# 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，如图2-4所示。从图2-4中可以看出，在每个SharedClient中，最多可以创建31个SharedBufferStack，每个Surface都对应一个SharedBufferStack，也就是一个window。

一个SharedClient对应一个Android应用程序，而一个Android应用程序可能包含多个窗口，即Surface。也就是说SharedClient包含的是SharedBufferStack的集合。因为最多可以创建31个SharedBufferStack，这也意味着一个Android应用程序最多可以包含31个窗口，同时每个SharedBufferStack中又包含了两个（低于4.1版本）或者三个（4.1及以上版本）缓冲区，即在后面的显示刷新机制中会提到的双缓冲和**三重缓冲技术**。



图2-4　Android显示框架

最后总结起来显示整体流程分为三个模块：应用层绘制到缓存区，SurfaceFlinger把缓存区数据渲染到屏幕，由于是两个不同的进程，所以使用Android的匿名共享内存SharedClient缓存需要显示的数据来达到目的。

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

从图2-5中可以看出，绘制过程首先是CPU准备数据，通过Driver层把数据交给CPU渲染，其中CPU主要负责Measure、Layout、Record、Execute的数据计算工作，GPU负责Rasterization（栅格化）、渲染。由于图形API不允许CPU直接与GPU通信，而是通过中间的一个图形驱动层（Graphics Driver）来连接这两部分。图形驱动维护了一个队列，CPU把display list添加到队列中，GPU从这个队列取出数据进行绘制，最终才在显示屏上显示出来。



图2-5　渲染数据流程图

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



图2-6　理想状态下的绘制操作

但是Android系统很有可能无法及时完成那些复杂的界面渲染操作。Android系统每隔16ms发出VSYNC信号，触发对UI进行渲染，如果每次渲染都成功，这样就能够达到流畅的画面所需的60FPS。即为了实现60FPS，就意味着程序的大多数绘制操作都必须在16ms内完成。

如果某个操作花费的时间是24ms，系统在得到VSYNC信号时就无法进行正常渲染，这样就发生了丢帧现象。那么用户在32ms内看到的会是同一帧画面。主要场景在执行动画或者滑动ListView时更容易感知到卡顿不流畅，是因为这里的操作相对复杂，容易发生丢帧的现象，从而感觉卡顿。有很多原因可以导致CPU或者GPU负载过重从而出现丢帧现象：可能是你的Layout太过复杂，无法在16ms内完成渲染；可能是UI上有层叠太多的绘制单元；还有可能是动画执行的次数过多。

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

### 刷新机制

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

从Android 4.1（Jelly Bean）开始，Android OS开发团队便力图在每个版本中解决一个重要问题。作为严重影响Android口碑问题之一的UI流畅性差的问题，首先在Android 4.1版本中得到了有效处理。其解决方法即在4.1版本推出的Project **Butter**。Project Butter对Android Display系统进行了重构，引入三个核心元素：**VSYNC、Triple Buffer和Choreographer**。其中，VSYNC是理解Project Buffer的核心。VSYNC是Vertical Synchronization（垂直同步）的缩写，是一种在PC上已经**很早就广泛使用的技术**，读者可简单地把它认为是一种定时中断。Choreographer起调度的作用，将绘制工作统一到VSYNC的某个时间点上，使应用的绘制工作有序。接下来，本文将围绕VSYNC来介绍Android Display系统的工作方式。

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

·双缓冲：显示内容的数据内存，为什么要用双缓冲，我们知道在Linux上通常使用Framebuffer来做显示输出，当用户进程更新Framebuffer中的数据后，显示驱动会把Framebuffer中每个像素点的值更新到屏幕，但这样会带来一个问题，如果上一帧的数据还没有显示完，Framebuffer中的数据又更新了，就会带来**残影的问题，给用户直观的感觉就会有闪烁感**，所以普遍采用了双缓冲技术。双缓冲意味着要使用两个缓冲区（在SharedBufferStack中），其中一个称为Front Buffer，另外一个称为Back Buffer。**UI总是先在Back Buffer中绘制**，后台绘制好，然后再和Front Buffer交换，渲染到显示设备中。即只有当另一个buffer的数据准备好后，通过io\_ctrl来通知显示设备切换Buffer。

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

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

·CALLBACK\_INPUT：优先级最高，与输入事件有关。

·CALLBACK\_ANIMATION：第二优先级，与动画有关。

·CALLBACK\_TRAVERSAL：最低优先级，与UI控件绘制有关。

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



图2-7　没有VSync信息的刷新

（1）没有VSync信号同步

我们以16ms为单位来进行分析：

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

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

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

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

（2）有VSync信号同步

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

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



图2-8　有VSync的绘制



图2-9　双缓冲下的时序图（CPU的第三个A->B）



图2-10　双缓冲下CPU/GPU FPS小于刷新频率时序图

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

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

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

为什么CPU不能在第二个16ms处即VSync到来就开始工作呢？很明显，原因就是只有两个Buffer。如果有第三个Buffer存在，CPU就可以开始工作，而不至于空闲。于是在Andoird 4.1以后，引出了第三个缓冲区：Triple Buffer。Triple Buffer利用CPU/GPU的空闲等待时间提前准备好数据，并不一定会使用。

[注意　在大部分情况下，只使用到双缓存，只有在需要时，才会用三缓冲来增强，这时可以把输入的延迟降到最少，保持画面的流畅。

引入Triple Buffer后的刷新时序如图2-11所示。



图2-11　使用Triple Buffer时序图

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

从以上的分析来看，Android系统在显示机制上解决了Android UI不流畅的问题，并且从Google I/O2012给出的视频来看，其效果也达到预期。但实际在应用开发过程中仍然存在卡顿的现象。因为VSync中断处理的线程优先级一定要最高，否则即使接收到VSync中断，不能及时处理，也是徒劳无功。

# 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。

### 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类的构造函数实现在文件frameworks/base/cmds/bootanimation/BootAnimation.cpp中，如下所示：

Libs/Gui

在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。

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：

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。

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

1. sp<ISurfaceComposerClient> SurfaceFlinger::createConnection()
2. {
3. sp<ISurfaceComposerClient> bclient;
4. sp<Client> client(**new** Client(**this**));
5. status\_t err = client->initCheck();
6. **if** (err == NO\_ERROR) {
7. bclient = client;
8. }
9. **return** bclient;
10. }

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

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

Client类有两个成员变量mFlinger和mNameGenerator，它们的类型分别为sp<SurfaceFlinger>和int32\_t，前者指向了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

## 代码模板

Hg

1. Java start
2. startActivitySafely(Intent intent, Object tag) {
3. intent.addFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK);
4. try {
5. startActivity(intent);
6. } catch (ActivityNotFoundException e) {
7. ......
8. } catch (SecurityException e) {
9. ......
10. }
11. End

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。

## 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

F

Android系统的图形处理相关的模块，就不得不提surfaceflinger，这是由init进程所启动的 守护进程，在init.rc中该服务如下：

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);

// 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);

}

#### MQ.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();

};

...

}

### SF.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. //当应用和sf的vsync偏移量一致时，则只创建一个EventThread线程
40. if (vsyncPhaseOffsetNs != sfVsyncPhaseOffsetNs) {
41. sp<VSyncSource> vsyncSrc = new DispSyncSource(&mPrimaryDispSync,
42. vsyncPhaseOffsetNs, true, "app");
43. mEventThread = new EventThread(vsyncSrc);
44. sp<VSyncSource> sfVsyncSrc = new DispSyncSource(&mPrimaryDispSync,
45. sfVsyncPhaseOffsetNs, true, "sf");
46. mSFEventThread = new EventThread(sfVsyncSrc);
47. mEventQueue.setEventThread(mSFEventThread);
48. } else {
49. //创建DispSyncSource对象【2.6】
50. sp<VSyncSource> vsyncSrc = new DispSyncSource(&mPrimaryDispSync,
51. vsyncPhaseOffsetNs, true, "sf-app");
52. //创建线程EventThread 【见小节2.7】
53. mEventThread = new EventThread(vsyncSrc);
54. //设置EventThread 【见小节2.8】
55. mEventQueue.setEventThread(mEventThread);
56. }
57. //【见小节2.9】
58. mEventControlThread = new EventControlThread(this);
59. mEventControlThread->run("EventControl", PRIORITY\_URGENT\_DISPLAY);
60. //当不存在HWComposer时，则设置软件vsync
61. if (mHwc->initCheck() != NO\_ERROR) {
62. mPrimaryDispSync.setPeriod(16666667);
63. }
64. //初始化绘图状态
65. mDrawingState = mCurrentState;
66. //初始化显示设备
67. initializeDisplays();
68. //启动开机动画【2.10】
69. startBootAnim();
70. } ......
71. }

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)。

### 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

J

[-> 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

Df

[-> HWComposer.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。

### 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);

}

}

### 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

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

# Choreographer原理

http://gityuan.com/2017/02/25/choreographer/

# 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/

# 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。

# 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