferItem& item) {

{ // Autolock scope

Mutex::Autolock lock(mQueueItemLock);

if (item.mFrameNumber == 1) {

mLastFrameNumberReceived = 0;

}

while (item.mFrameNumber != mLastFrameNumberReceived + 1) {

status\_t result = mQueueItemCondition.waitRelative(mQueueItemLock,

ms2ns(500));

}

mQueueItems.push\_back(item);

android\_atomic\_inc(&mQueuedFrames); //加1操作

//唤醒所有pending的回调方法

mLastFrameNumberReceived = item.mFrameNumber;

mQueueItemCondition.broadcast();

}

mFlinger->signalLayerUpdate(); //【见小节2.3】

}

### signalLayerUpdate

void SurfaceFlinger::signalLayerUpdate() {

mEventQueue.invalidate();

}

#### invalidate

f

[-> MessageQueue.cpp]

void MessageQueue::invalidate() {

#if INVALIDATE\_ON\_VSYNC

mEvents->requestNextVsync();

#else

mHandler->dispatchInvalidate();

#endif

}

#### dispatchInvalidate

fd

void MessageQueue::Handler::dispatchInvalidate() {

if ((android\_atomic\_or(eventMaskInvalidate, &mEventMask) & eventMaskInvalidate) == 0) {

mQueue.mLooper->sendMessage(this, Message(MessageQueue::INVALIDATE));

}

}

发送消息到MessageQueue队列，Looper遍历消息后回调handleMessage方法来处理消息

#### handleMessage

void MessageQueue::Handler::handleMessage(const Message& message) {

switch (message.what) {

case INVALIDATE:

android\_atomic\_and(~eventMaskInvalidate, &mEventMask);

mQueue.mFlinger->onMessageReceived(message.what);

break;

case REFRESH:

android\_atomic\_and(~eventMaskRefresh, &mEventMask);

mQueue.mFlinger->onMessageReceived(message.what);

break;

case TRANSACTION:

android\_atomic\_and(~eventMaskTransaction, &mEventMask);

mQueue.mFlinger->onMessageReceived(message.what);

break;

}

}

#### SF.onMessageReceived

void SurfaceFlinger::onMessageReceived(int32\_t what) {

ATRACE\_CALL();

switch (what) {

case MessageQueue::TRANSACTION: {

handleMessageTransaction();

break;

}

case MessageQueue::INVALIDATE: {

//【见小节2.4】

bool refreshNeeded = handleMessageTransaction();

//【见小节2.5】

refreshNeeded |= handleMessageInvalidate();

refreshNeeded |= mRepaintEverything;

if (refreshNeeded) {

signalRefresh(); //【见小节2.6】

}

break;

}

case MessageQueue::REFRESH: {

handleMessageRefresh();

break;

}

}

}

当收到消息INVALIDATE时，则执行：

handleMessageTransaction；

handleMessageInvalidate；

根据是否需要刷新，来决定是否执行signalRefresh。

### handleMessageTransaction

d

bool SurfaceFlinger::handleMessageTransaction() {

uint32\_t transactionFlags = peekTransactionFlags(eTransactionMask);

if (transactionFlags) {

handleTransaction(transactionFlags); //【见小节2.4.1】

return true;

}

return false;

}

#### SF.handleTransaction

F

void SurfaceFlinger::handleTransaction(uint32\_t transactionFlags)

{

State drawingState(mDrawingState);

Mutex::Autolock \_l(mStateLock);

const nsecs\_t now = systemTime();

mDebugInTransaction = now;

transactionFlags = getTransactionFlags(eTransactionMask);

handleTransactionLocked(transactionFlags); //【见小节2.4.2】

mLastTransactionTime = systemTime() - now;

mDebugInTransaction = 0;

//设置mHwWorkListDirty=true

invalidateHwcGeometry();

}

#### SF.handleTransactionLocked

Df

void SurfaceFlinger::handleTransactionLocked(uint32\_t transactionFlags)

{

const LayerVector& currentLayers(mCurrentState.layersSortedByZ);

const size\_t count = currentLayers.size();

//遍历所有Layer来执行其doTransaction方法

if (transactionFlags & eTraversalNeeded) {

for (size\_t i=0 ; i<count ; i++) {

const sp<Layer>& layer(currentLayers[i]);

uint32\_t trFlags = layer->getTransactionFlags(eTransactionNeeded);

if (!trFlags) continue; //layer标志未改变则继续

//【见小节2.4.3】

const uint32\_t flags = layer->doTransaction(0);

if (flags & Layer::eVisibleRegion)

mVisibleRegionsDirty = true; //Layer成为可见，需更新

}

}

// 处理显示设备的改变

if (transactionFlags & eDisplayTransactionNeeded) {

//当前显示设备状态的列表

const KeyedVector< wp<IBinder>, DisplayDeviceState>& curr(mCurrentState.displays);

//之前使用过的显示设备状态的列表

const KeyedVector< wp<IBinder>, DisplayDeviceState>& draw(mDrawingState.displays);

if (!curr.isIdenticalTo(draw)) {

mVisibleRegionsDirty = true;

const size\_t cc = curr.size();

size\_t dc = draw.size();

for (size\_t i=0 ; i<dc ; i++) {

const ssize\_t j = curr.indexOfKey(draw.keyAt(i));

if (j < 0) {

if (!draw[i].isMainDisplay()) {

const sp<const DisplayDevice> defaultDisplay(getDefaultDisplayDevice());

defaultDisplay->makeCurrent(mEGLDisplay, mEGLContext);

sp<DisplayDevice> hw(getDisplayDevice(draw.keyAt(i)));

if (hw != NULL)

hw->disconnect(getHwComposer());

if (draw[i].type < DisplayDevice::NUM\_BUILTIN\_DISPLAY\_TYPES)

mEventThread->onHotplugReceived(draw[i].type, false);

mDisplays.removeItem(draw.keyAt(i));

}

} else {

const DisplayDeviceState& state(curr[j]);

const wp<IBinder>& display(curr.keyAt(j));

const sp<IBinder> state\_binder = IInterface::asBinder(state.surface);

const sp<IBinder> draw\_binder = IInterface::asBinder(draw[i].surface);

if (state\_binder != draw\_binder) {

sp<DisplayDevice> hw(getDisplayDevice(display));

if (hw != NULL)

hw->disconnect(getHwComposer());

mDisplays.removeItem(display);

mDrawingState.displays.removeItemsAt(i);

dc--; i--;

continue;

}

const sp<DisplayDevice> disp(getDisplayDevice(display));

if (disp != NULL) {

if (state.layerStack != draw[i].layerStack) {

disp->setLayerStack(state.layerStack);

}

if ((state.orientation != draw[i].orientation)

|| (state.viewport != draw[i].viewport)

|| (state.frame != draw[i].frame))

{

disp->setProjection(state.orientation,

state.viewport, state.frame);

}

if (state.width != draw[i].width || state.height != draw[i].height) {

disp->setDisplaySize(state.width, state.height);

}

}

}

}

//增加的显示设备

for (size\_t i=0 ; i<cc ; i++) {

if (draw.indexOfKey(curr.keyAt(i)) < 0) {

const DisplayDeviceState& state(curr[i]);

sp<DisplaySurface> dispSurface;

sp<IGraphicBufferProducer> producer;

sp<IGraphicBufferProducer> bqProducer;

sp<IGraphicBufferConsumer> bqConsumer;

BufferQueue::createBufferQueue(&bqProducer, &bqConsumer,

new GraphicBufferAlloc());

int32\_t hwcDisplayId = -1;

if (state.isVirtualDisplay()) {

if (state.surface != NULL) {

int width = 0;

int status = state.surface->query(

NATIVE\_WINDOW\_WIDTH, &width);

int height = 0;

status = state.surface->query(

NATIVE\_WINDOW\_HEIGHT, &height);

if (MAX\_VIRTUAL\_DISPLAY\_DIMENSION == 0 ||

(width <= MAX\_VIRTUAL\_DISPLAY\_DIMENSION &&

height <= MAX\_VIRTUAL\_DISPLAY\_DIMENSION)) {

hwcDisplayId = allocateHwcDisplayId(state.type);

}

sp<VirtualDisplaySurface> vds = new VirtualDisplaySurface(

\*mHwc, hwcDisplayId, state.surface,

bqProducer, bqConsumer, state.displayName);

dispSurface = vds;

producer = vds;

}

} else {

hwcDisplayId = allocateHwcDisplayId(state.type);

dispSurface = new FramebufferSurface(\*mHwc, state.type,

bqConsumer);

producer = bqProducer;

}

const wp<IBinder>& display(curr.keyAt(i));

if (dispSurface != NULL) {

sp<DisplayDevice> hw = new DisplayDevice(this,

state.type, hwcDisplayId,

mHwc->getFormat(hwcDisplayId), state.isSecure,

display, dispSurface, producer,

mRenderEngine->getEGLConfig());

hw->setLayerStack(state.layerStack);

hw->setProjection(state.orientation,

state.viewport, state.frame);

hw->setDisplayName(state.displayName);

mDisplays.add(display, hw);

if (state.isVirtualDisplay()) {

if (hwcDisplayId >= 0) {

mHwc->setVirtualDisplayProperties(hwcDisplayId,

hw->getWidth(), hw->getHeight(),

hw->getFormat());

}

} else {

mEventThread->onHotplugReceived(state.type, true);

}

}

}

}

}

}

if (transactionFlags & (eTraversalNeeded|eDisplayTransactionNeeded)) {

sp<const DisplayDevice> disp;

uint32\_t currentlayerStack = 0;

for (size\_t i=0; i<count; i++) {

const sp<Layer>& layer(currentLayers[i]);

uint32\_t layerStack = layer->getDrawingState().layerStack;

if (i==0 || currentlayerStack != layerStack) {

currentlayerStack = layerStack;

disp.clear();

for (size\_t dpy=0 ; dpy<mDisplays.size() ; dpy++) {

sp<const DisplayDevice> hw(mDisplays[dpy]);

if (hw->getLayerStack() == currentlayerStack) {

if (disp == NULL) {

disp = hw;

} else {

disp = NULL;

break;

}

}

}

}

if (disp == NULL) {

disp = getDefaultDisplayDevice();

}

layer->updateTransformHint(disp);

}

}

const LayerVector& layers(mDrawingState.layersSortedByZ);

// layers增加

if (currentLayers.size() > layers.size()) {

// layers have been added

mVisibleRegionsDirty = true;

}

// layers移除

if (mLayersRemoved) {

mLayersRemoved = false;

mVisibleRegionsDirty = true;

const size\_t count = layers.size();

for (size\_t i=0 ; i<count ; i++) {

const sp<Layer>& layer(layers[i]);

if (currentLayers.indexOf(layer) < 0) {

const Layer::State& s(layer->getDrawingState());

Region visibleReg = s.transform.transform(

Region(Rect(s.active.w, s.active.h)));

invalidateLayerStack(s.layerStack, visibleReg);

}

}

}

commitTransaction(); //【见小节2.4.4】

updateCursorAsync(); //【见小节2.4.5】

}

handleTransactionLocked方法的主要工作：

遍历所有Layer来执行其doTransaction方法；

处理显示设备的改变；

处理layers的改变；

提交transaction，并更新光标情况。

#### doTransaction

dsaf

[-> Layer.cpp]

uint32\_t Layer::doTransaction(uint32\_t flags) {

const Layer::State& s(getDrawingState()); //上次绘制状态

const Layer::State& c(getCurrentState()); //当前设置的绘制状态

const bool sizeChanged = (c.requested.w != s.requested.w) ||

(c.requested.h != s.requested.h);

if (sizeChanged) {

//当Layer尺寸改变，则调整surface缓存区大小

mSurfaceFlingerConsumer->setDefaultBufferSize(

c.requested.w, c.requested.h);

}

if (!isFixedSize()) {

const bool resizePending = (c.requested.w != c.active.w) ||

(c.requested.h != c.active.h);

//当Layer不是固定大小，且请求大小和实际大小不同。

if (resizePending && mSidebandStream == NULL) {

flags |= eDontUpdateGeometryState;

}

}

if (flags & eDontUpdateGeometryState) {

} else {

Layer::State& editCurrentState(getCurrentState());

editCurrentState.active = c.requested;

}

if (s.active != c.active) {

flags |= Layer::eVisibleRegion; // invalidate且重新计算可见区域

}

if (c.sequence != s.sequence) {

flags |= eVisibleRegion; // invalidate且重新计算可见区域

this->contentDirty = true;

const uint8\_t type = c.transform.getType();

mNeedsFiltering = (!c.transform.preserveRects() ||

(type >= Transform::SCALE));

}

commitTransaction(); //【见小节2.4.4】

return flags;

}

#### commitTransaction

f

void SurfaceFlinger::commitTransaction()

{

if (!mLayersPendingRemoval.isEmpty()) {

// Notify removed layers now that they can't be drawn from

for (size\_t i = 0; i < mLayersPendingRemoval.size(); i++) {

mLayersPendingRemoval[i]->onRemoved();

}

mLayersPendingRemoval.clear();

}

mAnimCompositionPending = mAnimTransactionPending;

mDrawingState = mCurrentState; //设置状态

mTransactionPending = false;

mAnimTransactionPending = false;

mTransactionCV.broadcast(); //唤醒

}

2.4.5 updateCursorAsync

void SurfaceFlinger::updateCursorAsync()

{

HWComposer& hwc(getHwComposer());

for (size\_t dpy=0 ; dpy<mDisplays.size() ; dpy++) {

sp<const DisplayDevice> hw(mDisplays[dpy]);

const int32\_t id = hw->getHwcDisplayId();

if (id < 0) {

continue;

}

const Vector< sp<Layer> >& currentLayers(

hw->getVisibleLayersSortedByZ());

const size\_t count = currentLayers.size();

HWComposer::LayerListIterator cur = hwc.begin(id);

const HWComposer::LayerListIterator end = hwc.end(id);

for (size\_t i=0 ; cur!=end && i<count ; ++i, ++cur) {

if (cur->getCompositionType() != HWC\_CURSOR\_OVERLAY) {

continue;

}

const sp<Layer>& layer(currentLayers[i]);

Rect cursorPos = layer->getPosition(hw);

hwc.setCursorPositionAsync(id, cursorPos);

break;

}

}

}

2.5 SF.handleMessageInvalidate

bool SurfaceFlinger::handleMessageInvalidate() {

ATRACE\_CALL();

return handlePageFlip(); //【见小节2.5.1】

}

#### 2.5.1 SF.handlePageFlip

bool SurfaceFlinger::handlePageFlip()

{

Region dirtyRegion;

bool visibleRegions = false;

const LayerVector& layers(mDrawingState.layersSortedByZ);

bool frameQueued = false;

Vector<Layer\*> layersWithQueuedFrames;

for (size\_t i = 0, count = layers.size(); i<count ; i++) {

const sp<Layer>& layer(layers[i]);

//判断Layer是否有需要更新的图层

if (layer->hasQueuedFrame()) {

frameQueued = true;

if (layer->shouldPresentNow(mPrimaryDispSync)) {

//将需要更新的图层放入layersWithQueuedFrames

layersWithQueuedFrames.push\_back(layer.get());

} else {

layer->useEmptyDamage();

}

} else {

layer->useEmptyDamage();

}

}

for (size\_t i = 0, count = layersWithQueuedFrames.size() ; i<count ; i++) {

Layer\* layer = layersWithQueuedFrames[i];

//调用latchBuffer来处理

const Region dirty(layer->latchBuffer(visibleRegions));

layer->useSurfaceDamage();

const Layer::State& s(layer->getDrawingState());

invalidateLayerStack(s.layerStack, dirty);

}

mVisibleRegionsDirty |= visibleRegions;

if (frameQueued && layersWithQueuedFrames.empty()) {

signalLayerUpdate();

}

return !layersWithQueuedFrames.empty();

}

#### 2.6 signalRefresh

void SurfaceFlinger::signalRefresh() {

mEventQueue.refresh();

}

处理过程类似于[小节2.3] signalLayerUpdate，最终会调用[小节1.1] SF.handleMessageRefresh()。

## rebuildLayerStacks

fg

void SurfaceFlinger::rebuildLayerStacks() {

if (CC\_UNLIKELY(mVisibleRegionsDirty)) {

mVisibleRegionsDirty = false;

invalidateHwcGeometry();

//每个显示屏中的所有可见图层列表

const LayerVector& layers(mDrawingState.layersSortedByZ);

for (size\_t dpy=0 ; dpy<mDisplays.size() ; dpy++) {

Region opaqueRegion;

Region dirtyRegion;

Vector< sp<Layer> > layersSortedByZ;

const sp<DisplayDevice>& hw(mDisplays[dpy]);

const Transform& tr(hw->getTransform());

const Rect bounds(hw->getBounds());

if (hw->isDisplayOn()) {

//计算每个Layer的可见区域

SurfaceFlinger::computeVisibleRegions(layers,

hw->getLayerStack(), dirtyRegion, opaqueRegion);

const size\_t count = layers.size();

for (size\_t i=0 ; i<count ; i++) {

const sp<Layer>& layer(layers[i]);

const Layer::State& s(layer->getDrawingState());

//当前layer和显示设备的layerStack相同

if (s.layerStack == hw->getLayerStack()) {

Region drawRegion(tr.transform(

layer->visibleNonTransparentRegion));

drawRegion.andSelf(bounds);

if (!drawRegion.isEmpty()) {

layersSortedByZ.add(layer);

}

}

}

}

hw->setVisibleLayersSortedByZ(layersSortedByZ);

hw->undefinedRegion.set(bounds);

hw->undefinedRegion.subtractSelf(tr.transform(opaqueRegion));

hw->dirtyRegion.orSelf(dirtyRegion);

}

}

}

重建所有显示屏的各个可见Layer列表，根据Z轴排序。

## setUpHWComposer

d

void SurfaceFlinger::setUpHWComposer() {

for (size\_t dpy=0 ; dpy<mDisplays.size() ; dpy++) {

bool dirty = !mDisplays[dpy]->getDirtyRegion(false).isEmpty();

bool empty = mDisplays[dpy]->getVisibleLayersSortedByZ().size() == 0;

bool wasEmpty = !mDisplays[dpy]->lastCompositionHadVisibleLayers;

bool mustRecompose = dirty && !(empty && wasEmpty);

mDisplays[dpy]->beginFrame(mustRecompose);

if (mustRecompose) {

mDisplays[dpy]->lastCompositionHadVisibleLayers = !empty;

}

}

HWComposer& hwc(getHwComposer());

if (hwc.initCheck() == NO\_ERROR) {

if (CC\_UNLIKELY(mHwWorkListDirty)) {

mHwWorkListDirty = false;

for (size\_t dpy=0 ; dpy<mDisplays.size() ; dpy++) {

sp<const DisplayDevice> hw(mDisplays[dpy]);

const int32\_t id = hw->getHwcDisplayId();

if (id >= 0) {

const Vector< sp<Layer> >& currentLayers(

hw->getVisibleLayersSortedByZ());

const size\_t count = currentLayers.size();

//在HWComposer中创建列表

if (hwc.createWorkList(id, count) == NO\_ERROR) {

HWComposer::LayerListIterator cur = hwc.begin(id);

const HWComposer::LayerListIterator end = hwc.end(id);

for (size\_t i=0 ; cur!=end && i<count ; ++i, ++cur) {

const sp<Layer>& layer(currentLayers[i]);

layer->setGeometry(hw, \*cur);

}

}

}

}

}

//设置每帧的数据

for (size\_t dpy=0 ; dpy<mDisplays.size() ; dpy++) {

sp<const DisplayDevice> hw(mDisplays[dpy]);

const int32\_t id = hw->getHwcDisplayId();

if (id >= 0) {

const Vector< sp<Layer> >& currentLayers(

hw->getVisibleLayersSortedByZ());

const size\_t count = currentLayers.size();

HWComposer::LayerListIterator cur = hwc.begin(id);

const HWComposer::LayerListIterator end = hwc.end(id);

for (size\_t i=0 ; cur!=end && i<count ; ++i, ++cur) {

/\* \* update the per-frame h/w composer data for each layer \* and build the transparent region of the FB /为每一个layer,更新每帧h/w合成器的数据 const sp<Layer>& layer(currentLayers[i]); layer->setPerFrameData(hw, \*cur); } } } //在每一个显示屏上，尝试使用cursor overlay for (size\_t dpy=0 ; dpy<mDisplays.size() ; dpy++) { sp<const DisplayDevice> hw(mDisplays[dpy]); const int32\_t id = hw->getHwcDisplayId(); if (id >= 0) { const Vector< sp<Layer> >& currentLayers( hw->getVisibleLayersSortedByZ()); const size\_t count = currentLayers.size(); HWComposer::LayerListIterator cur = hwc.begin(id); const HWComposer::LayerListIterator end = hwc.end(id); for (size\_t i=0 ; cur!=end && i<count ; ++i, ++cur) { const sp<Layer>& layer(currentLayers[i]); if (layer->isPotentialCursor()) { cur->setIsCursorLayerHint(); break; } } } } status\_t err = hwc.prepare(); //【见小节4.1】 for (size\_t dpy=0 ; dpy<mDisplays.size() ; dpy++) { sp<const DisplayDevice> hw(mDisplays[dpy]); hw->prepareFrame(hwc); } } }

4.1 HWC.prepare

status\_t HWComposer::prepare() {

Mutex::Autolock \_l(mDisplayLock);

for (size\_t i=0 ; i<mNumDisplays ; i++) {

DisplayData& disp(mDisplayData[i]);

if (disp.framebufferTarget) {

disp.framebufferTarget->compositionType = HWC\_FRAMEBUFFER\_TARGET;

}

mLists[i] = disp.list;

if (mLists[i]) {

if (hwcHasApiVersion(mHwc, HWC\_DEVICE\_API\_VERSION\_1\_3)) {

mLists[i]->outbuf = disp.outbufHandle;

mLists[i]->outbufAcquireFenceFd = -1;

} else if (hwcHasApiVersion(mHwc, HWC\_DEVICE\_API\_VERSION\_1\_1)) {

mLists[i]->dpy = (hwc\_display\_t)0xDEADBEEF;

mLists[i]->sur = (hwc\_surface\_t)0xDEADBEEF;

} else {

mLists[i]->dpy = EGL\_NO\_DISPLAY;

mLists[i]->sur = EGL\_NO\_SURFACE;

}

}

}

int err = mHwc->prepare(mHwc, mNumDisplays, mLists);

if (err == NO\_ERROR) {

for (size\_t i=0 ; i<mNumDisplays ; i++) {

DisplayData& disp(mDisplayData[i]);

disp.hasFbComp = false;

disp.hasOvComp = false;

if (disp.list) {

for (size\_t i=0 ; i<disp.list->numHwLayers ; i++) {

hwc\_layer\_1\_t& l = disp.list->hwLayers[i];

if (l.flags & HWC\_SKIP\_LAYER) {

l.compositionType = HWC\_FRAMEBUFFER;

}

if (l.compositionType == HWC\_FRAMEBUFFER) {

disp.hasFbComp = true;

}

if (l.compositionType == HWC\_OVERLAY) {

disp.hasOvComp = true;

}

if (l.compositionType == HWC\_CURSOR\_OVERLAY) {

disp.hasOvComp = true;

}

}

if (disp.list->numHwLayers == (disp.framebufferTarget ? 1 : 0)) {

disp.hasFbComp = true;

}

} else {

disp.hasFbComp = true;

}

}

}

return (status\_t)err;

}

五. doComposition

void SurfaceFlinger::doComposition() {

const bool repaintEverything = android\_atomic\_and(0, &mRepaintEverything);

for (size\_t dpy=0 ; dpy<mDisplays.size() ; dpy++) {

const sp<DisplayDevice>& hw(mDisplays[dpy]);

if (hw->isDisplayOn()) {

//将脏区域转换为此屏幕的坐标空间

const Region dirtyRegion(hw->getDirtyRegion(repaintEverything));

//如果需要，则重绘framebuffer【见小节5.1】

doDisplayComposition(hw, dirtyRegion);

hw->dirtyRegion.clear();

hw->flip(hw->swapRegion);

hw->swapRegion.clear();

}

//通知h/w我们已完成合成操作

hw->compositionComplete();

}

postFramebuffer(); //【见小节5.3】

}

5.1 doDisplayComposition

void SurfaceFlinger::doDisplayComposition(const sp<const DisplayDevice>& hw,

const Region& inDirtyRegion)

{

bool isHwcDisplay = hw->getHwcDisplayId() >= 0;

if (!isHwcDisplay && inDirtyRegion.isEmpty()) {

return;

}

Region dirtyRegion(inDirtyRegion);

//计算需要更新的区域

hw->swapRegion.orSelf(dirtyRegion);

uint32\_t flags = hw->getFlags();

if (flags & DisplayDevice::SWAP\_RECTANGLE) {

//矩形更新模式

dirtyRegion.set(hw->swapRegion.bounds());

} else {

if (flags & DisplayDevice::PARTIAL\_UPDATES) {

//部分更新模式

dirtyRegion.set(hw->swapRegion.bounds());

} else {

//更新区域为整个屏幕

dirtyRegion.set(hw->bounds());

hw->swapRegion = dirtyRegion;

}

}

if (CC\_LIKELY(!mDaltonize && !mHasColorMatrix)) {

if (!doComposeSurfaces(hw, dirtyRegion)) return;

} else {

RenderEngine& engine(getRenderEngine());

mat4 colorMatrix = mColorMatrix;

if (mDaltonize) {

colorMatrix = colorMatrix \* mDaltonizer();

}

mat4 oldMatrix = engine.setupColorTransform(colorMatrix);

doComposeSurfaces(hw, dirtyRegion);

engine.setupColorTransform(oldMatrix);

}

//更新交换区域，并清除脏区域

hw->swapRegion.orSelf(dirtyRegion);

//交换buffer，输出图像

hw->swapBuffers(getHwComposer());

}

5.2 doComposeSurfaces

bool SurfaceFlinger::doComposeSurfaces(const sp<const DisplayDevice>& hw, const Region& dirty)

{

RenderEngine& engine(getRenderEngine());

const int32\_t id = hw->getHwcDisplayId();

HWComposer& hwc(getHwComposer());

HWComposer::LayerListIterator cur = hwc.begin(id);

const HWComposer::LayerListIterator end = hwc.end(id);

bool hasGlesComposition = hwc.hasGlesComposition(id);

if (hasGlesComposition) {

if (!hw->makeCurrent(mEGLDisplay, mEGLContext)) {

eglMakeCurrent(mEGLDisplay, EGL\_NO\_SURFACE, EGL\_NO\_SURFACE, EGL\_NO\_CONTEXT);

if(!getDefaultDisplayDevice()->makeCurrent(mEGLDisplay, mEGLContext)) {

}

return false;

}

const bool hasHwcComposition = hwc.hasHwcComposition(id);

if (hasHwcComposition) {

engine.clearWithColor(0, 0, 0, 0);

} else {

//全屏

const Region bounds(hw->getBounds());

const Region letterbox(bounds.subtract(hw->getScissor()));

Region region(hw->undefinedRegion.merge(letterbox));

region.andSelf(dirty);

if (!region.isEmpty()) {

//能发生在SurfaceView

drawWormhole(hw, region);

}

}

if (hw->getDisplayType() != DisplayDevice::DISPLAY\_PRIMARY) {

const Rect& bounds(hw->getBounds());

const Rect& scissor(hw->getScissor());

if (scissor != bounds) {

const uint32\_t height = hw->getHeight();

engine.setScissor(scissor.left, height - scissor.bottom,

scissor.getWidth(), scissor.getHeight());

}

}

}

const Vector< sp<Layer> >& layers(hw->getVisibleLayersSortedByZ());

const size\_t count = layers.size();

const Transform& tr = hw->getTransform();

if (cur != end) {

//使用h/w composer

for (size\_t i=0 ; i<count && cur!=end ; ++i, ++cur) {

const sp<Layer>& layer(layers[i]);

const Region clip(dirty.intersect(tr.transform(layer->visibleRegion)));

if (!clip.isEmpty()) {

switch (cur->getCompositionType()) {

case HWC\_CURSOR\_OVERLAY:

case HWC\_OVERLAY: {

const Layer::State& state(layer->getDrawingState());

if ((cur->getHints() & HWC\_HINT\_CLEAR\_FB)

&& i

&& layer->isOpaque(state) && (state.alpha == 0xFF)

&& hasGlesComposition) {

layer->clearWithOpenGL(hw, clip);

}

break;

}

case HWC\_FRAMEBUFFER: {

layer->draw(hw, clip); //执行图像绘制

break;

}

case HWC\_FRAMEBUFFER\_TARGET: {

break;

}

}

}

layer->setAcquireFence(hw, \*cur);

}

} else {

//不使用h/w composer

for (size\_t i=0 ; i<count ; ++i) {

const sp<Layer>& layer(layers[i]);

const Region clip(dirty.intersect(

tr.transform(layer->visibleRegion)));

if (!clip.isEmpty()) {

layer->draw(hw, clip);

}

}

}

engine.disableScissor();

return true;

}

5.3 postFramebuffer

void SurfaceFlinger::postFramebuffer()

{

const nsecs\_t now = systemTime();

mDebugInSwapBuffers = now;

HWComposer& hwc(getHwComposer());

if (hwc.initCheck() == NO\_ERROR) {

if (!hwc.supportsFramebufferTarget()) {

getDefaultDisplayDevice()->makeCurrent(mEGLDisplay, mEGLContext);

}

hwc.commit(); //【见小节5.4】

}

getDefaultDisplayDevice()->makeCurrent(mEGLDisplay, mEGLContext);

for (size\_t dpy=0 ; dpy<mDisplays.size() ; dpy++) {

sp<const DisplayDevice> hw(mDisplays[dpy]);

const Vector< sp<Layer> >& currentLayers(hw->getVisibleLayersSortedByZ());

hw->onSwapBuffersCompleted(hwc);

const size\_t count = currentLayers.size();

int32\_t id = hw->getHwcDisplayId();

if (id >=0 && hwc.initCheck() == NO\_ERROR) {

HWComposer::LayerListIterator cur = hwc.begin(id);

const HWComposer::LayerListIterator end = hwc.end(id);

for (size\_t i = 0; cur != end && i < count; ++i, ++cur) {

currentLayers[i]->onLayerDisplayed(hw, &\*cur);

}

} else {

for (size\_t i = 0; i < count; i++) {

currentLayers[i]->onLayerDisplayed(hw, NULL);

}

}

}

mLastSwapBufferTime = systemTime() - now;

mDebugInSwapBuffers = 0;

uint32\_t flipCount = getDefaultDisplayDevice()->getPageFlipCount();

if (flipCount % LOG\_FRAME\_STATS\_PERIOD == 0) {

logFrameStats();

}

}

将数据写入framebuffer则完成物理屏幕的图像显示。

5.4 HWC.commit

status\_t HWComposer::commit() {

int err = NO\_ERROR;

if (mHwc) {

if (!hwcHasApiVersion(mHwc, HWC\_DEVICE\_API\_VERSION\_1\_1)) {

mLists[0]->dpy = eglGetCurrentDisplay();

mLists[0]->sur = eglGetCurrentSurface(EGL\_DRAW);

}

for (size\_t i=VIRTUAL\_DISPLAY\_ID\_BASE; i<mNumDisplays; i++) {

DisplayData& disp(mDisplayData[i]);

if (disp.outbufHandle) {

mLists[i]->outbuf = disp.outbufHandle;

mLists[i]->outbufAcquireFenceFd =

disp.outbufAcquireFence->dup();

}

}

//处理图像输出到FrameBuffer

err = mHwc->set(mHwc, mNumDisplays, mLists);

for (size\_t i=0 ; i<mNumDisplays ; i++) {

DisplayData& disp(mDisplayData[i]);

disp.lastDisplayFence = disp.lastRetireFence;

disp.lastRetireFence = Fence::NO\_FENCE;

if (disp.list) {

if (disp.list->retireFenceFd != -1) {

disp.lastRetireFence = new Fence(disp.list->retireFenceFd);

disp.list->retireFenceFd = -1;

}

disp.list->flags &= ~HWC\_GEOMETRY\_CHANGED;

}

}

}

return (status\_t)err;

}

六. postComposition

void SurfaceFlinger::postComposition()

{

const LayerVector& layers(mDrawingState.layersSortedByZ);

const size\_t count = layers.size();

for (size\_t i=0 ; i<count ; i++) {

layers[i]->onPostComposition();

}

const HWComposer& hwc = getHwComposer();

sp<Fence> presentFence = hwc.getDisplayFence(HWC\_DISPLAY\_PRIMARY);

if (presentFence->isValid()) {

if (mPrimaryDispSync.addPresentFence(presentFence)) {

enableHardwareVsync();

} else {

disableHardwareVsync(false);

}

}

const sp<const DisplayDevice> hw(getDefaultDisplayDevice());

if (kIgnorePresentFences) {

if (hw->isDisplayOn()) {

enableHardwareVsync();

}

}

if (mAnimCompositionPending) {

mAnimCompositionPending = false;

if (presentFence->isValid()) {

mAnimFrameTracker.setActualPresentFence(presentFence);

} else {

nsecs\_t presentTime = hwc.getRefreshTimestamp(HWC\_DISPLAY\_PRIMARY);

mAnimFrameTracker.setActualPresentTime(presentTime);

}

mAnimFrameTracker.advanceFrame();

}

if (hw->getPowerMode() == HWC\_POWER\_MODE\_OFF) {

return;

}

nsecs\_t currentTime = systemTime();

if (mHasPoweredOff) {

mHasPoweredOff = false;

} else {

nsecs\_t period = mPrimaryDispSync.getPeriod();

nsecs\_t elapsedTime = currentTime - mLastSwapTime;

size\_t numPeriods = static\_cast<size\_t>(elapsedTime / period);

if (numPeriods < NUM\_BUCKETS - 1) {

mFrameBuckets[numPeriods] += elapsedTime;

} else {

mFrameBuckets[NUM\_BUCKETS - 1] += elapsedTime;

}

mTotalTime += elapsedTime;

}

mLastSwapTime = currentTime;

}

### HWC.prepare

F

status\_t HWComposer::prepare() {

Mutex::Autolock \_l(mDisplayLock);

for (size\_t i=0 ; i<mNumDisplays ; i++) {

DisplayData& disp(mDisplayData[i]);

if (disp.framebufferTarget) {

disp.framebufferTarget->compositionType = HWC\_FRAMEBUFFER\_TARGET;

}

mLists[i] = disp.list;

if (mLists[i]) {

if (hwcHasApiVersion(mHwc, HWC\_DEVICE\_API\_VERSION\_1\_3)) {

mLists[i]->outbuf = disp.outbufHandle;

mLists[i]->outbufAcquireFenceFd = -1;

} else if (hwcHasApiVersion(mHwc, HWC\_DEVICE\_API\_VERSION\_1\_1)) {

mLists[i]->dpy = (hwc\_display\_t)0xDEADBEEF;

mLists[i]->sur = (hwc\_surface\_t)0xDEADBEEF;

} else {

mLists[i]->dpy = EGL\_NO\_DISPLAY;

mLists[i]->sur = EGL\_NO\_SURFACE;

}

}

}

int err = mHwc->prepare(mHwc, mNumDisplays, mLists);

if (err == NO\_ERROR) {

for (size\_t i=0 ; i<mNumDisplays ; i++) {

DisplayData& disp(mDisplayData[i]);

disp.hasFbComp = false;

disp.hasOvComp = false;

if (disp.list) {

for (size\_t i=0 ; i<disp.list->numHwLayers ; i++) {

hwc\_layer\_1\_t& l = disp.list->hwLayers[i];

if (l.flags & HWC\_SKIP\_LAYER) {

l.compositionType = HWC\_FRAMEBUFFER;

}

if (l.compositionType == HWC\_FRAMEBUFFER) {

disp.hasFbComp = true;

}

if (l.compositionType == HWC\_OVERLAY) {

disp.hasOvComp = true;

}

if (l.compositionType == HWC\_CURSOR\_OVERLAY) {

disp.hasOvComp = true;

}

}

if (disp.list->numHwLayers == (disp.framebufferTarget ? 1 : 0)) {

disp.hasFbComp = true;

}

} else {

disp.hasFbComp = true;

}

}

}

return (status\_t)err;

}

## doComposition

df

void SurfaceFlinger::doComposition() {

const bool repaintEverything = android\_atomic\_and(0, &mRepaintEverything);

for (size\_t dpy=0 ; dpy<mDisplays.size() ; dpy++) {

const sp<DisplayDevice>& hw(mDisplays[dpy]);

if (hw->isDisplayOn()) {

//将脏区域转换为此屏幕的坐标空间

const Region dirtyRegion(hw->getDirtyRegion(repaintEverything));

//如果需要，则重绘framebuffer【见小节5.1】

doDisplayComposition(hw, dirtyRegion);

hw->dirtyRegion.clear();

hw->flip(hw->swapRegion);

hw->swapRegion.clear();

}

//通知h/w我们已完成合成操作

hw->compositionComplete();

}

postFramebuffer(); //【见小节5.3】

}

### doDisplayComposition

ff

void SurfaceFlinger::doDisplayComposition(const sp<const DisplayDevice>& hw,

const Region& inDirtyRegion)

{

bool isHwcDisplay = hw->getHwcDisplayId() >= 0;

if (!isHwcDisplay && inDirtyRegion.isEmpty()) {

return;

}

Region dirtyRegion(inDirtyRegion);

//计算需要更新的区域

hw->swapRegion.orSelf(dirtyRegion);

uint32\_t flags = hw->getFlags();

if (flags & DisplayDevice::SWAP\_RECTANGLE) {

//矩形更新模式

dirtyRegion.set(hw->swapRegion.bounds());

} else {

if (flags & DisplayDevice::PARTIAL\_UPDATES) {

//部分更新模式

dirtyRegion.set(hw->swapRegion.bounds());

} else {

//更新区域为整个屏幕

dirtyRegion.set(hw->bounds());

hw->swapRegion = dirtyRegion;

}

}

if (CC\_LIKELY(!mDaltonize && !mHasColorMatrix)) {

if (!doComposeSurfaces(hw, dirtyRegion)) return;

} else {

RenderEngine& engine(getRenderEngine());

mat4 colorMatrix = mColorMatrix;

if (mDaltonize) {

colorMatrix = colorMatrix \* mDaltonizer();

}

mat4 oldMatrix = engine.setupColorTransform(colorMatrix);

doComposeSurfaces(hw, dirtyRegion);

engine.setupColorTransform(oldMatrix);

}

//更新交换区域，并清除脏区域

hw->swapRegion.orSelf(dirtyRegion);

//交换buffer，输出图像

hw->swapBuffers(getHwComposer());

}

### doComposeSurfaces

ff

bool SurfaceFlinger::doComposeSurfaces(const sp<const DisplayDevice>& hw, const Region& dirty)

{

RenderEngine& engine(getRenderEngine());

const int32\_t id = hw->getHwcDisplayId();

HWComposer& hwc(getHwComposer());

HWComposer::LayerListIterator cur = hwc.begin(id);

const HWComposer::LayerListIterator end = hwc.end(id);

bool hasGlesComposition = hwc.hasGlesComposition(id);

if (hasGlesComposition) {

if (!hw->makeCurrent(mEGLDisplay, mEGLContext)) {

eglMakeCurrent(mEGLDisplay, EGL\_NO\_SURFACE, EGL\_NO\_SURFACE, EGL\_NO\_CONTEXT);

if(!getDefaultDisplayDevice()->makeCurrent(mEGLDisplay, mEGLContext)) {

}

return false;

}

const bool hasHwcComposition = hwc.hasHwcComposition(id);

if (hasHwcComposition) {

engine.clearWithColor(0, 0, 0, 0);

} else {

//全屏

const Region bounds(hw->getBounds());

const Region letterbox(bounds.subtract(hw->getScissor()));

Region region(hw->undefinedRegion.merge(letterbox));

region.andSelf(dirty);

if (!region.isEmpty()) {

//能发生在SurfaceView

drawWormhole(hw, region);

}

}

if (hw->getDisplayType() != DisplayDevice::DISPLAY\_PRIMARY) {

const Rect& bounds(hw->getBounds());

const Rect& scissor(hw->getScissor());

if (scissor != bounds) {

const uint32\_t height = hw->getHeight();

engine.setScissor(scissor.left, height - scissor.bottom,

scissor.getWidth(), scissor.getHeight());

}

}

}

const Vector< sp<Layer> >& layers(hw->getVisibleLayersSortedByZ());

const size\_t count = layers.size();

const Transform& tr = hw->getTransform();

if (cur != end) {

//使用h/w composer

for (size\_t i=0 ; i<count && cur!=end ; ++i, ++cur) {

const sp<Layer>& layer(layers[i]);

const Region clip(dirty.intersect(tr.transform(layer->visibleRegion)));

if (!clip.isEmpty()) {

switch (cur->getCompositionType()) {

case HWC\_CURSOR\_OVERLAY:

case HWC\_OVERLAY: {

const Layer::State& state(layer->getDrawingState());

if ((cur->getHints() & HWC\_HINT\_CLEAR\_FB)

&& i

&& layer->isOpaque(state) && (state.alpha == 0xFF)

&& hasGlesComposition) {

layer->clearWithOpenGL(hw, clip);

}

break;

}

case HWC\_FRAMEBUFFER: {

layer->draw(hw, clip); //执行图像绘制

break;

}

case HWC\_FRAMEBUFFER\_TARGET: {

break;

}

}

}

layer->setAcquireFence(hw, \*cur);

}

} else {

//不使用h/w composer

for (size\_t i=0 ; i<count ; ++i) {

const sp<Layer>& layer(layers[i]);

const Region clip(dirty.intersect(

tr.transform(layer->visibleRegion)));

if (!clip.isEmpty()) {

layer->draw(hw, clip);

}

}

}

engine.disableScissor();

return true;

}

### postFramebuffer

g

void SurfaceFlinger::postFramebuffer()

{

const nsecs\_t now = systemTime();

mDebugInSwapBuffers = now;

HWComposer& hwc(getHwComposer());

if (hwc.initCheck() == NO\_ERROR) {

if (!hwc.supportsFramebufferTarget()) {

getDefaultDisplayDevice()->makeCurrent(mEGLDisplay, mEGLContext);

}

hwc.commit(); //【见小节5.4】

}

getDefaultDisplayDevice()->makeCurrent(mEGLDisplay, mEGLContext);

for (size\_t dpy=0 ; dpy<mDisplays.size() ; dpy++) {

sp<const DisplayDevice> hw(mDisplays[dpy]);

const Vector< sp<Layer> >& currentLayers(hw->getVisibleLayersSortedByZ());

hw->onSwapBuffersCompleted(hwc);

const size\_t count = currentLayers.size();

int32\_t id = hw->getHwcDisplayId();

if (id >=0 && hwc.initCheck() == NO\_ERROR) {

HWComposer::LayerListIterator cur = hwc.begin(id);

const HWComposer::LayerListIterator end = hwc.end(id);

for (size\_t i = 0; cur != end && i < count; ++i, ++cur) {

currentLayers[i]->onLayerDisplayed(hw, &\*cur);

}

} else {

for (size\_t i = 0; i < count; i++) {

currentLayers[i]->onLayerDisplayed(hw, NULL);

}

}

}

mLastSwapBufferTime = systemTime() - now;

mDebugInSwapBuffers = 0;

uint32\_t flipCount = getDefaultDisplayDevice()->getPageFlipCount();

if (flipCount % LOG\_FRAME\_STATS\_PERIOD == 0) {

logFrameStats();

}

}

将数据写入framebuffer则完成物理屏幕的图像显示。

### HWC.commit

R

status\_t HWComposer::commit() {

int err = NO\_ERROR;

if (mHwc) {

if (!hwcHasApiVersion(mHwc, HWC\_DEVICE\_API\_VERSION\_1\_1)) {

mLists[0]->dpy = eglGetCurrentDisplay();

mLists[0]->sur = eglGetCurrentSurface(EGL\_DRAW);

}

for (size\_t i=VIRTUAL\_DISPLAY\_ID\_BASE; i<mNumDisplays; i++) {

DisplayData& disp(mDisplayData[i]);

if (disp.outbufHandle) {

mLists[i]->outbuf = disp.outbufHandle;

mLists[i]->outbufAcquireFenceFd =

disp.outbufAcquireFence->dup();

}

}

//处理图像输出到FrameBuffer

err = mHwc->set(mHwc, mNumDisplays, mLists);

for (size\_t i=0 ; i<mNumDisplays ; i++) {

DisplayData& disp(mDisplayData[i]);

disp.lastDisplayFence = disp.lastRetireFence;

disp.lastRetireFence = Fence::NO\_FENCE;

if (disp.list) {

if (disp.list->retireFenceFd != -1) {

disp.lastRetireFence = new Fence(disp.list->retireFenceFd);

disp.list->retireFenceFd = -1;

}

disp.list->flags &= ~HWC\_GEOMETRY\_CHANGED;

}

}

}

return (status\_t)err;

}

## postComposition

fg

void SurfaceFlinger::postComposition()

{

const LayerVector& layers(mDrawingState.layersSortedByZ);

const size\_t count = layers.size();

for (size\_t i=0 ; i<count ; i++) {

layers[i]->onPostComposition();

}

const HWComposer& hwc = getHwComposer();

sp<Fence> presentFence = hwc.getDisplayFence(HWC\_DISPLAY\_PRIMARY);

if (presentFence->isValid()) {

if (mPrimaryDispSync.addPresentFence(presentFence)) {

enableHardwareVsync();

} else {

disableHardwareVsync(false);

}

}

const sp<const DisplayDevice> hw(getDefaultDisplayDevice());

if (kIgnorePresentFences) {

if (hw->isDisplayOn()) {

enableHardwareVsync();

}

}

if (mAnimCompositionPending) {

mAnimCompositionPending = false;

if (presentFence->isValid()) {

mAnimFrameTracker.setActualPresentFence(presentFence);

} else {

nsecs\_t presentTime = hwc.getRefreshTimestamp(HWC\_DISPLAY\_PRIMARY);

mAnimFrameTracker.setActualPresentTime(presentTime);

}

mAnimFrameTracker.advanceFrame();

}

if (hw->getPowerMode() == HWC\_POWER\_MODE\_OFF) {

return;

}

nsecs\_t currentTime = systemTime();

if (mHasPoweredOff) {

mHasPoweredOff = false;

} else {

nsecs\_t period = mPrimaryDispSync.getPeriod();

nsecs\_t elapsedTime = currentTime - mLastSwapTime;

size\_t numPeriods = static\_cast<size\_t>(elapsedTime / period);

if (numPeriods < NUM\_BUCKETS - 1) {

mFrameBuckets[numPeriods] += elapsedTime;

} else {

mFrameBuckets[NUM\_BUCKETS - 1] += elapsedTime;

}

mTotalTime += elapsedTime;

}

mLastSwapTime = currentTime;

}

## 小结

J

1. preComposition：根据上次绘制的图层中是否有更新，来决定是否执行invalidate过程；
2. rebuildLayerStacks： 重建每个显示屏的所有可见Layer列表；
3. setUpHWComposer：更新HWComposer的图层
4. doComposition：合成所有图层的图像
5. postComposition：回调每个layer的onPostComposition。

[SurfaceFlinger绘图篇](http://gityuan.com/2017/02/18/surface_flinger_2/)

# Choreographer原理

在Android4.1之后增加了Choreographer机制，用于同Vsync机制配合，实现统一调度界面绘图.

前面两篇文章介绍了SurfaceFlinger原理，讲述了SurfaceFlinger的启动过程，绘制过程，以及Vsync处理过程。 本文再介绍一下Choreographer的启动与Vsync处理过程。

## 前言

为什么叫舞蹈编导，因为舞蹈是由节奏的，节奏是每个点位动作的快慢控制，跳舞时节奏很重要，编舞者控制节奏。视图刷新也是如此，不是说你想刷就能刷，一切要按照底层信号要求的节奏来。

### 理解屏幕刷新频率

**刷新频率：**每秒钟刷新屏幕的次数，从缓存中取出每一帧，显示到屏幕上的速度。  
**帧率：**GPU/CPU生成每一帧画面图像，存入缓存中的速度。  
一般情况下帧率是大于刷新频率的，每个设备的刷新频率固定，与硬件相关，若帧率是刷新率的两倍，两张图像画面，只能有一个现实到屏幕上，也就是说生产画面的速度要大于显示画面的速度。**典型的生产者-消费者模式。**  
**屏幕刷新过程：**从左到右刷新一行，然后垂直刷新，再一行..，直到屏幕刷新完毕。每一次刷新，都是这一个过程，刷新一次后，**中间有一个期间**，刷新率太高，每秒60帧左右，人眼无法感知。  
**tearing：**若帧率大于刷新率，会导致在刷新前一帧还未开始时，缓存已经被新一帧覆盖一部分，那么在刷新前一帧时，现显示的一部分是新一帧内容。会导致**画面前后帧上下重叠**。这种情况技术上称之tearing，画面撕裂的意思。  
**解决方案：**增加**缓存到两个缓存**，CPU生成帧存入一个缓存A，屏幕取出帧存入另一个缓存B，解决了一个缓存导致生产者与消费者不同步的问题。增加**Vsync同步信号**，负责调度将A缓存拷贝到B缓存，显示取出的就是一个完整帧的画面。Vsync信号在一帧刷新完的中间期间产生，A到B的复制(交换地址即可)，进入下一次刷新，并通知生产者gpu/cpu继续生产帧，只有收到Vsync信号，生产者才会生产帧。因此，可使帧率与刷新率保持同步，消耗一次才生成一次。  
**掉帧：**若Vsync信号发出时，A缓存正在被生产者**锁住生产**，gpu绘制生产帧时间超过信号发出时刻一点，此时不会复制。导致B缓存仍是老帧，下一次周期与前一次**刷新相同的帧**。当gpu生产结束后，此时刷新老数据的刷新周期中，还没有Vsync信号，则**gpu空闲**。  
**掉帧解决方案：**再增加一个缓存到**三个缓存**。当有缓存锁住时复制**他前面的**上一次被锁住的另一个缓存。

App进程若要通过gpu实现图像绘制，需要在接收到Vsync信号的条件下进行，因此，App进程访问SurfaceFlinger进程获取这个信号，再进行gpu绘制。

Choreographer就是负责获取Vsync同步信号并控制App线程(主线程)完成图像绘制的类。

在Android系统中主要是主线程进行UI绘制，其他线程也可以绘制，比如SurfaceView，本文以主线程UI绘制进行介绍。

## Choreographer启动流程

在Activity启动过程，执行完onResume后，会调用Activity.makeVisible()，然后再调用到addView()， 层层调用会进入如下方法：

public ViewRootImpl(Context context, Display display) {

//这里便出现获取Choreographer实例【见小节2.1】

mChoreographer = Choreographer.getInstance();

}

其他的调用如下：

android/app/ActivityThread.java

**public void** handleResumeActivity(IBinder token, **boolean** finalStateRequest, **boolean** isForward,  
 String reason) {

}

**if** (r.**activity**.**mVisibleFromClient**) {  
 r.**activity**.makeVisible();  
}

android/app/Activity.java

**void** makeVisible() {  
 **if** (!**mWindowAdded**) {  
 ViewManager wm = getWindowManager();  
 wm.addView(**mDecor**, getWindow().getAttributes());  
 **mWindowAdded** = **true**;  
 }  
 **mDecor**.setVisibility(View.***VISIBLE***);  
}

### getInstance

[-> Choreographer.java]

public static Choreographer getInstance() {

return sThreadInstance.get(); //线程单例模式

}

**private static final** ThreadLocal<Choreographer> ***sThreadInstance*** =  
 **new** ThreadLocal<Choreographer>() {  
 @Override  
 **protected** Choreographer initialValue() {  
 Looper looper = Looper.*myLooper*();//获取当前线程的Looper  
 **if** (looper == **null**) {  
 **throw new** IllegalStateException(**"The current thread must have a looper!"**);  
 }  
 Choreographer choreographer = **new** Choreographer(looper, ***VSYNC\_SOURCE\_APP***);  
 **if** (looper == Looper.*getMainLooper*()) {  
 *mMainInstance* = choreographer;  
 }  
 **return** choreographer;  
 }  
};

当前所在线程为UI线程，也就是常说的主线程。线程本地存储ThreadLocal变量，Choreographer类型，在主线程中初始化变量时，**创建Choreographer对象，绑定主线程Looper。**

同一个App的每个窗体旗下ViewRootImpl使用的同一个Choregrapher对象，他控制者整个App中大部分视图的绘制节奏。每个线程中保存一个Choreographer实例对象。

### 创建Choreographer

[-> Choreographer.java]

private Choreographer(Looper looper) {

mLooper = looper;

//创建Handler对象【见小节2.3】

mHandler = new FrameHandler(looper);

//创建用于接收VSync信号的对象【见小节2.4】

mDisplayEventReceiver = USE\_VSYNC ? new FrameDisplayEventReceiver(looper) : null;

mLastFrameTimeNanos = Long.MIN\_VALUE;

mFrameIntervalNanos = (long)(1000000000 / getRefreshRate());

//创建回调对象

mCallbackQueues = new CallbackQueue[CALLBACK\_LAST + 1];

for (int i = 0; i <= CALLBACK\_LAST; i++) {

mCallbackQueues[i] = new CallbackQueue();

}

}

mLastFrameTimeNanos：是指上一次帧绘制时间点；

mFrameIntervalNanos：帧间时长，一般等于16.7ms.

### FrameHandler

[-> Choreographer.java ::FrameHandler]

private final class FrameHandler extends Handler {

public FrameHandler(Looper looper) {

super(looper);

}

@Override

public void handleMessage(Message msg) {

switch (msg.what) {

case MSG\_DO\_FRAME:

doFrame(System.nanoTime(), 0);

break;

case MSG\_DO\_SCHEDULE\_VSYNC:

doScheduleVsync();

break;

case MSG\_DO\_SCHEDULE\_CALLBACK:

doScheduleCallback(msg.arg1);

break;

}

}

}

### 创建FrameDisplayEventReceiver

[-> Choreographer.java ::FrameDisplayEventReceiver]

private final class FrameDisplayEventReceiver extends DisplayEventReceiver implements Runnable {

public FrameDisplayEventReceiver(Looper looper) {

super(looper); //【见小节2.4.1】

}

}

#### DisplayEventReceiver

[-> DisplayEventReceiver.java]

public DisplayEventReceiver(Looper looper) {

mMessageQueue = looper.getQueue(); //获取主线程的消息队列

//【见小节2.4.2】

mReceiverPtr = nativeInit(new WeakReference<DisplayEventReceiver>(this), mMessageQueue);

}

经过JNI调用进入如下Native方法。

#### nativeInit

[-> android\_view\_DisplayEventReceiver.cpp]

static jlong nativeInit(JNIEnv\* env, jclass clazz, jobject receiverWeak, jobject messageQueueObj) {

sp<MessageQueue> messageQueue = android\_os\_MessageQueue\_getMessageQueue(env, messageQueueObj);

...

//【见小节2.4.3】

sp<NativeDisplayEventReceiver> receiver = new NativeDisplayEventReceiver(env,

receiverWeak, messageQueue);

//【见小节2.4.4】

status\_t status = receiver->initialize();

...

//获取DisplayEventReceiver对象的引用

receiver->incStrong(gDisplayEventReceiverClassInfo.clazz);

return reinterpret\_cast<jlong>(receiver.get());

}

#### 创建NativeDisplayEventReceiver

[-> android\_view\_DisplayEventReceiver.cpp]

NativeDisplayEventReceiver::NativeDisplayEventReceiver(JNIEnv\* env,

jobject receiverWeak, const sp<MessageQueue>& messageQueue) :

mReceiverWeakGlobal(env->NewGlobalRef(receiverWeak)),

mMessageQueue(messageQueue), mWaitingForVsync(false) {

ALOGV("receiver %p ~ Initializing display event receiver.", this);

}

NativeDisplayEventReceiver继承于LooperCallback对象，此处mReceiverWeakGlobal记录的是Java层 DisplayEventReceiver对象的全局引用。

#### initialize

[-> android\_view\_DisplayEventReceiver.cpp]

status\_t NativeDisplayEventReceiver::initialize() {

//mReceiver为DisplayEventReceiver类型

status\_t result = mReceiver.initCheck();

...

//监听mReceiver的所获取的文件句柄。

int rc = mMessageQueue->getLooper()->addFd(mReceiver.getFd(), 0, Looper::EVENT\_INPUT, this, NULL);

...

return OK;

}

此处跟文章[SurfaceFlinger原理(一)](http://gityuan.com/2017/02/11/surface_flinger/)的【小节2.8】的监听原理一样。 监听mReceiver的所获取的文件句柄，一旦有数据到来，则回调this(此处NativeDisplayEventReceiver)中所复写LooperCallback对象的 **handleEvent**。

### handleEvent

[-> android\_view\_DisplayEventReceiver.cpp]

int NativeDisplayEventReceiver::handleEvent(int receiveFd, int events, void\* data) {

...

nsecs\_t vsyncTimestamp;

int32\_t vsyncDisplayId;

uint32\_t vsyncCount;

//清除所有的pending事件，只保留最后一次vsync【见小节2.5.1】

if (processPendingEvents(&vsyncTimestamp, &vsyncDisplayId, &vsyncCount)) {

mWaitingForVsync = false;

//分发Vsync【见小节2.5.2】

dispatchVsync(vsyncTimestamp, vsyncDisplayId, vsyncCount);

}

return 1;

}

#### processPendingEvents

bool NativeDisplayEventReceiver::processPendingEvents(

nsecs\_t\* outTimestamp, int32\_t\* outId, uint32\_t\* outCount) {

bool gotVsync = false;

DisplayEventReceiver::Event buf[EVENT\_BUFFER\_SIZE];

ssize\_t n;

while ((n = mReceiver.getEvents(buf, EVENT\_BUFFER\_SIZE)) > 0) {

for (ssize\_t i = 0; i < n; i++) {

const DisplayEventReceiver::Event& ev = buf[i];

switch (ev.header.type) {

case DisplayEventReceiver::DISPLAY\_EVENT\_VSYNC:

gotVsync = true; //获取VSync信号

\*outTimestamp = ev.header.timestamp;

\*outId = ev.header.id;

\*outCount = ev.vsync.count;

break;

case DisplayEventReceiver::DISPLAY\_EVENT\_HOTPLUG:

dispatchHotplug(ev.header.timestamp, ev.header.id, ev.hotplug.connected);

break;

default:

break;

}

}

}

return gotVsync;

}

遍历所有的事件，当有多个VSync事件到来，则只关注最近一次的事件。

#### dispatchVsync

void NativeDisplayEventReceiver::dispatchVsync(nsecs\_t timestamp, int32\_t id, uint32\_t count) {

JNIEnv\* env = AndroidRuntime::getJNIEnv();

ScopedLocalRef<jobject> receiverObj(env, jniGetReferent(env, mReceiverWeakGlobal));

if (receiverObj.get()) {

//【见小节2.6】

env->CallVoidMethod(receiverObj.get(),

gDisplayEventReceiverClassInfo.dispatchVsync, timestamp, id, count);

}

mMessageQueue->raiseAndClearException(env, "dispatchVsync");

}

此处调用到Java层的DisplayEventReceiver对象的dispatchVsync()方法，接下来进入Java层。

### dispatchVsync

[-> DisplayEventReceiver.java]

private void (long timestampNanos, int builtInDisplayId, int frame) {

//【见小节2.7】

onVsync(timestampNanos, builtInDisplayId, frame);

}

再回到【小节2.2】，可知Choreographer对象实例化的过程，创建的对象是DisplayEventReceiver子类 FrameDisplayEventReceiver对象，接下来进入该对象。

### onVsync

[-> Choreographer.java ::FrameDisplayEventReceiver]

private final class FrameDisplayEventReceiver extends DisplayEventReceiver implements Runnable {

private boolean mHavePendingVsync;

private long mTimestampNanos;

private int mFrame;

@Override

public void onVsync(long timestampNanos, int builtInDisplayId, int frame) {

//忽略来自第二显示屏的Vsync

if (builtInDisplayId != SurfaceControl.BUILT\_IN\_DISPLAY\_ID\_MAIN) {

scheduleVsync();

return;

}

...

mTimestampNanos = timestampNanos;

mFrame = frame;

//该消息的callback为当前对象FrameDisplayEventReceiver

Message msg = Message.obtain(mHandler, this);

msg.setAsynchronous(true);

//此处mHandler为FrameHandler

mHandler.sendMessageAtTime(msg, timestampNanos / TimeUtils.NANOS\_PER\_MS);

}

@Override

public void run() {

mHavePendingVsync = false;

doFrame(mTimestampNanos, mFrame); //【见小节2.8】

}

}

可见onVsync()过程是通过FrameHandler向主线程Looper发送了一个自带callback的消息 callback为FrameDisplayEventReceiver。 当主线程Looper执行到该消息时，则调用FrameDisplayEventReceiver.run()方法，紧接着便是调用doFrame，如下：

### doFrame

void doFrame(long frameTimeNanos, int frame) {

final long startNanos;

synchronized (mLock) {

if (!mFrameScheduled) {

return; // mFrameScheduled=false，则直接返回。

}

long intendedFrameTimeNanos = frameTimeNanos; //原本计划的绘帧时间点

startNanos = System.nanoTime();

final long jitterNanos = startNanos - frameTimeNanos;

if (jitterNanos >= mFrameIntervalNanos) {

final long skippedFrames = jitterNanos / mFrameIntervalNanos;

//当掉帧个数超过30，则输出相应log

if (skippedFrames >= SKIPPED\_FRAME\_WARNING\_LIMIT) {

Log.i(TAG, "Skipped " + skippedFrames + " frames! "

+ "The application may be doing too much work on its main thread.");

}

final long lastFrameOffset = jitterNanos % mFrameIntervalNanos;

frameTimeNanos = startNanos - lastFrameOffset; //对齐帧的时间间隔

}

if (frameTimeNanos < mLastFrameTimeNanos) {

scheduleVsyncLocked();

return;

}

mFrameInfo.setVsync(intendedFrameTimeNanos, frameTimeNanos);

mFrameScheduled = false;

mLastFrameTimeNanos = frameTimeNanos;

}

try {

Trace.traceBegin(Trace.TRACE\_TAG\_VIEW, "Choreographer#doFrame");

mFrameInfo.markInputHandlingStart();

**doCallbacks(**Choreographer.CALLBACK\_INPUT, frameTimeNanos);

mFrameInfo.markAnimationsStart();

doCallbacks(Choreographer.CALLBACK\_ANIMATION, frameTimeNanos);

mFrameInfo.markPerformTraversalsStart();

doCallbacks(Choreographer.CALLBACK\_TRAVERSAL, frameTimeNanos);

doCallbacks(Choreographer.CALLBACK\_COMMIT, frameTimeNanos);

} finally {

Trace.traceEnd(Trace.TRACE\_TAG\_VIEW);

}

}

每调用一次scheduleFrameLocked()，则mFrameScheduled为true，能执行一次 doFrame()操作。最终有4个回调方法，依次为如下：

INPUT：输入事件

ANIMATION：动画

TRAVERSAL：窗口刷新

COMMIT

#### doCallbacks

[-> Choreographer.java]

void doCallbacks(int callbackType, long frameTimeNanos) {

CallbackRecord callbacks;

synchronized (mLock) {

final long now = System.nanoTime();

// 从队列查找相应类型的CallbackRecord对象【见小节2.8.2】

callbacks = mCallbackQueues[callbackType].**extractDueCallbacksLocked**(

now / TimeUtils.NANOS\_PER\_MS);

if (callbacks == null) {

return; //当队列为空，则直接返回

}

mCallbacksRunning = true;

if (callbackType == Choreographer.CALLBACK\_COMMIT) {

final long jitterNanos = now - frameTimeNanos;

//当commit类型回调执行的时间点超过2帧，则更新mLastFrameTimeNanos。

if (jitterNanos >= 2 \* mFrameIntervalNanos) {

final long lastFrameOffset = jitterNanos % mFrameIntervalNanos

+ mFrameIntervalNanos;

frameTimeNanos = now - lastFrameOffset;

mLastFrameTimeNanos = frameTimeNanos;

}

}

}

try {

for (CallbackRecord c = callbacks; c != null; c = c.next) {

c.run(frameTimeNanos); //【见小节2.9】

}

} finally {

... //回收callbacks，加入mCallbackPool对象池

}

}

#### extractDueCallbacksLocked

[-> Choreographer.java ::CallbackQueue]

private final class CallbackQueue {

public CallbackRecord extractDueCallbacksLocked(long now) {

CallbackRecord callbacks = mHead;

//当队列头部的callbacks对象为空，或者执行时间还没到达，则直接返回

if (callbacks == null || callbacks.dueTime > now) {

return null;

}

CallbackRecord last = callbacks;

CallbackRecord next = last.next;

while (next != null) {

if (next.dueTime > now) {

last.next = null;

break;

}

last = next;

next = next.next;

}

mHead = next;

return callbacks;

}

}

### CallbackRecord.run

[-> Choreographer.java ::CallbackRecord]

private static final class CallbackRecord {

public CallbackRecord next;

public long dueTime;

public Object action; // Runnable or FrameCallback

public Object token;

public void run(long frameTimeNanos) {

i**f (token == FRAME\_CALLBACK\_TOKEN) {**

((FrameCallback)action).doFrame(frameTimeNanos);

} else {

((Runnable)action).run();

}

}

}

这里的回调方法run()有两种执行情况

* 当token的数据类型为FRAME\_CALLBACK\_TOKEN，则执行该对象的doFrame()方法;
* 当token为其他类型，则执行该对象的run()方法。

在前面小节【2.8】doFrame()过程有一个判断变量mFrameScheduled，有两种执行情况：

* 当该值为true则执行动画，执行完本次操作则再次设置该值为false;
* 否则并不会执行动画。

对于底层Vsync信号每间隔16.7ms，上层都会接收到该信号。但对于系统会有需要，才会更新动画， 那么需要的场景便是由WMS调用scheduleAnimationLocked()方法来设置mFrameScheduled=true来触发动画， 接下来说说动画控制的过程

## 动画显示过程

调用栈

WMS.scheduleAnimationLocked

postFrameCallback

postFrameCallbackDelayed

postCallbackDelayedInternal

scheduleFrameLocked

### WMS.scheduleAnimationLocked

[-> WindowManagerService.java]

void scheduleAnimationLocked() {

if (!mAnimationScheduled) {

mAnimationScheduled = true;

//【见小节3.2】

mChoreographer.postFrameCallback(mAnimator.mAnimationFrameCallback);

}

}

只有当mAnimationScheduled=false时，才会执行postFrameCallback()，其中参数为mAnimator对象的 成员变量mAnimationFrameCallback，该对象的初始化过程：

#### 创建WMS

private WindowManagerService( ...

mAnimator = new WindowAnimator(this); //【见小节3.1.2】

}

#### 创建WindowAnimator

[-> WindowAnimator.java]

WindowAnimator(final WindowManagerService service) {

mService = service;

mContext = service.mContext;

mPolicy = service.mPolicy;

mAnimationFrameCallback = new Choreographer.FrameCallback() {

public void doFrame(long frameTimeNs) {

synchronized (mService.mWindowMap) {

mService.mAnimationScheduled = false;

animateLocked(frameTimeNs);

}

}

};

}

mAnimationFrameCallback的数据类型为Choreographer.FrameCallback

### postFrameCallback

[-> Choreographer.java]

public void postFrameCallback(FrameCallback callback) {

postFrameCallbackDelayed(callback, 0);

}

public void postFrameCallbackDelayed(FrameCallback callback, long delayMillis) {

...

//【见小节3.3】

postCallbackDelayedInternal(CALLBACK\_ANIMATION,

callback, FRAME\_CALLBACK\_TOKEN, delayMillis);

}

### 3.3 postCallbackDelayedInternal

[-> Choreographer.java]

// callbackType为动画，action为mAnimationFrameCallback

// token为FRAME\_CALLBACK\_TOKEN，delayMillis=0

private void postCallbackDelayedInternal(int callbackType, Object action, Object token, long delayMillis) {

synchronized (mLock) {

final long now = SystemClock.uptimeMillis();

final long dueTime = now + delayMillis;

//添加到mCallbackQueues队列

mCallbackQueues[callbackType].addCallbackLocked(dueTime, action, token);

if (dueTime <= now) {

scheduleFrameLocked(now);

} else {

//发送消息MSG\_DO\_SCHEDULE\_CALLBACK

Message msg = mHandler.obtainMessage(MSG\_DO\_SCHEDULE\_CALLBACK, action);

msg.arg1 = callbackType;

msg.setAsynchronous(true);

mHandler.sendMessageAtTime(msg, dueTime);

}

}

}

发送MSG\_DO\_SCHEDULE\_CALLBACK消息后，主线程接收后进入FrameHandler的handleMessage()操作，如下方法。

### 3.4 MSG\_DO\_SCHEDULE\_CALLBACK

[-> Choreographer.java ::FrameHandler]

private final class FrameHandler extends Handler {

public void handleMessage(Message msg) {

switch (msg.what) {

case MSG\_DO\_FRAME:

doFrame(System.nanoTime(), 0);

break;

case MSG\_DO\_SCHEDULE\_VSYNC:

doScheduleVsync();

break;

case MSG\_DO\_SCHEDULE\_CALLBACK:

doScheduleCallback(msg.arg1); //【见小节3.5】

break;

}

}

}

### 3.5 doScheduleCallback

[-> Choreographer.java]

void doScheduleCallback(int callbackType) {

synchronized (mLock) {

if (!mFrameScheduled) {

final long now = SystemClock.uptimeMillis();

if (mCallbackQueues[callbackType].hasDueCallbacksLocked(now)) {

scheduleFrameLocked(now); //【见小节3.6】

}

}

}

}

### 3.6 scheduleFrameLocked

[-> Choreographer.java]

private void scheduleFrameLocked(long now) {

if (!mFrameScheduled) {

mFrameScheduled = true;

if (USE\_VSYNC) {

if (isRunningOnLooperThreadLocked()) {

//当运行在Looper线程，则立刻调度vsync

scheduleVsyncLocked();

} else {

//否则，发送消息到UI线程

Message msg = mHandler.obtainMessage(MSG\_DO\_SCHEDULE\_VSYNC);

msg.setAsynchronous(true);

mHandler.sendMessageAtFrontOfQueue(msg);

}

} else {

final long nextFrameTime = Math.max(

mLastFrameTimeNanos / TimeUtils.NANOS\_PER\_MS + sFrameDelay, now);

Message msg = mHandler.obtainMessage(MSG\_DO\_FRAME);

msg.setAsynchronous(true);

mHandler.sendMessageAtTime(msg, nextFrameTime);

}

}

}

该方法的功能

1. 当运行在Looper线程，则立刻调度scheduleVsyncLocked();
2. 当运行在其他线程，则通过发送一个消息到Looper线程，然后再执行scheduleVsyncLocked();

### 3.7 scheduleVsyncLocked

[-> Choreographer.java]

private void scheduleVsyncLocked() {

mDisplayEventReceiver.scheduleVsync(); //【见小节3.8】

}

mDisplayEventReceiver对象是在【小节2.2】Choreographer的实例化过程所创建的。

### 3.8 scheduleVsync

[-> DisplayEventReceiver.java]

public void scheduleVsync() {

if (mReceiverPtr == 0) {

...

} else {

nativeScheduleVsync(mReceiverPtr);

}

}

### 3.9 nativeScheduleVsync

[-> android\_view\_DisplayEventReceiver.cpp]

static void nativeScheduleVsync(JNIEnv\* env, jclass clazz, jlong receiverPtr) {

sp<NativeDisplayEventReceiver> receiver =

reinterpret\_cast<NativeDisplayEventReceiver\*>(receiverPtr);

status\_t status = receiver->scheduleVsync();

...

}

### 3.10 scheduleVsync

[-> android\_view\_DisplayEventReceiver.cpp]

status\_t NativeDisplayEventReceiver::scheduleVsync() {

if (!mWaitingForVsync) {

nsecs\_t vsyncTimestamp;

int32\_t vsyncDisplayId;

uint32\_t vsyncCount;

processPendingEvents(&vsyncTimestamp, &vsyncDisplayId, &vsyncCount);

//【见小节3.11】

status\_t status = mReceiver.requestNextVsync();

...

mWaitingForVsync = true;

}

return OK;

}

### 3.11 requestNextVsync

[-> DisplayEventReceiver.cpp]

status\_t DisplayEventReceiver::requestNextVsync() {

if (mEventConnection != NULL) {

mEventConnection->requestNextVsync();

return NO\_ERROR;

}

return NO\_INIT;

}

这里就先不继续往下写了，该方法的作用请求下一次Vsync信息处理。 当Vsync信号到来，由于mFrameScheduled=true,则继续【小节2.9】CallbackRecord.run()方法。

## 动画处理

根据[小节3.1.2]mAnimationFrameCallback.FrameCallback方法，进而调用animateLocked().

### 4.1 animateLocked

[-> WindowAnimator.java]

private void animateLocked(long frameTimeNs) {

if (!mInitialized) {

return;

}

mCurrentTime = frameTimeNs / TimeUtils.NANOS\_PER\_MS;

mBulkUpdateParams = SET\_ORIENTATION\_CHANGE\_COMPLETE;

boolean wasAnimating = mAnimating;

mAnimating = false;

mAppWindowAnimating = false;

SurfaceControl.openTransaction(); //打开transacion

SurfaceControl.setAnimationTransaction();

try {

final int numDisplays = mDisplayContentsAnimators.size();

for (int i = 0; i < numDisplays; i++) {

final int displayId = mDisplayContentsAnimators.keyAt(i);

updateAppWindowsLocked(displayId);

DisplayContentsAnimator displayAnimator = mDisplayContentsAnimators.valueAt(i);

final ScreenRotationAnimation screenRotationAnimation =

displayAnimator.mScreenRotationAnimation;

if (screenRotationAnimation != null && screenRotationAnimation.isAnimating()) {

if (screenRotationAnimation.stepAnimationLocked(mCurrentTime)) {

mAnimating = true;

} else {

mBulkUpdateParams |= SET\_UPDATE\_ROTATION;

screenRotationAnimation.kill();

displayAnimator.mScreenRotationAnimation = null;

if (mService.mAccessibilityController != null

&& displayId == Display.DEFAULT\_DISPLAY) {

mService.mAccessibilityController.onRotationChangedLocked(

mService.getDefaultDisplayContentLocked(), mService.mRotation);

}

}

}

//更新所有应用的动画，包括正在退出的应用

updateWindowsLocked(displayId);

updateWallpaperLocked(displayId);

final WindowList windows = mService.getWindowListLocked(displayId);

final int N = windows.size();

for (int j = 0; j < N; j++) {

//输出动画

windows.get(j).mWinAnimator.prepareSurfaceLocked(true);

}

}

for (int i = 0; i < numDisplays; i++) {

final int displayId = mDisplayContentsAnimators.keyAt(i);

testTokenMayBeDrawnLocked(displayId);

final ScreenRotationAnimation screenRotationAnimation =

mDisplayContentsAnimators.valueAt(i).mScreenRotationAnimation;

if (screenRotationAnimation != null) {

screenRotationAnimation.updateSurfacesInTransaction();

}

mAnimating |= mService.getDisplayContentLocked(displayId).animateDimLayers();

if (mService.mAccessibilityController != null

&& displayId == Display.DEFAULT\_DISPLAY) {

mService.mAccessibilityController.drawMagnifiedRegionBorderIfNeededLocked();

}

}

if (mAnimating) {

mService.scheduleAnimationLocked();

}

mService.setFocusedStackLayer();

if (mService.mWatermark != null) {

mService.mWatermark.drawIfNeeded();

}

} catch (RuntimeException e) {

Slog.wtf(TAG, "Unhandled exception in Window Manager", e);

} finally {

SurfaceControl.closeTransaction(); //关闭transacion

}

boolean hasPendingLayoutChanges = false;

final int numDisplays = mService.mDisplayContents.size();

for (int displayNdx = 0; displayNdx < numDisplays; ++displayNdx) {

final DisplayContent displayContent = mService.mDisplayContents.valueAt(displayNdx);

final int pendingChanges = getPendingLayoutChanges(displayContent.getDisplayId());

if ((pendingChanges & WindowManagerPolicy.FINISH\_LAYOUT\_REDO\_WALLPAPER) != 0) {

mBulkUpdateParams |= SET\_WALLPAPER\_ACTION\_PENDING;

}

if (pendingChanges != 0) {

hasPendingLayoutChanges = true;

}

}

boolean doRequest = false;

if (mBulkUpdateParams != 0) {

doRequest = mService.copyAnimToLayoutParamsLocked();

}

if (hasPendingLayoutChanges || doRequest) {

mService.requestTraversalLocked();

}

if (!mAnimating && wasAnimating) {

mService.requestTraversalLocked();

}

}

未完待续。。。

[Choreographer原理](http://gityuan.com/2017/02/25/choreographer/)

## 节奏大师的应用

Choreographer就是一个普通的工具类，封装了屏幕Vsync的方法，在system\_server和TextView都有应用

### 跑马灯

TextView自带了跑马灯功能，只要把它的ellipsize属性设置为marquee就可以了。但有个前提，就是TextView要处于被选中状态才能有效果

<TextView

    android:ellipsize="marquee"

    android:marqueeRepeatLimit="marquee\_forever"

    android:text="正在定位..."

/>

case com.android.internal.R.styleable.TextView\_marqueeRepeatLimit:

setMarqueeRepeatLimit(a.getInt(attr, mMarqueeRepeatLimit));//==3

case com.android.internal.R.styleable.TextView\_ellipsize:

ellipsize = a.getInt(attr, ellipsize);

switch (ellipsize) {

case ELLIPSIZE\_MARQUEE:

setEllipsize(TextUtils.TruncateAt.MARQUEE);

}

private void startMarquee() {

// Do not ellipsize EditText

if (getKeyListener() != null) return;

if (compressText(getWidth() - getCompoundPaddingLeft() - getCompoundPaddingRight())) {

return;

}

if ((mMarquee == null || mMarquee.isStopped()) && (isFocused() || isSelected())

&& getLineCount() == 1 && canMarquee()) {

if (mMarqueeFadeMode == MARQUEE\_FADE\_SWITCH\_SHOW\_ELLIPSIS) {

mMarqueeFadeMode = MARQUEE\_FADE\_SWITCH\_SHOW\_FADE;

final Layout tmp = mLayout;

mLayout = mSavedMarqueeModeLayout;

mSavedMarqueeModeLayout = tmp;

setHorizontalFadingEdgeEnabled(true);

requestLayout();

invalidate();

}

if (mMarquee == null) mMarquee = new Marquee(this);

mMarquee.start(mMarqueeRepeatLimit);

}

}

### ViewRootImpl安排一次绘制

一次绘制，就是完成一个树形视图的测量、布局、绘制的过程，遍历视图树的每一个节点，当然，可以根据条件判断，省略掉其中一个或几个环节，比如，只刷新绘制，不测量和布局。

#### App层scheduleTraversals

**void** scheduleTraversals() {  
 **if** (!**mTraversalScheduled**) {  
 **mTraversalScheduled** = **true**;  
 **mTraversalBarrier** = **mHandler**.getLooper().getQueue().postSyncBarrier();  
 **mChoreographer**.postCallback(  
 Choreographer.***CALLBACK\_TRAVERSAL***, **mTraversalRunnable**, **null**);  
 **if** (!**mUnbufferedInputDispatch**) {  
 scheduleConsumeBatchedInput();  
 }  
 notifyRendererOfFramePending();  
 pokeDrawLockIfNeeded();  
 }  
}

ViewRootImpl对象的scheduleTraversals安排一次绘制，安排后，将设置标志位mTraversalScheduled，防止多次安排，发送同步栅栏。  
当执行绘制doTraversal方法或unscheduleTraversals方法主动取消绘制时，关掉标志位，取消同步栅栏。委托Choreographer安排绘制，请求信号。  
postCallbackDelayedInternal方法，调用postCallbackDelayedInternal。

**private void** postCallbackDelayedInternal(**int** callbackType,  
 Object action, Object token, **long** delayMillis) {  
 **synchronized** (**mLock**) {  
 **final long** now = SystemClock.*uptimeMillis*();  
 **final long** dueTime = now + delayMillis;  
 **mCallbackQueues**[callbackType].addCallbackLocked(dueTime, action, token);  
  
 **if** (dueTime <= now) {  
 scheduleFrameLocked(now);// //没有延迟，以当前时间安排帧。  
 } **else** {  
 Message msg = **mHandler**.obtainMessage(***MSG\_DO\_SCHEDULE\_CALLBACK***, action);  
 msg.**arg1** = callbackType;  
 msg.setAsynchronous(**true**);  
 **mHandler**.sendMessageAtTime(msg, dueTime);  
 }  
 }  
}

该postCallback方法向回调请求数组CallbackQueues中存入一个对应类型的记录CallbackRecord，该纪录封装回调任务、当前时间，等待信号安排一帧绘制。数组元素是CallbackQueue，内部包含指向CallbackRecord链表的头指针。四种callbackType类型CALLBACK\_INPUT、CALLBACK\_ANIMATION、CALLBACK\_TRAVERSAL、CALLBACK\_COMMIT。每一种callbackType类型代表数组的一个索引。

CallbackQueue的addCallbackLocked方法，创建一个CallbackRecord，**封装dueTime执行时间(当前时间+延迟)，任务action和token。**插入链表，按照dueTime时间排序，dueTime时间小的在链表头部。  
本文只关注CALLBACK\_TRAVERSAL类型和TraversalRunnable回调任务，当收到Vsync信号时，将触发任务的doTraversal方法。

**没有延迟**，scheduleFrameLocked方法立即安排。  
**有延迟**，等待delayMillis时间，在特定时间dueTime，通过Choreographer内部FrameHandler发送消息。

FrameHandler处理延迟发送的CALLBACK，触发doScheduleCallback方法。

有延迟时，当到达时间dueTime，在handleMessage方法开始执行，最终实现一次绘制，和没有延迟时一样，触发scheduleFrameLocked方法。注意，经过一段时间的延迟，中间有不确定性，增加两个条件判断。  
**mFrameScheduled标志，**表示此时已经有过一次scheduled，请求一次有回复在doFrame处重置标志。这时，不再scheduled。

**当链表头结点的dueTime比当前时间now大**，表示当时以(now + delayMillis)插入的CallbackRecord节点已经不在链表中了，否则在(now + delayMillis)时刻执行获取的当前时间一定会<=now，**链表每个元素的dueTime都大于now**，或者头节点是空，这种情况下不再scheduled。

**scheduleFrameLocked方法，schedule一次具体的帧绘制。**

**private void** scheduleFrameLocked(**long** now) {  
 **if** (!**mFrameScheduled**) {  
 **mFrameScheduled** = **true**;**if** (isRunningOnLooperThreadLocked()) {  
 scheduleVsyncLocked();  
 } **else** {  
 Message msg = **mHandler**.obtainMessage(***MSG\_DO\_SCHEDULE\_VSYNC***);  
 msg.setAsynchronous(**true**);  
 **mHandler**.sendMessageAtFrontOfQueue(msg);  
 }  
 }  
}

入参now是执行该方法前获取的当前时间，设置mFrameScheduled标志，底层回调后重置标志。  
默认USE\_VSYNC，当前线程和Choreographer绑定线程一致，直接调用Choreographer的scheduleVsyncLocked方法，线程不同，通过FrameHandler发送消息，在Choreographer线程处理事务消息。

case MSG\_DO\_SCHEDULE\_VSYNC:

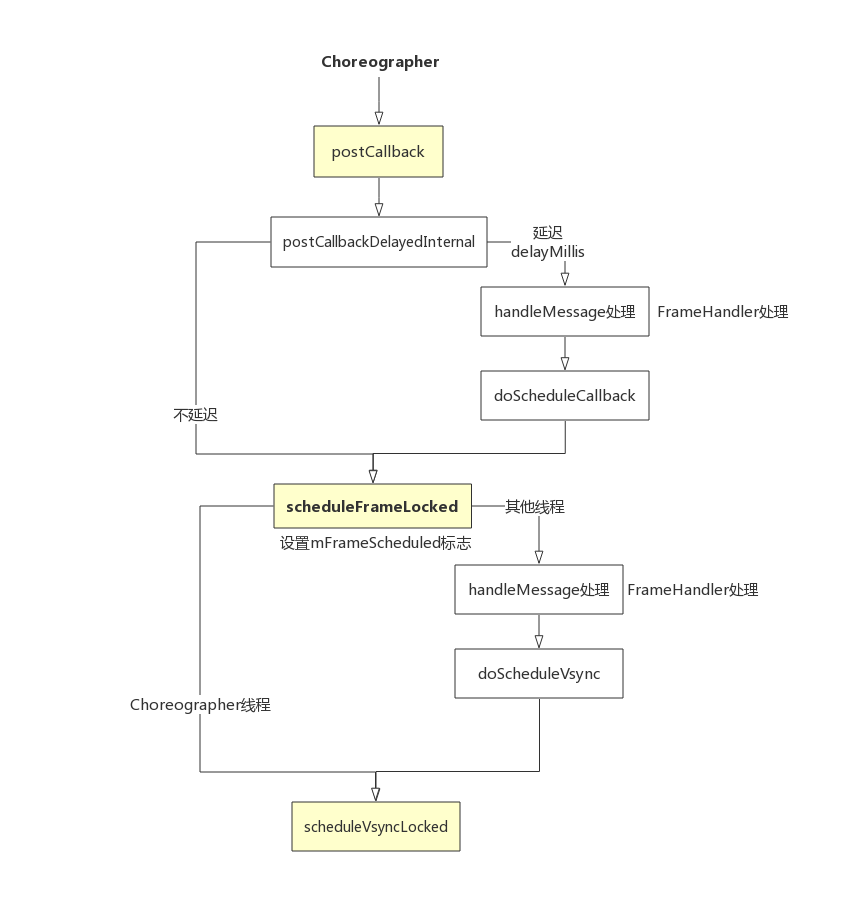
doScheduleVsync();

同样，在doScheduleVsync会调用scheduleVsyncLocked方法。

**void** doScheduleVsync() {  
 **synchronized** (**mLock**) {  
 **if** (**mFrameScheduled**) {  
 scheduleVsyncLocked();  
 }  
 }  
}

**private void** scheduleVsyncLocked() {  
 **mDisplayEventReceiver**.scheduleVsync();  
}

同样，在doScheduleVsync会调用scheduleVsyncLocked方法。postCallback方法流程图。



Choreographer的postCallback方法流程.jpg

最终统一调用到scheduleVsyncLocked方法，**它通过注册的接收器JNI方法访问底层，请求垂直同步信号。**

#### 显示事件接收器基础架构

scheduleVsyncLocked方法，通过DisplayEventReceiver的scheduleVsync实现请求同步信号。

private void scheduleVsyncLocked() {

mDisplayEventReceiver.scheduleVsync();

}

它是FrameDisplayEventReceiver类型，继承DisplayEventReceiver，在编舞者的构造方法中初始化，Java层DisplayEventReceiver构造方法。

public DisplayEventReceiver(Looper looper) {

..//Looper是空抛异常

mMessageQueue = looper.getQueue();

mReceiverPtr = nativeInit(new WeakReference<DisplayEventReceiver>(this), mMessageQueue);

}

DisplayEventReceiver会绑定Choreographer线程消息队列。创建一个弱引用，和消息队列一起，JNI#nativeInit方法初始化传入底层

static jlong nativeInit(JNIEnv\* env, jclass clazz, jobject receiverWeak,

jobject messageQueueObj) {

sp<MessageQueue> messageQueue = android\_os\_MessageQueue\_getMessageQueue(env, messageQueueObj);

...

sp<NativeDisplayEventReceiver> receiver = new NativeDisplayEventReceiver(env,

receiverWeak, messageQueue);

status\_t status = receiver->initialize();

...

receiver->incStrong(gDisplayEventReceiverClassInfo.clazz);

return reinterpret\_cast<jlong>(receiver.get());

}

根据Java层MQ获取底层消息队列，创建一个NativeDisplayEventReceiver，JNI层接收器，**继承LooperCallback，**将mReceiverPtr指针返回Java层。JNI层接收器封装Java层弱引用receiverWeak、底层消息队列，底层接收器。  
JNI层接收器的initialize初始化方法。

status\_t NativeDisplayEventReceiver::initialize() {

status\_t result = mReceiver.initCheck();

...

int rc = mMessageQueue->getLooper()->addFd(mReceiver.getFd(), 0, Looper::EVENT\_INPUT,

this, NULL);

return OK;

}

向Choreographer线程Looper监听**底层DisplayEventReceiver对象中BitTube的mReceiverFd描述符。**一旦接收到消息，将触发Looper的handleEvent回调方法。

底层DisplayEventReceive的构造方法。

DisplayEventReceiver::DisplayEventReceiver() {

sp<ISurfaceComposer> sf(ComposerService::getComposerService());

if (sf != NULL) {

mEventConnection = sf->createDisplayEventConnection();

if (mEventConnection != NULL) {

mDataChannel = mEventConnection->getDataChannel();

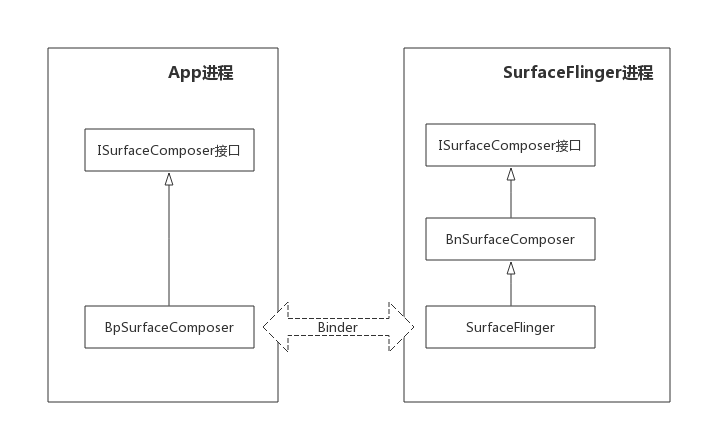
}

}

}

初始化两个重要指针，IDisplayEventConnection类型(mEventConnection)和BitTube类型(mDataChannel)。  
IDisplayEventConnection负责和SurfaceFlinger进程通信，真正向底层发起请求，定义了进程通信的业务接口，IDisplayEventConnection业务层接口方法。

上面代码中sf是App进程ISurfaceComposer业务代理，真实的业务对象是SurfaceFlinger对象，继承BnSurfaceComposer，在SurfaceFlinger进程。ISurfaceComposer业务进程通信。

在App进程，ISurfaceComposer代理的createDisplayEventConnection方法，返回IDisplayEventConnection业务代理，继承BpDisplayEventConnection，也就是mEventConnection。然后，App进程就能通过IDisplayEventConnection的三个业务方法，**requestNextVsync方法，**请求下一次垂直同步信号。**setVsyncRate方法，**设置垂直同步帧率。**getDataChannel方法，**获取通信管道。

在SurfaceFlinger进程，SurfaceFlinger对象的createDisplayEventConnection方法，创建IDisplayEventConnection业务真实对象Connection，继承BnDisplayEventConnection。和App进程通信时，参数是Binder类型的，Parcal的writeStrongBinder和readStrongBinder方法可以实现Binder对象的传输，内核红黑树创建服务和引用节点，App进程创建出BpDisplayEventConnection业务代理。  
**SurfaceFlinger进程的createDisplayEventConnection方法。**

sp<IDisplayEventConnection> SurfaceFlinger::createDisplayEventConnection() {

return mEventThread->createEventConnection();

}

sp<EventThread::Connection> EventThread::createEventConnection() const {

//在创建前会向EventThread注册该连接，加入到mDisplayEventConnections中。

return new Connection(const\_cast<EventThread\*>(this));

}

创建Connection对象，它是EventThread内部类，该类的构造方法。

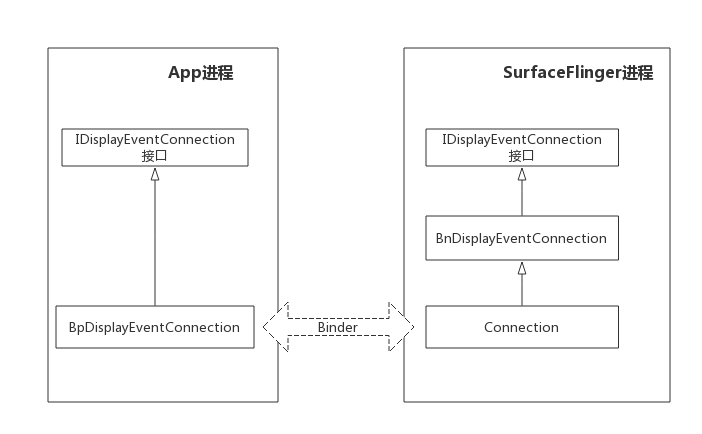
EventThread::Connection::Connection(const sp<EventThread>& eventThread)

: count(-1), mEventThread(eventThread), mChannel(new BitTube())

{

}

创建数据通道BitTube，**内部一对Socket描述符**，提供读写方法，用于数据通讯。  
IDisplayEventConnection业务进程通信图。

BitTube的构造方法和初始化方法。

BitTube::BitTube()

: mSendFd(-1), mReceiveFd(-1){

init(DEFAULT\_SOCKET\_BUFFER\_SIZE, DEFAULT\_SOCKET\_BUFFER\_SIZE);

}

void BitTube::init(size\_t rcvbuf, size\_t sndbuf) {

int sockets[2];

if (socketpair(AF\_UNIX, SOCK\_SEQPACKET, 0, sockets) == 0) {

...

mReceiveFd = sockets[0];

mSendFd = sockets[1];

} else {

mReceiveFd = -errno;

}

}

初始化一对socket发送/接收描述符，提供读/写数据的管道功能。Tube的意思是管，BitTube即字节管道。socket缓存4M，mSendFd用于写入，mReceiveFd用于接收，写入后，接收端mReceivedFd能读取。

App进程，通过BpDisplayEventConnection的getDataChannel方法获取通信管道，SurfaceFlinger进程，通过writeDupFileDescriptor写入mReceivedFd描述符，App进程从Parcel中读取，创建新BitTube对象封装mReceivedFd描述符。

virtual sp<BitTube> getDataChannel() const {

Parcel data, reply;

data.writeInterfaceToken(IDisplayEventConnection::getInterfaceDescriptor());

remote()->transact(GET\_DATA\_CHANNEL, data, &reply);

return new BitTube(reply);

}

Parcel的writeDupFileDescriptor和readFileDescriptor方法负责FileDescriptor类型读写存储。BitTube的构造方法(带Parcel参数)。

BitTube::BitTube(const Parcel& data)

: mSendFd(-1), mReceiveFd(-1) {

mReceiveFd = dup(data.readFileDescriptor());

if (mReceiveFd < 0) {

mReceiveFd = -errno;

ALOGE("BitTube(Parcel): can't dup filedescriptor (%s)",

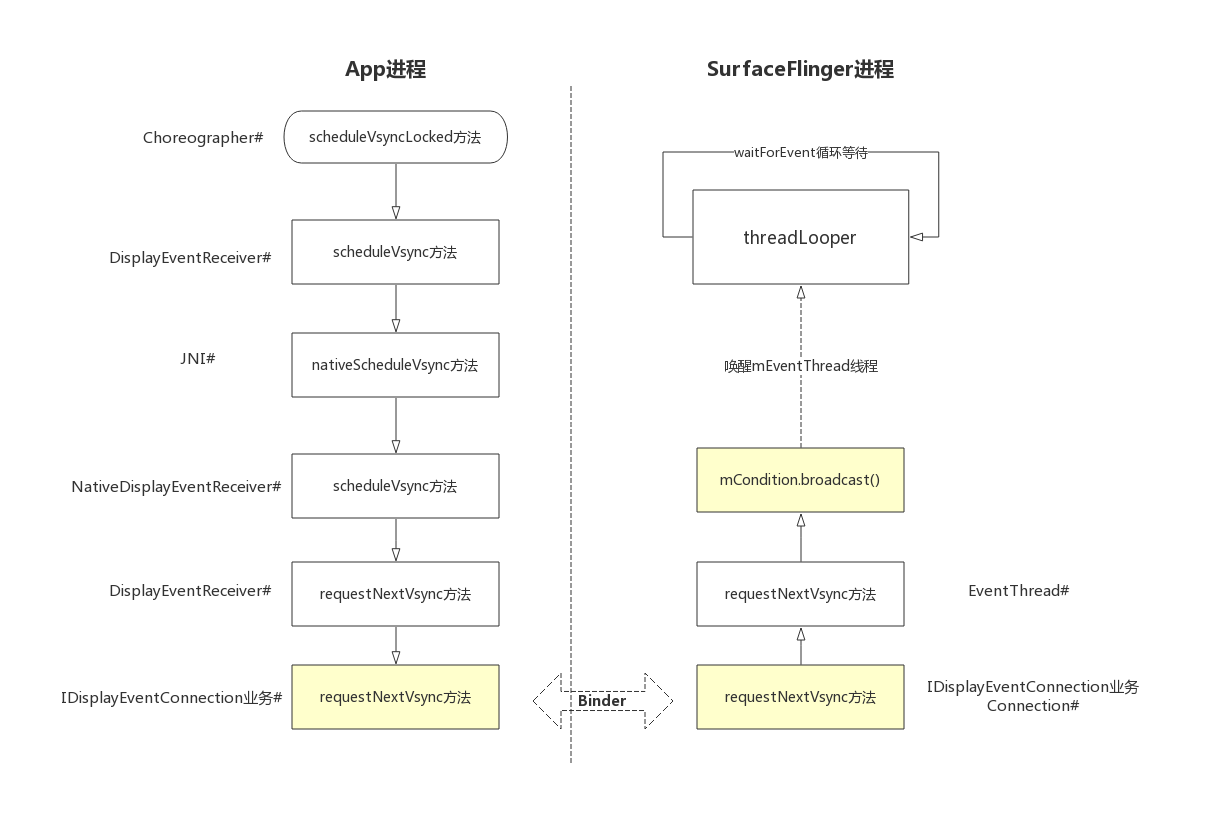
strerror(-mReceiveFd));

}

}

Choreographer接收器与SurfaceFlinger通信的架构图。

#### https://upload-images.jianshu.io/upload_images/5964029-f81bdd2a248dc509.jpg?imageMogr2/auto-orient/请求垂直同步信号流程

Java层DisplayEventReceiver请求一次垂直同步信号的过程。  


前两个方法已经看过了，直接看一下JNI#nativeScheduleVsync方法。

static void nativeScheduleVsync(JNIEnv\* env, jclass clazz, jlong receiverPtr) {

sp<NativeDisplayEventReceiver> receiver =

reinterpret\_cast<NativeDisplayEventReceiver\*>(receiverPtr);

status\_t status = receiver->scheduleVsync();

...

}

根据Java层receiverPtr指针，获取JNI层的NativeDisplayEventReceiver对象

status\_t NativeDisplayEventReceiver::scheduleVsync() {

if (!mWaitingForVsync) {

...

processPendingEvents(&vsyncTimestamp, &vsyncDisplayId, &vsyncCount);

status\_t status = mReceiver.requestNextVsync();

...

mWaitingForVsync = true;

}

return OK;

}

JNI层接收器封装底层DisplayEventReceiver，调用requestNextVsync方法。

status\_t DisplayEventReceiver::requestNextVsync() {

if (mEventConnection != NULL) {

mEventConnection->requestNextVsync();

return NO\_ERROR;

}

return NO\_INIT;

}

底层DisplayEventReceiver，内部两个重要指针已经介绍过。  
**IDisplayEventConnection业务#requestNextVsync方法**，和SurfaceFlinger进程通信，该进程实现业务接口的是Connection对象。sf进程Connection的requestNextVsync方法。

void EventThread::Connection::requestNextVsync() {

mEventThread->requestNextVsync(this);

}

Connection内部EventThread的requestNextVsync方法。

void EventThread::requestNextVsync(

const sp<EventThread::Connection>& connection) {

Mutex::Autolock \_l(mLock);

if (connection->count < 0) {

connection->count = 0;

mCondition.broadcast();

}

}

通知mCondition条件，**broadcast方法唤醒SurfaceFlinger进程的一个循环线程mEventThread，**该线程在**waitForEvent处等待**，被唤醒后可利用Connection发送事件。

**SurfaceFlinger循环线程EventThread的threadLoop方法**

bool EventThread::threadLoop() {

DisplayEventReceiver::Event event;

Vector< sp<EventThread::Connection> > signalConnections;

signalConnections = waitForEvent(&event);//等待事件，等待得到的事件保存在event指针处

const size\_t count = signalConnections.size();

for (size\_t i=0 ; i<count ; i++) {//遍历有信号的Connection，每个都发送event

const sp<Connection>& conn(signalConnections[i]);

status\_t err = conn->postEvent(event);//写入的内容是Event类型

if (err == -EAGAIN || err == -EWOULDBLOCK) {

...

} else if (err < 0) {

removeDisplayEventConnection(signalConnections[i]);

}

}

return true;

}

**该循环线程唯一的工作是在waitForEvent方法处等待VSYNC信号**，当信号发生时，发送给BitTube#mSendFd句柄。  
注意，SurfaceFlinger有两个EventThread线程，运行在各自的循环中。

收到信号时，遍历收到信号的Connection，调用它的postEvent方法。

status\_t EventThread::Connection::postEvent(

const DisplayEventReceiver::Event& event) {

ssize\_t size = DisplayEventReceiver::sendEvents(mChannel, &event, 1);

return size < 0 ? status\_t(size) : status\_t(NO\_ERROR);

}

DisplayEventReceiver的sendEvents方法。

ssize\_t DisplayEventReceiver::sendEvents(const sp<BitTube>& dataChannel,

Event const\* events, size\_t count) {

return BitTube::sendObjects(dataChannel, events, count);

}

利用Connection内部BitTube(即mChannel)，BitTube的sendObjects将触发BitTube#write方法，向mSendFd写入数据，App进程mReceiveFd可收到数据，实现SurfaceFlinger进程到App进程的数据传输。

#### App进程Choreographer线程监听消息

在接收器架构中，JNI层的NativeDisplayEventReceiver继承LooperCallback，在初始化addFd时，将本身加入Looper回调，当App进程的mReceiveFd描述符收到消息后，Choreographer线程的底层Looper将触发LooperCallback的handleEvent方法。也就是NativeDisplayEventReceiver的handleEvent方法。  
调用dispatchVsync方法。

void NativeDisplayEventReceiver::dispatchVsync(nsecs\_t timestamp, int32\_t

id, uint32\_t count) {

JNIEnv\* env = AndroidRuntime::getJNIEnv();

ScopedLocalRef<jobject> receiverObj(env, jniGetReferent(env, mReceiverWeakGlobal));

if (receiverObj.get()) {

env->CallVoidMethod(receiverObj.get(),

gDisplayEventReceiverClassInfo.dispatchVsync, timestamp, id, count);

}

...

}

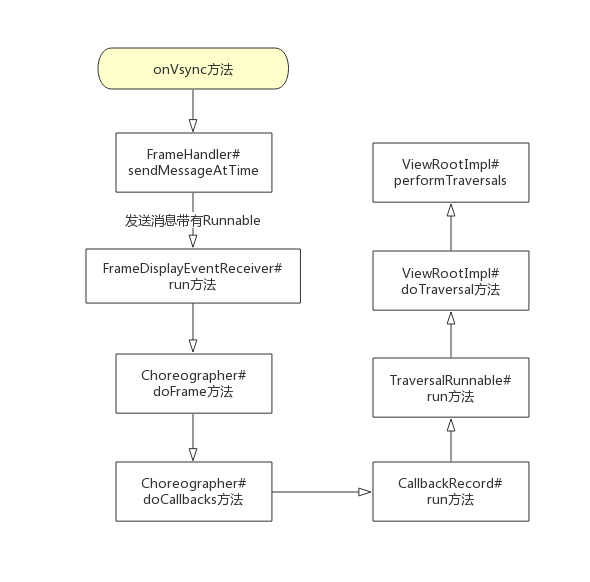
CallVoidMethod方法将调用Java层DisplayEventReceiver对象dispatchVsync方法。

private void dispatchVsync(long timestampNanos, int builtInDisplayId, int frame) {

onVsync(timestampNanos, builtInDisplayId, frame);

}

在Choreographer类，接收器就是FrameDisplayEventReceiver类型，它重写onVsync方法，**被底层Looper监听到的，在Choreographer线程执行。**onVsync方法的流程图。

@Override

public void onVsync(long timestampNanos, int builtInDisplayId, int frame) {

...

long now = System.nanoTime();

if (timestampNanos > now) {

timestampNanos = now;

}

...

mTimestampNanos = timestampNanos;

mFrame = frame;

Message msg = Message.obtain(mHandler, this);

msg.setAsynchronous(true);

mHandler.sendMessageAtTime(msg, timestampNanos / TimeUtils.NANOS\_PER\_MS);

}

特定的时刻发送消息，Message携带任务，而FrameDisplayEventReceiver实现Runnable，因此，最后在特定的时刻运行FrameDisplayEventReceiver的run方法。

@Override

public void run() {

mHavePendingVsync = false;

doFrame(mTimestampNanos, mFrame);

}

触发doFrame方法。

void doFrame(long frameTimeNanos, int frame) {

final long startNanos;

synchronized (mLock) {

if (!mFrameScheduled) {

return; // no work to do

}

long intendedFrameTimeNanos = frameTimeNanos;

startNanos = System.nanoTime();

final long jitterNanos = startNanos - frameTimeNanos;

if (jitterNanos >= mFrameIntervalNanos) {

final long skippedFrames = jitterNanos / mFrameIntervalNanos;

if (skippedFrames >= SKIPPED\_FRAME\_WARNING\_LIMIT) {

Log.i(TAG, "Skipped " + skippedFrames + " frames! "

+ "The application may be doing too much work on its main thread.");

}

final long lastFrameOffset = jitterNanos % mFrameIntervalNanos;

if (DEBUG\_JANK) {

Log.d(TAG, "Missed vsync by " + (jitterNanos \* 0.000001f) + " ms "

+ "which is more than the frame interval of "

+ (mFrameIntervalNanos \* 0.000001f) + " ms! "

+ "Skipping " + skippedFrames + " frames and setting frame "

+ "time to " + (lastFrameOffset \* 0.000001f) + " ms in the past.");

}

frameTimeNanos = startNanos - lastFrameOffset;

}

if (frameTimeNanos < mLastFrameTimeNanos) {

if (DEBUG\_JANK) {

Log.d(TAG, "Frame time appears to be going backwards. May be due to a "

+ "previously skipped frame. Waiting for next vsync.");

}

scheduleVsyncLocked();

return;

}

mFrameInfo.setVsync(intendedFrameTimeNanos, frameTimeNanos);

mFrameScheduled = false;

mLastFrameTimeNanos = frameTimeNanos;

}

try {

mFrameInfo.markInputHandlingStart();

doCallbacks(Choreographer.CALLBACK\_INPUT, frameTimeNanos);

mFrameInfo.markAnimationsStart();

doCallbacks(Choreographer.CALLBACK\_ANIMATION, frameTimeNanos);

mFrameInfo.markPerformTraversalsStart();

doCallbacks(Choreographer.CALLBACK\_TRAVERSAL, frameTimeNanos);

doCallbacks(Choreographer.CALLBACK\_COMMIT, frameTimeNanos);

} finally {

}

}

恢复mFrameScheduled标志，入参代表上一个帧的时间，如果当前时间已经>=frameTimeNanos一个间隔，表示发生了跳帧，根据时间差值/帧间隔计算跳过帧数，重置frameTimeNanos为不发生挑帧时，最近的一个帧时间点。最后，设置mLastFrameTimeNanos值，记录上一帧的时间。  
下面的事情就是回调处理每个类型的Callbacks，触发CallbackRecord的run方法，CallbackRecord封装了任务action。对于CALLBACK\_TRAVERSAL类型，最终会回调到ViewRootImpl#TraversalRunnable的run方法。

final class TraversalRunnable implements Runnable {

@Override

public void run() {

doTraversal();

}

}

ViewRootImpl的doTraversal方法，实现一次遍历绘制。

#### 总结

1，Java层DisplayEventReceiver通过JNI调用，根据底层垂直同步信号请求，实现帧画面的显示控制，在需要绘制时，发起请求，当底层发出信号时，同步回调到上层执行。底层初始化(nativeInit)，发起请求(scheduleVsync)，实现回调(dispatchVsync)。  
2，请求信号时，利用Binder机制同surfaceflinger进程通信，在surfaceflinger进程的业务对象是Connection，代表和App的一个连接。  
3，底层通知上层是通过socketpair建立一对匿名已经连接套接字mReceiveFd与mSendFd，实现SurfaceFlinger与App进程的双向通信管道。在App进程中，有**mReceiveFd描述符**，监听描述符来自SurfaceFlinger进程**mSendFd端**的消息。  
4，App进程Choreographer线程Looper负责监听来自mReceiveFd描述符

#### 参考

界面显示\_视图Choreographer控制

链接：<https://www.jianshu.com/p/47c866f6fb67>

# 实战

## 支持HDR

services/surfaceflinger/Layer.cpp第三方

-+#define ALIGN(val, align) (((val) + ((align) - 1)) & ~((align) - 1))

-+

- static TexBufferImag yuvTeximg[TexBufferMax] = {{NULL,EGL\_NO\_IMAGE\_KHR},{NULL,EGL\_NO\_IMAGE\_KHR}};

- #endif

-

Layer::onDraw(const sp<const DisplayDevice>& hw, const Region& clip,

- dlclose(dso);

- return;

- }

-- rockchipxxx((u8\*)src\_vaddr, (u8\*)dst\_vaddr, src\_r - src\_l, src\_b - src\_t, src\_stride, (src\_r - src\_l)\*2, 0);

-+ /\* align w to 64 \*/

-+ w = ALIGN(w, 64);

-+ if(w <= yuvTeximg[yuvIndex].yuvTexBuffer->getStride()/2)

-+ {

-+ rockchipxxx((u8\*)src\_vaddr, (u8\*)dst\_vaddr, w, src\_b - src\_t, src\_stride, yuvTeximg[yuvIndex].yuvTexBuffer->getStride(), 0);

-+ }

-+ else

-+ ALOGE("%s(%d):unsupport resolution for 4k", \_\_FUNCTION\_\_, \_\_LINE\_\_);

- #elif RK\_NV12\_10\_TO\_NV12\_BY\_NENO

- if(rockchipxxx3288 == NULL)

发给hj

发送

# 参考

[Choreographer原理](http://gityuan.com/2017/02/25/choreographer/)

http://gityuan.com/2017/02/25/choreographer/

# 录屏原理

MediaProjectionManager调用系统

MediaProjectionManager是个系统级服务，同TelephoneManager一样，通过getSystemService()方法获取；   
MediaProjection 实现截屏和录屏的类

getMediaProjection来取出intent中的数据：

@Override

public void onActivityResult(int requestCode, int resultCode, Intent data) {

if (requestCode != PERMISSION\_CODE) {

Log.e(TAG, "Unknown request code: " + requestCode);

return;

}

if (resultCode != RESULT\_OK) {

Toast.makeText(this,

"User denied screen sharing permission", Toast.LENGTH\_SHORT).show();

return;

}

mMediaProjection = mProjectionManager.getMediaProjection(resultCode, data);

mMediaProjection.registerCallback(new MediaProjectionCallback(), null);

mVirtualDisplay = createVirtualDisplay();

}

我们通过getMediaProjection获取到mediaProjection，并注册了一个callback回调。   
看看createVirtualDisplay做了什么：

private VirtualDisplay createVirtualDisplay() {

return mMediaProjection.createVirtualDisplay("ScreenSharingDemo",

mDisplayWidth, mDisplayHeight, mScreenDensity,

DisplayManager.VIRTUAL\_DISPLAY\_FLAG\_AUTO\_MIRROR,

mSurface, null /\*Callbacks\*/, null /\*Handler\*/);

}

可以看到，我们调用了MediaProjection的createVirtualDisplay方法，来创建了一个VirtualDisplay的实例，说几个createVirtualDisplay的参数含义：

name 是生成的VirtualDisplay实例的名称；   
width和height分别是生成实例的宽高，必须大于0；   
dpi，生成实例的像素密度，必须大于0，一般都取1；   
**surface，这个比较重要，是你生成的VirtualDisplay的载体，我的理解是，VirtualDisplay的内容是一帧帧的屏幕截图（所以你看到是有宽高，像素密度等设置），所以MediaProjection获取到的其实是一帧帧的图，然后通过surface（surface你可以理解成是android的一个画布，默认它会以每秒60帧来刷新，这里我们不再展开细说），来顺序播放这些图片，形成视频。**

# A n d r o i d 图 形 系 统 简 介

http://windrunnerlihuan.com/2017/12/01/Android-SurfaceFlinger-%E5%AD%A6%E4%B9%A0%E4%B9%8B%E8%B7%AF-%E5%8D%81%E4%B8%80-%E5%90%88%E6%88%90Layer%E4%B9%8B%E5%87%86%E5%A4%87%E5%90%88%E6%88%90/

# 自定义vsync

#ifdef USE\_VSYNC  
**void** ACodec::startVsyncThread(){  
 **if** (mvysnc != NULL) {  
 stopVsync();  
 mvysnc = NULL;  
 }  
   
 mvysnc = **new** VsyncReciver(**this**);  
 startalign = **true**;  
 phase = 0;  
 ALOGV(**"startVsyncThread"**);  
}  
**void** ACodec::stopVsync(){  
 Mutex::Autolock autoLock(mLock);  
   
 **delete** mvysnc;  
 mvysnc = NULL;  
 mRenderQueue.clear();  
  
 ALOGV(**"stopVsync end"**);  
}  
#endif

VsyncReciver

# 截屏

ScreenshotClient类位于frameworks/native/libs/gui/SurfaceComposerClient.cpp中。



从567行来看，该函数是直接调用了SurfaceFlinger服务的代理对象中的captureScreen()函数。

其中getComposerService（）函数的源码：



从代码上可以看出，instance.mComposerService是SurfaceFlinger服务的代理对象。

如果有兴趣的话可以继续跟进去里面的代码，会发现上面的第5行代码就是通过getService("SurfaceFlinger", &mComposerService)获取服务代理对象。

说到“SurfaceFlinger”服务，这个服务器是比较庞大的架构。主要的作用是进行UI绘制。（后续可以深入分析）

所以：

Screencap的实现就是通过访问“SurfaceFlinger”服务或者读取fb0文件进行截屏的。

# dumpsys SurfaceFlinger

adb shell dumpsys SurfaceFlinger >

用于查看叠加画面显示的层数。用法：

dumpsys SurfaceFlinger [--list、--latency、--latency-clear]

dumpsys SurfaceFlinger后面的参数通过-h，查询不到，我是在SurfaceFlinger服务的dump实现查看到。

用法

dumpsys SurfaceFlinger --list

com.android.systemui.ImageWallpaper

**com.tencent.mm/com.tencent.mm.plugin.webview.ui.tools.WebViewUI**

...

第二行即为显示最前面的应用程序。当出现黑屏的时候，可以快速定位目前运行的是哪个应用程序。

SurfaceFlinger的dump信息详解

## 特殊宏的打开

一般dump的第一行都是这样的:

1. Build configuration: [sf] [libui] [libgui]

这说明其实没有打开任何特殊的宏,实际上,如果一下特殊宏打开,第一行log会打印出来:

1. FRAMEBUFFER\_FORCE\_FORMAT,HAS\_CONTEXT\_PRIORITY,NEVER\_DEFAULT\_TO\_ASYNC\_MODE,TARGET\_DISABLE\_TRIPLE\_BUFFERING,DONT\_USE\_FENCE\_SYNC

一般情况下,这些宏一个都不会打开.

## Sync机制

第二行一般是这样的:

1. Sync configuration: [using: EGL\_ANDROID\_native\_fence\_sync EGL\_KHR\_wait\_sync]

这行其实打印了目前使用的sync机制,这个值源于这段逻辑:

1. if (useNativeFenceSync()) {
2. mString.append(" EGL\_ANDROID\_native\_fence\_sync");
3. }
4. if (useFenceSync()) {
5. mString.append(" EGL\_KHR\_fence\_sync");
6. }
7. if (useWaitSync()) {
8. mString.append(" EGL\_KHR\_wait\_sync");
9. }

注意,一二是互斥的,三可以与一二共存.

## DispSync参数

第三行是打印的是Vsync相关的参数:

1. DispSync configuration: app phase 0 ns, sf phase 0 ns, present offset 0 ns (refresh 16666667 ns)

这些参数我们还是比较熟悉的,有意思的是打印这些参数时候使用的语法:

1. result.appendFormat("app phase %" PRId64 " ns, sf phase %" PRId64 " ns, "
2. "present offset %d ns (refresh %" PRId64 " ns)",
3. vsyncPhaseOffsetNs, sfVsyncPhaseOffsetNs, PRESENT\_TIME\_OFFSET\_FROM\_VSYNC\_NS,
4. mHwc->getRefreshPeriod(HWC\_DISPLAY\_PRIMARY));

PRId64的用法很独特,这是一种跨平台的打印64位整数的做法:

1. printf("%" PRId64 "\n", value);
2. *// 相当于64位的：*
3. printf("%" "ld" "\n", value);
4. *// 或32位的：*
5. printf("%" "lld" "\n", value);

## layer的dump

接下来就是很长的一段layer的dump,一般以这样一句话开始:

1. Visible layers (count = 9)

count的值来源于layersSortedByZ中layer的数量.   
接下来就进入各个layer的dump,我们参考代码并以launcher所在的layer为例来解释下各行的意义:

1. + Layer 0xb3f92000 (com.sec.android.app.launcher/com.android.launcher2.Launcher) id=87

0xb3f92000指向当前layer对象的值,括号中是当前layer的名称,id是创建layer时产生的序列号.

### 区域信息

1. Region transparentRegion (this=0xb3f92164, count=1)
2. [ 0, 0, 0, 0]
3. Region visibleRegion (this=0xb3f92008, count=1)
4. [ 0, 0, 1440, 2560]

接下来的两段是两个Region的dump,每个region可能包含多个区域,所以这里count也可能不等于1.   
前两行的值来源于activeTransparentRegion,表示的是这个layer里面透明区域的大小.   
后两行值来源于visibleRegion,表示可见区域的大小.

### 基本信息

1. layerStack= 0, z= 21010, pos=(0,0), size=(1440,2560), crop=(0, 0,1440,2560), isOpaque=0,
2. invalidate=0, alpha=0xff, flags=0x00000000, tr=[1.00, 0.00][0.00, 1.00]
3. client=0xb11160c0

上面这段dump源自这段代码:

1. result.appendFormat( " "
2. "layerStack=%4d, z=%9d, pos=(%g,%g), size=(%4d,%4d), crop=(%4d,%4d,%4d,%4d), "
3. "isOpaque=%1d, invalidate=%1d, "
4. "alpha=0x%02x, flags=0x%08x, tr=[%.2f, %.2f][%.2f, %.2f]\n"
5. " client=%p\n",
6. s.layerStack, s.z, s.transform.tx(), s.transform.ty(), s.active.w, s.active.h,
7. s.active.crop.left, s.active.crop.top,
8. s.active.crop.right, s.active.crop.bottom,
9. isOpaque(s), contentDirty,
10. s.alpha, s.flags,
11. s.transform[0][0], s.transform[0][1],
12. s.transform[1][0], s.transform[1][1],
13. client.get());

* layerStack表示这个layer是保存在哪个layerstack中(不同的display是有不同的layerstack的,这点可以通过一个连接HDMI时的layerstack很容易确认).
* z表示Z轴坐标,z值越大,layer越靠上.
* pos的值是layer左上角的位置,这个值比较特殊的是ImageWallpaper这个layer的pos值,因为ImageWallpaper的大小大于屏幕大小,所以ImageWallpaper的pos值在屏幕的外面(note4是pos=(-560,0)).
* size自然是layer的大小
* crop代表裁剪区域,这点依然是对于壁纸很明显,因为壁纸layer大小大于屏幕,必须涉及到需要裁剪一部分显示在屏幕上,因此它的裁剪区域是crop=( 560, 0,2000,2560).
* isOpaque代表是否是不透明的,只有完全不透明的layer这个值才是1,比如壁纸,像状态栏和launcher他们都是0,代表不是完全不透明的
* invalidate表示这个layer的数据是失效的,这个值绝大多数情况下都是0.因为我们看到的一般都是绘制好的有效的数据.一种情况下这值特别频繁的多见为1,就是刚刚锁屏(解锁)时.因为突然锁屏,会导致绘制的内容和要显示的内容完全不同,导致layer的各种数据要重新计算,所以将layer置为失效.
* alpha表示了这张layer的透明度,这个值跟isOpaque是有区别的.isOpaque表示了这个layer可以是透明的,也就是没有显示数据的地方,可以透明;而alpha表示透明度,也即是有数据的地方也可以因为透明度而收到影响产生透明的效果.
* flag值含义丰富,它是众多flag或出来的结果,影响它值的包括:

1. enum {
2. eLayerHidden = 0x01, *// SURFACE\_HIDDEN in SurfaceControl.java*
3. eLayerOpaque = 0x02, *// SURFACE\_OPAQUE*
4. eLayerTransparent = 0x200, *// SURFACE\_TRANSPARENT*
5. };
6. enum {
7. ePositionChanged = 0x00000001,
8. eLayerChanged = 0x00000002,
9. eSizeChanged = 0x00000004,
10. eAlphaChanged = 0x00000008,
11. eMatrixChanged = 0x00000010,
12. eTransparentRegionChanged = 0x00000020,
13. eVisibilityChanged = 0x00000040,
14. eLayerStackChanged = 0x00000080,
15. eCropChanged = 0x00000100,
16. */\* SRIB : Smg Surface Animator : State that will indicate animation change \*/*
17. e3DAnimationChanged = 0x00001000,
18. */\* SRIB : Smg Surface Animator : Change End\*/*
19. eOpacityChanged = 0x00000200,
20. *// { SRUK-SFBLUR*
21. eTranslucentRegionChanged = 0x00000400,
22. *// SRUK-SFBLUR }*
23. eTransparencyChanged = 0x80000000,
24. };
25. enum { *// (keep in sync with Surface.java)*
26. eHidden = 0x00000004,
27. eDestroyBackbuffer = 0x00000020,
28. eSecure = 0x00000080,
29. eNonPremultiplied = 0x00000100,
30. eOpaque = 0x00000400,
31. eProtectedByApp = 0x00000800,
32. eProtectedByDRM = 0x00001000,
33. eCursorWindow = 0x00002000,
34. */\* SISO Changes for Internal\_Only - Start \*/*
35. eFXInternalDisplay = 0x80000000,
36. */\* SISO Changes for Internal\_Only - End \*/*
37. eFXSurfaceNormal = 0x00000000,
38. eFXSurfaceDim = 0x00020000,
39. eFXSurfaceMask = 0x000F0000,
40. *// begin of app fw : fixed orientation window*
41. eFixedOrientation = 0x40000000,
42. *// end of app fw*
43. *// begin of MDM remote control*
44. eNoRemoteControl = 0x08000000,
45. *// end of MDM remote control*
46. };

所有的这些值都可能影响layer的状态,涉及不同模块不同功能,这里不再展开.

* 接下来的一组tr数据代表屏幕的旋转和缩放程度.大多数的layer实际上是不需要旋转和缩放的,因为他们定义的大小就是跟屏幕一致的,所以他们的这组数据是[1.00, 0.00][0.00, 1.00],实际上如果你使用这组数据来做矩阵变换的话,矩阵是不会发生变化的.   
  需要旋转的比较典型的场景是照相机.横着拿相机时它的layer的变换矩阵是[-1.00, 0.00][-0.00, -1.00],也就是旋转180°.   
  这个值的来源是上层调用setMatrix函数设置的.
* client含义比较简单,值的来源是创建layer时,对应的SurfaceSession中mNativeClient.这东西也是跟SurfaceSession一一对应的,也就是跟SurfaceFlinger连接时一一对应的.从这个值我们可以判断,client值相同的layer,必然来自同一个进程(因为他们是由同一个连接创建出来的).

### 4.3 buffer信息

1. format= 1, activeBuffer=[1440x2560:1664, 1], queued-frames=0, mRefreshPending=0
2. mTexName=38 mCurrentTexture=2
3. mCurrentCrop=[0,0,0,0] mCurrentTransform=0
4. mAbandoned=0
5. -BufferQueue mMaxAcquiredBufferCount=1, mDequeueBufferCannotBlock=0,
6. default-size=[1440x2560], default-format=1, transform-hint=00, FIFO(0)={}
7. [00:0xb110e100] state=FREE , 0xb3eb1ec0 [1440x2560:1664, 1]
8. [01:0xb3ec7000] state=FREE , 0xb620d060 [1440x2560:1664, 1]
9. >[02:0xb110e200] state=ACQUIRED, 0xb1111100 [1440x2560:1664, 1]

#### 4.3.1 数据格式

首先是数据的format,值的来源是layer创建时赋予的,当然我们如果追溯的话,可以追溯到WindowManagerService创建SurfaceControl的过程,值也是创建时指定的.值的定义如下:

1. enum {
2. *//*
3. *// these constants need to match those*
4. *// in graphics/PixelFormat.java & pixelflinger/format.h*
5. *//*
6. PIXEL\_FORMAT\_UNKNOWN = 0,
7. PIXEL\_FORMAT\_NONE = 0,
8. *// logical pixel formats used by the SurfaceFlinger -----------------------*
9. PIXEL\_FORMAT\_CUSTOM = -4,
10. *// Custom pixel-format described by a PixelFormatInfo structure*
11. PIXEL\_FORMAT\_TRANSLUCENT = -3,
12. *// System chooses a format that supports translucency (many alpha bits)*
13. PIXEL\_FORMAT\_TRANSPARENT = -2,
14. *// System chooses a format that supports transparency*
15. *// (at least 1 alpha bit)*
16. PIXEL\_FORMAT\_OPAQUE = -1,
17. *// System chooses an opaque format (no alpha bits required)*
18. *// real pixel formats supported for rendering -----------------------------*
19. PIXEL\_FORMAT\_RGBA\_8888 = HAL\_PIXEL\_FORMAT\_RGBA\_8888, *// 4x8-bit RGBA*
20. PIXEL\_FORMAT\_RGBX\_8888 = HAL\_PIXEL\_FORMAT\_RGBX\_8888, *// 4x8-bit RGB0*
21. PIXEL\_FORMAT\_RGB\_888 = HAL\_PIXEL\_FORMAT\_RGB\_888, *// 3x8-bit RGB*
22. PIXEL\_FORMAT\_RGB\_565 = HAL\_PIXEL\_FORMAT\_RGB\_565, *// 16-bit RGB*
23. PIXEL\_FORMAT\_BGRA\_8888 = HAL\_PIXEL\_FORMAT\_BGRA\_8888, *// 4x8-bit BGRA*
24. PIXEL\_FORMAT\_RGBA\_5551 = 6, *// 16-bit ARGB*
25. PIXEL\_FORMAT\_RGBA\_4444 = 7, *// 16-bit ARGB*
26. PIXEL\_FORMAT\_sRGB\_A\_8888 = HAL\_PIXEL\_FORMAT\_sRGB\_A\_8888, *// 4x8-bit sRGB + A*
27. PIXEL\_FORMAT\_sRGB\_X\_8888 = HAL\_PIXEL\_FORMAT\_sRGB\_X\_8888, *// 4x8-bit sRGB, no A*
28. };

其实只有下面的值是真实可用的,其余值在SurfaceFlinger创建时会被转换:

1. switch (format) {
2. case PIXEL\_FORMAT\_TRANSPARENT:
3. case PIXEL\_FORMAT\_TRANSLUCENT:
4. format = PIXEL\_FORMAT\_RGBA\_8888;
5. break;
6. case PIXEL\_FORMAT\_OPAQUE:
7. format = PIXEL\_FORMAT\_RGBX\_8888;
8. break;
9. }

其实当前常见的format也就是这几种.

1. HAL\_PIXEL\_FORMAT\_RGBA\_8888 = 1,
2. HAL\_PIXEL\_FORMAT\_RGBX\_8888 = 2,
3. HAL\_PIXEL\_FORMAT\_RGB\_888 = 3,
4. HAL\_PIXEL\_FORMAT\_RGB\_565 = 4,
5. HAL\_PIXEL\_FORMAT\_BGRA\_8888 = 5,

0代表未知格式.   
常见的layer中,dimlayer一般是0,大多数layer是1,壁纸是2,照相机的预览数据是4,视频播放也是4.

#### 4.3.2 activeBuffer

activeBuffer=[1080x1920:1088, 1]

* activeBuffer的前两项表示了当前正在显示的buffer的宽和高.
* 第三项表示Stride.这个值很有意思,我们发现他有时候是等于宽的,有时候是大于宽的,我们先来看下这个值的解释.   
  The number of *pixels* that a line in the buffer takes in memory. This may be >= width.   
  我们知道内存申请使用是需要成块对齐的,也就是说不是说使我们申请多大的内存,就会给我们多大的内存,因为涉及到对齐,所以很可能这个内存实际上是大于我们的需要的.(暂时没有仔细研究,有待确认)像有些marvell型号,内存是按照64位对齐的,那么我们申请一个100宽的buffer,系统就会给我们留出128的buffer大小供我们使用.
* 第四项并没有什么特殊,表示format,跟前面的format应该是一致的.

#### 4.3.3 queued-frames 新的帧的数量

queued-frames的含义是是否有新的帧,如果当前没有新的帧,这个值是0.   
一般在画面持续变化时(照相预览,视频播放,窗口滑动,游戏),这个值会是1.表示有新的一帧.   
偶尔也可以见到这个值是2(这个值应该最大就是2,因为只有三个缓冲区).

#### 4.3.4 mRefreshPending刷新卡住了吗?

mRefreshPending几乎所有的常见情况下都是0,因为这个参数代表了一个layer执行了Invalidate却没有完成Refresh,除非发生错误这显然不可能.

### 4.4 SurfaceFlingerConsumer的dump

接下来开始对消费者进行dump,SurfaceFlingerConsumer是GLConsumer子类,所以这里实际上是调用了GLConsumer的dumpLocked函数.   
先来看下代码:

1. result.appendFormat(
2. "%smTexName=%d mCurrentTexture=%d\n"
3. "%smCurrentCrop=[%d,%d,%d,%d] mCurrentTransform=%#x\n",
4. prefix, mTexName, mCurrentTexture, prefix, mCurrentCrop.left,
5. mCurrentCrop.top, mCurrentCrop.right, mCurrentCrop.bottom,
6. mCurrentTransform);

它会对应打印出来这一段信息:

1. mTexName=38 mCurrentTexture=2
2. mCurrentCrop=[0,0,0,0] mCurrentTransform=0

#### 4.4.1 材质名称

mTexName的值来源是在消费者被创建时,我们知道最常见的创建消费者的时候是Layer::onFirstRef时会调用:

1. mSurfaceFlingerConsumer = new SurfaceFlingerConsumer(consumer, mTextureName);

创建一个消费者,有两个参数,其中mTextureName是我们目前关注的,如果追溯来源你会发现mTextureName的值来源于glGenTextures,这个函数的实现依赖平台,参考ligagl,它是这样的:

1. *// generate unique (shared) texture names*
2. c->surfaceManager->getToken(n, textures);

还是继续回来看SurfaceFlingerConsumer的创建:

1. SurfaceFlingerConsumer(const sp<IGraphicBufferConsumer>& consumer,
2. uint32\_t tex)
3. : GLConsumer(consumer, tex, GLConsumer::TEXTURE\_EXTERNAL, false, false),
4. mTransformToDisplayInverse(false)
5. GLConsumer::GLConsumer(const sp<IGraphicBufferConsumer>& bq, uint32\_t tex,
6. uint32\_t texTarget, bool useFenceSync, bool isControlledByApp) :
7. ConsumerBase(bq, isControlledByApp),
8. mCurrentTransform(0),
9. mCurrentScalingMode(NATIVE\_WINDOW\_SCALING\_MODE\_FREEZE),
10. mCurrentFence(Fence::NO\_FENCE),
11. mCurrentTimestamp(0),
12. mCurrentFrameNumber(0),
13. mDefaultWidth(1),
14. mDefaultHeight(1),
15. mFilteringEnabled(true),
16. mTexName(tex),
17. mUseFenceSync(useFenceSync),
18. mTexTarget(texTarget),
19. mEglDisplay(EGL\_NO\_DISPLAY),
20. mEglContext(EGL\_NO\_CONTEXT),
21. mCurrentTexture(BufferQueue::INVALID\_BUFFER\_SLOT),
22. mAttached(true)

我们现在可以看出来mTexName的值来源于前面创建的材质名称.   
mCurrentTexture的初始值是INVALID\_BUFFER\_SLOT,也就是-1,后面会在updateAndReleaseLocked时被更改,值的来源是使用的BufferItem的mBuf值,也就是mSlot,这应该是使用的buffer数组的slot值,这个变量的合理取值只有0,1,2三个值(mSlot is the slot index of this buffer ,default INVALID\_BUFFER\_SLOT).

#### 4.4.2 mCurrentCrop裁剪区域

mCurrentCrop的值来源同样是updateAndReleaseLocked调用时被赋值,值的来源是BufferItem的mCrop值.这个值基本一直都是0,只有在视频播放和照相机时会被设置(值的来源有待更深入的研究, mCrop is the current crop rectangle for this buffer slot).

#### 4.4.3 mCurrentTransform 旋转相关

mCurrentTransform的值和前面我们说过的tr值很类似. (mTransform is the current transform flags for this buffer slot. refer to NATIVE\_WINDOW\_TRANSFORM\_\* in ).   
它也跟旋转有关,我们来看下window.h中的定义:

1. */\* parameter for NATIVE\_WINDOW\_SET\_BUFFERS\_TRANSFORM \*/*
2. enum {
3. */\* flip source image horizontally \*/*
4. NATIVE\_WINDOW\_TRANSFORM\_FLIP\_H = HAL\_TRANSFORM\_FLIP\_H ,
5. */\* flip source image vertically \*/*
6. NATIVE\_WINDOW\_TRANSFORM\_FLIP\_V = HAL\_TRANSFORM\_FLIP\_V,
7. */\*rotate source image 90 degrees clock-wise, is applied after TRANSFORM\_FLIP\_{H|V} \*/*
8. NATIVE\_WINDOW\_TRANSFORM\_ROT\_90 = HAL\_TRANSFORM\_ROT\_90,
9. */\* rotate source image 180 degrees \*/*
10. NATIVE\_WINDOW\_TRANSFORM\_ROT\_180 = HAL\_TRANSFORM\_ROT\_180,
11. */\* rotate source image 270 degrees clock-wise \*/*
12. NATIVE\_WINDOW\_TRANSFORM\_ROT\_270 = HAL\_TRANSFORM\_ROT\_270,
13. */\* transforms source by the inverse transform of the screen it is displayed onto. This*
14. *\* transform is applied last \*/*
15. NATIVE\_WINDOW\_TRANSFORM\_INVERSE\_DISPLAY = 0x08
16. };
17. enum {
18. */\* flip source image horizontally (around the vertical axis) \*/*
19. HAL\_TRANSFORM\_FLIP\_H = 0x01,
20. */\* flip source image vertically (around the horizontal axis)\*/*
21. HAL\_TRANSFORM\_FLIP\_V = 0x02,
22. */\* rotate source image 90 degrees clockwise \*/*
23. HAL\_TRANSFORM\_ROT\_90 = 0x04,
24. */\* rotate source image 180 degrees \*/*
25. HAL\_TRANSFORM\_ROT\_180 = 0x03,
26. */\* rotate source image 270 degrees clockwise \*/*
27. HAL\_TRANSFORM\_ROT\_270 = 0x07,
28. */\* don't use. see system/window.h \*/*
29. HAL\_TRANSFORM\_RESERVED = 0x08,
30. };

### 4.5 ConsumerBase(消费者)的dump

子类GLConsumer dump完毕,调用了它的父类的dump函数,基本就是调用了IGraphicBufferConsumer的dump函数.   
生产者消费者这套体系我们之前以前讲过,这里我们就不再详细展开,如果不清楚看下Layer::onFirstRef这个函数就明白了,消费者这个值来自于BufferQueue::createBufferQueue的创建,其中创建了新的BufferQueueConsumer做为消费者.

子类GLConsumer dump完毕,调用了它的父类的dump函数,基本就是调用了IGraphicBufferConsumer的dump函数.   
生产者消费者这套体系我们之前以前讲过,这里我们就不再详细展开,如果不清楚看下Layer::onFirstRef这个函数就明白了,消费者这个值来自于BufferQueue::createBufferQueue的创建,其中创建了新的BufferQueueConsumer做为消费者.

1. void BufferQueue::createBufferQueue(sp<IGraphicBufferProducer>\* outProducer,
2. sp<IGraphicBufferConsumer>\* outConsumer,
3. const sp<IGraphicBufferAlloc>& allocator) {
4. sp<BufferQueueCore> core(new BufferQueueCore(allocator));
5. sp<IGraphicBufferProducer> producer(new BufferQueueProducer(core));
6. sp<IGraphicBufferConsumer> consumer(new BufferQueueConsumer(core));
7. \*outProducer = producer;
8. \*outConsumer = consumer;

当然BufferQueueConsumer的dump函数啥也没写,就调用了BufferQueueCore的dump函数.

打印出来的信息一般是这样的:

1. -BufferQueue mMaxAcquiredBufferCount=1, mDequeueBufferCannotBlock=0,
2. default-size=[1440x2560], default-format=1, transform-hint=00, FIFO(0)={}
3. [00:0xb110e100] state=FREE , 0xb3eb1ec0 [1440x2560:1664, 1]
4. [01:0xb3ec7000] state=FREE , 0xb620d060 [1440x2560:1664, 1]
5. >[02:0xb110e200] state=ACQUIRED, 0xb1111100 [1440x2560:1664, 1]

下面我们按照代码顺序详细解释一下:

#### 4.5.1 队列中的buffer

我们之前在解释queued-frames的含义时已经说过,在画面持续变化时(照相预览,视频播放,窗口滑动,游戏),queued-frames值会是1.表示有新的一帧.   
对应的,有新的frames自然需要有在队列中等待的buffer,对应这段代码会把这个队列打印出来:

1. Fifo::const\_iterator current(mQueue.begin());
2. while (current != mQueue.end()) {
3. fifo.appendFormat("%02d:%p crop=[%d,%d,%d,%d], "
4. "xform=0x%02x, time=%#" PRIx64 ", scale=%s\n",
5. current->mSlot, current->mGraphicBuffer.get(),
6. current->mCrop.left, current->mCrop.top, current->mCrop.right,
7. current->mCrop.bottom, current->mTransform, current->mTimestamp,
8. BufferItem::scalingModeName(current->mScalingMode));
9. ++current;
10. }

对应打印出来的dump信息是这样的:

1. 02:0xb631e480 crop=[0,0,0,0], xform=0x07, time=0xc4d5da9b1e0, scale=FREEZE

* 02是mSlot的值,crop是裁剪区域,xform是旋转,这三个我们上面已经讲过,这里不再展开.
* time是这个buffer被queue的时间(mTimestamp is the current timestamp for this buffer slot. This gets to set by queueBuffer each time this slot is queued. This value is guaranteed to be monotonically increasing for each newly acquired buffer.).
* scale是缩放模式,一般取值如下:

1. enum {
2. */\* the window content is not updated (frozen) until a buffer of*
3. *\* the window size is received (enqueued)*
4. *\*/*
5. NATIVE\_WINDOW\_SCALING\_MODE\_FREEZE = 0,
6. */\* the buffer is scaled in both dimensions to match the window size \*/*
7. NATIVE\_WINDOW\_SCALING\_MODE\_SCALE\_TO\_WINDOW = 1,
8. */\* the buffer is scaled uniformly such that the smaller dimension*
9. *\* of the buffer matches the window size (cropping in the process)*
10. *\*/*
11. NATIVE\_WINDOW\_SCALING\_MODE\_SCALE\_CROP = 2,
12. */\* the window is clipped to the size of the buffer's crop rectangle; pixels*
13. *\* outside the crop rectangle are treated as if they are completely*
14. *\* transparent.*
15. *\*/*
16. NATIVE\_WINDOW\_SCALING\_MODE\_NO\_SCALE\_CROP = 3,
17. };

#### 4.5.2 BufferQueue的基本默认信息

接下来的一段代码会打印BufferQueue的一些基本信息:

1. result.appendFormat("%s-BufferQueue mMaxAcquiredBufferCount=%d, "
2. "mDequeueBufferCannotBlock=%d, default-size=[%dx%d], "
3. "default-format=%d, transform-hint=%02x, FIFO(%zu)={%s}\n",
4. prefix, mMaxAcquiredBufferCount, mDequeueBufferCannotBlock,
5. mDefaultWidth, mDefaultHeight, mDefaultBufferFormat, mTransformHint,
6. mQueue.size(), fifo.string());

一般会打印如下:

1. -BufferQueue mMaxAcquiredBufferCount=1, mDequeueBufferCannotBlock=0, default-size=[1920x1080],
2. default-format=4, transform-hint=04

##### 4.5.2.1 允许同时acquire的buffer的数量

mMaxAcquiredBufferCount是允许同时acquire的buffer的数量,解释如下:

1. *// mMaxAcquiredBufferCount is the number of buffers that the consumer may*
2. *// acquire at one time. It defaults to 1, and can be changed by the consumer*
3. *// via setMaxAcquiredBufferCount, but this may only be done while no*
4. *// producer is connected to the BufferQueue. This value is used to derive*
5. *// the value returned for the MIN\_UNDEQUEUED\_BUFFERS query to the producer.*

基本这个值只能是1,不再深究.

##### 4.5.2.2 dequeueBuffer是否允许被block

1. *// mDequeueBufferCannotBlock indicates whether dequeueBuffer is allowed to*
2. *// block. This flag is set during connect when both the producer and*
3. *// consumer are controlled by the application.*
4. bool mDequeueBufferCannotBlock;

mDequeueBufferCannotBlock几乎总是为0,除非一个应用同时控制了生产者和消费者,这很罕见.

##### 4.5.2.3 buffer default-size默认buffer大小

这两个值的来源应该是BufferQueueConsumer::setDefaultBufferSize函数(不是特别确定,因为这段代码写的不好,严重破坏了封装性).

用处是这样的: mDefaultHeight holds the default height of allocated buffers. It is used in dequeueBuffer if a width and height of 0 are specified.

##### 4.5.2.4 mDefaultBufferFormat默认格式

mDefaultBufferFormat很简单,format含义可以参考前面的解释.   
mDefaultBufferFormat can be set so it will override the buffer format when it isn’t specified in dequeueBuffer.

##### 4.5.2.5 mTransformHint

同样用于旋转.   
mTransformHint is the transform probably applied to buffers of this window. this is only a hint, actual transform may differ.

#### 4.5.3 各个Buffer的信息

接下来是打印BufferSlot中各个buffer的信息,一般打印如下:

1. [00:0xb651d780] state=QUEUED , 0xb6321240 [1080x1920:1152, 1]
2. [01:0xb1513200] state=FREE , 0xb65189c0 [1080x1920:1152, 1]
3. >[02:0xb651d080] state=ACQUIRED, 0xb6518330 [1080x1920:1152, 1]

是由下面的代码打印出来的.

1. for (int s = 0; s < maxBufferCount; ++s) {
2. const BufferSlot& slot(mSlots[s]);
3. const sp<GraphicBuffer>& buffer(slot.mGraphicBuffer);
4. result.appendFormat("%s%s[%02d:%p] state=%-8s", prefix,
5. (slot.mBufferState == BufferSlot::ACQUIRED) ? ">" : " ",
6. s, buffer.get(),
7. BufferSlot::bufferStateName(slot.mBufferState));
8. if (buffer != NULL) {
9. result.appendFormat(", %p [%4ux%4u:%4u,%3X]", buffer->handle,
10. buffer->width, buffer->height, buffer->stride,
11. buffer->format);
12. }
13. result.append("\n");
14. }
15. }

ACQUIRED的buffer前面会打印>,表示这是当前在显示的buffer.   
state表示buffer的状态,取值包括DEQUEUED,QUEUED,FREE,ACQUIRED.我们知道ACQUIRED是在显示的,DEQUEUED是在绘制的,QUEUED绘制完成还未显示的,free是未使用的.   
后面的大小,stride,和format前面都讲过了,这里不再说明.

至此,layer的dump已经说明完毕,我们继续分析Displays的dump.

## Displays的dump.

### 设备名称

首先DisplayDevice是设备的名字,这个可以调用接口设置,但是比较常见的值一般有:Built-in Screen,HDMI Screen,Virtual Screen,wfdservice等等.

### 设备类型

type则是一个枚举值:

1. enum DisplayType {
2. DISPLAY\_ID\_INVALID = -1,
3. DISPLAY\_PRIMARY = HWC\_DISPLAY\_PRIMARY,
4. DISPLAY\_EXTERNAL = HWC\_DISPLAY\_EXTERNAL,
5. DISPLAY\_VIRTUAL = HWC\_DISPLAY\_VIRTUAL,
6. NUM\_BUILTIN\_DISPLAY\_TYPES = HWC\_NUM\_PHYSICAL\_DISPLAY\_TYPES,
7. };
8. enum {
9. HWC\_DISPLAY\_PRIMARY = 0,
10. HWC\_DISPLAY\_EXTERNAL = 1, *// HDMI, DP, etc.*
11. HWC\_DISPLAY\_VIRTUAL = 2, *// wfdservice*
12. HWC\_NUM\_PHYSICAL\_DISPLAY\_TYPES = 2,
13. HWC\_NUM\_DISPLAY\_TYPES = 3,
14. };

### layerStack

layerStack是存储layer的容器,我们知道每个display只会有一个layerstack来存储他要显示的layer,但是不同的display可以使用同一个layerStack,也可以使用不同的layerStack.   
上面我们贴的这个就是两个display使用了不同的layerstack,因为他们显示的内容不一样(电视播放幻灯片).  
后续我们可以研究下什么情况下会导致layerstack切换.

### 屏幕方向

* orient表示屏幕方向
* 后面括号里面的type,是和我们上面说的设备类型完全不同的东西,这个值是由Transform::type算出来的.   
  基本是下面这些值与或非出来的:

1. enum type\_mask {
2. IDENTITY = 0,
3. TRANSLATE = 0x1,
4. ROTATE = 0x2,
5. SCALE = 0x4,
6. UNKNOWN = 0x8
7. };

### powerMode

powerMode表示了屏幕当前的状态,它有以下取值:

1. enum {
2. */\* The display is turned off (blanked). \*/*
3. HWC\_POWER\_MODE\_OFF = 0,
4. */\* The display is turned on and configured in a low power state*
5. *\* that is suitable for presenting ambient information to the user,*
6. *\* possibly with lower fidelity than normal but greater efficiency. \*/*
7. HWC\_POWER\_MODE\_DOZE = 1,
8. */\* The display is turned on normally. \*/*
9. HWC\_POWER\_MODE\_NORMAL = 2,
10. */\* The display is configured as in HWC\_POWER\_MODE\_DOZE but may*
11. *\* stop applying frame buffer updates from the graphics subsystem.*
12. *\* This power mode is effectively a hint from the doze dream to*
13. *\* tell the hardware that it is done drawing to the display for the*
14. *\* time being and that the display should remain on in a low power*
15. *\* state and continue showing its current contents indefinitely*
16. *\* until the mode changes.*
17. *\**
18. *\* This mode may also be used as a signal to enable hardware-based doze*
19. *\* functionality. In this case, the doze dream is effectively*
20. *\* indicating that the hardware is free to take over the display*
21. *\* and manage it autonomously to implement low power always-on display*
22. *\* functionality. \*/*
23. HWC\_POWER\_MODE\_DOZE\_SUSPEND = 3,
24. };

常见的取值有0和2,代表屏幕熄灭和普通情况.   
目前还没看到1和3的情况.

### 其他一些参数

* 设备大小由eglQuerySurface得来,不展开.
* ANativeWindow代表要渲染的本地窗口,这个不同的display之间应该肯定不同.
* flips代表屏幕翻页的次数,其实也就是SurfaceFlinger调用doComposition的次数,也就是屏幕画面更新的次数
* hwcId需要注意的是,如果一个设备不是HWC合成的,这个值会是负数.需要指出的是,这个值不受开关overlay的影响,也就是说如果这个设备是支持HWC的,应该就不会是负数.目前来看,只有开发者选项模拟二级显示出现的display这个会是负数.
* mIsSecure是屏幕自身的属性,mSecureLayerVisible应该会跟播放DRM之类的场景相关
* activeConfig目前看到的总是0,还不清楚作用
* numLayers是这个display上可见的layer的数量
* v,f,s分别代表三个大小:Viewport,Frame,Scissor.

## 5555 display信息的dump

首先会打印当前display的数量,数量基于mDisplays的大小,这个容器在SurfaceFlinger初始化时会生成数据,后面根据收到不同的消息在handleTransactionLocked函数中也会调整.   
正常情况下是1,也就是只有一个display(Built-in Screen),当设备连接了HDMI或者使用了屏幕共享等功能时,会有额外的display加入,比如下面这个:

1. Displays (2 entries)
2. + DisplayDevice: HDMI Screen
3. type=1, hwcId=1, layerStack=6, (1920x1080), ANativeWindow=0xb4d94d08, orient= 0 (type=00000000), flips=1173, isSecure=1,
4. secureVis=0, powerMode=2, activeConfig=0, numLayers=1
5. v:[0,0,1920,1080], f:[0,0,1920,1080], s:[0,0,1920,1080],transform:[[1.000,0.000,-0.000][0.000,1.000,-0.000][0.000,0.000,1.000]]
6. mAbandoned=0
7. -BufferQueue mMaxAcquiredBufferCount=2, mDequeueBufferCannotBlock=0, default-size=[1920x1080], default-format=1, transform-hint=00,
8. FIFO(0)={}
9. [00:0xb6418c80] state=FREE , 0xb43ed880 [1920x1080:1920, 1]
10. [01:0xb43cb300] state=FREE , 0xb640d970 [1920x1080:1920, 1]
11. >[02:0xb43cb280] state=ACQUIRED, 0xb43ed830 [1920x1080:1920, 1]
12. + DisplayDevice: Built-in Screen
13. type=0, hwcId=0, layerStack=0, (1080x1920), ANativeWindow=0xb4d94608, orient= 0 (type=00000000), flips=3140, isSecure=1,
14. secureVis=0, powerMode=2, activeConfig=0, numLayers=2
15. v:[0,0,1080,1920], f:[0,0,1080,1920], s:[0,0,1080,1920],transform:[[1.000,0.000,-0.000][0.000,1.000,-0.000][0.000,0.000,1.000]]

这个是连接了HDMI后的数据.

## 7 EventThread的dump

EventThread的dump信息:

1. VSYNC state: disabled
2. soft-vsync: disabled
3. numListeners=33,
4. events-delivered: 18546

## 8 HWC的dump

HWC的dump从这句话开始:

1. h/w composer state:
2. h/w composer present and enabled

其中present和enable与否由以下参数决定:

1. result.appendFormat(" h/w composer %s and %s\n",
2. hwc.initCheck()==NO\_ERROR ? "present" : "not present",
3. (mDebugDisableHWC || mDebugRegion || mDaltonize
4. || mHasColorMatrix) ? "disabled" : "enabled");

只要init成功,就是present;而是否enable则由众多debug选项决定.   
mDebugDisableHWC我们很熟悉,就是我们在开发者选项里勾选的禁用硬件叠加.   
mDebugRegion是开发者选项里面的开发者选项中的显示屏幕更新,也可以通过属性debug.sf.showupdates来控制.   
mDaltonize被1014号命令控制,暂时先不研究.   
mHasColorMatrix被1015号命令控制.

上述四个选项有任意一个打开,就composer的状态就会使disable.

接下来进入HWComposer的dump函数:   
- 首先打印的是hwc的version,目前一般0103或者0104居多.   
- mDebugForceFakeVSync的值源自属性debug.sf.no\_hw\_vsync,打开之后会使用纯软件模拟VSync信号.   
- 接下来是打印各个display的一些基本信息:

1. Display[0] configurations (\* current):
2. \* 0: 1440x2560, xdpi=508.000000, ydpi=516.062988, secure=1 refresh=16666667

这些基本信息取自对应display设备的DisplayData的config信息.

接下来就开始打印各个layer的信息,这个也是我们最常见到的layer信息.   
依然是分display打印的,分别打印每个display的可见layer,通过getVisibleLayersSortedByZ获取的.

1. type | handle | hint | flag | tr | blnd | format | source crop(l,t,r,b) | frame | dirtyRect | name
2. ------------+----------+----------+----------+----+-------+----------+-----------------------------------+---------------------------+-------------------
3. HWC | aed1c650 | 0002 | 0000 | 00 | 0100 | ? 00000011 | 0.0, 0.0, 1920.0, 1080.0 | 0, 0, 960, 540 | [ 0, 0, 1920, 1080] | SurfaceView
4. HWC | aed1c470 | 0000 | 0000 | 00 | 0105 | RGBA\_8888 | 0.0, 0.0, 2560.0, 1440.0 | 0, 0, 960, 540 | [ 0, 0, 2560, 1440] | SurfaceView
5. FB TARGET | b3ec5240 | 0000 | 0000 | 00 | 0105 | RGBA\_8888 | 0.0, 0.0, 960.0, 540.0 | 0, 0, 960, 540 | [ 0, 0, 0, 0] | HWC\_FRAMEBUFFER\_TARGET

逐次来看下:   
- type表示合成类型,可取的值包括以下几种: “GLES”,”HWC”,”BKGND”, “FB TARGET”,”SIDEBAND”,”HWC\_CURSOR”,”FB\_BLIT”,”UNKNOWN”.(更详细的一些解释可以参见hwcomposer.h)   
- handle是个标识符,可以跟某个buffer的handle对应.(Handle of buffer to compose. This handle is guaranteed to have been allocated from gralloc using the GRALLOC\_USAGE\_HW\_COMPOSER usage flag. If the layer’s handle is unchanged across two consecutive prepare calls and the HWC\_GEOMETRY\_CHANGED flag is not set for the second call then the HWComposer implementation may assume that the contents of the buffer have not changed. )   
- hint is bit mask set by the HWC implementation during (\*prepare)().It is preserved between (\*prepare)() calls, unless the HWC\_GEOMETRY\_CHANGED flag is set, in which case it is reset to 0.   
下面是hint的可能取值.

1. */\**
2. *\* hwc\_layer\_t::hints values*
3. *\* Hints are set by the HAL and read by SurfaceFlinger*
4. *\*/*
5. enum {
6. */\**
7. *\* HWC can set the HWC\_HINT\_TRIPLE\_BUFFER hint to indicate to SurfaceFlinger*
8. *\* that it should triple buffer this layer. Typically HWC does this when*
9. *\* the layer will be unavailable for use for an extended period of time,*
10. *\* e.g. if the display will be fetching data directly from the layer and*
11. *\* the layer can not be modified until after the next set().*
12. *\*/*
13. HWC\_HINT\_TRIPLE\_BUFFER = 0x00000001,
14. */\**
15. *\* HWC sets HWC\_HINT\_CLEAR\_FB to tell SurfaceFlinger that it should clear the*
16. *\* framebuffer with transparent pixels where this layer would be.*
17. *\* SurfaceFlinger will only honor this flag when the layer has no blending*
18. *\**
19. *\*/*
20. HWC\_HINT\_CLEAR\_FB = 0x00000002
21. };

* flag的常见取值只有0和1,1代表这个layer由SF处理.如果我们关闭硬件叠加,可以看到layer的这个值都是1.都是这个值不能跟type画等号,在layer众多的时候,layer都是GLES类型,但是只有很少的layer的flag是1(原因有待研究,1的详细解释是这样的:HWC\_SKIP\_LAYER is set by SurfaceFlnger to indicate that the HAL shall not consider this layer for composition as it will be handled by SurfaceFlinger ).
* tr是transform的缩写,但是注意这里的tr和前面单独layer dump时的tr或transform的值这不能划等号的,这里的tr的含义是transformation to apply to the buffer during composition.
* blnd的含义如下:

1. enum {
2. */\* no blending \*/*
3. HWC\_BLENDING\_NONE = 0x0100,
4. */\* ONE / ONE\_MINUS\_SRC\_ALPHA \*/*
5. HWC\_BLENDING\_PREMULT = 0x0105,
6. */\* SRC\_ALPHA / ONE\_MINUS\_SRC\_ALPHA \*/*
7. HWC\_BLENDING\_COVERAGE = 0x0405
8. };

* format前面说过很多次了,这里不再说了.

## REF

[SurfaceFlinger的dump信息详解](http://lee_3do.leanote.com/post/SurfaceFlinger%E7%9A%84dump%E4%BF%A1%E6%81%AF%E8%AF%A6%E8%A7%A3)

[引发的问题：如果只是简单的将视频小窗口层输出到video层，壁纸和桌面混合输出到Graphic层，就会导致video被完全遮挡，因为video层在最底下。](http://www.chongchonggou.com/g_492198516.html)

# 参考

[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)

[Android应用程序与SurfaceFlinger服务的连接过程分析](http://blog.csdn.net/Luoshengyang/article/details/7857163)

TASk

GPU 配置电压到UI显示原理