首页 业界 移动 云计算 研发 论坛 博客 下载 更多

您还未登录！| 登录 | 注册 | 帮助

个人资料

zirconsdu

访问：46579次
积分：1106分
排名：第9937名

原创：55篇 转载：72篇
译文：0篇 评论：20条

GUI显示系统之SurfaceFlinger

分类： android display 2013-06-09 13:58 657人阅读 评论(0) 收藏 举报

SurfaceFlinger

目录(?) [-]

转载自http://blog.csdn.net/uiop78uiop78/article/details/8954508

介绍了Surface, SurfaceTexture, BufferQueue, VSYNC generation, Render等内容。

后面的几篇博文中我将按照如下的顺序贴出各小节内容。

文章都是通过阅读源码分析出来的，还在不断完善与改进中，其中难免有些地方理解得不对，欢迎大家批评指正

第1章 GUI系统之SurfaceFlinger

在进入GUI系统的学习前，建议大家可以先阅读本书应用篇中的“OpenGLES”章节，并参阅OpenGL ES官方指

南。因为Android的GUI系统是基于OpenGL/EGL来实现的，如果没有一定基础的话，分析源码时有可能会“事倍功半”。

1.1 OpenGLES与EGL

SurfaceFlinger虽然是GUI的核心，但相对于OpenGL ES来讲，它其实只是一个“应用”。

对于没有做过OpenGLES开发的人来讲，理解这部分的内容还是有一定难度的，特别是容易对系统中既有EGL/OpenGLES，又有 SurfaceFlinger、GraphicPlane、DisplayHardware、Gralloc、FramebufferNativeWindow等一系列陌生的模块感到混乱而无序。

的确如此，假如不先理清这些模块的相互关系，对于我们深入研究整个Android显示系统就是一个很大的障碍。有鉴于此，我们先来从框架的高度审视一下它们之间看似错综复杂、剪不断理还乱的依赖。

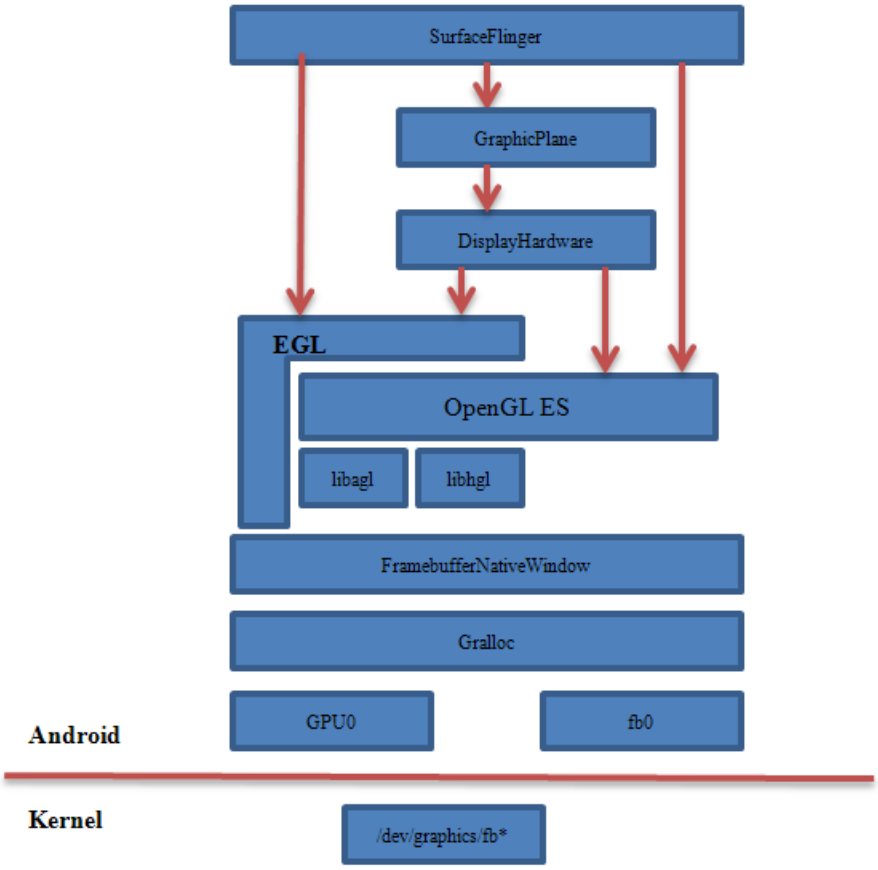

图 11　1 SurfaceFlinger与OpenGLES等模块关系

我们根据上面这个图，由底层往上层来逐步分析整个架构：

1. Linux内核提供了统一的framebuffer显示驱动，设备节点/dev/graphics/fb*或者/dev/fb*，以fb0表示第一个Monitor，当前实现中只用到了一个显示屏

2. Android的HAL层提供了Gralloc，分为fb和gralloc两个设备。前者负责打开内核中的framebuffer、初始化配置，以及提供 post、setSwapInterval等操作，后者则管理帧缓冲区的分配和释放。上层只能通过Gralloc访问帧缓冲区，这样一来就实现了有序的封 装保护

3. 由于OpenGL ES是一个通用的函数库，在不同的平台系统上需要被“本地化”——即把它与具体平台上的窗口系统建立起关联，这样才能保证它正常工作。从 FramebufferNativeWindow这个名称就能判断出来，它就是

将OpenGL ES在Android平台上本地化的中介之一。后面我们还会看到应用程序端所使用的另一个“本地窗口”。为OpengGL ES配置本地窗口的是EGL

4. OpenGL或者OpenGL ES 更多的只是一个接口协议，实现上既可以采用软件，也能依托于硬件。这一方面给产品开发带来了灵活性，我们可以根据成本与市场定位来决定具体的硬件设计，从 而达到很好的定制需求;另一方面，既然有多种实现的可能，那么OpenGL ES在运行时是如何取舍的呢？这也是EGL的作用之一。它会去读取egl.cfg这个配置文件，然后根据用户的设定来动态加载libagl(软件实现)或 者libhgl(硬件实现)。然后上层才可以正常使用各种glXXX接口

5. SurfaceFlinger中持有一个GraphicPlane成员变量mGraphicPlanes来描述“显示屏”;GraphicPlane类中 又包含了一个DisplayHardware对象实例(mHw)。具体是在SurfaceFlinger::readyToRun中，完成对它们的创建与 初始化。并且DisplayHardware在初始化时还将调用eglInitialize、eglCreateWindowSurface等接口，利用 EGL来完成对OpenGLES环境的搭建。其中:

```
surface =eglCreateWindowSurface(display, config, mNativeWindow.get(), NULL);
```

mNativeWindow 就是一个FramebufferNativeWindow对象。DisplayHardware为OpenGL ES设置了“本地化”所需的窗口

6. 很多模块都可以调用OpenGLES提供的API(这些接口以“gl”为前缀，比如glViewport、glClear、glMatrixMode、glLoadIdentity等等)，包括SurfaceFlinger、DisplayHardware等

7. 与OpenGL ES相关的模块，可以分为如下几类:

Ø 配置类

即帮助OpenGL ES完成配置的，包括EGL、DisplayHardware都可以认为是这一类

Ø 依赖类

也就是OpenGL ES要正常运行起来所依赖的“本地化”的东西，上图中是指FramebufferNativeWindow

Ø 使用类

使用者也可能是配置者，比如DisplayHardware既扮演了“帮助”OpenGL的角色，同时它也是其使用方。另外只要处在与OpenGL ES同一个环境(Context)中的模块，都可以使用它来完成操作，比如SurfaceFlinger

如果是对EGL的作用、工作方式以及它所提供的重要接口等有不明白的，强烈建议大家先阅读官方文档以及本书应用篇中的章节，否则会大大影响后面的学习和理解。

1.1 Gralloc与Framebuffer

相信做过Linux开发的人对framebuffer不会太陌生，它是内核系统提供的一个与硬件无关的显示抽象层。之所以称之为buffer，是由 于它也是系统存储空间的一部分，是一块包含屏幕显示信息的缓冲区。由此可见，在“一切都是文件”的Linux系统中，Framebuffer被看成了终端 monitor的“化身”。它借助于文件系统向上层提供统一而方便的操作接口，从而让用户空间程序可以不用修改就能适应多种屏幕——无论这些屏幕是哪家厂 商、什么型号，都由framebuffer内部来兼容。

在Android系统中，framebuffer提供的设备文件节点是/dev/graphics/fb*。因为理论上支持多个屏幕显示，所以fb按数字序号进行排列，即fb0、fb1等等。其中第一个fb0是主显示屏幕，必须存在。如下是某设备的fb设备截图:

图 11　2 fb节点

根据前面章节学习过的知识，Android中各子系统通常不会直接基于Linux驱动来实现，而是由HAL层间接引用底层架构，在显示系统中也同样如此——它借助于HAL层来操作帧缓冲区，而完成这一中介任务的就是Gralloc，下面我们分几个方面来介绍。

<1> Gralloc的加载

Gralloc对应的模块是由FramebufferNativeWindow(OpenGLES的本地窗口之一，后面小节有详细介绍)在构造时加载的，即：

hw_get_module(HWC_HARDWARE_MODULE_ID, &mModule);

这个hw_get_module函数我们在前面已经见过很多次了，它是上层加载HAL库的入口，这里传入的模块ID名为：

#define GRALLOC_HARDWARE_MODULE_ID "gralloc"

按照hw_get_module的作法，它会在如下路径中查找与ID值匹配的库：

#define HAL_LIBRARY_PATH1 "/system/lib/hw"

#define HAL_LIBRARY_PATH2 "/vendor/lib/hw"

lib库名有如下几种形式：

gralloc.[ro.hardware].so

gralloc.[ro.product.board].so

gralloc.[ro.board.platform].so

gralloc.[ro.arch].so

或者当上述的系统属性组成的文件名都不存在时，就使用默认的：

gralloc.default.so

最后这个库是Android原生态的实现，位置在hardware/libhardware/modules/gralloc/中，它由gralloc.cpp、framebuffer.cpp和mapper.cpp三个主要源文件编译生成。

<2> Gralloc提供的接口

Gralloc对应的库被加载后，我们来看下它都提供了哪些接口方法。

由于Gralloc代表的是一个hw_module_t，这是HAL中统一定义的硬件模块描述体，所以和其它module所能提供的接口是完全一致的：

```
/*hardware/libhardware/include/hardware/Hardware.h*/

typedef struct hw_module_t {…
```

```
    structhw_module_methods_t* methods;
            …
} hw_module_t;

typedef struct hw_module_methods_t {
    int (*open)(const structhw_module_t* module, const char* id,
            structhw_device_t** device);
} hw_module_methods_t;
```

这个open接口可以帮助上层打开两个设备，分别是：

#defineGRALLOC_HARDWARE_FB0 "fb0"

以及 #define GRALLOC_HARDWARE_GPU0 "gpu0"

“fb0”就是我们前面说的主屏幕，gpu0负责图形缓冲区的分配和释放。这两个设备将由FramebufferNativeWindow中的fbDev和grDev成员变量来管理。

```
/*frameworks/native/libs/ui/FramebufferNativeWindow.cpp*/
FramebufferNativeWindow::FramebufferNativeWindow()
        : BASE(),fbDev(0), grDev(0), mUpdateOnDemand(false)
{…
    err = framebuffer_open(module, &fbDev);
    err =gralloc_open(module, &grDev);
```

这两个open函数分别是由hardware/libhardware/include/hardware目录下的Fb.h和Gralloc.h头 文件提供的打开fb及gralloc设备的便捷实现。其中fb对应的设备名为GRALLOC_HARDWARE_FB0，gralloc则是GRALLOC_HARDWARE_GPU0。各硬件生产商可以根据自己的平台配置来实现fb和gralloc的打开、关闭以及管理，比如 hardware/msm7k/libgralloc就是一个很好的参考例子。

原生态的实现在hardware/libhardware/modules/gralloc中，对应的是gralloc_device_open@Gralloc.cpp。在这个函数中，根据设备名来判断是打开fb或者gralloc。

```
/*hardware/libhardware/modules/gralloc/Gralloc.cpp*/
int gralloc_device_open(const hw_module_t* module, const char* name,hw_device_t** device)
{
    int status = -EINVAL;
    if (!strcmp(name, GRALLOC_HARDWARE_GPU0)) {//打开gralloc设备
        …
```

```
    } else {
        status = fb_device_open(module, name, device);//否则就是fb设备
    }
    return status;
}
```

先来大概看下framebuffer设备的打开过程：

```
/*hardware/libhardware/modules/gralloc/Framebuffer.cpp*/
int fb_device_open(hw_module_t const* module, const char* name,hw_device_t** device)
{
    int status = -EINVAL;
    if (!strcmp(name,GRALLOC_HARDWARE_FB0)) {//设备名是否正确
        fb_context_t *dev =(fb_context_t*)malloc(sizeof(*dev));//分配hw_device_t空间，这是一个“壳”
        memset(dev, 0,sizeof(*dev));//初始化，良好的编程习惯
                ...
       dev->device.common.close = fb_close;//这几个接口是fb设备的核心
       dev->device.setSwapInterval = fb_setSwapInterval;
        dev->device.post          = fb_post;
                ...
        private_module_t* m =(private_module_t*)module;
        status = mapFrameBuffer(m);//内存映射，以及参数配置
        if (status >= 0) {
            ...
            *device =&dev->device.common;//“壳”和“核心”的关系
        }
    }
    return status;
}
```

其中fb_context_t是framebuffer内部使用的一个类，它包含了众多信息，而最终返回的device只是其内部的device.common。这种“通用和差异”并存的编码风格在HAL层非常常见，大家要做到习以为常。

Struct类型fb_context_t里的唯一成员就是framebuffer_device_t，这是对frambuffer设备的统一描述。一个标准的fb设备通常要提供如下的函数实现：

Ø int(*post)(struct framebuffer_device_t* dev, buffer_handle_t buffer);

将buffer数据post到显示屏上。要求buffer必须与屏幕尺寸一致，并且没有被locked。这样的话buffer内容将在下一次VSYNC中被显示出来

Ø int(*setSwapInterval)(struct framebuffer_device_t* window, int interval);

设置两个缓冲区交换的时间间隔

Ø int(*setUpdateRect)(struct framebuffer_device_t* window, int left, int top,

int width, int height);

设置刷新区域，需要framebuffer驱动支持“update-on-demand”。也就是说在这个区域外的数据很可能被认为无效

我们再来解释下framebuffer_device_t中一些重要的成员变量，如下表：

表格 11　1 framebuffer_device_t中的重要成员变量

变量	描述
uint32_t flags	标志位，指示framebuffer的属性配置
uint32_t width; uint32_t height;	framebuffer的宽和高，以像素为单位
int format	framebuffer的像素格式，比如：HAL_PIXEL_FORMAT_RGBA_8888 HAL_PIXEL_FORMAT_RGBX_8888 HAL_PIXEL_FORMAT_RGB_888 HAL_PIXEL_FORMAT_RGB_565等等
float xdpi; float ydpi;	x和y轴的密度(pixel per inch)
float fps	屏幕的每秒刷新频率，假如无法正常从设备获取的话，默认设置为60Hz
int minSwapInterval; int maxSwapInterval;	该framebuffer支持的最小和最大缓冲交换时间

到目前为止，我们还没看到系统是如何打开具体的fb设备、以及如何对fb进行配置，这些工作都是在mapFrameBuffer()完成的。这个函数首先尝试打开(调用open，权限为O_RDWR)如下路径中的fb设备：

"/dev/graphics/fb%u"或者 "/dev/fb%u",其中%u当前的实现中只用了“0”，也就是只会打开一个fb，虽然Android从趋势上看是要支持多屏幕的。成功打开fb后，我们通过：

ioctl(fd, FBIOGET_FSCREENINFO, &finfo);

ioctl(fd, FBIOGET_VSCREENINFO, &info)

来得到显示屏的一系列参数，同时通过

ioctl(fd, FBIOPUT_VSCREENINFO, &info)来对底层fb进行配置。

这个函数的另一重要任务，就是对fb做内存映射，主要语句如下：

```
   void* vaddr = mmap(0,fbSize, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);

  module->framebuffer->base = intptr_t(vaddr);

   memset(vaddr, 0, fbSize);
```

所以映射地址是module->framebuffer->base，这个module对应的是前面hw_get_module(GRALLOC_HARDWARE_MODULE_ID,&module)得到的hw_module_t(被强制类 型转化为private_module_t，大家可以自己看下这个struct)。

接下来再看下对gralloc设备的打开操作，它相对fb简单些，如下所示：

```
/*hardware/libhardware/modules/gralloc/Gralloc.cpp*/

int gralloc_device_open(const hw_module_t* module, const char* name,hw_device_t** device)

{

   int status = -EINVAL;

   if (!strcmp(name,GRALLOC_HARDWARE_GPU0)) {

       gralloc_context_t*dev;//做法和fb类似

       dev =(gralloc_context_t*)malloc(sizeof(*dev));//分配空间

       /* initialize ourstate here */

       memset(dev, 0,sizeof(*dev));

                   …

      dev->device.alloc   =gralloc_alloc; //从提供的接口来看，gralloc和分配/释放有关系

      dev->device.free    =gralloc_free;

…

}
```

与fb相似的部分我们就不多做介绍了。因为gralloc担负着图形缓冲区的分配与释放，所以它提供了两个最重要的实现即alloc和free。这里我们先不深入分析了，只要知道gralloc所提供的功能就可以了。

我们以下面简图来小结对Gralloc的分析。

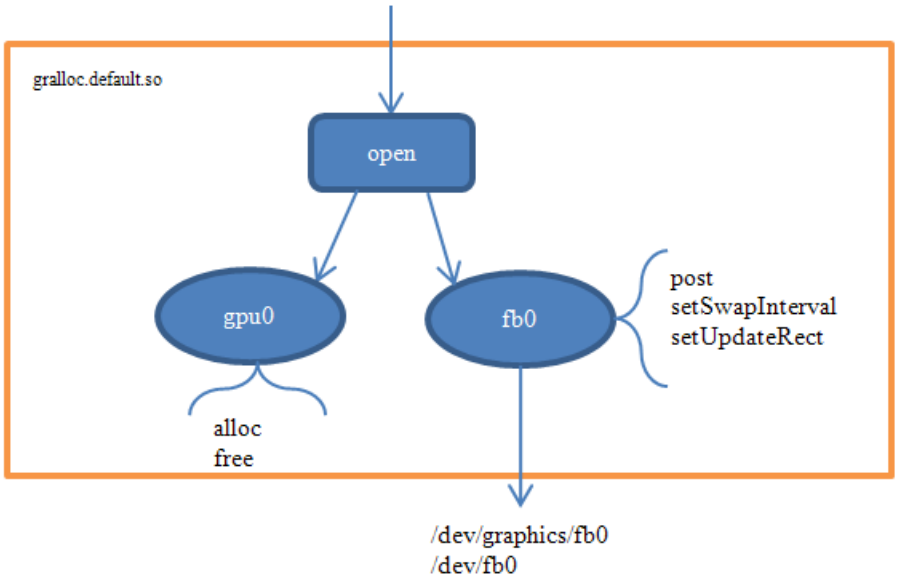

图 11　3 Gralloc简图

1.1 Android中的本地窗口

在OpenGL的学习过程中，我们不断提及“本地窗口”(NativeWindow)这一概念。那么对于Android系统来说，它是如何将OpenGL ES本地化的呢，或者说，它提供了什么样的本地窗口？

根据整个Android系统的GUI设计理念，我们不难猜想到至少需要两种本地窗口：

Ø 面向管理者(SurfaceFlinger)

既然SurfaceFlinger扮演了系统中所有UI界面的管理者，那么它无可厚非地需要直接或间接地持有“本地窗口”。从前一小节我们已经知道，这个窗口就是FramebufferNativeWindow

Ø 面向应用程序

我们先给出答案，这类窗口是SurfaceTextureClient

有不少读者可能会觉得困惑，为什么需要两种窗口，同一个系统不是应该只有一种窗口吗？比如这样子：

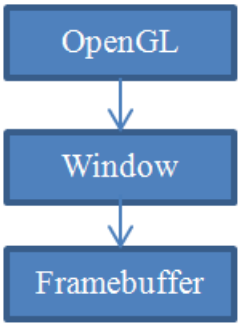

图 11　4理想的窗口系统

这个图中，由Window来管理Framebuffer。我们打个比方来说，OpenGL就像是一台通用的打印机一样，只要输入正确的指令，它就能 按照要求输出结果;而Window则是“纸”，它是用来承载OpenGL的输出结果的。OpenGL并不介意Window是A4纸或者是A6，甚至是塑料 纸也没有关系，对它来说都只是“本地窗口”。

理解了这个图后，我们再来思考下，这样的模型是否能符合Android的要求？假如整个系统仅有一个需要显示UI的程序，我们有理由相信它是可以胜任的。但是如果有N个UI程序的情况呢？Framebuffer显然只有一个，不可能让各个应用程序自己单独管理。

这样子问题就来了，该如何改进呢？下面这个方法如何？

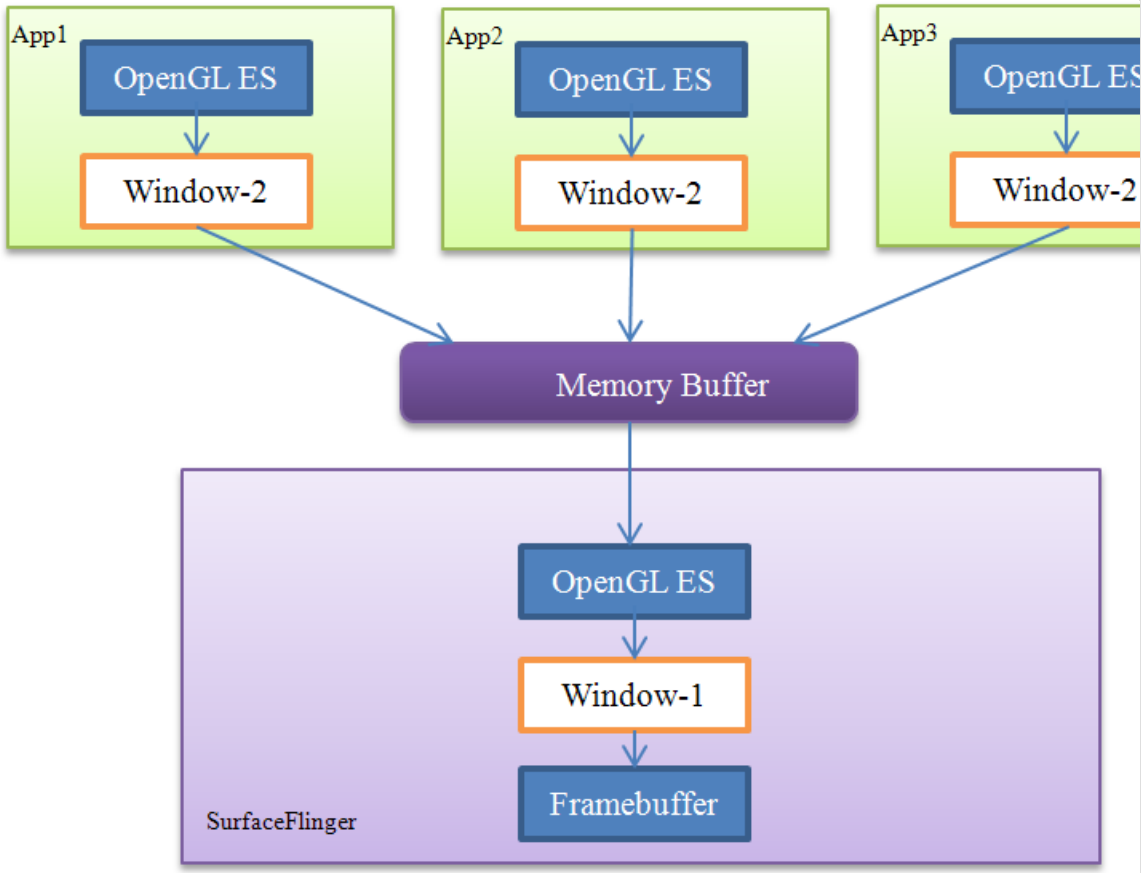

图 11　5 改进的窗口系统

在这个改进的窗口系统中，我们有了两类本地窗口，即Window-1和Window-2。第一种窗口是能直接显示在终端屏幕上的——它使用了帧缓冲 区，而后一种Window实际上是从内存缓冲区分配的空间。当系统中存在多个应用程序时，这能保证它们都可以获得一个“本地窗口”，并且这些窗口最终也能 显示到屏幕上——SurfaceFlinger会收集所有程序的显示需求，对它们做统一的图像混合操作(有点类似于AudioFlinger)，然后输出 到自己的Window-1上。

当然，这个改进的窗口系统有一个前提，即应用程序与SurfaceFlinger都是基于OpenGL ES来实现的。有没有其它选择呢？答案是肯定的，比如应用程序端完全可以采用Skia等第三方的图形库，只要保持它们与SurfaceFlinger间的 “协议”不变就可以了，如下所示：

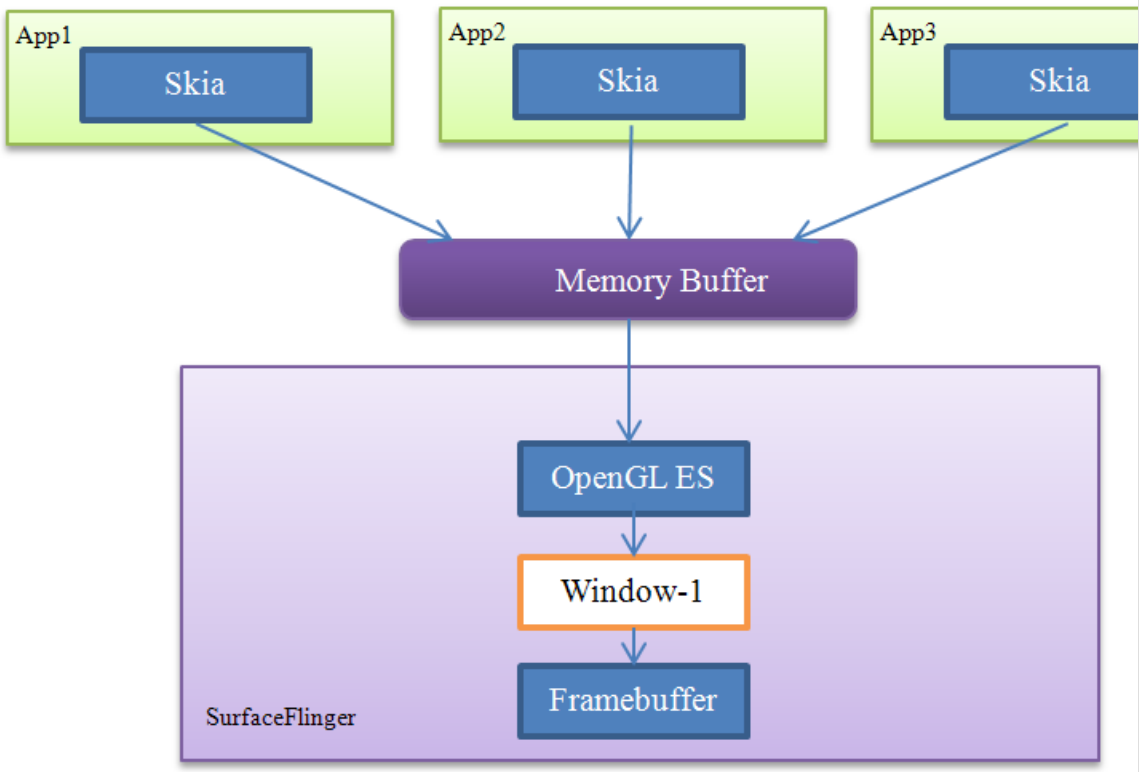

图 11　6 另一种改进的窗口系统

理论上来说，采用哪一种方式都是可行的。不过对于开发人员，特别是没有OpenGLES项目经验的人而言，前一种系统的门槛相对较高。事实 上，Android系统同时提供了这两种实现来供上层选择。正常情况下我们按照SDK向导生成的apk应用，就属于后面的情况;而对于希望使用 OpenGLES来完成复杂的界面渲染的应用开发者，也可以使用GLSurfaceView来达到目标。

在接下来的源码分析中，我们将对上面所提出的假设做进一步验证。

1.1.1 FramebufferNativeWindow

先把EGL创建一个Window Surface的函数原型列出如下：

EGLSurface eglCreateWindowSurface(EGLDisplay dpy, EGLConfig config,

NativeWindowType window, const EGLint *attrib_list);

显然不论是哪一种本地窗口，它都必须要与NativeWindowType保持一致，否则就无法正常使用EGL了。先从数据类型的定义来看下这个window参数有什么特别之处：

```
/*frameworks/native/opengl/include/egl/Eglplatform.h*/
typedef  EGLNativeWindowType  NativeWindowType;//注意这两种类型其实是一样的
…
#if defined(_WIN32) || defined(__VC32__) &&!defined(__CYGWIN__) && !defined(__SCITECH_SNAP__)
            /* Win32 和WinCE系统下的定义 */
   …
   typedef  HWND    EGLNativeWindowType;
#elif defined(__WINSCW__) || defined(__SYMBIAN32__)  /* Symbian系统*/
   …
    typedef  void *  EGLNativeWindowType;
#elif defined(__ANDROID__) || defined(ANDROID) /* Android系统 */
   struct ANativeWindow;
   …
   typedef struct ANativeWindow*          EGLNativeWindowType;
   …
#elif defined(__unix__) /* Unix系统*/
   …
   typedef  Window   EGLNativeWindowType;
#else
   #error "Platform notrecognized"
#endif
```

我们以下表来概括在不同的操作系统平台下EGLNativeWindowType所对应的具体数据类型：

表格 11　2 不同平台下的EGLNativeWindowType

操作系统	数据类型
Win32, WinCE	HWND，即句柄
Symbian	Void*

Android	ANativeWindow*
Unix	Window
其它	暂时不支持

由于OpenGL ES并不是只针对某一个操作系统平台设计的，它在很多地方都要考虑兼容性和可移植性，这个EGLNativeWindowType就是其中一个例子。它在 不同的系统中对应的是不一样的数据类型，比如Android中就指的是ANativeWindow指针。

ANativeWindow的定义在Window.h中：

```
/*system/core/include/system/Window.h*/
struct ANativeWindow
{...
    const uint32_t flags; //与Surface或updater有关的属性
    const int   minSwapInterval;//所支持的最小交换间隔时间
    const int   maxSwapInterval;//所支持的最大交换间隔时间
    const float xdpi; //水平方向的密度，以dpi为单位
    const float ydpi;//垂直方向的密度，以dpi为单位
    intptr_t    oem[4];//为OEM定制驱动所保留的空间
    int     (*setSwapInterval)(struct ANativeWindow*window, int interval);
    int     (*dequeueBuffer)(struct ANativeWindow*window, struct ANativeWindowBuffer** buffer);
    int     (*lockBuffer)(struct ANativeWindow*window, struct ANativeWindowBuffer* buffer);
    int    (*queueBuffer)(struct ANativeWindow* window, struct ANativeWindowBuffer*buffer);
    int     (*query)(const struct ANativeWindow*window, int what, int* value);
    int     (*perform)(struct ANativeWindow* window,int operation, ... );
    int     (*cancelBuffer)(struct ANativeWindow*window, struct ANativeWindowBuffer* buffer);
    void* reserved_proc[2];
};
```

我们在下表中详细解释这个类的成员函数。

表格 11　3 ANativeWindow类成员函数解析

Member Function	Description

setSwapInterval	设置交换间隔时间，后面我们会讲解swap的作用
dequeueBuffer	EGL通过这个接口来申请一个buffer。以前面我们所举的例子来说，两个本地窗口所提供的buffer分别来自于帧缓冲区和内存空间。单词“dequeue”的字面意思是“出队列”，这从侧面告诉我们，一个Window所包含的buffer很可能不只一份
lockBuffer	申请到的buffer并没有被锁定，这种情况下是不允许我们去修改其中的内容的。所以我们必须要先调用lockBuffer来获得一个锁
queueBuffer	当EGL对一块buffer渲染完成后，它调用这个接口来unlock和post buffer
query	用于向本地窗口咨询相关信息
perform	用于执行本地窗口支持的各种操作，比如： NATIVE_WINDOW_SET_USAGE NATIVE_WINDOW_SET_CROP NATIVE_WINDOW_SET_BUFFER_COUNT NATIVE_WINDOW_SET_BUFFERS_TRANSFORM NATIVE_WINDOW_SET_BUFFERS_TIMESTAMP 等等
cancelBuffer	这个接口可以用来取消一个已经dequeued的buffer，要特别注意同步的问题

从上面对ANativeWindow的描述可以看出，它更像是一份“协议”，规定了一个本地窗口的形态和功能。这对于支持多种本地窗口的系统是必须的，因为只有这样子我们才能针对某种特定的平台窗口，来填充具体的实现。

这个小节中我们先来看下FramebufferNativeWindow是如何履行这份“协议”的。

FramebufferNativeWindow本身代码并不多，下面分别选取其构造函数及dequeue()两个函数来分析，其它部分的实现都是类似的，大家可以自行阅读。

(1) FramebufferNativeWindow构造函数

基于FramebufferNativeWindow的功能，可以大概推测出它的构造函数里应该至少完成如下的初始化操作：

Ø 加载GRALLOC_HARDWARE_MODULE_ID模块，详细流程我们在Gralloc小节已经解释过了

Ø 分别打开fb和gralloc设备。我们在Gralloc小节也已经分析过了，打开后的设备由全局变量fbDev和grDev管理

Ø 根据设备的属性来给FramebufferNativeWindow赋初值

Ø 根据FramebufferNativeWindow的实现来填充ANativeWindow中的“协议”

Ø 其它一些必要的初始化

下面从源码入手看下每个步骤具体是怎样实现的。

```
/*frameworks/native/libs/ui/FramebufferNativeWindow.cpp*/
FramebufferNativeWindow::FramebufferNativeWindow()
    : BASE(), fbDev(0),grDev(0), mUpdateOnDemand(false)
{
    hw_module_t const* module;
if (hw_get_module(GRALLOC_HARDWARE_MODULE_ID, &module) == 0) {…
  int err;
        int i;
        err = framebuffer_open(module, &fbDev);
        err = gralloc_open(module, &grDev);
        /*上面这部分我们在前几个小节已经分析过了，不清楚的可以回头看下*/
        …
        mNumBuffers = NUM_FRAME_BUFFERS; //buffer个数，目前为2
        mNumFreeBuffers =NUM_FRAME_BUFFERS; //可用的buffer个数，初始时所有buffer可用
        mBufferHead =mNumBuffers-1;
        …
        for (i = 0; i <mNumBuffers; i++) //给每个buffer初始化
        {
            buffers[i] = new NativeBuffer(fbDev->width,fbDev->height, fbDev->format,
GRALLOC_USAGE_HW_FB);
        }// NativeBuffer是什么?

        for (i = 0; i <mNumBuffers; i++) //给每个buffer分配空间
        {
             err =grDev->alloc(grDev, fbDev->width, fbDev->height, fbDev->format,
                    GRALLOC_USAGE_HW_FB, &buffers[i]->handle,&buffers[i]->stride);
                                        …
        }
```

```
            /*为本地窗口赋属性值*/
        const_cast<uint32_t&>(ANativeWindow::flags) = fbDev->flags;
         const_cast<float&>(ANativeWindow::xdpi)= fbDev->xdpi;
        const_cast<float&>(ANativeWindow::ydpi) = fbDev->ydpi;
        const_cast<int&>(ANativeWindow::minSwapInterval) =fbDev->minSwapInterval;
        const_cast<int&>(ANativeWindow::maxSwapInterval) =fbDev->maxSwapInterval;
    } else {
         ALOGE("Couldn'tget gralloc module");
    }
    /*以下履行窗口“协议”*/
   ANativeWindow::setSwapInterval = setSwapInterval;
   ANativeWindow::dequeueBuffer = dequeueBuffer;
    ANativeWindow::lockBuffer= lockBuffer;
    ANativeWindow::queueBuffer= queueBuffer;
    ANativeWindow::query =query;
    ANativeWindow::perform =perform;
}
```

这个函数逻辑上很简单，开头一部分我们已经分析过了，就不再赘述。需要注意的是FramebufferNativeWindow是如何分配buffer的，换句话说，后面的dequeue所获得的缓冲区是从何而来。

成员变量mNumBuffers代表了FramebufferNativeWindow所管理的buffer总 数，NUM_FRAME_BUFFERS当前定义为2。有人可能会觉得奇怪，既然FramebufferNativeWindow对应的是真实的物理屏 幕，那么为什么需要两个buffer呢？

假设我们需要绘制这样一个画面，包括两个三角形和三个圆形，最终结果如下图所示：

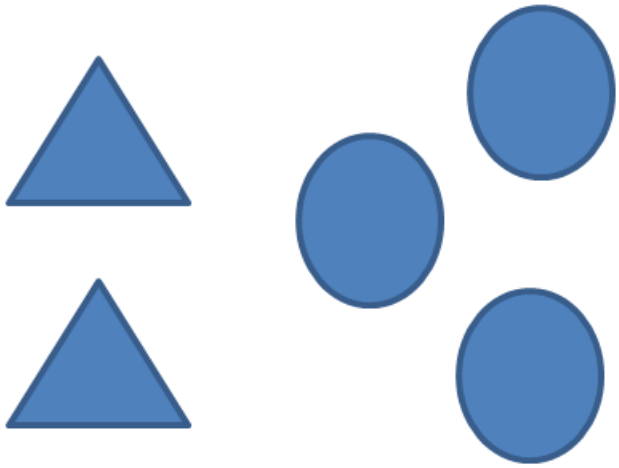

图 11　7 希望在屏幕上显示的完整结果

先来看只有一个buffer的情况，这意味着我们是直接以屏幕为画板来实时做画的——我们画什么，屏幕上就显示什么。以绘制上图中的每一个三角形或圆形都需要0.5秒为例，那么总计耗时应该是0.5*5=2.5秒。换句话说，用户在不同时间点所看到的屏幕是这样子的：

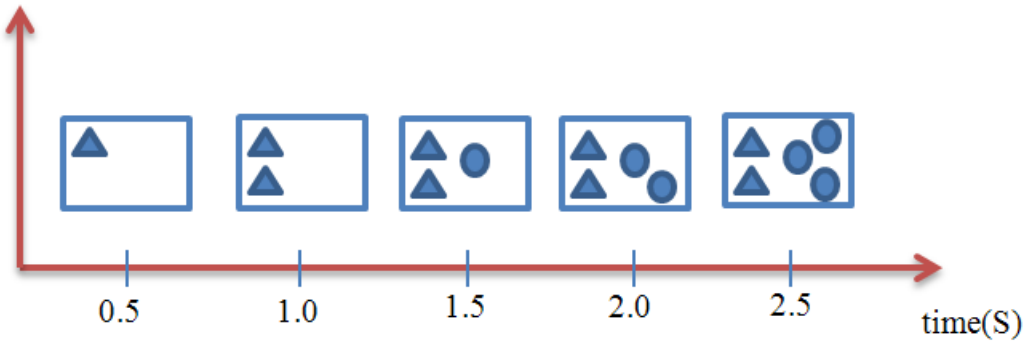

图 11　8 只有一个buffer的情况

对于用户来说，他将看到一个不断刷新的画面。通俗来讲，就是画面很“卡”。对于图像刷新很频繁的情况，比如游戏场景，用户的体验就会更差。那么有什 么解决的办法呢？我们知道，出现这种现象的原因就是程序直接以屏幕为绘图板，把还没有准备就绪的图像直接呈现给了用户。换句话说，如果可以等待整幅图绘制 完成以后再刷新到屏幕上，那么对于用户来说，他在任何时候看到的都是正确而完整的图像，问题也就解决了。下图解释了当采用两个缓冲区时的情况：

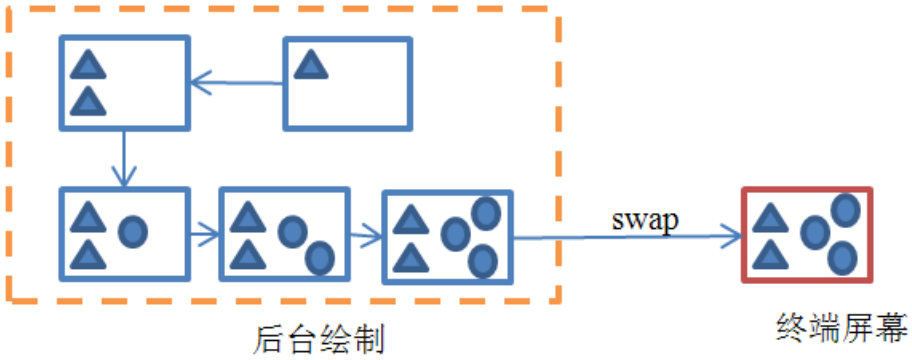

图 11　9 采用两个缓冲区的情况

上图中所述的就是通常所称的“双缓冲”(Double-Buffering)技术。除此以外，其实还有三缓冲(TripleBuffering)、四缓冲(Quad Buffering)等等，我们将它们统称为“多缓冲”(MultipleBuffering)机制。

理解了为什么需要双缓冲以后，我们再回过头来看FramebufferNativeWindow的构造函数。接下来就要解决另一个问题，即两个缓冲 区空间是从哪里分配的？根据前几个小节的知识，应该是要向HAL层的Gralloc申请。两个缓冲区以全局变量 buffers[NUM_FRAME_BUFFERS]来记录，每个数据元素是一个NativeBuffer，这个类定义如下：

```
class NativeBuffer : public ANativeObjectBase< ANativeWindowBuffer,
        NativeBuffer,LightRefBase<NativeBuffer> >
{…
```

所以这个“本地缓冲区”继承了ANativeWindowBuffer的特性，后者的定义在/system/core/include/system/Window.h中：

```
typedef struct ANativeWindowBuffer
{…
    int width; //宽
    int height;//高
    …
    buffer_handle_t handle;/*代表内存块的句柄，比如ashmem机制。
                    可以参考本书的共享内存章节*/
…
} ANativeWindowBuffer_t;
```

第一个for循环里先给各buffer创建相应的实例(new NativeBuffer)，其中的属性值都来源于fbDev，比如宽、高、格式等等。紧随其后的就是调用Gralloc设备的alloc()方法：

```
err = grDev->alloc(grDev, fbDev->width,fbDev->height, fbDev->format,
            GRALLOC_USAGE_HW_FB, &buffers[i]->handle, &buffers[i]->stride);
```

注意第5个参数，它代表所要申请的缓冲区的用途，定义在hardware/libhardware/include/hardware/Gralloc.h中，目前已经支持几十种，比如：

Ø GRALLOC_USAGE_HW_TEXTURE

缓冲区将用于OpenGL ES Texture

Ø GRALLOC_USAGE_HW_RENDER

缓冲区将用于OpenGL ES的渲染

Ø GRALLOC_USAGE_HW_2D

缓冲区会提供给2D 硬件图形设备

Ø GRALLOC_USAGE_HW_COMPOSER

缓冲区用于HWComposer HAL模块

Ø GRALLOC_USAGE_HW_FB

缓冲区用于framebuffer设备

Ø GRALLOC_USAGE_HW_VIDEO_ENCODER

缓冲区用于硬件视频编码器

等等。。。

这里是要用于在终端屏幕上显示的，所以申请的usage类型是GRALLOC_USAGE_HW_FB，对应的Gralloc中的实现是 gralloc_alloc_framebuffer@Gralloc.cpp;假如是其它用途的申请，则对应gralloc_alloc_buffer@Gralloc.cpp。不过，如果底层只允许一个buffer(不支持page-flipping的情 况)，那么gralloc_alloc_framebuffer也同样可能只返回一个ashmem中申请的“内存空间”，真正的“帧缓冲区”要在post 时才会被用到。

另外，当前可用(free)的buffer数量由mNumFreeBuffers管理，这个变量的初始值也是NUM_FRAME_BUFFERS， 即总共有2个可用缓冲区。在程序后续的运行过程中，始终由mBufferHead来指向下一个将被申请的buffer(注意，不是下一个可用 buffer)。也就是说每当用户向FramebufferNativeWindow申请一个buffer时(dequeueBuffer)，这个 mBufferHead就会增加1;一旦它的值超过NUM_FRAME_BUFFERS，则还会变成0，如此就实现了循环管理，后面 dequeueBuffer时我们再详细解释。

一个本地窗口包含了很多属性值，比如各种标志(flags)、横纵坐标的密度值等等。这些数值都可以从fb设备上取到，我们需要将它赋予刚生成的FramebufferNativeWindow实例。

最后，就是履行ANativeWindow的协议了。FramebufferNativeWindow会将其成员函数填充到ANativeWindow中的函数指针中，比如：

ANativeWindow::setSwapInterval = setSwapInterval;

ANativeWindow::dequeueBuffer = dequeueBuffer;

这样子OpenGL ES才能通过一个ANativeWindow来正确地与本地窗口建立连接，下面我们就详细分析下其中的dequeueBuffer。

(2)dequeueBuffer

这个函数很短，只有二十几行，不过是FramebufferNativeWindow中的核心。OpenGL ES就是通过它来分配一个用于渲染的缓冲区的，与之相对应的是queueBuffer。

```
int FramebufferNativeWindow::dequeueBuffer(ANativeWindow* window,ANativeWindowBuffer** buffer)
{
    FramebufferNativeWindow*self = getSelf(window); /*Step1*/
    Mutex::Autolock_l(self->mutex); /*Step2*/
…
/*Step3. 计算mBufferHead */
    int index =self->mBufferHead++;
    if (self->mBufferHead>= self->mNumBuffers)
        self->mBufferHead =0;

    /*Step4. 当前没有可用缓冲区*/
    while (!self->mNumFreeBuffers){
       self->mCondition.wait(self->mutex);
}
/*Step5. 有人释放了缓冲区*/
   self->mNumFreeBuffers--;
   self->mCurrentBufferIndex = index;
    *buffer =self->buffers[index].get();

    return 0;
}
```

Step1@ FramebufferNativeWindow::dequeueBuffer, 这里先将入参中ANativeWindow 类型的变量window强制转化为FramebufferNativeWindow。因为前者是后者的父类，所以这样的转化当然是有效的。不过细心的读者 可能会发现，为什么函数入参中还要特别传入一个ANativeWindow对象的内存地址，直接使用FramebufferNativeWindow的 this指针不行吗？这个问题我还没有确定真正的原因是什么，一个猜测是为了兼容各种平台的需求。大家应该注意到了ANativeWindow是一个 Struct数据类型，在C语言中Struct是没有成员函数的，所以我们通常是用函数指针的形式来模拟一个成员函数，比如这个 dequeueBuffer在ANativeWindow的定义就是一个函数指针。而且我们没有办法确定最终填充到ANativeWindow中函数指针 的实现是否有this指针，所以在参数中带入一个window变量就是必要的了。

Step2@ FramebufferNativeWindow::dequeueBuffer,获得一个Mutex锁。因为接下来的操作涉及到互斥区，自然需要有一个 保护措施。这里采用的是Autolock，意味着dequeueBuffer函数结束后会自动释放Mutex。

Step3@ FramebufferNativeWindow::dequeueBuffer,前面我们介绍过mBufferHead变量，这里来看下对它的实际使 用。首先index得到的是mBufferHead所代表的当前位置，然后mBufferHead增加1。由于我们是循环利用两个缓冲区的，所以如果这个 变量的值超过mNumBuffers，那么就需要把它置0。也就是说在这个场景下mBufferHead的值永远只能是0或者1。

Step4@ FramebufferNativeWindow::dequeueBuffer,mBufferHead并不代表它所指向的缓冲区是可用的。假如当前的 mNumFreeBuffers表明已经没有多余的缓冲区空间，那么我们就需要等待有人释放buffer。这里使用到了Condition这一同步机制， 如果有不清楚的请参考本书进程章节的详细描述。可以肯定的是这里调用了mCondition.wait，那么必然有其它地方要唤醒它——具体的就是在 queueBuffer()中，大家可以自己验证下是否如此。

Step5@ FramebufferNativeWindow::dequeueBuffer,一旦成功获得buffer后，要把可用的buffer计数减1(mNumFreeBuffers--)，因为mBufferHead前面已经自增过了，这里就不用再特别处理。

这样子我们就完成了对Android系统中本地窗口FramebufferNativeWindow的分析，接下来就讲解另一个重要的Native Window。

1.1.1 SurfaceTextureClient

针对应用程序端的本地窗口是SurfaceTextureClient，和FramebufferNativeWindow一样，它必须继承ANativeWindow：

```
class SurfaceTextureClient

    : publicANativeObjectBase<ANativeWindow, SurfaceTextureClient,RefBase>
```

这个本地窗口当然也需要实现ANativeWindow所制定的“协议”，我们的重点是关注它与前面的FramebufferNativeWindow有什么不同。SurfaceTextureClient的构造函数只是简单地调用了init函数，后者则对ANativeWindow::dequeueBuffer等函数指针及内部变量赋了初值。由于整个函数的功能很简单，我们只摘录其中的一部分：

```
/*frameworks/native/libs/gui/SurfaceTextureClient.cpp*/

void SurfaceTextureClient::init() {

    /*给ANativeWindow中的函数指针赋值*/

   ANativeWindow::setSwapInterval  =hook_setSwapInterval;

    ANativeWindow::dequeueBuffer    = hook_dequeueBuffer;

    …

    /*为各内部变量赋值，因为此时用户还没有真正发起申请，所以基本是0*/

    mReqWidth = 0;

    mReqHeight = 0;

    …
```

```
    mDefaultWidth = 0;
    mDefaultHeight = 0;
    mUserWidth = 0;
    mUserHeight = 0;…
}
```

SurfaceTextureClient是面向Android系统中所有UI应用程序的，也就是说它承担着单个应用进程中的UI显示需求。基于这点考虑，可以推测出它的内部实现至少会有以下几点：

Ø 提供给上层(主要是java层)绘制图像的“画板”

前面说过，这个本地窗口分配的内存应该不是来自于帧缓冲区，那么具体是由谁分配的，又是如何管理的呢？

Ø 它与SurfaceFlinger间是如何分工的

显然SurfaceFlinger需要收集系统中所有应用程序绘制的图像数据，然后集中显示到物理屏幕上。在这个过程中，SurfaceTextureClient扮演了什么样的角色呢？

我们先来解释下这个类中的一些重要的成员变量，如下表所示：

表格 11　4 SurfaceTextureClient部分成员变量一览

成员变量	说明
sp<ISurfaceTexture> mSurfaceTexture	这个变量是SurfaceTextureClient的核心，很多“协议”就是通过它实现的，后面会有详细讲解
BufferSlot mSlots[NUM_BUFFER_SLOTS]	从名称上可以看出，这是Client内部用于存储buffer的地方，容量NUM_BUFFER_SLOTS最多达32个。BufferSlot类 内部又由一个GraphicBuffer和一个dirtyRegion组成，当有用户dequeueBuffer时就会分配真正的空间
uint32_t mReqWidth	SurfaceTextureClient中有多组相似的宽高变量，它们之间是有区别的。这里的宽和高是指下一次dequeue时将会申请的尺寸，初始值都是1
uint32_t mReqHeight	
uint32_t mReqFormat	和上面的两变量类似，这是指下次dequeue时将会申请的buffer的像素格式，初始值是PIXEL_FORMAT_RGBA_8888
uint32_t mReqUsage	指下次dequeue时将会指定的usage类型
Rect mCrop	Crop表示“修剪”，这个变量将在下次queue时用于修剪缓冲区，可以调用setCrop来设置具体的值
int mScalingMode	同样，这个变量将用于下次queue时对缓冲区进行scale，可以调

	用setScalingMode来设置具体的值
uint32_t mTransform	用于下次queue时的图形翻转等操作(Transform)
uint32_t mDefaultWidth	默认情况下的缓冲区宽高值
uint32_t mDefaultHeight	
uint32_t mUserWidth	如果不为零的话，就是应用层指定的值，将会覆盖前面的mDefaultWidth/ mDefaultHeight
uint32_t mUserHeight	
sp<GraphicBuffer> mLockedBuffer	这三个值需要锁的保护，接下来还会有分析
sp<GraphicBuffer> mPostedBuffer	
Region mDirtyRegion	

从这些内部变量的描述中，我们可以大概了解到两点：SurfaceTextureClient中将通过mSurfaceTexture来获得buffer，而且这些缓冲区会被记录在mSlots数组中。接下来就来分析其中的实现细节。

前面SurfaceTextureClient构造函数里我们看到ANativeWindow中的函数指针赋予的是各种以hook开头的函数，这些 函数内部又直接调用了SurfaceTextureClient中真正的实现，比如hook_dequeueBuffer对应的是dequeueBuffer。这就好像是“钩子”一样，所以称之为hook。

```
int SurfaceTextureClient::dequeueBuffer(android_native_buffer_t**buffer) {…
    Mutex::Autolocklock(mMutex);
int buf = -1;
/*Step1. 宽高计算*/
    int reqW = mReqWidth ?mReqWidth : mUserWidth;
int reqH = mReqHeight ? mReqHeight :mUserHeight;
/*Step2. dequeueBuffer得到一个缓冲区*/
    status_t result =mSurfaceTexture->dequeueBuffer(&buf, reqW, reqH,mReqFormat, mReqUsage);
    …
   sp<GraphicBuffer>& gbuf(mSlots[buf].buffer);//注意buf只是一个int值，代表的是mSlots数组序号
…
/*Step3. requestBuffer*/
    if ((result &ISurfaceTexture::BUFFER_NEEDS_REALLOCATION) || gbuf == 0) {
```

```
        result =mSurfaceTexture->requestBuffer(buf, &gbuf);
        ...
    }
    *buffer = gbuf.get();
    return OK;
}
```

Step1@ SurfaceTextureClient::dequeueBuffer。用于UI绘制的图形缓冲区一定有宽高属性，具体的值由mReqWidth/mReqHeight或者mUserWidth/mUserHeight决定，其中前者的优先级比后者高

Step2@ SurfaceTextureClient::dequeueBuffer。可以看到，真正执行dequeueBuffer操作的确实是mSurfaceTexture(ISurfaceTexture)。这个变量的赋值有两个来源：作为SurfaceTextureClient的构造函 数参数传入，然后间接调用setISurfaceTexture来设置的;或者SurfaceTextureClient的子类通过直接调用setISurfaceTexture来生成。

在应用进程环境中，属于后面一种情况。

具体流程就是：当Java层的Surface进行init时，实际上执行的函数是 Surface_init@android_view_Surface.cpp。这个JNI函数将进一步调用 SurfaceComposerClient::createSurface生成一个SurfaceControl，后者是用于管理Surface的类。 它将在SurfaceControl::getSurface时生成一个Surface实例，在构造时通过SurfaceControl::getSurfaceTexture来获得一个ISurfaceTexture。而Surface类实际上又继承自 SurfaceTextureClient，所以它可以调用setISurfaceTexture。

由于Android源码有多处称为Surface的地方，取名极其混乱，我们下面通过一张完整的流程图来帮助大家把这些关系理顺：

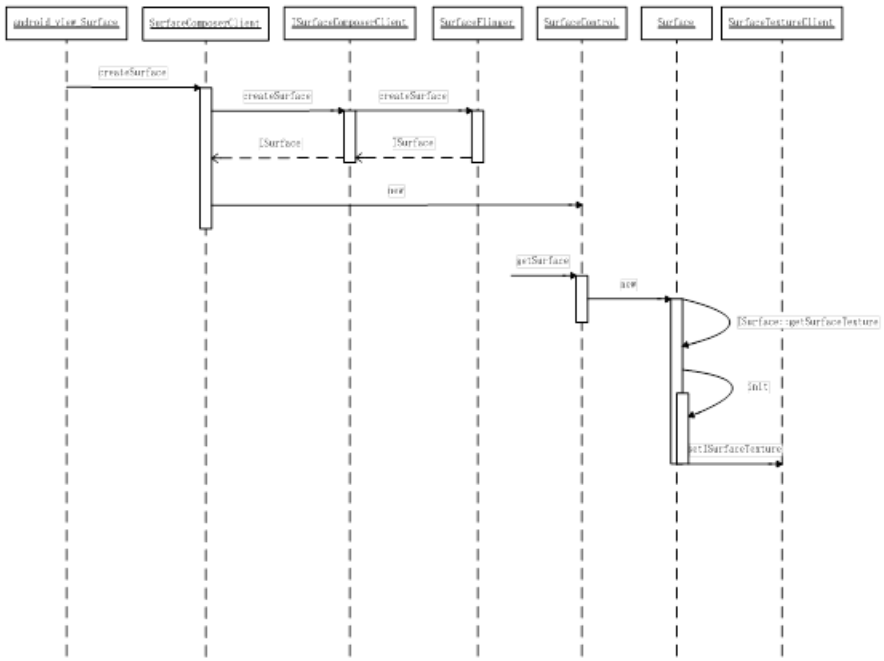

图 11　10 ISurfaceTexture创建流程

从这个图中可以看到，ISurfaceTexture是由ISurface::getSurfaceTexture生成的，而ISurface则是由SurfaceFlinger生成的。在这一过程中，总共使用到了三个匿名binderserver，它们所提供的接口整理如下表：

表格 11　5 与Surface相关的三个匿名binder

匿名**Binder**	提供的接口
ISurfaceComposerClient	sp<ISurface> createSurface(…); status_t destroySurface(SurfaceID sid);
ISurface	sp<ISurfaceTexture> getSurfaceTexture(…);

ISurfaceTexture	status_t requestBuffer(int slot, sp<GraphicBuffer>* buf);
	status_t setBufferCount(int bufferCount);
	status_t dequeueBuffer(…);
	status_t queueBuffer(…);
	void cancelBuffer(int slot);
	int query(int what, int* value);
	status_t setSynchronousMode(bool enabled);
	status_t connect(int api, QueueBufferOutput* output);
	status_t disconnect(int api);

由此可见，这三个匿名binder是一环扣一环的，也就是说我们访问的顺序只能是ISurfaceComposerClientàISurfaceàISurfaceTexture。当然，第一个匿名binder就一定是需要由一个实名binder来提供，它就是SurfaceFlinger，而SurfaceFlinger则是在ServiceManager中“注册在案”的。具体 是在SurfaceComposerClient::onFirstRef()这个函数中，通过向ServiceManager查询名称为“SurfaceFlinger”的binder server来获得的。不过和其它常见binder server不同的是，SurfaceFlinger虽然在ServiceManager中注册的名称为“SurfaceFlinger”，但它在 server端实现的binder接口却是ISurfaceComposer，因而SurfaceComposerClient得到的其实是 ISurfaceComposer，这点大家要特别注意，否则可能会搞乱。

```
//我们可以从SurfaceFlinger的继承关系中看出这一区别，如下代码片断
class SurfaceFlinger :
    publicBinderService<SurfaceFlinger>, //在ServiceManager中注册为“SurfaceFlinger”
    public BnSurfaceComposer, //实现的接口却叫ISurfaceComposer，不知道为什么要这么设计。。。
```

绕了一大圈后，我们接着分析前面dequeueBuffer函数的实现。很显然SurfaceTextureClient只是一个中介，它间接调用 mSurfaceTexture也就是ISurfaceTexture的服务。那么ISurfaceTexture在Server端又是由谁来完成的呢？

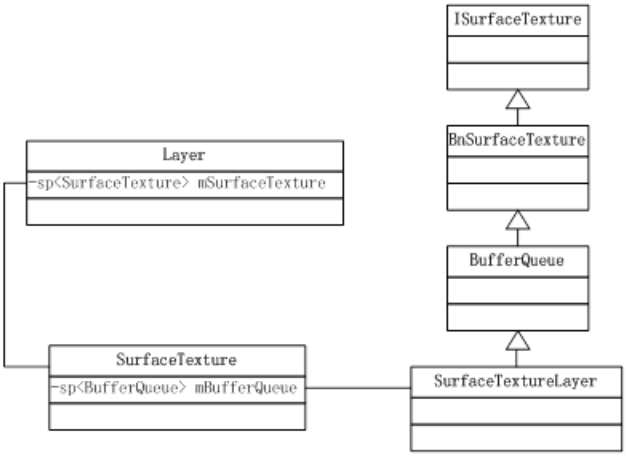

图 11　11 ISurfaceTexture的本地端实现

因为这里面牵扯到很多新的类，我们先不做过多解释，到后面BufferQueue小节再详细分析其中的依赖关系。

当mSurfaceTexture->dequeueBuffer返回后，buf变量就是mSlots[]数组中可用的成员序号。接下来就要通过这个序号来获取真正的buffer地址，即mSlots[buf].buffer。

Step3@ SurfaceTextureClient::dequeueBuffer。假如返回值result中的标志包含了BUFFER_NEEDS_REALLOCATION，说明BufferQueue为这个Slot重新分配了空间，具体细节请参见下一个小节。此时我们还 需要另外调用requestBuffer来确定gbuf的值，这其中又牵涉到很多东西，我们放在下面小节统一解释原因。

通过这两个小节，我们学习了显示系统中两个重要的本地窗口，即FramebufferNativewindow和SurfaceTextureClient。第一个窗口是专门为SurfaceFlinger服务的，它由Gralloc提供支持，相对逻辑上很好理解。 而SurfaceTextureClient则是为应用程序服务的，同时它从本质上还是由SurfaceFlinger服务统一管理的，因而涉及到很多跨 进程的通信细节。这个小节我们只是简单地勾勒出其中的框架，接下去就要分几个方面来做完整的分析了。

Ø BufferQueue

为应用程序服务的本地窗口SurfaceTextureClient在server端的实现是BufferQueue。我们将详细解析BufferQueue的内部实现，并结合应用程序端的使用流程来理解清楚它们之间的关系。

Ø Buffer、Consumer、Producer是“生产者-消费者”模型中的三个参与对象，如何协调好它们的工作是应用程序能否正常显示UI的关键。 在接下来内容的安排上，我们先讲解Buffer(BufferQueue)与Producer(应用程序)间的交互，然后再专门切入 Consumer(SurfaceFlinger)做详细分析

1.1 BufferQueue详解

上一小节我们已经看到了BufferQueue，它是SurfaceTextureClient实现本地窗口的关键。从逻辑上来推断，BufferQueue应该是驻留在SurfaceFlinger这边的进程中。我们需要进一步解决的疑惑是：

Ø 每个应用程序可以对应几个BufferQueue，它们是一对一、多对一或者是一对多？

Ø 应用程序所需要的绘图空间是由谁分配的？

在音频系统的学习中，我们知道AudioTrack和AudioFlinger是通过共享内存的形式来进行数据传递的，那么显示系统中是否也是类似情况？

Ø 应用程序与SurfaceFlinger如何互斥共享数据区

和在Audio系统中遇到的问题一样，我们面临的是经典的“生产者-消费者”模型。显示系统又是如何协调好这两者间的互斥访问的呢？

1.1.1 BufferQueue的内部原理

先来解析下BufferQueue的内部构造，如下图所示：

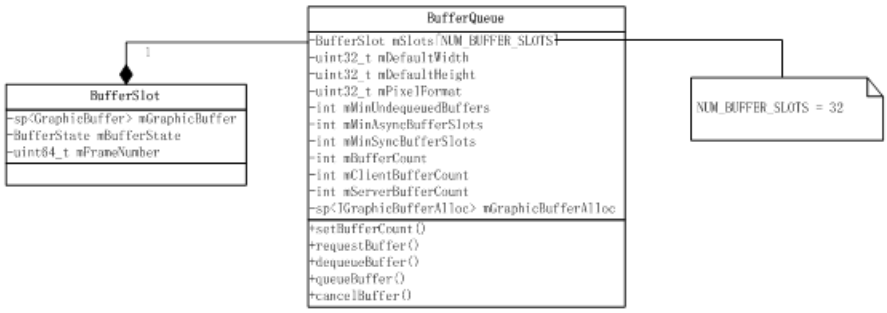

图 11　12 BufferQueue内部变量

因为BufferQueue是ISurfaceTexture的本地实现，所以它必须重载接口中的各虚函数，比如queueBuffer、requestBuffer、dequeueBuffer等等。另外，这个类的内部有一个非常重要的成员数组，即mSlots[NUM_BUFFER_SLOTS]，大家是否还记得前面SurfaceTextureClient类中也有一个一模一样的数组：

```
class SurfaceTextureClient…{

        BufferSlot  mSlots[NUM_BUFFER_SLOTS];
```

数组的成员是BufferSlot，其中包含的GraphicBuffer变量(mGraphicBuffer)用于记录这个Slot所涉及的缓冲区，另外还有一个BufferState变量mBufferState用于跟踪每个缓冲区的状态，比如：

```
    enum BufferState {
        FREE = 0, /*Buffer当前可用，也就是说可以被dequeued。此时Buffer的owner
                可认为是BufferQueue*/
        DEQUEUED = 1, /*Buffer已经被dequeued，还未被queued或canceld。此时
                Buffer的owner可认为是producer(应用程序)，这意味着server
                端(BufferQueue)不可以对这块缓冲区进行操作*/
        QUEUED = 2, /*Buffer已经被客户端queued，除特别情况外此时还不能对它进
                行dequeue，而可以acquired。此时的owner是BufferQueue*/
```

ACQUIRED = 3/*Buffer的owner改为consumer，可以released，

然后状态又返回FREE*/

};

从上面的状态可以看出，一块Buffer大致经历的过程就是FREE->DEQUEUED->QUEUED->ACQUIRED->FREE。从owner的角度来讲，有点类似于下图的描述：

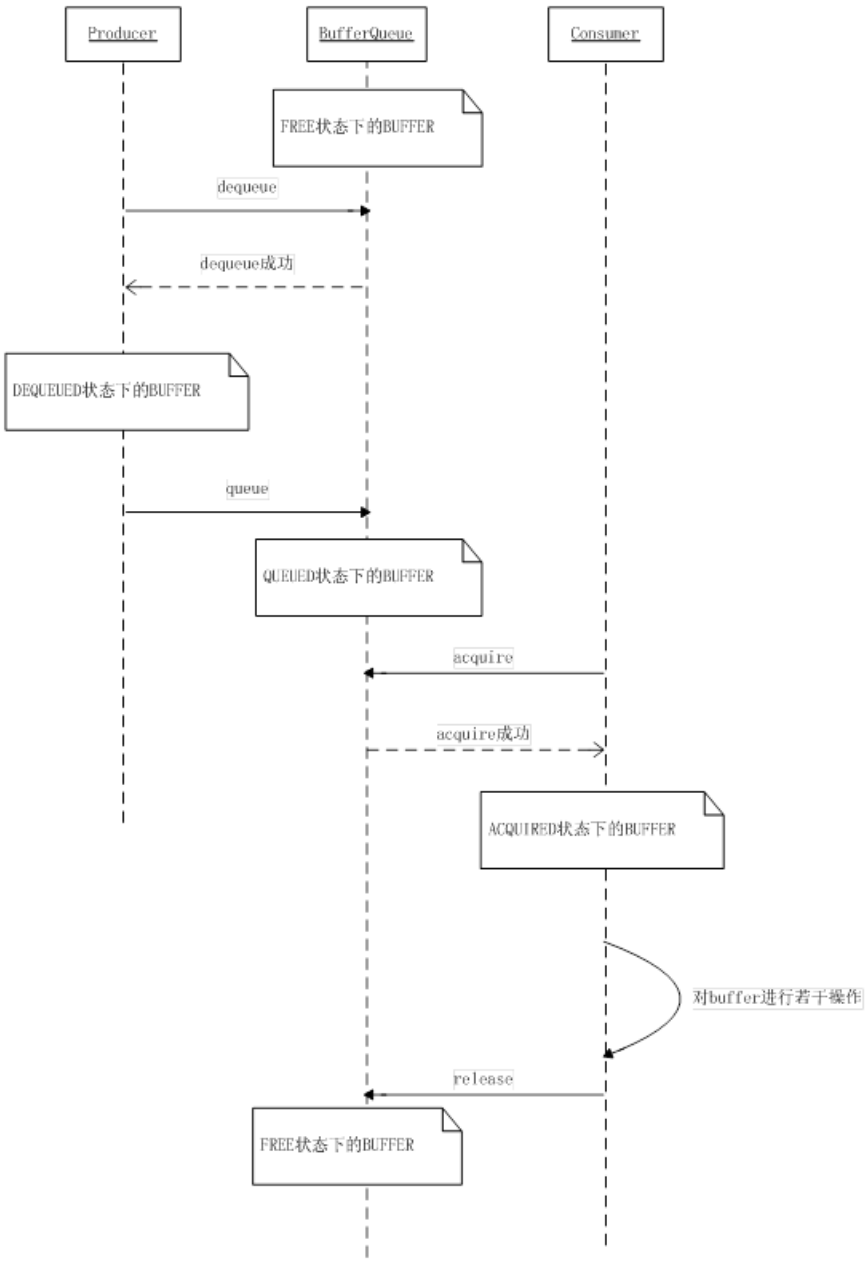

图 11　13 Buffer的状态迁移图

我们来分析下上面的buffer状态迁移简图。

从图中可以大致看出一个buffer的各个状态、引起状态迁移的条件以及各状态下的owner。参与对buffer进行管理

的对象有三个：

Ø BufferQueue

可以认为BufferQueue是一个服务中心，其它两个owner必须要通过它来管理buffer。比如说当producer想要获取一个 buffer时，它不能越过BufferQueue直接与consumer进行联系，反之亦然。这有点像房产中介一样，房主与买方的任何交易都需要经过中 介的同意，私自达成的协议都是违反规定的

Ø Producer

生产者就是“填充”buffer空间的人，通常情况下当然就是应用程序。因为应用程序不断地刷新UI，从而将产生的显示数据源源不断地写到 buffer中。当Producer需要使用一块buffer时，它首先会向中介BufferQueue发起dequeue申请，然后才能对指定的缓冲区 进行操作。这种情况下buffer就属于producer一个人的了，它可以对buffer进行任何必要的操作，而其它owner此刻绝不能擅自插手。

当生产者认为一块buffer已经写入完成后，它进一步调用BufferQueue的queue。从字面上看这个函数是“入列”的意思，形象地表达 了buffer此时的操作——把buffer归还到BufferQueue的队列中。一旦queue成功后，owner也就随之改变为 BufferQueue了

Ø Consumer

消费者是与生产者相对应的，它的操作同样受到BufferQueue的管控。当一块buffer已经就绪后，Consumer就可以开始工作了，具 体的细节我们会在SurfaceFlinger中描述。这里需要特别留意的是，从各个对象所扮演的角色来看，BufferQueue是中介机构，属于服务 提供方;Producer属于buffer内容的产出方，它对缓冲区的操作是一个“主动”的过程;反之，Consumer对buffer的处理则是“被 动”的、“等待式”的——它必须要等到一块buffer填充完成后才能做工作。在这样的模型下，我们怎么保证Consumer可以及时的处理buffer 呢？换句话说，当一块buffer数据ready后，应该怎么告知Consumer来操作呢？

仔细观察的话，可以看到BufferQueue里还同时提供了一个特别的类，名称为ConsumerListener，其中的函数接口包括：

```
struct ConsumerListener :public virtual RefBase {

    virtual voidonFrameAvailable() = 0;/*当一块buffer可以被消费时，这个函数会被调用，特别注意此

                                时没有共享锁的保护*/

    virtual voidonBuffersReleased() = 0;/*BufferQueue通知consumer它已经释放其slot中的一个或多个

                                GraphicBuffer 引用*/

};
```

这样子就很清楚了，当有一帧数据准备就绪后，BufferQueue就会调用onFrameAvailable()来通知Consumer进行消费。

1.1.1 BufferQueue中的缓冲区分配

我们知道，BufferQueue中有一个mSlots数组用于管理其内的各缓冲区，最大容量为32。从它的声明方式来看，这个mSlots在程序 一开始就静态分配了32个BufferSlot大小的空间。不过这并不代表缓冲区也是一次性静态分配的，恰恰相反，从BufferSlot的内部变量指针 mGraphicBuffer可以看出，缓冲区的空间分配应当是动态的(从下面的注释也能看出一些端倪)：

```
// mGraphicBuffer points to the buffer allocated for this slot or isNULL if no buffer has been allocated.
```

```
    sp<GraphicBuffer> mGraphicBuffer;
```

现在的问题就转移为，在什么情况下会给一个Slot实际的分配空间呢？

首先能想到的就是dequeueBuffer。理由如下：

Ø 缓冲区的空间分配应该既要满足使用者的需求，又要防止浪费。后面这一点mSlots已经满足了，因为它并没有采取一开始就静态预分配的方式

Ø 既然Producer对buffer的操作是“主动”的，那么就意味着它是整个需求的发起者。换句话说，只要它没有dequeueBuffer，或者dequeueBuffer时能获取到可用的缓冲区，那当然就没有必要再重新分配空间了

来详细分析下这个函数：

```
/*frameworks/native/libs/gui/BufferQueue.cpp*/
status_t BufferQueue::dequeueBuffer(int *outBuf, uint32_t w,uint32_t h, uint32_t format, uint32_t usage) {…
status_t  returnFlags(OK);
…
   { // Scope for the lock
      Mutex::Autolocklock(mMutex); /*这里采用了自动锁，所以上面需要加个“{”，这样当lock变量生
                                命期结束后锁也就自动释放了。这种写法非常常见*/
                …
      int found = -1;
      int foundSync = -1;
      int dequeuedCount = 0;
      bool tryAgain = true;
      while (tryAgain) {/*Step1. 循环查找符合要求的slot*/
         …
         found = INVALID_BUFFER_SLOT;//初始值
         foundSync =INVALID_BUFFER_SLOT;
         dequeuedCount = 0;
         for (int i = 0; i< mBufferCount; i++) {
            const intstate = mSlots[i].mBufferState;
            /*Step2.统计dequeued buffer数量，后面会用到*/
            if (state ==BufferSlot::DEQUEUED) {
               dequeuedCount++;
```

```
        }


        if (false) {
          //肯定不会走这边
        } else {
          if (state== BufferSlot::FREE) { /*Step3.寻找符合要求的Slot* /
            boolisOlder = mSlots[i].mFrameNumber <mSlots[found].mFrameNumber;
            if(found < 0 || isOlder) {
              foundSync = i;
              found = i; //找到符合要求的Slot
            }
          }// if (state ==BufferSlot::FREE)结束
        }//if(false)else结束
      }// for循环结束

     /*Step4.如果Client没有设置buffer count的话，就不允许dequeue一个以上的buffer*/
      if(!mClientBufferCount && dequeuedCount) {
        ST_LOGE("dequeueBuffer:can't dequeue multiple buffers without setting the buffer count");
        return-EINVAL;
      }
      …
      /*Step5. 判断是否要重试*/
      tryAgain = found== INVALID_BUFFER_SLOT;
      if (tryAgain) {
        mDequeueCondition.wait(mMutex);
      }
    }//while循环结束


    if (found ==INVALID_BUFFER_SLOT) {
      /*因为前面while循环如果没找到的话是不会返回的，所以理论上不会出现这种情况*/
      ST_LOGE("dequeueBuffer: no available buffer slots");
```

```
        return -EBUSY;
    }

    const int buf = found;
    *outBuf = found; //返回值
   /*成功找到可用的Slot序号，接下来就开始对这个指定的Slot进行初始操作，及状态变迁等*/
            …
   mSlots[buf].mBufferState = BufferSlot::DEQUEUED;/*Step6.Buffer状态改变*/
    constsp<GraphicBuffer>& buffer(mSlots[buf].mGraphicBuffer);
    if ((buffer == NULL)|| (uint32_t(buffer->width)  != w) ||(uint32_t(buffer->height) != h) ||
       (uint32_t(buffer->format) != format) || ((uint32_t(buffer->usage)& usage) != usage))
    {
        status_t error;
         /*Step7. 分配GraphicBuffer空间*/
       sp<GraphicBuffer> graphicBuffer(mGraphicBufferAlloc->createGraphicBuffer(
                    w,h, format, usage, &error));
        …
        mSlots[buf].mGraphicBuffer = graphicBuffer; //这个Slot终于分配到空间了
        …
        returnFlags |=ISurfaceTexture::BUFFER_NEEDS_REALLOCATION;
    }
            …
  } // 自动锁lock结束的地方
  …
  return returnFlags;
}
```

因为这个函数很长，我们尽量只留下最核心的部分。从大的方向上看，Step1-Step5是在查找一个可用的Slot序号。从Step6开始，就针对这一指定的Slot进行操作了。下面我们分步来解释。

Step1@BufferQueue::dequeueBuffer。进入while循环，退出的条件是tryAgain为false。这个变量默 认值是true，如果一轮循环结束后found的值不再是INVALID_BUFFER_SLOT，就会变成false，从而结束整个while循环。

循环的主要功能就是查找符合要求的Slot，其中found变量是一个int值，指的是这个BufferSlot在mSlots数组中的序号。

Step2@BufferQueue::dequeueBuffer。统计当前已经被dequeued的buffer数量，这将用于后面的判断，即假如Client没有设置buffer count的话，那么它会被禁止dequeue一个以上的buffer。

Step3@BufferQueue::dequeueBuffer。假如当前的buffer状态是FREE，那么这个Slot就可以进入备选了。 为什么只是备选而不是直接返回这一结果呢？因为当前的mSlots中很可能有多个符合条件的Slot，当然需要挑选其中最匹配的。判断的依据是当前符合要 求的Slot的mFrameNumber是否比上一次选中的最优Slot的mFrameNumber小。代码如下：

bool isOlder =mSlots[i].mFrameNumber <mSlots[found].mFrameNumber;

Step4@BufferQueue::dequeueBuffer。这里的判断来源于前面第二步的计算结果，一旦发现dequeue的数量“超标”，就直接出错返回

Step5@BufferQueue::dequeueBuffer。经过上述几个步骤，我们已经扫描了一遍mSlots中的所有成员了，这时就要 决定是否可以退出循环。前面已经说过，如果成功找到有效的Slot就可以不用再循环查找了，否则tryAgain仍然是true。假如是后一种情况，证明 当前已经没有FREE的Slot了，这时如果直接进入下一次循环，结果通常也是一样的，反而浪费了CPU资源。所以就需要使用条件锁，代码如下：

mDequeueCondition.wait(mMutex);

当有Buffer被释放时，这个锁的条件就会满足。

Step6@BufferQueue::dequeueBuffer。根据Buffer的状态迁移图，当处于FREE状态的Buffer被dequeue成功后，它将进入DEQUEUED,所以这里我们需要改变mBufferState。

Step7@BufferQueue::dequeueBuffer。通过上述几个步骤的努力，现在我们已经成功地寻找到可用的Slot序号了，但是这并不代表这个Slot可以直接使用，为什么？最明显的一个原因就是这个Slot可能还没有分配空间。

因为BufferSlot:: mGraphicBuffer初始值是NULL，假如我们是第一次使用它，必然是需要为它分配空间的。另外，即便mGraphicBuffer不为空，但 如果用户所需要的Buffer属性(比如width、height、format等等)和当前这个不符合，那么也还是要重新分配。

分配空间使用的是mGraphicBufferAlloc这个Allocator，这里暂不深究其中的实现了。

如果重新分配了空间，那么最后的返回值中需要加上BUFFER_NEEDS_REALLOCATION标志。客户端在发现这个标志后，它还应调用requestBuffer()来取得最新的buffer地址。

我们前一小节SurfaceTextureClient::dequeueBuffer()的Step3就是其中一个例子，这里统一解释一下为什么。为了方便阅读，再把这部分代码简单地列出来：

```
int SurfaceTextureClient::dequeueBuffer(android_native_buffer_t**buffer) {…
```

```
/*Step2. dequeueBuffer得到一个缓冲区*/
   status_t result =mSurfaceTexture->dequeueBuffer(&buf, reqW, reqH,mReqFormat, mReqUsage);
   …
  sp<GraphicBuffer>& gbuf(mSlots[buf].buffer);//注意buf只是一个int值，代表的是mSlots数组序号
…
/*Step3. requestBuffer*/
   if ((result &ISurfaceTexture::BUFFER_NEEDS_REALLOCATION) || gbuf == 0) {
       result =mSurfaceTexture->requestBuffer(buf, &gbuf);
       …
   }
   *buffer = gbuf.get();
   return OK;
}
```

当mSurfaceTexture->dequeueBuffer成功返回后，buf得到了mSlots中可用数组成员的序号(对应这一小节的found变量)。但一个很显然的问题是，既然客户端和BufferQueue运行于两个不同的进程中，那么它们两者中的mSlots[buf]会指向 同一块物理内存吗？

先来看下BpSurfaceTexture中是如何发起binder申请的：

```
/*frameworks/native/libs/gui/ISurfaceTexture.cpp*/
class BpSurfaceTexture : public BpInterface<ISurfaceTexture>
{ …
            virtualstatus_t requestBuffer(int bufferIdx, sp<GraphicBuffer>* buf) {//函数入参有两个
       Parcel data, reply;
      data.writeInterfaceToken(ISurfaceTexture::getInterfaceDescriptor());
      data.writeInt32(bufferIdx);//只写入了bufferIdx，也就是BnSurfaceTexture实际上是看不到buf的
       status_t result=remote()->transact(REQUEST_BUFFER, data, &reply);
       …
       bool nonNull =reply.readInt32();//读取的是什么？我们可以去BnSurfaceTexture中去确认下
       if (nonNull) {
          *buf = newGraphicBuffer(); //生成一个GraphicBuffer，看到没，这是一个本地实例
          reply.read(**buf);/*buf是一个sp指针,那么**sp实际上得到的就是这个智能指针所指向的
                        对象。在这个例子中指的是mSlots[buf].buffer*/
```

```
    }
    result =reply.readInt32();/*读取结果*/
    return result;
  }
```

Native层的BpXXX/BnXXX与Java层的不同之处在于，后者通常都是依赖于aidl来自动生成这两个类，而前者则是手工完成的。也正 因为是手工，使用起来才也更灵活。比如在ISurfaceTexture这个例子中，开发者就耍了点技巧——SurfaceTextureClient中 调用了ISurfaceTexture::requestBuffer(intslot, sp<GraphicBuffer>* buf)，这个函数虽然形式上有两个参数，但只有第一个是入参，后一个则是出参。在实际的binder通信中，只有slot这个int值传递给了对方进 程，而buf则自始至终都是SurfaceTextureClient进程在处理，只不过从调用者的角度来讲，好像是由ISurfaceTexture的 Server端完成了对buf的赋值。

从BpSurfaceTexture::requestBuffer这个函数实现中可以看到，Client端进程向server端请求了一个REQUEST_BUFFER服务，然后通过读取返回值来获得缓冲区信息。为了让大家能看清楚这其中的细节，我们有必要先分析下 BnSurfaceTexture这边具体是如何响应这个服务请求的，如下所示：

```
/*frameworks/native/libs/gui/ISurfaceTexture.cpp*/
status_t BnSurfaceTexture::onTransact(uint32_t code, constParcel& data, Parcel* reply, uint32_t flags)
{
   switch(code) {
      case REQUEST_BUFFER: {
         CHECK_INTERFACE(ISurfaceTexture, data, reply);
          int bufferIdx   =data.readInt32();//首先读取要处理的Slot序号
          sp<GraphicBuffer> buffer; //生成一个GraphicBuffer智能指针
          int result =requestBuffer(bufferIdx, &buffer);//调用本地端的实现
         reply->writeInt32(buffer != 0);//注意，第一个写入的值是判断buffer不为空，
                                  // 也就是一个bool值
          if (buffer != 0) {
             reply->write(*buffer); //好，真正的内容在这里，后面我们详细解释
          }
         reply->writeInt32(result);//写入结果值
          return NO_ERROR;
      } break;
```

针对BnSurfaceTexture的写入顺序，显然BpSurfaceTexture必须要按照同样的顺序来读取。

(1)因而它首先获取一个int32值，赋予nonNull变量，这个值对应的是buffer != 0逻辑判断。假如确实不为空的话，那说明我们可以接着读取GraphicBuffer了

(2)两边对GraphicBuffer变量的写和读分别是:

reply->write(*buffer);//写入

reply.read(**buf);//读取

(3)读取result结果值

在第二步中，Server端写入的GraphicBuffer对象需要在Client中完整地复现出来。根据我们在binder章节的学习，具备这种能力的binder对象应该是继承了Flattenable。实际上呢？

class GraphicBuffer: publicANativeObjectBase<ANativeWindowBuffer, GraphicBuffer,

LightRefBase<GraphicBuffer> >,public Flattenable

从GraphicBuffer声明来看，确实是证明了我们的猜测。

接下来我们只需要看下它是如何实现flatten和unflatten的，相信谜底就能揭晓了。

```
/*frameworks/native/libs/ui/GraphicBuffer.cpp*/

status_t GraphicBuffer::flatten(void* buffer, size_t size, intfds[], size_t count) const

{ …

   int* buf =static_cast<int*>(buffer);

   …

   if (handle) {

      buf[6] = handle->numFds;

      buf[7] = handle->numInts;

      native_handle_t const*const h = handle;

      memcpy(fds,     h->data,          h->numFds*sizeof(int));

      memcpy(&buf[8], h->data + h->numFds, h->numInts*sizeof(int));

   }

   return NO_ERROR;

}
```

这个函数中，我们最关心的是handle这个变量的flatten，它实际上是GraphicBuffer中打开的一个ashmem句柄，因而也就 是两边进程共享缓冲区的关键。与handle相关的分别是buf[6]-buf[8]以及fds，再来看下Client端是如何还原出一个 GraphicBuffer的。

```
status_t GraphicBuffer::unflatten(void const* buffer, size_t size,int fds[], size_t count)

{…

   int const* buf =static_cast<int const*>(buffer);

   …
```

```
    const size_t numFds  = buf[6];
    const size_t numInts =buf[7];
    …
    if (numFds || numInts) {…
        native_handle* h = native_handle_create(numFds, numInts);
       memcpy(h->data,          fds,    numFds*sizeof(int));
        memcpy(h->data +numFds, &buf[8], numInts*sizeof(int));
        handle = h;
    } else { …
    } …
    if (handle != 0) {
        mBufferMapper.registerBuffer(handle);
    }
    return NO_ERROR;
}
```

同样，unflatten中的操作依据的也是flatten时写入的格式。其中最重要的两个函数是native_handle_create()和registerBuffer()。前一个函数生成native_handle实例，并将相关数据拷贝到其内部。另一个registerBuffer则属于GraphicBufferMapper类中的实现，成员变量mBufferMapper是在GraphicBuffer在构造函数中生成的，它所承担 的任务是和Gralloc打交道，代码如下：

```
GraphicBufferMapper::GraphicBufferMapper()
{
    hw_module_t const* module;
    int err = hw_get_module(GRALLOC_HARDWARE_MODULE_ID, &module);
```

这里出现了我们熟悉的Gralloc的moduleid，不清楚的可以回头看下Gralloc这小节的介绍。GraphicBufferMapper::registerBuffer()只是一个中介作用，它会直接调用 gralloc_module_t::registerBuffer()，那么后者究竟完成了什么功能？因为这个函数的实现与具体平台有关，我们以 msm7k为例大概看下：

```
/*hardware/msm7k/libgralloc/Mapper.cpp*/
int gralloc_register_buffer(gralloc_module_t const* module,buffer_handle_t handle)
{…
    private_handle_t* hnd =(private_handle_t*)handle;
    …
    err =gralloc_map(module, handle, &vaddr);
```

```
    return err;
}
```

可以看到，通过handle句柄，Client端可以将指定的内存区域映射到自己的进程空间中，而这块区域与BufferQueue中所指向的物理空间是一致的，从而成功地实现了缓冲区的共享。

这样子在SurfaceTextureClient::dequeueBuffer()中，当遇到结果中包含有BUFFER_NEEDS_REALLOCATION的情况时，我们再通过requestBuffer()得到的结果来“刷新”Client这端mSlots[]所管辖的缓冲区信息，以保证SurfaceTextureClient与BufferQueue能在任何情况下都对32个BufferSlot保持数据上的高度一致。这也是后面它们能正确实施“生产者-消费者”模型的基础。

1.1.1 应用程序的典型绘图流程

我们知道，BufferQueue有最多达32个BufferSlot，这样设计的目的是什么？一个可能的原因就是提高图形渲染速度。因为假如只有 两个buffer，可以想象一下，当应用程序这个生产者的产出效率大于消费者的处理速度时，很快它就会dequeue完所有缓冲区而处于等待状态，从而导 致不必要的麻烦。当然，实际上32只是最大的容量，具体值是可以设置的，大家可以结合后面的ProjectButter小节来理解一下。

前面小节我们已经学习了BufferQueue的内部原理，那么应用程序又是如何与之配合的呢？

解决这个疑惑的关键就是了解应用程序是如何执行绘图流程的，这也是本节我们叙述的重点。不过大家应该有个心里准备，应用程序并不会直接使用 BufferQueue。和Android系统中很多其它地方一样，“层层包裹”在这里同样是存在的，因而我们要尽量抓住其中的重点，并辅以一定的手段， 才能更好更快地从诸多错综复杂的类关系中找出问题的答案。

出于以上原因的考虑，我们选取系统的开机动画这一应用程序，来分析整个图形绘制的流程。值得一提的是，这个开机动画的实现符合前面提到的两个改进的 图形系统中的第一个，即应用程序与SurfaceFlinger都是使用OpenGL ES来完成UI显示，不过因为它是一个C++程序，所以不需要上层GLSurfaceView的支持。

当一个Android设备上电后，正常情况下它会先后显示最多4个不一样的开机画面，分别是：

l boot-loader

这显然是第一个出现的画面。因为boot-loader只是负责系统后续模块的加载与启动，而且要求文件体积很小，所以一般我们只让它显示一张静态的图片

l kernel

在进入内核后，同样会在物理屏幕上有所显示。和boot-loader一样，默认情况下它也只是一张静态图片

l android(2个)

Android是系统启动的最后一个阶段，也是最耗时间的一个。它的开机画面既可以是静态文字描述、静态图片，也可以是动态画面。通常第一个是文字 或者静态图片(假如指定路径下的图片不存在的话，就显示文字。关于这方面的资料很多，大家可以自行查阅，我们这里不作过多叙述)，另外一个则是动画，如下 图所示：

图 11　14 原生态Android系统中的开机动画

这个开机动画的实现类是BootAnimation，它的内部就是借助SurfaceFlinger来完成的。另外，因为它并不是传统意义上的 Java层应用程序，这使得我们可以抛离很多上层的牵绊，以最直观的方式来审视BufferQueue的使用，是分析本节问题的最佳选择。

BootAnimation是一个C++程序，其工程源码路径是/frameworks/base/cmds/bootanimation。和很多native应用一样，它也是在init脚本中被启动的，大概来看下这一过程。

```
service bootanim /system/bin/bootanimation
    class main
    user graphics
    group graphics
    disabled
    oneshot
```

以上内容是从init.rc脚本中摘录出来的，完整地描述了bootanimation这个程序的启动属性。如果大家对其中的内容不清楚的话，可以参见本书的系统启动章节。

当bootanimation被启动后，它首先会进入main函数，即main@Bootanimation_main.cpp，生成一个BootAnimation对象，并开启线程池(因为它需要与SurfaceFlinger等系统服务进行跨进程的通信)。
在BootAnimation 的构造函数中，同时生成一个SurfaceComposerClient：

```
BootAnimation::BootAnimation() : Thread(false)
{
    mSession = newSurfaceComposerClient();
}
```

SurfaceComposerClient是每个UI应用程序与SurfaceFlinger间的独立纽带，后续很多操作都是通过它来完成的。不 过这个类只是一个封装，真正起作用的还是其内部的ISurfaceComposerClient。更多的分析我们将放在后续小节中，这里只要先知道它的功 能就可以了。值得一提的是，前面小节中我们讲到了ISurfaceTexture，这里又有一个ISurfaceComposerClient，两者有什 么区别呢？

简单来说，ISurfaceTexture是应用程序与BufferQueue的传输通道，而ISurfaceComposerClient则是它与SurfaceFlinger间的桥梁。这样子的设计是合理的，体现了模块化的思想——SurfaceFlinger的职责是“Flinger”，即把 系统中所有应用程序的最终的“绘图结果”进行“混合”，然后统一显示到物理屏幕上。它不应该，也没有办法分出太多的精力去一一关注各个应用程序的“绘画过 程”。这个光荣的任务自然而然地落在了BufferQueue的肩膀上，它是每个应用程序“一对一”的辅导老师，指导着UI程序的“画板申请”、“作画流 程”等一

系列细节。下面的图描述了这三者的关系：

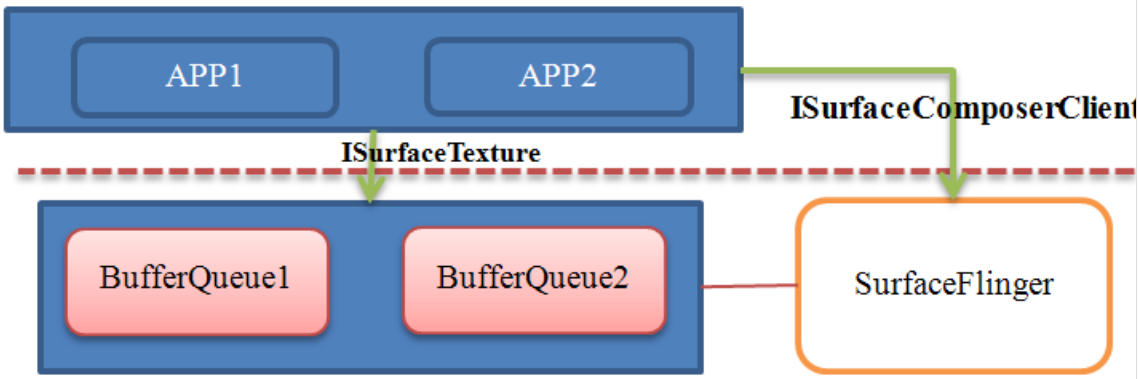

图 11　15 应用程序、BufferQueue及SurfaceFlinger间的关系

所以BootAnimation在其构造函数中就建立了与SurfaceFlinger的联系通道。那么它在什么时候会再去建立与BufferQueue的连接呢？因为BootAnimation继承自RefBase，当main函数中通过sp指针引用它时，会触发如下函数：

```
void BootAnimation::onFirstRef() {
    status_t err =mSession->linkToComposerDeath(this);//监听死亡事件
    if (err == NO_ERROR) {
        run("BootAnimation", PRIORITY_DISPLAY);//开启线程
    }
}
```

当一个client与远程server端的建立了binder联系后，它就可以使用这个server的服务了，但前提是服务端正常运行——换句话 说，假如出现了server异常的情况，client又如何知道呢？这就是linkToComposerDeath要解决的问题，它的第一个参数指明了接 收binder server死亡事件的人，在这个例子中就是BootAnimation自身。这是因为它继承了IBinder::DeathRecipient，并实现 了其中的binderDied接口。

如果上一步没有出错的话(err== NO_ERROR)，接下来就要启动一个新线程来承载业务了。为什么需要独立创建一个新的线程呢？前面main函数中大家应该发现了 BootAnimation启动了binder线程池，可以想象在只有一个线程的情况下，它是不可能既监听binder请求，又去做开机动画的绘制的。所 以当一个新的线程被run起来后，又触发了下列函数的调用：

```
status_t BootAnimation::readyToRun() {…
  /*第一部分，向server端获得buffer空间，从而得到EGL需要的本地窗口*/
   sp<SurfaceControl>control = session()->createSurface(0, dinfo.w,dinfo.h, PIXEL_FORMAT_RGB_565);
  SurfaceComposerClient::openGlobalTransaction();
   control->setLayer(0x40000000);
```

```
  SurfaceComposerClient::closeGlobalTransaction();
sp<Surface> s = control->getSurface();

  /*以下为第二部分，即EGL的使用流程*/
  const EGLint attribs[] = {…//属性值较多，节约篇幅，我们省略具体内容};
  EGLint w, h, dummy;
  EGLint numConfigs;//总共有多少个config
  EGLConfig config;
  EGLSurface surface;
  EGLContext context;
  EGLDisplay display =eglGetDisplay(EGL_DEFAULT_DISPLAY);//第一步，得到默认的物理屏幕
  eglInitialize(display, 0,0);//第二步，初始化
  eglChooseConfig(display,attribs, &config, 1, &numConfigs);//第三步，选取最佳的config
  surface =eglCreateWindowSurface(display, config, s.get(),NULL);//第四步，通过本地窗口创建Surface
  context =eglCreateContext(display, config, NULL, NULL);//第五步，创建context环境
  …
  if(eglMakeCurrent(display, surface, surface, context) == EGL_FALSE)//第六步，设置当前环境
     return NO_INIT;
  …
  return NO_ERROR;
}
```

从这个函数中不但可以看出应用程序是如何使用BufferQueue的，而且还有另外一个重要的学习点，即OpenglES与EGL的使用流程。在本书应用篇章中我们已经给出了EGL的使用实例，这里则可以做为第二个例子。

函数首先通过session()->createSurface()来获取一个SurfaceControl。其中session()得到的 是mSession变量，也就是前面构造函数中生成的SurfaceComposerClient对象，所以createSurface()最终就是由 SurfaceFlinger来实现的。只不过SurfaceFlinger中返回的其实是一个ISurface对象，本地端的 SurfaceComposerClient又包装了一层，变成了SurfaceControl，言下之意就是对ISurface进行管理。大家肯定会有 疑惑，ISurface从哪冒出来的，做什么的？要回答这点不难，只要看下SurfaceFlinger中最终是传了个什么对象过来就行：

```
sp<ISurface> SurfaceFlinger::createSurface(…)
{
        sp<LayerBaseClient>layer;
        sp<ISurface>surfaceHandle;
```

```
//中间省略layer的生成过程

        surfaceHandle =layer->getSurface();

        returnsurfaceHandle;...
```

我们省略了中间一大段过程，只保留与问题相关的部分，更详细的分析可以参见后面的SurfaceFlinger小节。从中可以清楚看 到，ISurface是通过layer->getSurface()得到的。Layer类在SurfaceFlinger中表示“层”的概念，而ISurface则是客户端与这个“层”进行沟通的通道。通俗地讲“层”就代表了一个“画面”，最终的显示结果就是通过对系统中同时存在的所有“画面”进 行处理得到的。打个比方来说，就好像是一排人各举着一张绘画作品，那么观察者从最前面往后看时，他首先可以看到的就是第一张画。而假如第一张画恰好比第二 张小，又或者第一张是透明/半透明的(这并非不可能，比如作者是在玻璃上创作的)，那么他才能看到第二张画，以此类推。。。

这个比喻告诉我们，layer是有层级的，越靠近用户的那个“层”就越有优势。

明白了这个道理，函数接下来调用setLayer就不难理解了。不过参数中传入了一个数值0x40000000，这又是什么意思？其实这个值就是 layer的层级，数字越大就越靠近用户，在显示系统中我们通常称为z-order。以后在Window Manager Service章节的分析中还会看到对setLayer的调用，因为此时系统中还只有开机画面一个应用程序，所以我们还不需要担心z-order的问题。

设置完层级后，我们接着control->getSurface()来得到一个Surface对象。相关的类越来越多了，而且由于命名上的不恰当，进一步加剧了大家理解上的困难。其实可以来猜测下这个类是做什么的？根据如下：

l 第二部分中的eglCreateWindowSurface使用了Surface，证明它必然是一个本地窗口

l 前几个小节我们介绍的两种本地窗口，一个是FramebufferNativeWindow，另一个是SurfaceTextureClient，那么看 来Surface必然要与其中一个有关联。很显然的，运行于应用程序端的本地窗口必然是SurfaceTextureClient，我们可以从 Surface的继承关系中得到验证：

```
class Surface : public SurfaceTextureClient
```

的确是太乱了，我们有必要先来整理下目前已经出现的容易混淆的相关类的关系。

ISurfaceComposerClient: 应用程序与SurfaceFlinger间的通道，在应用进程中则被封装在SurfaceComposerClient这个类中。这是一个匿名 binder server，由应用程序(具体位置在SurfaceComposerClient::onFirstRef中)调用SurfaceFlinger这个实 名binder的createConnection方法来获取到，服务端的实现是SurfaceFlinger::Client。

ISurface:由应用程序调用ISurfaceComposerClient::createSurface()得到，同时在SurfaceFlinger这一进程中将会有一个Layer被创建，代表了一个“画面”。ISurface就是控制这一画面的handle。

Surface:从逻辑关系上看，它是上述ISurface的使用者。从继承关系上看，它是一个SurfaceTextureClient，也就是 本地窗口。SurfaceTextureClient内部持有ISurfaceTexture，即BufferQueue的实现接口。换个角度来思考，当EGL想通过Surface这个native window完成某些功能时，后者实际上又利用ISurface和ISurfaceTexture来取得远程服务端的对应服务，以完成EGL的请求。

回到BootAnimation::readyToRun()中来。因为本地窗口Surface已经成功创建，接下来就该EGL上场了，具体流程我们在代码中都加了注释，这里就不赘述了。

当EGL准备好环境后，意味着程序可以正常使用opengl ES提供的各种API函数进行绘图了。这部分实现就集中在随后的threadLoop()以及android()/movie()中。因为不属于本小节的讨论范围，有兴趣的读者可以自行参阅学习。

最后来做下小结，一个典型的应用程序使用SurfaceFlinger进行绘图的流程如下图所示：

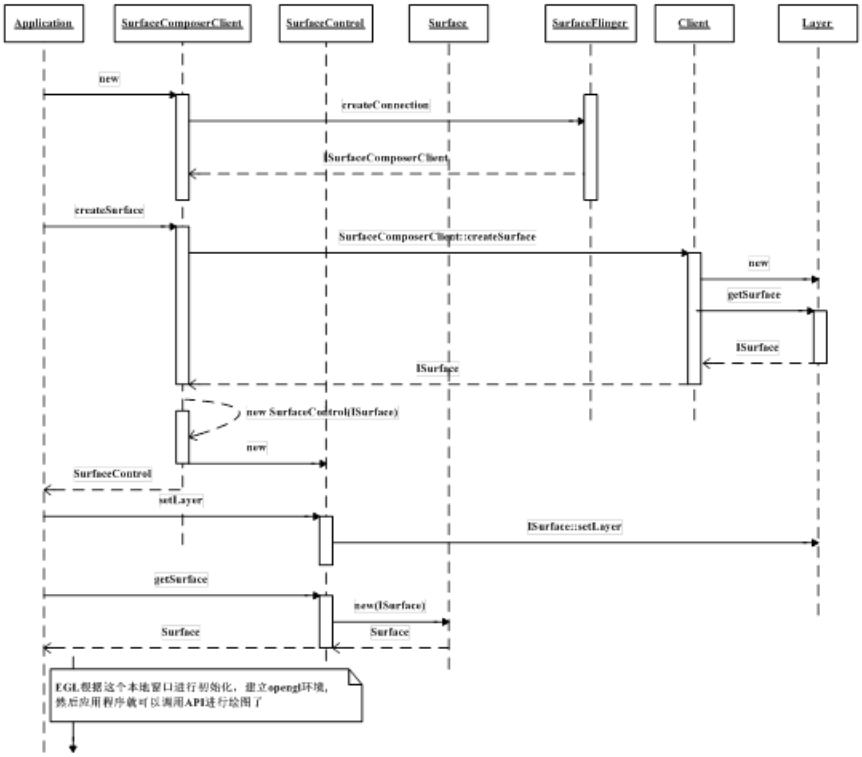

图 11　16 应用程序通过SurfaceFlinger进行绘图的典型流程

上图是从时序纵向角度总结出来的流程，我们再从横向的角度来看下，应该就更清楚了。

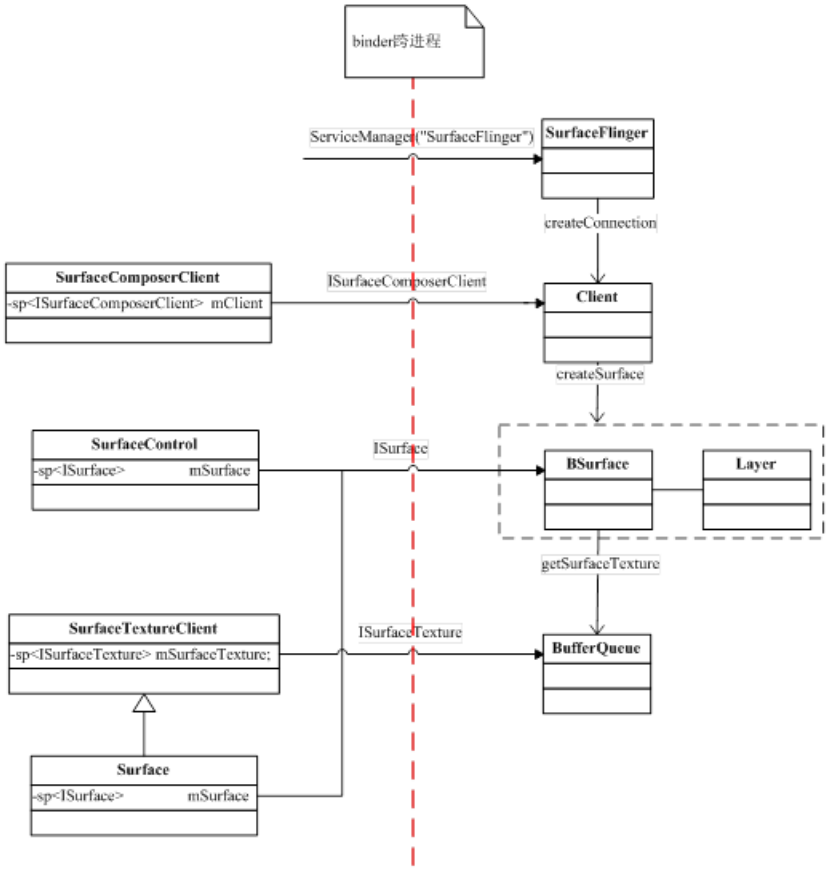

图 11　17 横向角度考查Surface相关类的关系

可以看到，涉及到的类还是比较多的，而且多数都涉及到跨进程通信。希望大家能熟悉这两张图，在后几个小节的学习中，如果觉得混乱的时候也可以回头来看下，加深印象。

1.1.1 应用程序与BufferQueue的关系

接着上一小节未解决完的问题继续讲解。

现在我们已经明白了应用程序利用SurfaceFlinger进行绘制工作的大致流程了，只不过在这个过程中直到最后才出现了BufferQueue。应用程序具体是如何借助BufferQueue来完成工作的呢？

仔细观察不难发现，当应用端通过ISurfaceComposerClient::createSurface()来发起创建Surface的请求时，SurfaceFlinger服务进程这边会创建一个Layer。既然Layer代表了一个画面图层，那么它肯定需要有存储图层数据的地方，因而我们 选择从这里做为入口。

```
/*frameworks/native/services/surfaceflinger/SurfaceFlinger.cpp*/

/*应用程序首先是要通过ISurfaceComposerClient来访问createSurface，这个接口实际上只是一个中介，它将应用 的请求传送到SurfaceFlinger的消息队列中，而不是直接调用SurfaceFlinger来处理。这样做是有必要的，因为一个系统中需要 SurfaceFlinger处理的来自各应用程序的消息是很多的，除非一些紧急情况，否则都应该排队等待*/

sp<ISurface>SurfaceFlinger::createSurface(ISurfaceComposerClient::surface_data_t* params,

constString8& name, const sp<Client>& client, DisplayID d,

uint32_tw, uint32_t h, PixelFormat format, uint32_t flags)

{

   sp<LayerBaseClient>layer;

   sp<ISurface>surfaceHandle;

   …

   sp<Layer>normalLayer;

   switch (flags &eFXSurfaceMask) { //Layer类型

      case eFXSurfaceNormal:

         normalLayer = createNormalSurface(client, d, w, h, flags, format);

         layer =normalLayer;

         break;

      case eFXSurfaceBlur://4.1系统中将Blur与Dim类型当成一种

      case eFXSurfaceDim:
```

```
            layer = createDimSurface(client, d, w, h, flags);
            break;
        case eFXSurfaceScreenshot:
            layer = createScreenshotSurface(client, d, w, h, flags);
            break;
    }
    if (layer != 0) {…
        surfaceHandle = layer->getSurface();
        …
    }
    return surfaceHandle;
}
```

可以看到，最后返回给应用程序的ISurface对应的变量是surfaceHandle，由layer通过getSurface()产生。从enum值定义来看，当前系统中有多达十几种Layer类型，只不过多数还没有真正实现，目前能用的只有四个，即eFXSurfaceNormal、 eFXSurfaceBlur、eFXSurfaceDim、eFXSurfaceScreenshot。第一种就是通常情况下的图层;第二和第三种在当 前系统中都被以eFXSurfaceDim来实现，从注释上看，很可能是Blur类型的Layer比较耗资源，所以暂时用Dim来取代。相信在后续的 Android版本中还会把它们再区分开来。

Layer和ISurface有什么联系？

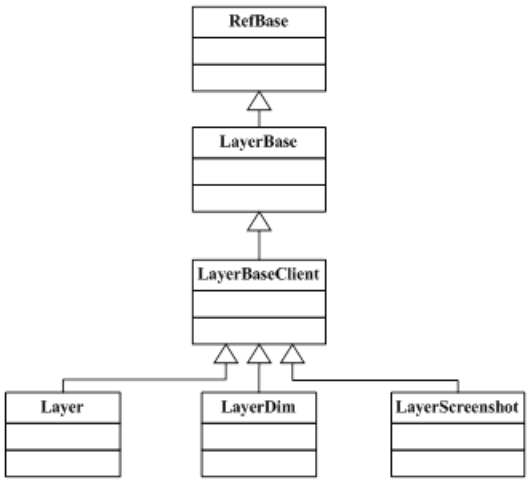

图 11　18 Layer类继承关系

```
/*frameworks/native/services/surfaceflinger/LayerBase.cpp*/
sp<ISurface> LayerBaseClient::getSurface()
{
    sp<ISurface> s;
```

```
    Mutex::Autolock _l(mLock);
    LOG_ALWAYS_FATAL_IF(mHasSurface, "LayerBaseClient::getSurface() hasalready been called");
    mHasSurface = true;
    s = createSurface(); //通过这个函数产生一个ISurface
    mClientSurfaceBinder= s->asBinder();
    return s;
}
```

这个函数只能被调用一次，由mHasSurface来控制。当被第二次调用时，就会发生assert错误。LayerBaseClient通过成员 变量mClientSurfaceBinder来记录ISurface。一个系统中同时存在的Layer和Surface通常不会只有一个，那么 ISurface是如何知道它所属的Layer的呢？

```
sp<ISurface> LayerBaseClient::createSurface()
{
    class BSurface : publicBnSurface, public LayerCleaner {
        virtualsp<ISurfaceTexture> getSurfaceTexture() const { return 0; }
    public:
        BSurface(constsp<SurfaceFlinger>& flinger, const sp<LayerBaseClient>&layer)
            :LayerCleaner(flinger, layer) { }
    };
    sp<ISurface> sur(newBSurface(mFlinger, this));
    return sur;
}
```

和一般的写法不同，ISurface的本地实现BSurface是定义在函数中的。不过这不是问题的关键，关键问题是BSurface中的 getSurfaceTexture居然直接返回了一个null(0)，显然如果是这样的话程序就没法继续执行了。唯一的解释是应该有 LayerBaseClient的子类重载实现了这一函数，来看下Layer中是不是这样的。

```
/*frameworks/native/services/surfaceflinger/Layer.cpp*/
sp<ISurface> Layer::createSurface()
{
    class BSurface : publicBnSurface, public LayerCleaner {
        wp<const Layer>mOwner;
        virtualsp<ISurfaceTexture> getSurfaceTexture() const {
            sp<ISurfaceTexture> res;
            sp<constLayer> that( mOwner.promote() );
```

```
        if (that != NULL){
            res =that->mSurfaceTexture->getBufferQueue();
        }
        return res;
    }
public:
    BSurface(constsp<SurfaceFlinger>& flinger, const sp<Layer>& layer)
        : LayerCleaner(flinger,layer), mOwner(layer) { }//mOwner在构造函数中以Layer赋值
};
sp<ISurface> sur(newBSurface(mFlinger, this));
return sur;
}
```

看来我们猜测的是对的，Layer的确重载了createSurface。而且这里的getSurfaceTexture就不是pseudo的了。函数的最后生成了一个BSurface，注意看下它的构造参数中传入了this指针，也就是Layer对象自身。这个指针是后期ISurface与它通信 的关键。同时作为入参的还有mFlinger，这样ISurface也可以访问到SurfaceFlinger了。因为这几个类都运行于同一个进程空间 中，可以看到它们都直接是内存地址的传递，不需要IPC通信。

当应用程序通过ISurfaceComposerClient::createSurface()得到一个ISurface后，它会接着使用ISurface::getSurfaceTexture()来取得可用的texture(具体是由Surface在构造时获取，然后再通过 Surface::init->setISurfaceTexture来将ISurfaceTexture赋值给 SurfaceTextureClient::mSurfaceTexture),这个ISurfaceTexture是应用程序中的opengl本地窗 口SurfaceTextureClient实现ANativeWindow规定的“协议”的基础）。

在getSurfaceTexture中，mOwner是一个指向Layer对象的弱指针，that->mSurfaceTexture指的 是Layer中的SurfaceTexture类成员变量，所以函数最终是通过SurfaceTexture::getBufferQueue()来获得 ISurfaceTexture。更具体地过程是，当Layer第一次被引用时，在onFirstRef()中会对mSurfaceTexture赋值。 因为ISurface(即BSurface)持有这个Layer对象(mOwner)，所以当应用程序申请getSurfaceTexture时，它才能 间接地向Layer要求取得ISurfaceTexture。我们前面说过，在binder server端，这个接口的实现类就是BufferQueue，如下所示：

```
/*frameworks/native/libs/gui/SurfaceTexture.cpp*/
sp<BufferQueue> SurfaceTexture::getBufferQueue() const {
    Mutex::Autolocklock(mMutex);
    return mBufferQueue;
}
```

函数直接返回了mBufferQueue，因为这个成员变量在SurfaceTexture构造时就已经初始化过了，它指向一个BufferQueue对象。大家一定要记住，ISurfaceTexture的server端实现是BufferQueue，由于命名上的差异，这点很 容易搞错。

这样应用程序与BufferQueue的关系就比较明朗了。虽然中间经历了多次跨进程通信，但对于应用程序来说最终只使用到了BufferQueue(通过ISurfaceTexture)。从本小节的内容中，我们也可以从侧面证明如下几个关键点：

(1) 应用程序可以调用createSurface来建立多个Layer，它们是一对多的关系。理由就是createSurface中没有任何机制来限制应用程序的多次调用，相反，它会把一个应用程序多次申请而产生的Layer统一管理，如下：

```
sp<ISurface> SurfaceFlinger::createSurface(…constsp<Client>& client)
{
   sp<LayerBaseClient>layer;
   … /*省略的这部分是生成layer的过程，前面已经分析过了*/
   if (layer != 0) {…
      ssize_t  token = addClientLayer(client,layer);//将client与layer记录起来
      surfaceHandle =layer->getSurface();
      if (surfaceHandle !=0) {…
         if (normalLayer !=0) {//只对normal layer生效
            Mutex::Autolock _l(mStateLock);
            mLayerMap.add(layer->getSurfaceBinder(), normalLayer);
         }
      }…
}
```

为应用程序申请的layer，一方面需要告知SurfaceFlinger，另一方面也要记录到各Client内部中，这两个步骤是由addClientLayer()分别调用Client::attachLayer()和SurfaceFlinger::addLayer_l()来完 成的。对于SurfaceFlinger，它需要对系统中当前所有的Layer进行Z-order排序，以决定用户所能看到的“画面”是什么样的。对于 Client，它则利用内部的mLayers成员变量来一一记录新增(attachLayer)和移除(detachLayer)的图层。

(2) 每个Layer对应一个BufferQueue，换句话说，一个应用程序可能对应多个BufferQueue。Layer没有直接持有BufferQueue，而是由其内部的mSurfaceTexture来管理。

我们以下面这个关系图来结束本小节的学习：

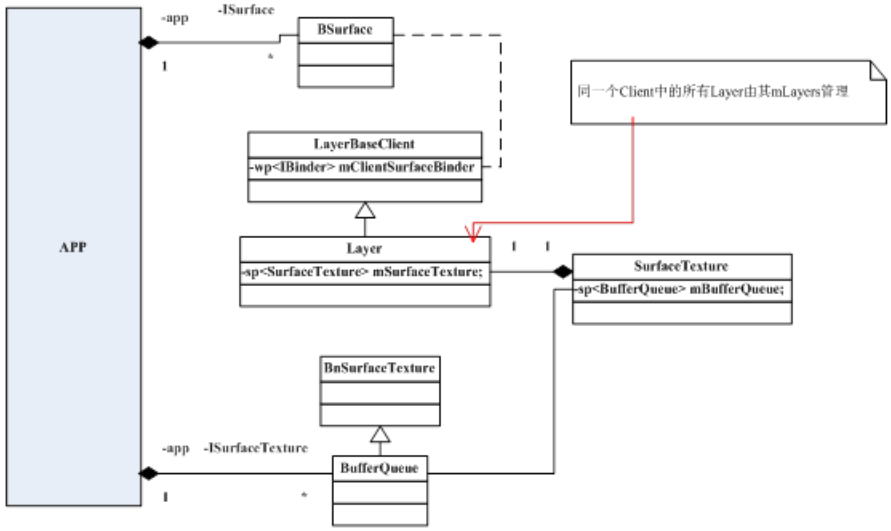

图 11　19 应用程序与BufferQueue的对应关系

1.1 SurfaceFlinger

从这一小节开始，我们正式切入SurfaceFlinger的分析。为了保持讲解的连贯性，部分内容可能在前面的章节中已经有所涉及了，接下来将会对其中的细节做更多的扩展讲解。

内容组织如下：

l 首先介绍Android 4.1引入的新特性(Project Butter)，理解这个项目是必要的，可以说SurfaceFlinger有很大一部分的内容就是围绕它来的

l SurfaceFlinger的启动过程及工作方式

l SurfaceFlinger与BufferQueue及应用程序间的关系

l SurfaceFlinger对VSYNC信号的处理过程(重点)

1.1.1 ProjectButter

直译过来，就是“黄油计划”，为什么叫这个名字呢？这个Project的目的是为了改善用户抱怨最多的Android几大缺陷之一，即UI响应速度 ——Google希望这一新计划可以让Android系统摆脱UI交互上给人带来的“滞后”感，而能像黄油一般“顺滑”。Google在2012年的 I/O大会上宣布了这一计划，并在Android 4.1中正式搭载了实现机制。

Butter中有两个重要的组成部分，即VSync和Triple Buffering。下面先分别介绍引入它们的原因。

喜欢玩游戏或者看电影的读者可能遇到过这样的情形：

Ø 某些游戏场面好像是几个场景“拼凑”而成的

Ø 电影画面不连贯，好像被“割裂”了

这样子描述有点抽象，我们引用widipedia上的一张图来看下实际的效果：

图 11　20 Screen Tearing实例

引自http://en.wikipedia.org/wiki/File:Tearing_%28simulated%29.jpg

我们把这种显示错误称为“screentearing”，那么为什么会出现这样的情况呢？

相信大家都能得出结论，那就是屏幕上显示的画面实际上来源于多个“帧”。

在一个典型的显示系统中，frame buffer代表了屏幕即将要显示的一帧画面。假如CPU/GPU绘图过程与屏幕刷新所使用的buffer是同一块，那么当它们的速度不同步的时候，是很 可能出现类似的画面“割裂”的。举个具体的例子来说，假设显示器的刷新率为66Hz，而CPU/GPU绘图能力则达到100Hz，也就是它们处理完成一帧 数据分别需要0.015秒和0.01秒。

以时间为横坐标来描述接下来会发生的事情，如下图：

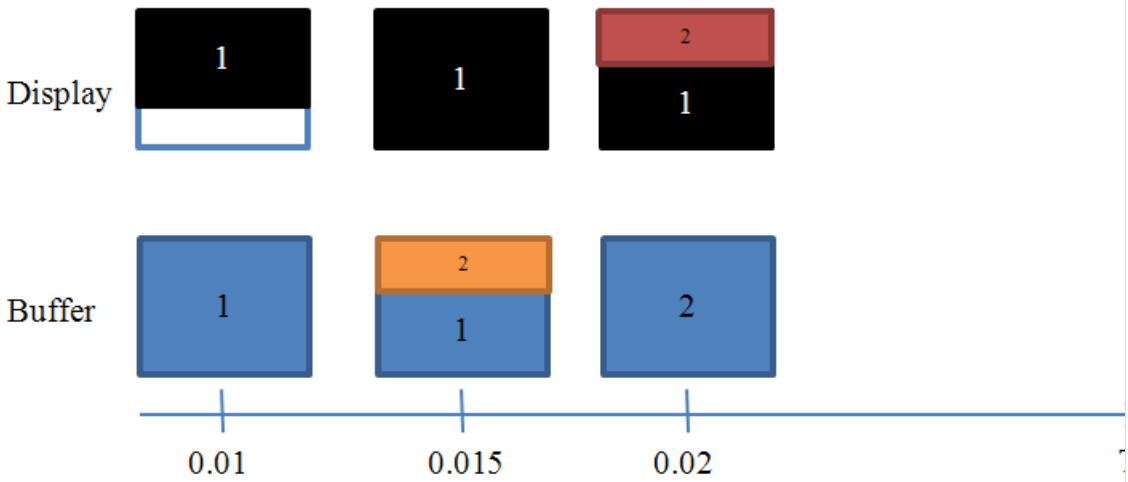

图 11　21 Screen Tearing产生过程分析

上半部分的方框表示在不同的时间点时显示屏的内容(加深的部分)，下半部分则是同一时间点时frame buffer中的数据状态，编号表示第几个frame，不考虑清屏。

·　　0.01秒

由于两者速率相差不小，此时buffer中已经准备好了第1帧数据，显示器只显示了第1帧画面的2/3

· 0.015秒

第1帧画面完整地显示出来了，此时buffer中有1/3的部分已经被填充上第2帧数据了

· 0.02秒

Buffer中已经准备好第2帧数据，而显示屏出现了screen tearing，有三分之一是第2帧内容，其余的则属于第1帧画面

在单缓冲区的情况下，这个问题很难规避。所以之前我们介绍了双缓冲技术，基本原理就是采用两块buffer。一块back buffer用于CPU/GPU后台绘制，另一块framebuffer则用于显示，当back buffer准备就绪后，它们才进行交换。不可否认，doublebuffering可以在很大程度上降低screen tearing错误，但是它是万能的吗？

一个需要考虑的问题是我们什么时候进行两个缓冲区的交换呢？假如是back buffer准备完成一帧数据以后就进行，那么如果此时屏幕还没有完整显示上一帧内容的话，肯定是会出问题的。看来只能是等到屏幕处理完一帧数据后，才可以执行这一操作了。

我们知道，一个典型的显示器有两个重要特性，行频和场频。行频(Horizontal ScanningFrequency)又称为“水平扫描频率”，是屏幕每秒钟从左至右扫描的次数; 场频(Vertical Scanning Frequency)也称为“垂直扫描频率”，是每秒钟整个屏幕刷新的次数。由此也可以得出它们的关系：行频=场频*纵坐标分辨率。

当扫描完一个屏幕后，设备需要重新回到第一行以进入下一次的循环，此时有一段时间空隙，称为VerticalBlanking Interval(VBI)。大家应该能想到了，这个时间点就是我们进行缓冲区交换的最佳时间。因为此时屏幕没有在刷新，也就避免了交换过程中出现 screentearing的状况。VSync(垂直同步)是VerticalSynchronization的简写，它利用VBI时期出现的 vertical sync pulse来保证双缓冲在最佳时间点才进行交换。

所以说V-sync这个概念并不是Google首创的，它在早些年前的PC机领域就已经出现了。不过Android 4.1给它赋予了新的功用，稍后就可以看到。

上面我们讨论的情况基于的假设是绘图速度大于显示速度，那么如果反过来呢？

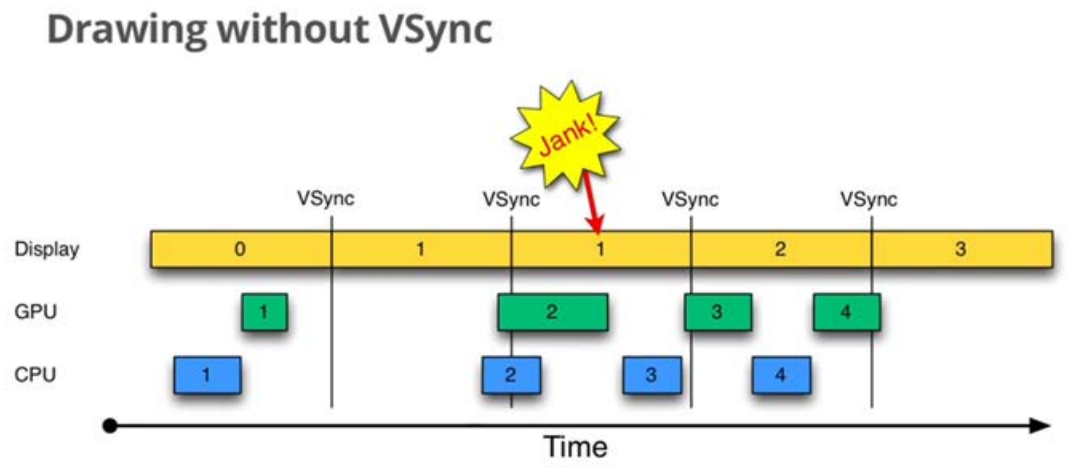

图 11　22绘图过程没有采用VSync同步的情况

引用自Google2012 I/O，作者Chet Haase和Romain Guy，AndroidUI Toolkit Engineers

这个图中有三个元素，Display是显示屏幕，GPU和CPU负责渲染帧数据，每个帧以方框表示，并以数字进行编号，如0、1、2等等。VSync用于指导双缓冲区的交换。

以时间的顺序来看下将会发生的异常：

Step1. Display显示第0帧数据，此时CPU和GPU渲染第1帧画面，而且赶在Display显示下一帧前完成

Step2. 因为渲染及时，Display在第0帧显示完成后，也就是第1个VSync后，正常显示第1帧

Step3. 由于某些原因，比如CPU资源被占用，系统没有及时地开始处理第2帧，直到第2个VSync快来前才开始处理

Step4. 第2个VSync来时，由于第2帧数据还没有准备就绪，显示的还是第1帧。这种情况被Android开发组命名为“Jank”。

Step5. 当第2帧数据准备完成后，它并不会马上被显示，而是要等待下一个VSync。

所以总的来说，就是屏幕平白无故地多显示了一次第1帧。原因大家应该都看到了，就是CPU没有及时地开始着手处理第2帧的渲染工作，以致“延误军机”。 Android系统中一直存在着这个问题，即便是上一版本的Ice Cream Sandwich。

从Android 4.1Jelly Bean开始，VSync得到了进一步的应用。系统在收到VSync pulse后，将马上开始下一帧的渲染。

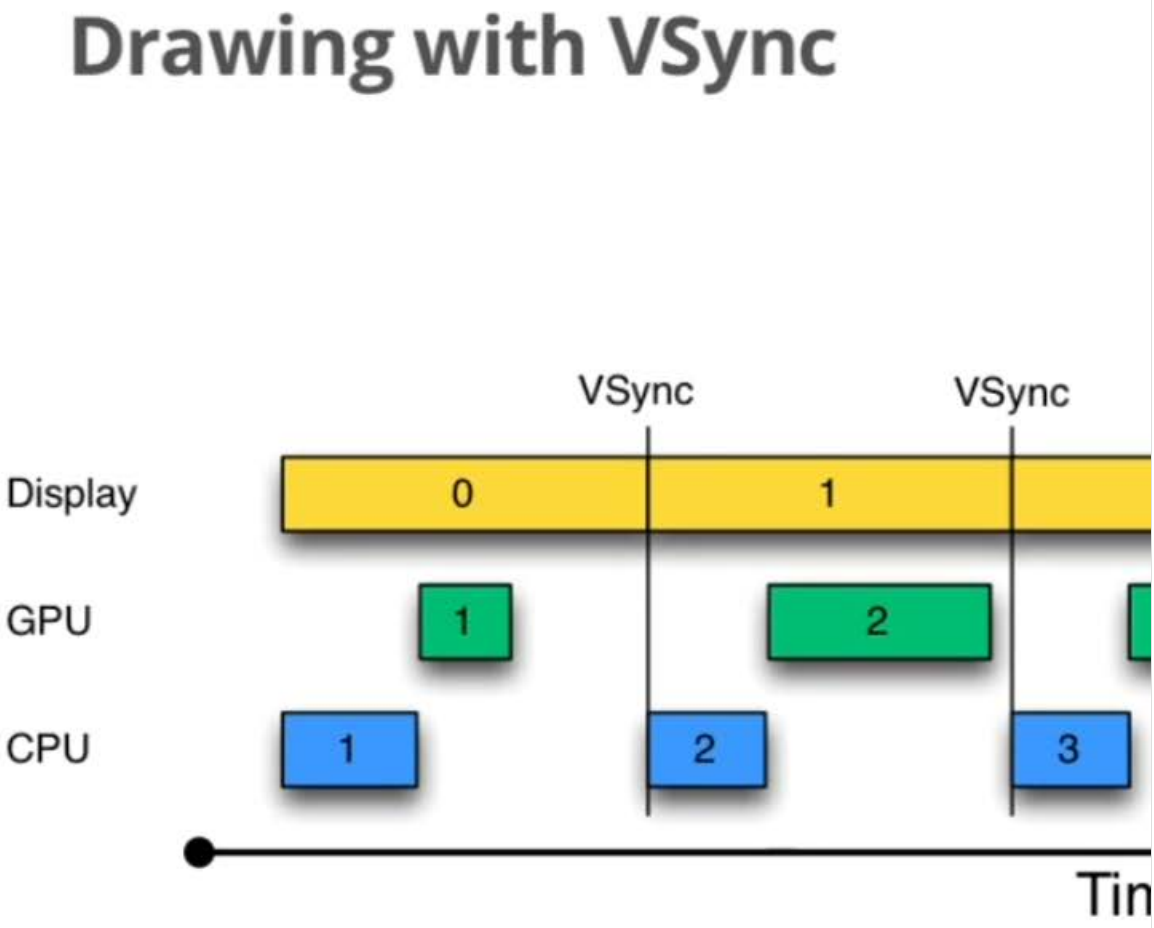

图 11　23 整个显示系统都以VSync进行同步

如上图所示，一旦VSync出现后，CPU不再犹豫，紧接着就开始执行buffer的准备工作。大部分的Android显示设备刷新率是60Hz, 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(FramesPer Second)高于这个值，那么这个方案是完美的，显示效果将很好。

可是我们没有办法保证所有设备的硬件配置都能达到要求。假如CPU/GPU的性能无法满足上图的条件，又是什么情况呢？

在分析这一问题之前，我们先来看下正常情况下，采用双缓冲区的系统的运行情况。

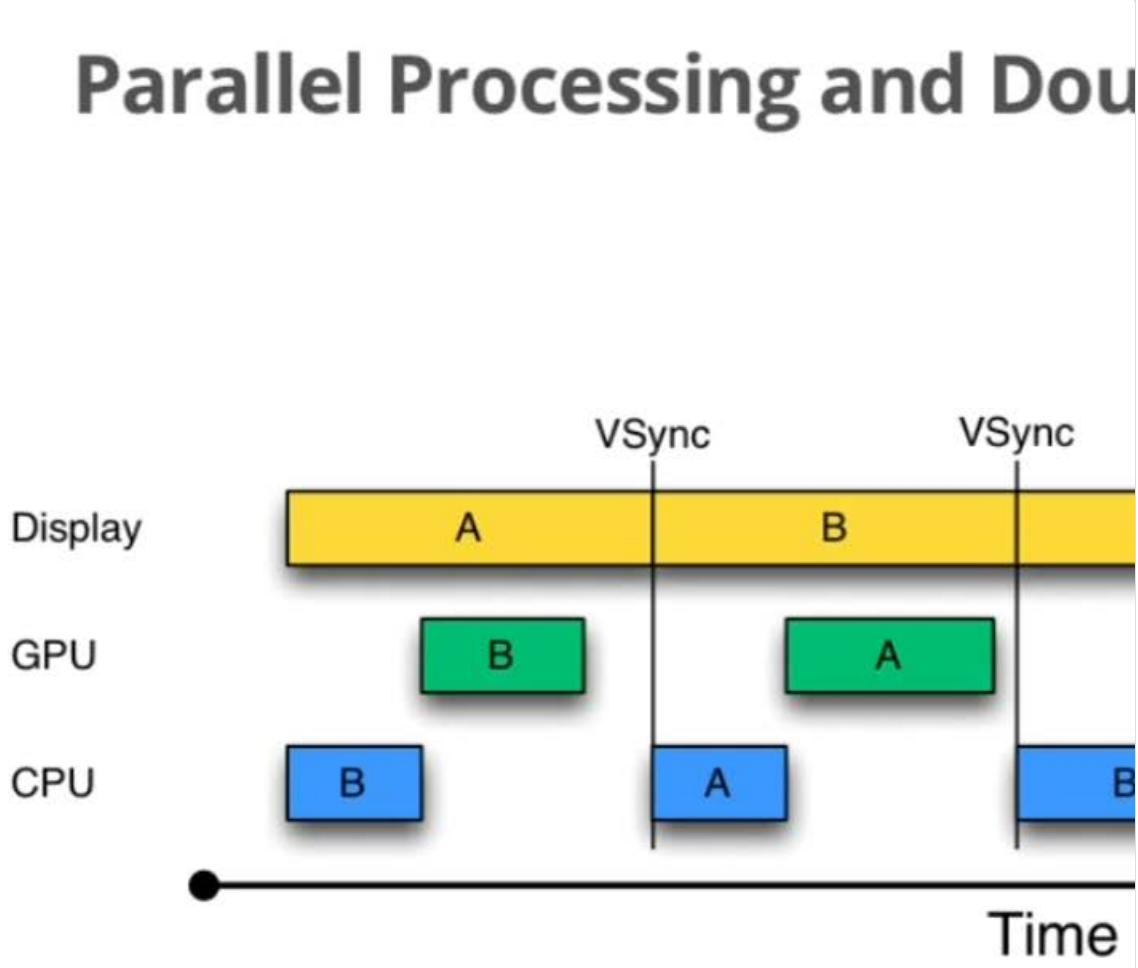

图 11　24 双缓冲展示

这个图采用了双缓冲，以及前面介绍的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，但总的来说都在16ms以内，因而不影响显示效果。A和B分别代表两个缓冲区，它们不断地交换来正确显示画面。

现在我们可以继续分析FPS低于屏幕刷新率的情况。

如下图所示：

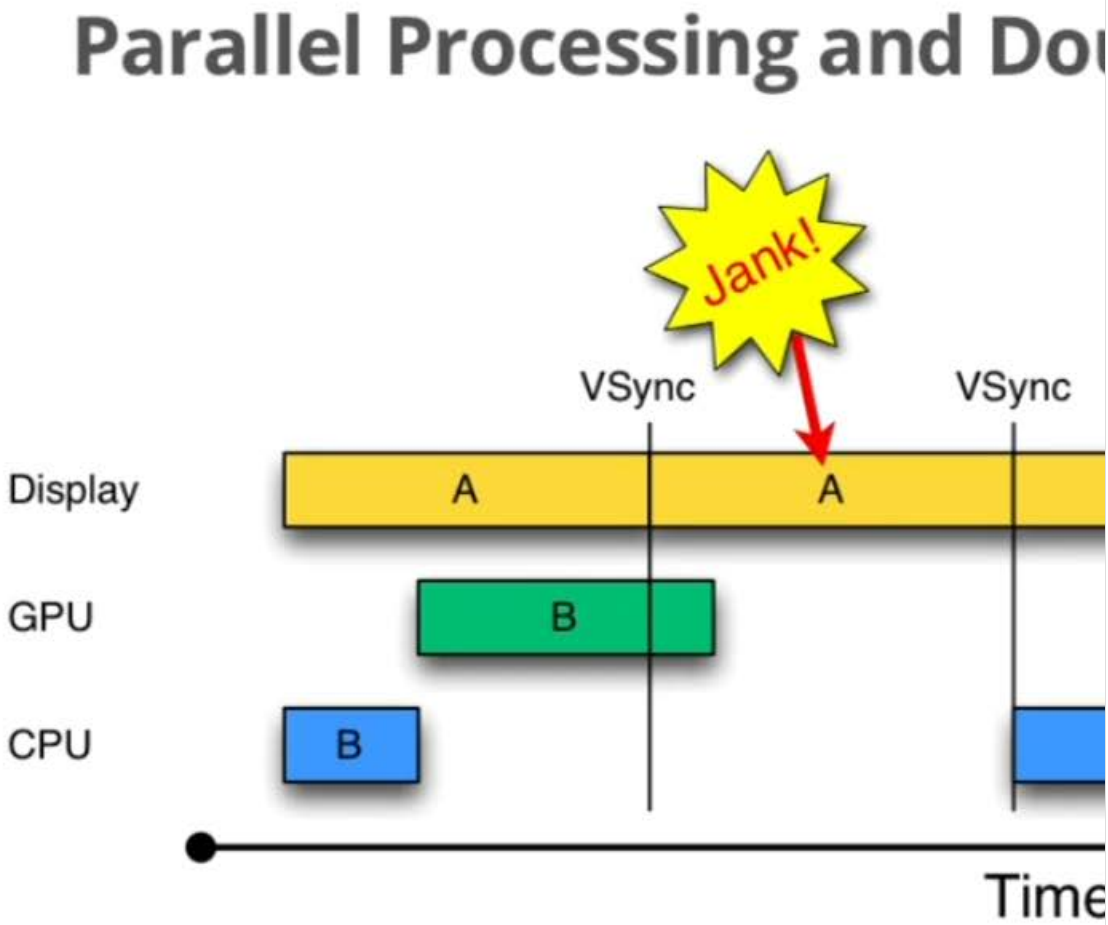

图 11　25 FPS低于屏幕刷新率的情况

当CPU/GPU的处理时间超过16ms时，第一个VSync到来时，缓冲区B中的数据还没有准备好，于是只能继续显示之前A缓冲区中的内容。而B 完成后，又因为缺乏VSync pulse信号，它只能等待下一个signal的来临。于是在这一过程中，有一大段时间是被浪费的。当下一个VSync出现时，CPU/GPU马上执行操 作，此时它可操作的buffer是A，相应的显示屏对应的就是B。这时看起来就是正常的。只不过由于执行时间仍然超过16ms，导致下一次应该执行的缓冲 区交换又被推迟了——如此循环反复，便出现了越来越多的“Jank”。

那么有没有规避的办法呢？

很显然，第一次的Jank看起来是没有办法的，除非升级硬件配置来加快FPS。我们关注的重点是被CPU/GPU浪费的时间段，怎么才能充分利用起 来呢？分析上述的过程，造成CPU/GPU无事可做的假象是因为当前已经没有可用的buffer了。换句话说，如果增加一个buffer，情况会不会好转 呢？

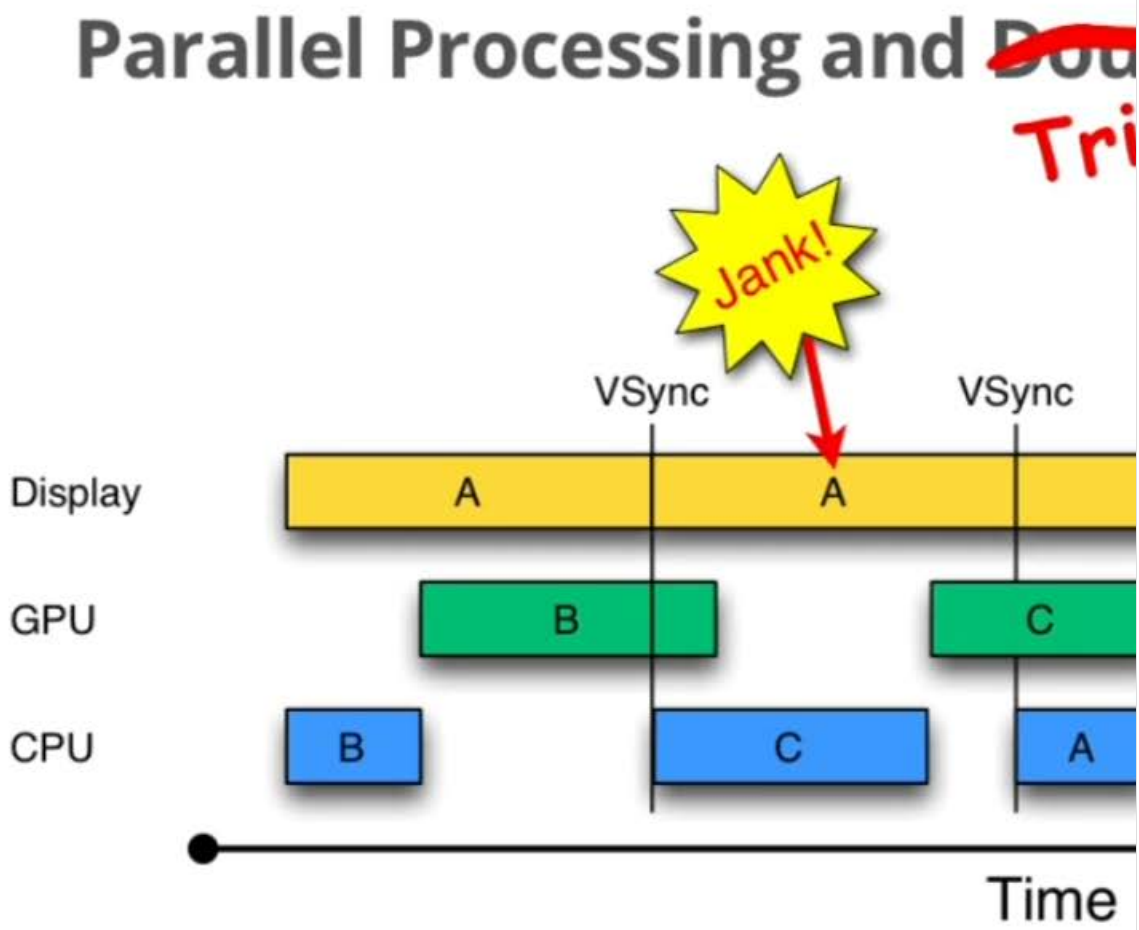

图 11　26 Triple Buffering

Triple Buffering是MultipleBuffering的一种，指的是系统使用3个缓冲区用于显示工作。我们来逐步分析下这个新机制是否有效。首先和预 料中的一致，第一次“Jank”无可厚非。不过让人欣慰的是，当第一次VSync发生后，CPU不用再等待了，它会使用第三个buffer C来进行下一帧数据的准备工作。虽然对缓冲区C的处理所需时间同样超过了16ms，但这并不影响显示屏——第2次VSync到来后，它选择buffer B进行显示;而第3次VSync时，它会接着采用C，而不是像double buffering中所看到的情况一样只能再显示一遍B了。这样子就有效地降低了系统显示错误的机率。

前面小节我们看到BufferQueue中最多有32个BufferSlot，不过在实际使用时具体值是可以设置的。

·　　TARGET_DISABLE_TRIPLE_BUFFERING

这个宏用于disable triple buffering。如果宏打开的话，Layer.cpp在onFirstRef有如下操作：

```
#ifdefTARGET_DISABLE_TRIPLE_BUFFERING
#warning"disabling triple buffering"
  mSurfaceTexture->setBufferCountServer(2);
#else
```

```
    mSurfaceTexture->setBufferCountServer(3);
#endif
```

也就是将mSurfaceTexture(即BufferQueue)中的mServerBufferCount设为2，否则就是3

· 对于应用程序来 说，它也可以通过ISurfaceTexture::setBufferCount来告诉BufferQueue它希望的Slot值，对应的则是 mClientBufferCount。默认情况下这个变量是0，表示应用端不关心到底有多少buffer可用。

· BufferQueue 中还有另一个变量mBufferCount，默认值是MIN_ASYNC_BUFFER_SLOTS。在具体的实现中，以上这三个变量都是要考虑到 的，BufferQueue会通过权衡各个值来选择最佳的解决方式

请大家务必理解本小节的几个场景分析，明白采用TripleBuffering、VSync机制的原因。带着这些理解进入SurfaceFlinger的学习，可以帮助我们“有的放矢”，对源码的分析也能事半功倍。

1.1.1 SurfaceFlinger的启动

SurfaceFlinger的启动和ServiceManager有点类似，它们都属于系统的底层支撑服务，必需在设备开机的早期就运行起来。

```
/*frameworks/base/cmds/system_server/library/System_init.cpp*/
extern "C" status_t  system_init()
{…
   property_get("system_init.startsurfaceflinger", propBuf,"1");
    if (strcmp(propBuf,"1") == 0) {
        SurfaceFlinger::instantiate();
}…
```

这个System_init.cpp会被编译到libsystem_server库中，然后由SystemServer在JNI层进行加载调用，从而启动包括SurfaceFlinger、SensorService等在内的系统服务。

和AudioFlinger/AudioPolicyService看到的情况一样，它调用instantiate来创建一个binder server，名称为“SurfaceFlinger”。而且强指针的特性让它在第一次被引用时会调用onFirstRef：

```
void SurfaceFlinger::onFirstRef()
{
```

```
    mEventQueue.init(this);//初始化事件队列
    run("SurfaceFlinger",PRIORITY_URGENT_DISPLAY);//启动一个新的业务线程
    mReadyToRunBarrier.wait();//等待新线程启动完毕
}
```

成员变量mEventQueue是一个MessageQueue类型的对象，我们在进程章节已经详细分析过消息队列与Looper、Handler 等类的使用，大家可以先回头参考下(虽然Java层的这些类与SurfaceFlinger中用到的有一定差异，但其本质原理是一样的)。既然有消息队 列，那就一定会有配套的事件处理器Handler以及循环体Looper，这些是在MessageQueue::init函数中创建的，即：

```
/*frameworks/native/services/surfaceflinger/MessageQueue.cpp*/
void MessageQueue::init(const sp<SurfaceFlinger>& flinger)
{
    mFlinger = flinger;
    mLooper = newLooper(true);
    mHandler = newHandler(*this);
}
```

也就是说这个MessageQueue类不但提供了消息队列，其内部还囊括了消息的处理机制，可以说是个“大杂烩”。那么这个Looper会在什么 时候运行起来呢？显然SurfaceFlinger需要先自行创建一个新的线程来承载这一“业务”，否则就会阻塞SystemServer的主线程，这一 点和AudioFlinger是有区别的。函数最后的mReadyToRunBarrier.wait()也可以证明这一点—— mReadyToRunBarrier在等待一个事件，在事件没有发生前其所在的线程就会处于等待状态。这是Android系统里两个线程间的一种典型交 互方式。举个例子来说，A线程将启动B线程，并且A接下来的工作会依赖于B进行。换句话说，A必顺要等到B说“好了，我已经ok”了，它才能继续往下走， 否则就会出错。由此可见，SurfaceFlinger新启动的这个线程中一定还会调用mReadyToRunBarrier。

这样我们也能推断出SurfaceFlinger一定是继承自Thread线程类的，如下所示：

```
class SurfaceFlinger :
        publicBinderService<SurfaceFlinger>,
        …
        protected Thread
```

所以上面代码中可以调用Thread::run()方法，进而启动一个名为“SurfaceFlinger”的线程，优先级别为PRIORITY_URGENT_DISPLAY。这个优先级是在ThreadDefs.h中定义的，如下表所示：

表格 11　6 Android系统的线程优先级定义

Priority	Value	Description
ANDROID_PRIORITY_LOWEST	19	可以使用最后的

ANDROID_PRIORITY_BACKGROUND	10	用于background tasks
ANDROID_PRIORITY_NORMAL	0	大部分线程都以这个优先级运行
ANDROID_PRIORITY_FOREGROUND	-2	用户正在交互的线程
ANDROID_PRIORITY_DISPLAY	-4	UI主线程
ANDROID_PRIORITY_URGENT_DISPLAY	-8	这个值由HAL_PRIORITY_URGENT_DISPLAY来指定，当前版本中是-8。只在部分紧急状态下使用
ANDROID_PRIORITY_AUDIO	-16	正常情况下的声音线程
ANDROID_PRIORITY_URGENT_AUDIO	-19	声音线程(通常情况不用)
ANDROID_PRIORITY_HIGHEST	-20	最高优先级，禁止使用
ANDROID_PRIORITY_DEFAULT	0	默认情况下就是ANDROID_PRIORITY_NORMAL
ANDROID_PRIORITY_MORE_FAVORABLE	-1	在上述优先级的基础上，用于加大优先级
ANDROID_PRIORITY_LESS_FAVORABLE	+1	在上述优先级的基础上，用于减小优先级

数值越大的，优先级越小。因为各等级间的数值并不是连续的，我们可以通过ANDROID_PRIORITY_MORE_FAVORABLE(-1)来适当地提高优先级，或者是利用ANDROID_PRIORITY_LESS_FAVORABLE(+1)来降低优先级。

由此可见，SurfaceFlinger工作线程所采用的优先级是相对较高的。这样做是必然的，因为屏幕显示无疑是人机交互中最直观的用户体验，任何滞后的响应速度都将大大降低产品的吸引力。

在执行了run()以后，Thread会自动调用threadLoop()接口，即：

```
bool SurfaceFlinger::threadLoop()

{

    waitForEvent();

    return true;

}
```

相当简洁的两句话，所有SurfaceFlinger接下来要执行的工作都涵括在这里了。其中waitForEvent()是SurfaceFlinger中的成员函数，它进一步调用mEventQueue.waitMessage():

```
void MessageQueue::waitMessage() {
```

```
    do {
        IPCThreadState::self()->flushCommands();
        int32_t ret =mLooper->pollOnce(-1);
        switch (ret) {
            caseALOOPER_POLL_WAKE:
            caseALOOPER_POLL_CALLBACK:
                continue;
            caseALOOPER_POLL_ERROR:
               ALOGE("ALOOPER_POLL_ERROR");
            caseALOOPER_POLL_TIMEOUT:
                // timeout(should not happen)
                continue;
            default:
                // should nothappen
               ALOGE("Looper::pollOnce() returned unknown status %d", ret);
                continue;
        }
    } while (true);
}
```

可以看到程序在这里进入了一个死循环，而且即便pollOnce的执行结果是ALOOPER_POLL_TIMEOUT，也同样不会跳出循环。这是Android系统在对待系统级严重错误时的一种普遍态度——一旦发生，就没救了，听天由命吧。。。

下面这句将在内部调用MessageQueue::mHandler来处理消息：

mLooper->pollOnce(-1);

注意pollOnce函数同样使用了一个死循环，它不断地取消息进行处理，关系如下：

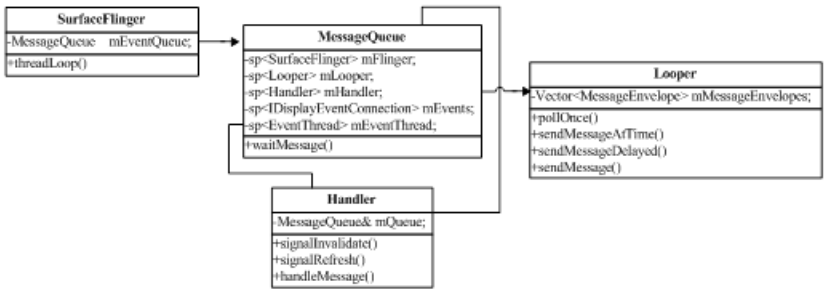

图 11　27 SurfaceFlinger中的消息循环机制

实际上SurfaceFlinger中的MessageQueue类不应该叫这个名字，因为会和我们传统的消息队列产生混淆和歧义。变量 mEventQueue是消息循环处理机制的管理者，其下包含了一个Looper和一个Handler。Looper采用的是/frameworks /native/libs/utils/Looper.cpp中的实现，SurfaceFlinger没有重新定义这个类。Looper中的mMessageEnvelope才是真正存储消息的地方。

太绕了，再好的设计也应该考虑可读性L

这样子就构建了一个完整的循环消息处理框架，SurfaceFlinger就是基于这个框架完成来自系统中各个程序的显示请求的。大家可能会有疑 问，mHandler是由MessageQueue在init()中直接通过newHandler()生成的，这样的话如何能处理特定的 SurfaceFlinger消息请求呢？个人感觉有这个困惑也是由于Handler类取名不当引起的。实际上此Handler并非我们经常看到的那个 Handler，这里的Handler是MessageQueue中自定义的一个事件处理器，也就是说它是专门为SurfaceFlinger设计的。

```
/*frameworks/native/services/surfaceflinger/MessageQueue.cpp*/

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;

   }

}
```

如上代码段所示，mHandler当收到INVALIDATE和REFRESH请求时，进一步回调了SurfaceFlinger中的onMessageReceived。等于是绕了一个大圈，又回到SurfaceFlinger中 了。

我们到目前为止还是没看到SurfaceFlinger是如何通知SystemServer线程解除等待的。这个工作是在下面的函数完成的：

```
status_t SurfaceFlinger::readyToRun()

{…

   mReadyToRunBarrier.open();//好了，现在可以解禁线程A了

}
```

函数readyToRun是在一个线程进入run循环前调用的，它为SurfaceFlinger的正常工作提供了各种必要的基础。我们在后续小节 还会看到其中的更多内容，这里先分析与消息处理有关的部分。前面所说的mReadyToRunBarrier果然在这里又被调用了，open()是告诉所 有正在等待的线程可以继续运行了。Barrier类内部实际上也是使用了Condition::broadcast()、 Condition::wait()等常规互斥方法，只是加了一层封装而已。

1.1.1 SurfaceComposerClient

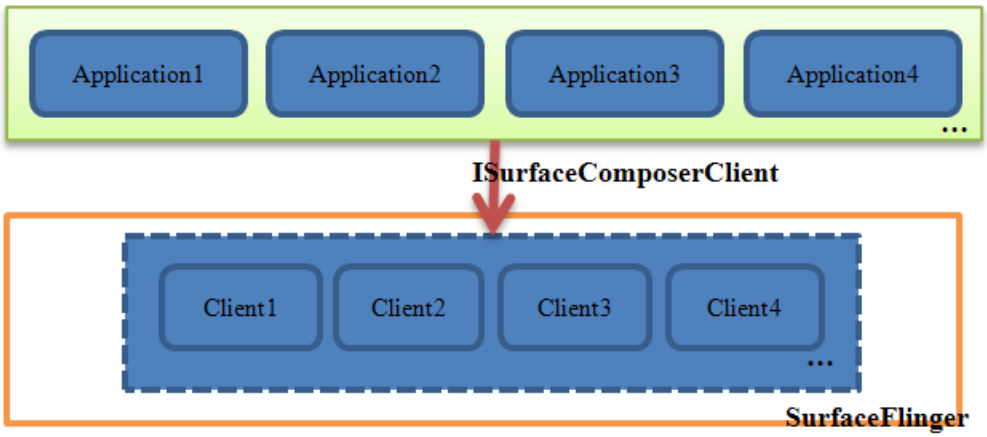

图 11　28 每个应用程序在SurfaceFlinger中都对应一个Client

SurfaceFlinger运行于SystemServer这一系统进程中，需要UI界面显示的应用程序则通过binder服务与它进行跨进程通 信。在音频系统的学习中，每一个AudioTrack在AudioFlinger中都可以找到一个对应的Track实现。这种设计方式同样适用于显示系 统，即任何有UI界面的程序都在SurfaceFlinger中有且仅有一个Client实例。

Client这个类名并没有完全表达出它的含义，因为在Android系统的很多其它地方你都可以找到同名的类。应用程序与 SurfaceFlinger间的接口是ISurfaceComposerClient，Client的父类是 BnSurfaceComposerClient,它是这一接口的本地端实现。

```
/*frameworks/native/include/gui/ISurfaceComposerClient.h*/

class ISurfaceComposerClient : public IInterface

{   …

virtual sp<ISurface>createSurface( surface_data_t* data, const String8& name, DisplayIDdisplay,

                        uint32_t w,uint32_t h, PixelFormat format, uint32_t flags) = 0;

   virtual status_t    destroySurface(SurfaceID sid) = 0;

};
```

接口中最重要的两个方法createSurface()和destroySurface()分别用于向SurfaceFlinger申请和销毁一个ISurface。那么既然有了Client，为什么还要再引出另一个binder对象呢？

这是因为每个SurfaceFlinger的客户程序都只会有唯一一个Client连接，但它们内部拥有的Surface数量却很可能有多个。通常 情况下，同一个Activity中的UI布局共用系统分配的Surface进行绘图，但像SurfaceView这种UI组件就是特例——它独占一个 Surface进行绘制。举个例子来说，如果我们制作一个带SurfaceView的视频播放器，其所在的应用程序最终就会有不止一个的Surface存 在。这样设计是必须的，因为播放视频对刷新频率要求很高，采用单独的Surface既可以保证视频的流畅度，也同时能让用户的交互动作(比如触摸屏操作) 及时得到响应。

Client的继承关系图如下所示：

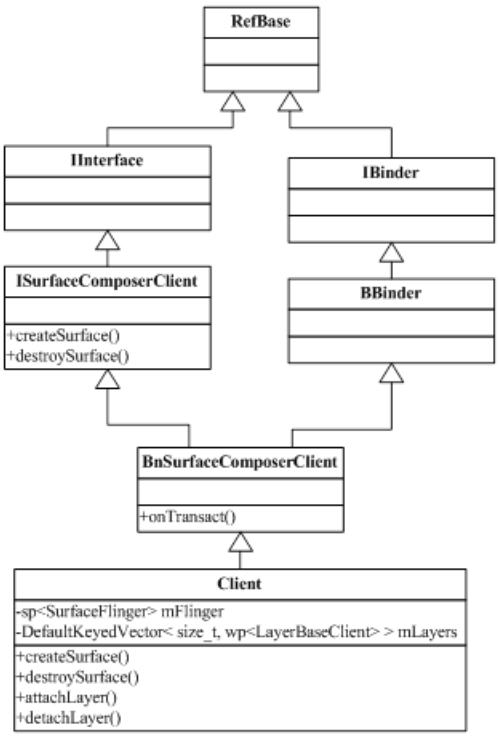

图 11　29 ISurfaceComposerClient的本地端实现

1. SurfaceFlinger::createConnection

由于Client属于匿名binder服务，外界的进程不可能直接获取到，因而首先需要借助SurfaceFlinger这一实名binder Service，源码实现如下：

```
sp<ISurfaceComposerClient> SurfaceFlinger::createConnection()
{
  sp<ISurfaceComposerClient> bclient;
   sp<Client>client(new Client(this));
   status_t err =client->initCheck();
   if (err == NO_ERROR) {
       bclient = client;
   }
   return bclient;
```

```
}
```

首先新建一个Client本地对象，initCheck初始化完成后如果没有错误的话，就可以ISurfaceComposerClient强指针返回。其中initCheck在目前版本中并没有做本质工作，它会直接返回NO_ERROR。

2. Client::createSurface

Client只是建立起了UI程序与SurfaceFlinger间的纽带，真正的图形层(Surface)创建需要另外申请，即调用Client提供的如下接口：

```
/*frameworks/native/services/surfaceflinger/SurfaceFlinger.cpp*/
sp<ISurface> Client::createSurface(…)
{
   class MessageCreateSurface: public MessageBase {…
      SurfaceFlinger*flinger;
      Client* client;
      …
   public:
      MessageCreateSurface(…): flinger(flinger), params(params),client(client), name(name),
          display(display), w(w), h(h), format(format), flags(flags)
      {//构造函数主要是对成员变量进行初始化
      }
      sp<ISurface>getResult() const { return result; }
      virtual bool handler(){
         result =flinger->createSurface(params, name, client, display, w, h, format, flags);
         return true;
      }
   };

sp<MessageBase> msg = newMessageCreateSurface(mFlinger.get(),params, name, this,
                                    display,w, h, format, flags);
   mFlinger->postMessageSync(msg); //将这一Message推送到SurfaceFlinger线程中
   return  static_cast<MessageCreateSurface*>(msg.get() )->getResult();
}
```

这个函数比较特别的地方，是它先在内部创建了一个MessageCreateSurface类，剩余部分代码就是围绕这个类展开的。那么 MessageCreateSurface有什么作用呢？从名称上看，它应该和Message有关，其父类

是MessageBase，定义在 MessageQueue.h中，如下关系图所示：

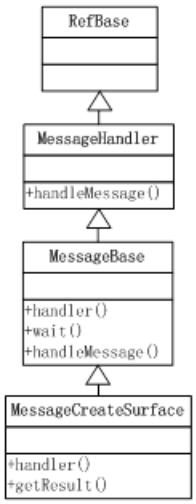

图 11　30 MessageCreateSurface

从这个继承图中可以大概看出一点端倪，即MessageCreateSurface是一个Message的承载体，并且内部提供了处理这条 Message的handler()函数。这是不是和Java层的Handler很像？可能也是这个原因，所以其“祖谱”中有一个类取名为 MessageHandler。回过头来看createSurface中的最后几行，程序将一个MessageCreateSurface对象msg发送 到了SurfaceFlinger中：

mFlinger->postMessageSync(msg);

为什么要这么做呢？

大家还记不记得在进程章节讲述Message、Looper、Handler的关系时曾经举过一个例子，为了方便阅读我们再把它列出来：

“。。。

打个比方来说，某天你和朋友去健身房运动，正当你在跑步机上气喘吁吁时，旁边有个朋友跟你说：“哥们，最近手头紧，借点钱花花”。那么这时候你就有两个选择：

Ø 马上执行

这就意味着你要从跑步机上下来，问清借多少钱，然后马上打开电脑进行转账汇款

Ø 稍后执行

上面的方法在某些场合下是有用的，不过大部分情况下“借钱”这种事并不是刻不容缓的，因而你可以跟你朋友说：“借钱没问题，你先和我秘书约时间，改 天我们具体谈细节”。那么在这种情况下，这个“借钱事件”就通过秘书这个MessageQueue进入了排队，所以你并不需要从跑步机上下来，中断健身运 动。后期秘书会一件件通知你MessageQueue上的待办事宜(当然你也可以主动问秘书)，直到“借钱事件”时你才需要和这位朋友进一步商谈。在一些 场合下这样的处理方式是合情合理的。比如说健身房一小时花费是10万，而你朋友只借一百块，那么在这时处理这一事件就不合理了

”

这个例子在createSurface()这个场景中也是同样适用的，如下图所示：

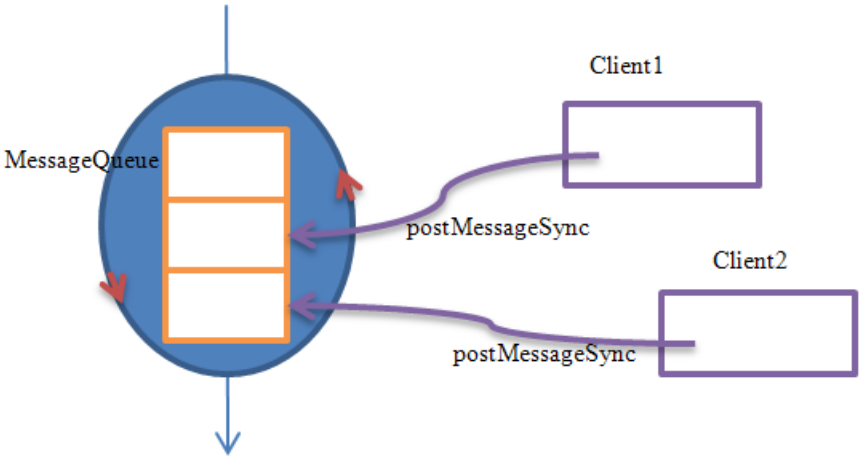

图 11　31 createSurface中的消息投递与执行

函数postMessageSync是SurfaceFlinger提供的一个接口，它通过mEventQueue将Msg压入队列中，并且还会进入等待状态，如下：

```
status_t SurfaceFlinger::postMessageSync(constsp<MessageBase>& msg, nsecs_t reltime, uint32_t flags) {
    status_t res =mEventQueue.postMessage(msg, reltime);
    if (res == NO_ERROR) {
        msg->wait();
    }
    return res;
}
```

MessageBase::wait()调用内部的Barrier::wait来实现等待，这意味着发送消息的线程将暂时停止执行，那么什么时候才能继续呢？显然得有人唤醒它才行。这个唤醒的地方隐藏在MessageBase::handleMessage()中，即：

```
/*frameworks/native/services/surfaceflinger/MessageQueue.cpp*/
void MessageBase::handleMessage(const Message&) {
    this->handler();
    barrier.open();
};
```

这其中的流程有点乱，我们以下图做下整理：

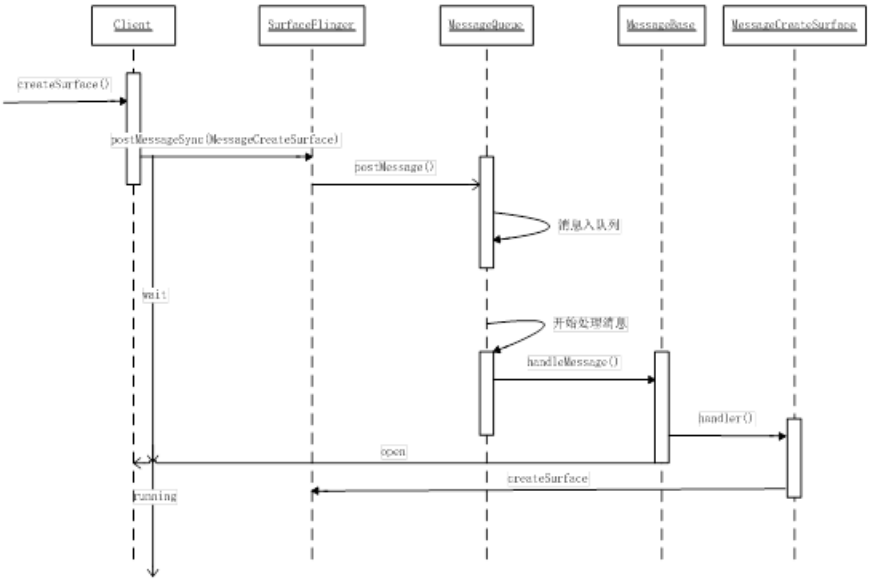

图 11　32 Client::createSurface流程图

另外SurfaceFlinger中还有一个相似的消息推送函数postMessageAsync，从名称上就可以看出它是要求异步执行的，通俗来讲就是推送过后就直接返回了，无需等待执行结果，读者可以自行分析下源码。

绕了一圈后，终于轮到SurfaceFlinger工作线程来处理createSurface()了，这个函数我们在前面讲解“应用程序与BufferQueue的关系”时已经详细分析过了，这里不再赘述。

1.1 VSync的产生和处理

前面小节ProjectButter中我们学习了Android 4.1显示系统中的新特性，其中一个就是加入了VSync同步。我们从理论的角度分析了采用这一机制的必要性和运作机理，那么SurfaceFlinger具体是如何实施的呢？

先来想一下有哪些东西要考虑：

·　　VSync信号的产生和分发

如果有硬件主动发出这一信号，那是最好的了;否则就得通过软件定时模拟来产生

·　　VSync信号的处理

当信号产生后，SurfaceFlinger如何在最短的时间内响应，具体处理流程是怎么样子的

1.1.1 VSync信号的产生和分发

在Android源码surfaceflinger目录下有一个displayhardware文件夹，其中HWComposer的主要职责之一，就是用于产生VSync信号。

```
/*frameworks/native/services/surfaceflinger/displayhardware/HWComposer.cpp*/

HWComposer::HWComposer(const sp<SurfaceFlinger>& flinger,EventHandler& handler, nsecs_t
```

```
refreshPeriod)
: mFlinger(flinger), mModule(0), mHwc(0), mList(0), mCapacity(0),mNumOVLayers(0),
 mNumFBLayers(0), mDpy(EGL_NO_DISPLAY),mSur(EGL_NO_SURFACE),
      mEventHandler(handler),mRefreshPeriod(refreshPeriod),
      mVSyncCount(0),mDebugForceFakeVSync(false)
{
    charvalue[PROPERTY_VALUE_MAX];
   property_get("debug.sf.no_hw_vsync", value, "0"); //系统属性
    mDebugForceFakeVSync =atoi(value);
    bool needVSyncThread =false;//是否需要软件模拟VSync
    int err = hw_get_module(HWC_HARDWARE_MODULE_ID, &mModule);//加载HAL模块
    if (err == 0) {
        err = hwc_open(mModule, &mHwc);//打开module
        if (err == 0) {
if(mHwc->registerProcs) { //注册硬件设备事件回调
                mCBContext.hwc= this;
                mCBContext.procs.invalidate = &hook_invalidate;
                mCBContext.procs.vsync = &hook_vsync;
                mHwc->registerProcs(mHwc, &mCBContext.procs);
                memset(mCBContext.procs.zero, 0, sizeof(mCBContext.procs.zero));
            }
            if(mHwc->common.version >= HWC_DEVICE_API_VERSION_0_3) {
                if(mDebugForceFakeVSync) {//用于调试
                    mHwc->methods->eventControl(mHwc, HWC_EVENT_VSYNC, 0);
                }
            } else {//有可能支持VSync的硬件模块是这个版本以后才加入的，老版本仍然需要软件模拟
                needVSyncThread = true;
            }
        }
    } else {
        needVSyncThread =true; //硬件模块打开失败，只能用软件模拟
```

```
    }
    if (needVSyncThread) {
        mVSyncThread = new VSyncThread(*this);//创建一个产生VSync信号的线程
    }
}
```

这个函数的核心就是决定VSync的“信号发生源”——硬件或者软件模拟。

假如当前系统可以成功加载HWC_HARDWARE_MODULE_ID=“hwcomposer”，并且通过这个库模块能顺利打开设备 (hwc_composer_device_t)，其版本号又大于HWC_DEVICE_API_VERSION_0_3的话，我们就采用“硬件源”(此 时needVSyncThread为false)，否则需要创建一个新的VSync线程来模拟产生信号。

(1)硬件源

如果mHwc->registerProcs不为空的话，我们注册硬件回调mCBContext.procs。定义如下：

```
    struct cb_context{
        callbacksprocs;
        HWComposer*hwc;
    };
```

调用registerProcs()时，传入的参数是&mCBContext.procs。后期当有事件产生时，比如vsync或者invalidate，硬件模块将分别通过procs.vsync和procs.invalidate来通知HWComposer。

```
void HWComposer::hook_vsync(struct hwc_procs* procs, int dpy,int64_t timestamp) {
    reinterpret_cast<cb_context *>(procs)->hwc->vsync(dpy,timestamp);
}
```

上面这个函数中，procs即前面的&mCBContext.procs，从指针地址上看它和&mCBContext是一致的，因而我们可以强制类型转换为cb_context来进行操作，并由此访问到hwc中的vsync实现：

```
void HWComposer::vsync(int dpy, int64_t timestamp) {
    mEventHandler.onVSyncReceived(dpy, timestamp);
}
```

HWComposer将VSync信号直接通知给mEventHandler，这个Handler由HWComposer构造时传入，换句话说，我们需要看下是谁创建了HWComposer。

```
/*frameworks/native/services/surfaceflinger/displayhardware/DisplayHardware.cpp*/
void DisplayHardware::init(uint32_t dpy)
{...
mHwc = newHWComposer(mFlinger, *this, mRefreshPeriod);
```

从这里可以看出来，HWComposer中的mEventHandler就是DisplayHardware对象，所以后者必须要继承

自HWComposer::EventHandler，以此处理产生的onVSyncReceived事件。

(2)软件源

软件源和硬件源的最大区别是它需要启动一个新线程VSyncThread，其运行优先级与SurfaceFlinger的工作线程是一样的，都是 -9。从理论的角度讲，任何通过软件定时来实现的机制都不可能是100%可靠的，即使优先级再高也可能出现延迟和意外。不过如果“不可靠”的机率很小，而 且就算出现意外时不至于是致命错误，那么还是可以接受的。所以说VSyncThread从实践的角度来讲，的确起到了很好的作用。

```
bool HWComposer::VSyncThread::threadLoop() {
 /*Step1. 系统是否使能了VSync信号发生机制*/
 { // 自动锁控制范围
     Mutex::Autolock_l(mLock);
     while (!mEnabled) {//VSync信号开关
        mCondition.wait(mLock);
     }
 }
 /*Step2. 计算产生VSync信号的时间*/
 const nsecs_t period = mRefreshPeriod;//信号的产生间隔
 const nsecs_t now =systemTime(CLOCK_MONOTONIC);
 nsecs_t next_vsync =mNextFakeVSync;//产生信号的时间
 nsecs_t sleep = next_vsync- now; //需要休眠的时长
 if (sleep < 0) {//已经过了时间点
     sleep = (period - ((now - next_vsync) %period));
     next_vsync = now +sleep;
 }
 mNextFakeVSync =next_vsync + period; //再下一次的VSync时间
 struct timespec spec;
 spec.tv_sec  = next_vsync / 1000000000;
 spec.tv_nsec = next_vsync% 1000000000;


 int err;
 do {
     err = clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &spec, NULL);//进入休眠
 } while (err<0&& errno == EINTR);
```

```
    if (err == 0) {
        mHwc.mEventHandler.onVSyncReceived(0, next_vsync);//和硬件源是一样的回调
    }
    return true;
}
```

Step1@ VSyncThread::threadLoop. 关于自动锁的使用我们已经分析过很多次了，不再赘述。这里要注意的是mEnabled这个变量，它是用于控制是否产生VSync信号的一个使能变量。当系 统希望关闭VSync信号发生源时，调用VSyncThread::setEnabled(false)，否则传入true。假如mEnabled为 false时，VSyncThread就处于等待状态，直到有人再次使能这个线程。

Step2@ VSyncThread::threadLoop. 接下来的代码用于真正产生一个VSync信号。可以想象一下，无非就是这些步骤：

- 计算下一次产生VSync信号的时间
- 进入休眠
- 休眠时间到了后，就代表应该发出VSync信号了，通知感兴趣的人
- 循环往复

变量mRefreshPeriod指定了产生VSync信号的间隔。它是在DisplayHardware::init中计算出来的：

mRefreshPeriod = nsecs_t(1e9 / mRefreshRate);

如果mRefreshRate为60Hz的话，mRefreshPeriod就差不多是16ms。

因为mNextFakeVSync代表的是“下一次”产生信号的时间点，所以首先将next_vsync=mNextFakeVSync。接着计算 sleep，也就是离产生信号的时间点还有多长(同时也是需要休眠的时间)。那么如果sleep的结果小于0呢？代表我们已经错过了这一次产生信号的最佳 时间点，这是有可能发生的。在这种情况下，就计算下一次最近的VSync离现在还剩多少时间，公式如下：

sleep = (period - ((now - next_vsync) % period));

我们以下图来表述下采用这个公式的依据：

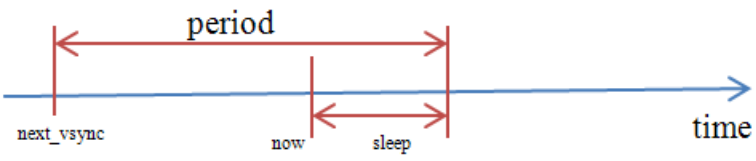

图 11　33 休眠时间推算简图

这个图的前提是now超时时间不超过一个period。因而公式中还要加上%period。

计算完成sleep后，mNextFakeVSync= next_vsync +period。这是因为mNextFakeVSync代表的是下一次threadLoop需要用到的时间点，而next_vsync是指下一次(最近一次)产生VSync的时间点。

如何在指定的时间点再产生信号呢？有两种方法，其一是采用定时器回调，其二就是采用休眠的形式主动等待，这里使用的是后一种。

可想而知这里的时间要尽可能精准，单位是nanosecond，即纳秒级。函数clock_nanosleep的第一个入参是CLOCK_MONOTONIC，这种时钟更加稳定，且不受系统时间的影响。

当休眠时间到了后，表示产生信号的时刻到了。根据前面的分析，就是通过mEventHandler.onVSyncReceived()回调来通知对消息感兴趣的人，这个做法软硬件都一样。

一次信号产生完成后，函数直接返回true，似乎没有看到循环的地方？这是因为当threadLoop返回值为“真”时，它将被系统再一次调用，从而循环起来。不清楚的可以参阅一下Thread类的实现。

接下来看下DisplayHardware如何处理这个VSync信号的。

中间过程很简单，我们就不一一解释。在DisplayHardware::onVSyncReceived中，它又再次调用内部mVSyncHandler的onVSyncReceived()，将消息向上一层传递。这个变量由EventThread在onFirstRef时通过DisplayHardware::setVSyncHandler()设置，代表的是EventThread对象本身，如下:

```
void EventThread::onFirstRef() {
    mHw.setVSyncHandler(this);//this指针代表EventThread对象
```

所以VSync信号被进一步递交到了EventThread中。显然，它也不是终点。

```
void EventThread::onVSyncReceived(int, nsecs_t timestamp) {
    Mutex::Autolock _l(mLock);
    mVSyncTimestamp =timestamp;
    mCondition.broadcast();//有人在等待事件的到来
}
```

等待VSync事件的地方很多，其中最重要的是EventThread::threadLoop()，这个函数将负责对VSync进行分发，决定谁有权利来最终处理这一事件。

这个函数的主体逻辑还是比较简单的，不过因为很长，内部又夹杂着多个循环体，显得不好理解，因此我们只摘选最重要的一部分来加快大家的阅读。

```
bool EventThread::threadLoop() {
    nsecs_t timestamp;
    DisplayEventReceiver::Event vsync;
```

```
Vector<wp<EventThread::Connection> > displayEventConnections;
do {//Step1. 第一个循环体
    Mutex::Autolock_l(mLock);
    do {…//Step2. 第二个循环体，决定是否上报VSync
    } while(true);

    //跳出循环，接下来就要准备分发VSync了
    mDeliveredEvents++;
    mLastVSyncTimestamp =timestamp;

    const size_t count =mDisplayEventConnections.size();
    for (size_t i=0 ;i<count ; i++) {
        …//Step3. 第三个循环中逐个判断各connection是否需要上报
        if (reportVsync) {
            displayEventConnections.add(connection);
        }
    }
} while(!displayEventConnections.size());//只要size不等于0就可以退出循环了

// 终于开始分发了。。。
vsync.header.type =DisplayEventReceiver::DISPLAY_EVENT_VSYNC;
vsync.header.timestamp =timestamp;
vsync.vsync.count =mDeliveredEvents;
const size_t count =displayEventConnections.size();
for (size_t i=0 ;i<count ; i++) {//Step4. 第四个循环体，分发事件
    sp<Connection>conn(displayEventConnections[i].promote());
    if (conn != NULL) {
        status_t err =conn->postEvent(vsync);//通知connection发起者
        if (err == -EAGAIN|| err == -EWOULDBLOCK) {
            //这两个错误是指对方当前不接受事件，有可能是暂时性的
```

```
            } else if (err< 0) {
                //发生了致命错误，一律移除
                removeDisplayEventConnection(displayEventConnections[i]);
            }
        } else {//connection已经死了，将它移除
            removeDisplayEventConnection(displayEventConnections[i]);
        }
    }
    …
    return true;
}
```

一共有四个循环体，看起来很乱，我们先以伪代码的形式来重新表述一遍：

```
do {//第一个循环体
    do {
//第二个循环体，判断当前系统是否允许上报VSync
    } while(true);
    for (size_t i=0 ; i<count ; i++) {
        //第三个循环体，逐个计算需要上报的connection个数
    }
} while (!displayEventConnections.size());/*一旦需要上报的连接数超过0，
就可以退出循环了*/
for (size_t i=0 ; i<count ; i++) {
/*第四个循环，开始实际的分发。这时要先考虑connection是否死亡，然后就是判断分发后是否有异常返回，比如EWOULDBLOCK等 等。对于暂态的错误，理论上是要再重发的，不过当前系统还没有这么做。的确，一方面这将使程序逻辑变得复杂，另一方面，即便丢一两个VSYNC，也无伤大 局。所以从源代码注释来看，将来这部分也不会改善。*/
}
```

相信大家结合这段伪代码再来对照源码，就比较清楚了。

对VSYNC信号感兴趣的人，可以通过registerDisplayEventConnection()来与EventThread建立一个连 接。搜索代码可以发现，当前系统中建立了连接的对象是MessageQueue，具体代码在MessageQueue::setEventThread()中。

我们以下图来总结本小节的内容：

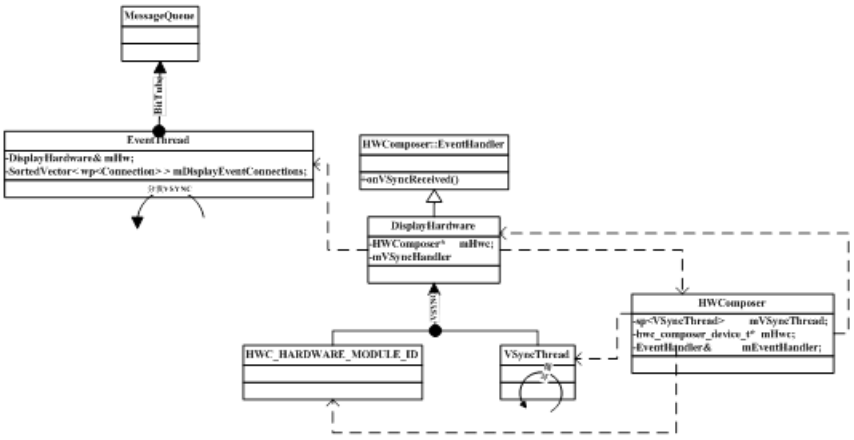

图 11　34 VSYNC信号的产生与分发

整体逻辑关系相对复杂，建议大家在做源码分析时，以下面两条线索进行：

l VSync信号的传递流向

l 各个类的静态依赖关系。比如DisplayHardware持有一个HWComposer对象，同时这个对象的mEventHandler成员变量又指向DisplayHardware

1.1.1 VSync信号的处理

经过上一小节的分析，现在我们已经明白了系统是如何通过硬件设备或者软件模拟来产生VSync信号的，也明白了它的流转过程。VSync最终会被EventThread::threadLoop()分发给各监听者，在当前版本中是MessageQueue。

MessageQueue通过与EventThread建立一个Connection来监听事件，简图如下：

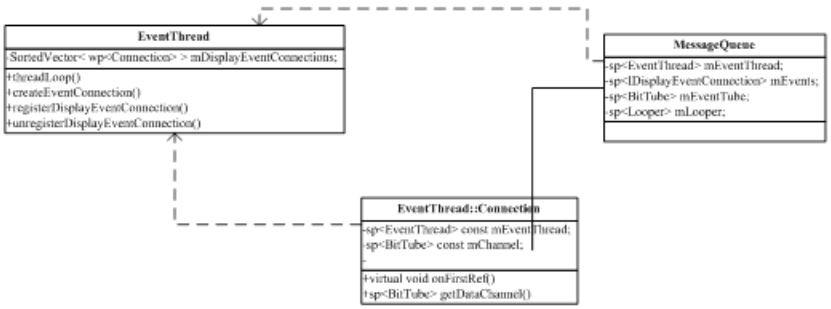

图 11　35 Connection代表了EventThread和对事件感兴趣的对象的连接

对VSYNC等事件感兴趣的对象，比如MessageQueue，首先要通过 EventThread::createEventConnection()来建立一个连接，实际上就是生成了一个 EventThread::Connection对象。这个对象将对双方产生如下影响：

l 当Connection::onFirstRef()时，它会主动调用 EventThread::registerDisplayEventConnection()来把自己加入到mDisplayEventConnections中，这是保证事件发生后EventThread能找到“连接”的关键一步

l 当MessageQueue得到Connection后，它会马上调用getDataChannel来获得一个BitTube。从逻辑关系上看，Connection只是双方业务上连接，而BitTube则是数据传输通道，各种Event信息就是通过这里传输的

```
void MessageQueue::setEventThread(const sp<EventThread>&eventThread)
{
    mEventThread =eventThread;
    mEvents = eventThread->createEventConnection(); //建立一个Connection
    mEventTube = mEvents->getDataChannel();//马上获取BitTube
    mLooper->addFd(mEventTube->getFd(), 0,ALOOPER_EVENT_INPUT, MessageQueue::cb_eventReceiver,
this);
}
```

从扮演的角色上来看，EventThread是Server，不断地往Tube中写入数据;而MessageQueue是Client，负责读取数 据。可能有人会很好奇，MessageQueue如何得知有Event到来，然后去读取它呢？答案就是它们之间的数据读写模式采用的是 Socket(AF_UNIX域)。

上面这个函数的末尾，通过Looper添加了一个fd，这实际上就是Socket pair中的一端。然后Looper将这个fd与其callback函数(即MessageQueue::cb_eventReceiver)加入全局的mRequests进行管理。

KeyedVector<int, Request> mRequests;

这个Vector会集中所有需要监测的fd，这样当Looper进行pollInner时，只要有事件需要处理，它就可以通过回调函数通知“接收者”。这里面的实现细节主要包括BitTube.cpp和Looper.cpp，有兴趣的读者可以自行研究下。

当Event发生后，MessageQueue::cb_eventReceiver开始执行，进而调用eventReceiver。如果 event的类型是DisplayEventReceiver::DISPLAY_EVENT_VSYNC，这是我们想要监听的事件，所以再调用 mHandler->signalRefresh()：

```
void MessageQueue::Handler::signalRefresh() {
    if((android_atomic_or(eventMaskRefresh, &mEventMask) & eventMaskRefresh)== 0) {
        mQueue.mLooper->sendMessage(this, Message(MessageQueue::REFRESH));
    }
}
```

根据前几个小节我们分析的SurfaceThread工作模式，这个Message会进入它的消息处理队列，然后在SurfaceFlinger::onMessageReceived()中得到处理：

```
void SurfaceFlinger::onMessageReceived(int32_t what)
```

```
{…
   switch (what) {
      caseMessageQueue::REFRESH: {
         const uint32_tmask = eTransactionNeeded | eTraversalNeeded;
         uint32_ttransactionFlags = peekTransactionFlags(mask);
         if(CC_UNLIKELY(transactionFlags)) {
            handleTransaction(transactionFlags);
         }
         handlePageFlip();
         handleRefresh();
         constDisplayHardware& hw(graphicPlane(0).displayHardware());
         …
         if(CC_LIKELY(hw.canDraw())) {
            handleRepaint();
           hw.compositionComplete();
            postFramebuffer();
         } else {
           hw.compositionComplete();
         }
      } break;
   }
}
```

经过几个版本的变迁，现在SurfaceFlinger只处理REFRESH一个消息，不过源码中遗留下了很多没用的代码J，大家注意辨别。我们将重点的部分通过高显帮大家划出来，即下面几个函数：

l handleTransaction

即处理事务，什么样的事务呢？在SurfaceFlinger::setTransactionState()中我们可以看到，假如当前的orientation和新的不符合时，会将eTransactionNeeded置位;当应用程序请求createSurface、removeSurface，或者addLayer、removeLayer时也会把它置位。另一个flag被置位的情况则包括：layer的size、alpha、matrix、transparentregion、visibility变化等等。

总结起来，就是当与系统显示相关的状态(比如新增/减少了Surface，显示屏的变化等等)改变，或者某个Layer自身状态(比如它的大小尺寸、可见性、透明度等等)改变时，就需要执行Transaction。

l handlePageFlip

由前面的分析我们知道，每个Layer对应着最多32个BufferSlot，这样系统在进行一次刷新时，必须先决定使用哪个buffer，并利用这一缓冲区更新纹理。另外，我们还需要计算所有图层的可见区域和“脏区域”，以便最终的合成显示。

l handleRefresh

版本更新遗留下的函数，当前实现中没有起到作用，相信在后续升级中会进一步完善。

l handleWorkList

创建HWComposer中的mList，这个列表将用于后续的layer合成。这个函数比较简单，我们不单独介绍。

l handleRepaint

计算出最终的脏区域，并执行实际的合成工作(composeSurfaces)，我们将做详细源码分析。

l postFramebuffer

将上一步中生成的缓冲区数据post到framebuffer中，这样才能真正在物理屏幕上显示出来。分为两条路径，即HWComposer::commit和直接调用eglSwapBuffers()

这些函数基本涵盖了SurfaceFlinger的所有功能，接下来的几个小节我们将详细分析它们，各个击破。

1.1.1 handleTransaction

有两个相似的函数，handleTransaction需要获取mStateLock锁，执行handleTransactionLocked，最后再将mHwWorkListDirty置为true。

后一个handleTransactionLocked才是真正处理业务的地方。从前面我们对transactionflags的解释来推断，它的具体工作应该会分为两部分，即traversal(对应eTraversalNeeded)和transaction(对应 eTransactionNeeded)。

```
void SurfaceFlinger::handleTransactionLocked(uint32_ttransactionFlags)
{
    const LayerVector&currentLayers(mCurrentState.layersSortedByZ);
    const size_t count =currentLayers.size();

    /*第一部分，traversal所有layer*/
```

```
const boollayersNeedTransaction = transactionFlags & eTraversalNeeded;
if (layersNeedTransaction){//是否需要traversal
    for (size_t i=0 ; i<count ; i++) {//逐个layer来处理
        constsp<LayerBase>& layer = currentLayers[i];
        uint32_t trFlags = layer->getTransactionFlags(eTransactionNeeded);/*这个layer是否
                                                需要doTransaction*/
        if (!trFlags)continue;//如果不需要的话，直接进入下一个

        const uint32_tflags = layer->doTransaction(0);//由各layer做内部处理，下面会做详细分析
        if (flags &Layer::eVisibleRegion)//各layer计算可见区域是否变化
            mVisibleRegionsDirty = true;//可见区域发生变化
    }
}

/*第二部分，SurfaceFlinger执行transaction*/
if (transactionFlags &eTransactionNeeded) {
    if (mCurrentState.orientation != mDrawingState.orientation) {
        /*角度产生变化，需要重新计算所有的可见区域，并且重绘*/
        const int dpy = 0;//第一个Display
        const intorientation = mCurrentState.orientation; //新的orientation
        GraphicPlane&plane(graphicPlane(dpy));
       plane.setOrientation(orientation);//将改变后的旋转角度设置到底层管理者

        // update theshared control block
        constDisplayHardware& hw(plane.displayHardware());
         /*下面的dcblk是一个供应用程序查询当前显示属性的“服务机构”，因而需要同步更新它*/
        volatiledisplay_cblk_t* dcblk = mServerCblk->displays + dpy;
       dcblk->orientation = orientation;
        dcblk->w =plane.getWidth();
        dcblk->h =plane.getHeight();
```

```
            mVisibleRegionsDirty= true; //旋转角度变化，当然可见区域也会变化
            mDirtyRegion.set(hw.bounds());//整个屏幕区域都是“脏”的
        }

        if (currentLayers.size() > mDrawingState.layersSortedByZ.size()) {// 有新的layer增加
            mVisibleRegionsDirty = true;//可见区域需要重新计算
        }

        if (mLayersRemoved) {/*也可能有layers已经被移除，这样可见区域也会有变化。
                             比如原本被遮盖的部分或许就暴露出来了*/
            mLayersRemoved =false; //
            mVisibleRegionsDirty = true;
            constLayerVector& previousLayers(mDrawingState.layersSortedByZ);
            const size_t count= previousLayers.size();
            for (size_t i=0 ;i<count ; i++) {
                constsp<LayerBase>& layer(previousLayers[i]);
                if(currentLayers.indexOf( layer ) < 0) {/*在之前状态中存在，在现有状态中找不到，
                                                        说明它被removed了*/
                    mDirtyRegionRemovedLayer.orSelf(layer->visibleRegionScreen);/* 被移除layer的可见
                                                                                区域*/
                }
            }
        }
    }
    commitTransaction();//和Layer::doTransaction()的做法基本一样，多了几步操作
}
```

和我们预料的一样，整个函数分别处理eTraversalNeeded和eTransactionNeeded两种情况。注意layersNeedTransaction这个变量其实应该叫layersNeedTraversal更贴切一点，而后面 eTransactionNeeded的情况下则没有另外声明bool变量来做判断。之所以叫做eTraversalNeeded，应该是因为这部分的源 码是要遍历所有layer来实现的。不过遍历的过程中调用的各layer内部函数又叫做doTransaction()。后一个eTransactionNeeded有点类似于做整体处理，所以整个handleTransactionLocked的函数逻辑就是“分(遍历各layer)à总(SurfaceFlinger来做综合处理)”。名字有点混乱，大家稍微整理一下。

(1) eTraversalNeeded

先来看eTraversalNeeded的处理。SurfaceFlinger中记录当前系统中layers状态的有两个全局变量，分别是：

State mDrawingState;

State mCurrentState;

前者是上一次“drawing”时的状态，而后者则是当前状态。这样我们只要通过对比这两个State，就知道系统中发生了什么变化，然后采取相应 的措施。它们内部都包含了一个LayerVector类型的layersSortedByZ成员变量，从变量名可以看出是所有layers按照Z- order顺序排列的Vector。

所以我们可以从mCurrentState.layersSortedByZ来访问到所有layer，然后对有需要执行transaction的图 层再调用内部的doTransaction()。显然，并不是每个layer在每次handleTransactionLocked中都需要调用doTransaction，判断的标准就是LayerBase::getTransactionFlags返回的标志中是否要求了eTransactionNeeded。大家要注意SurfaceFlinger和各Layer都有一个mTransactionFlags变量，不过含 义不同。另外，Layer中也同样有mCurrentState和mDrawingState，虽然它们也分别表示上一次和当前的状态，但所属的State结构体是完全不同的。特对比如下：

LayerBase::State	
-Geometry	active;
-Geometry	requested;
-uint32_t	z;
-uint8_t	alpha;
-uint8_t	flags;
-uint8_t	reserved[2];
-int32_t	sequence;
-Transform	transform;
-Region	transparentRegion;

SurfaceFlinger::State	
-LayerVector	layersSortedByZ;
-uint8_t	orientation;
-uint8_t	orientationFlags;

图 11　36 两个State结构体对比

为了理解Layer究竟在doTransaction中做了些什么,我们先插入分析下它的实现：

```
uint32_t  Layer::doTransaction(uint32_tflags)

{…

   const Layer::State&front(drawingState());/*mDrawingState */

   const Layer::State&temp(currentState());/*mCurrentState */



   const bool sizeChanged =(temp.requested.w != front.requested.w) || (temp.requested.h !=front.requested.h);

   if (sizeChanged) {

      …

      mSurfaceTexture->setDefaultBufferSize(temp.requested.w, temp.requested.h);
```

```
    }
    …
    return LayerBase::doTransaction(flags);
}
```

首先判断图层的尺寸是否发生了改变(sizeChanged变量)，即当前状态(temp)中的宽/高与上一次状态(front)一致与否。假如 size发生了变化，我们调用mSurfaceTexture->setDefaultBufferSize()来使它生效。其中mSurfaceTexture是在Layer::onFirstRef()时生成的SurfaceTexture对象;SurfaceTexture内 部又包含了一个BufferQueue成员变量(mBufferQueue)，所以SurfaceTexture就是BufferQueue的管理封装 类。这些知识我们在前几个小节已经详细分析过了，不清楚的可以回头看下。

BufferQueue::setDefaultBufferSize()改变的是内部的mDefaultWidth和 mDefaultHeight两个默认的宽高值。当我们调用requestBuffers()请求Buffer时，如果没有指定width和 height(值为0)，BufferQueue就会使用这两个默认配置。

最后函数调用其父类LayerBase的doTransaction()，这才是整个实现的核心，我们来看下:

```
uint32_t LayerBase::doTransaction(uint32_t flags)
{
    const Layer::State&front(drawingState());//和前面函数是一样的
    const Layer::State&temp(currentState());
    …
    if (temp.sequence != front.sequence) {
        flags |= eVisibleRegion;
        this->contentDirty = true;
        const uint8_t type =temp.transform.getType();
        mNeedsFiltering =(!temp.transform.preserveRects() || (type >= Transform::SCALE));
    }
    commitTransaction();
    return flags;
}
```

这个函数也不长，主要难点就是理解sequence。这个变量是一个int值，当Layer的position、z-order、alpha、matrix、transparent region、flags、crop等一系列属性发生变化时(这些属性的设置函数都以setXXX开头，比如setCrop、 setFlags)，mCurrentState.sequence就会自增。因而当doTransaction时，它和mDrawing.sequence的值就不一样了。相比于每次属性变化时都马上做处理，这样的设计是合理的。因为它在平时只收集属性的变化，而到了特定 的时刻(VSYNC信号产生时)才做统一处理。一方面这节约了系统资源;另一方面，部分属性的频繁变化会给整个画面带来不稳定感，采用这种方式就能规避这 些问题。

仔细观察Layer和LayerBase的这两个doTransaction()实现，我们可以大致得出它们的目的，就是通过分析当前与上一次的状 态变化，来制定下一步的操作——比如是否要重新计算可见区域(eVisibleRegion)、是否需要重

绘(contentDirty)等等。

这些操作有的是要由SurfaceFlinger统一部署的，因而通过flags返回给调用者，比如eVisibleRegion;有些属于Layer的“家务”，因而只要内部做标记就可以了。

最后，commitTransaction()把mCurrentState赋给mDrawingState，这样它们两个的状态就一样了。在下一轮doTransaction()前，mCurrentState又会跟着属性的变更而改变，如此循环往复。

这样Layer::doTransaction()函数就结束了，带着flags值返回到前面的SurfaceFlinger::handleTransactionLocked()中。一旦某个layer明确表明可见区域发生了改变(eVisibleRegion)，SurfaceFlinger中将其mVisibleRegionsDirty设为true，这个标志将影响后面的操 作。

(2) eTransactionNeeded

完成当前系统中所有layer的遍历后，SurfaceFlinger就进入自己的“内务”处理了，即handleTransactionLocked()源码中的第二部分。我们在代码中列出了它需要完成的工作，也就是：

l orientation

当显示屏旋转方向发生改变时，我们要把这一消息告知底层的管理者，即前几个小节讲过的GraphicPlane。变量dpy表示Display的编号，默认下都使用0。

接下来还需要同步更新的是mServerCblk中的信息，它是提供给各应用程序查询当前显示属性的，包括宽、高、格式、旋转角度、fps、密度等一系列数据。

显示屏旋转角度变化影响的范围是整个区域，所以在计算mDirtyRegion时，就直接设为整个屏幕(hw.bounds())

l 新增layer

与上一次相比，系统可能有新增的layer,我们只要对比两个State中的layer队列数目是否一致就可以得出结论了。假如图层有新增的话，可见区域需要重新计算，我们将mVisibleRegionsDirty置为true

l 移除layer

和上面的情况类似，有些layer也可能被移除了。针对这种情况，我们也需要重新计算可见区域。怎么知道哪些layer被移除了呢？有一个简单的办 法就是比较两个State中的layersSortedByZ——假如一个layer在上一次的mDrawingState中还有，到了这次 mCurrentState找不到了，那么就可以认定它被removed了。我们用mDirtyRegionRemovedLayer来记录这些被剔除图 层的可见区域，因为一旦图层被移除，意味着被它遮盖的区域就有可能重新显露出来

在handleTransactionLocked()末尾，它也调用了commitTransaction()来结束整个业务处理。和LayerBase相同的地方是：

mDrawingState = mCurrentState;

这样两个状态就一样了。

另外，SurfaceFlinger还需要通知所有被移除的layer,相应的回调函数是onRemoved()。Layer在得到这一消息

后，就知道它已经不会再被SurfaceFlinger处理了。最后，唤醒所有正在等待Transaction结束的线程：

mTransactionCV.broadcast();

SurfaceFlinger:: handleTransaction ()的流程图如下：

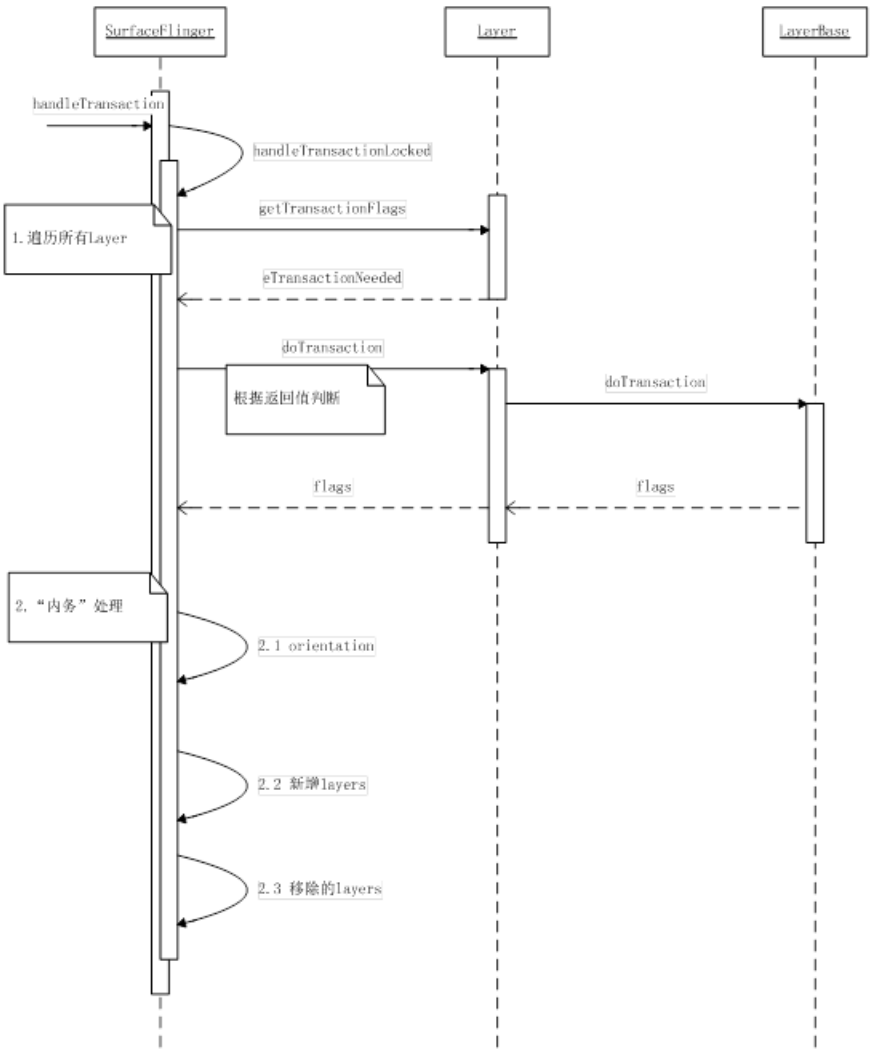

图 11　37 handleTransaction流程图

1.1.1 handlePageFlip

PageFlip可以理解为“翻页”。从这个意思上来看，它应该与图层缓冲区有关系——因为是多缓冲机制，在适当的时机，我们就需要做“翻页”的动作。

```
void SurfaceFlinger::handlePageFlip()

{…

    const DisplayHardware&hw = graphicPlane(0).displayHardware();//编号为0的Display

    const RegionscreenRegion(hw.bounds());//整个屏幕区域
```

```
const LayerVector&currentLayers(mDrawingState.layersSortedByZ);/*当前所有layers*/


const bool visibleRegions = lockPageFlip(currentLayers);/*Step1.下面会详细分析这个函数，注意它的返回
                                        值是一个bool类型变量。*/
       if (visibleRegions ||mVisibleRegionsDirty) {//可见区域发生变化
           RegionopaqueRegion;//不透明区域
           computeVisibleRegions(currentLayers, mDirtyRegion, opaqueRegion);/*Step2.计算可见区域*/


           /*Step3.重建mVisibleLayersSortedByZ，即所有可见图层的排序*/
           const size_t count= currentLayers.size();
           mVisibleLayersSortedByZ.clear();//清空
          mVisibleLayersSortedByZ.setCapacity(count);//容量
           for (size_t i=0 ;i<count ; i++) {
              if(!currentLayers[i]->visibleRegionScreen.isEmpty())//当前图层有可见区域
                 mVisibleLayersSortedByZ.add(currentLayers[i]);
           }
           mWormholeRegion = screenRegion.subtract(opaqueRegion);/*Step4.虫洞计算*/
          mVisibleRegionsDirty = false;
          invalidateHwcGeometry();
       }
   unlockPageFlip(currentLayers);/*Step5.与lockPageFlip相对应 */
    …
  mDirtyRegion.andSelf(screenRegion);//排除屏幕范围之外的区域
}
```

Step1@SurfaceFlinger::handlePageFlip，通过lockPageFlip分别锁定各layer当前要处理的缓冲区。SurfaceFlinger::lockPageFlip逻辑比较简单，我们直接来看下Layer::lockPageFlip:

```
void Layer::lockPageFlip(bool& recomputeVisibleRegions)
{…
   if (mQueuedFrames > 0) {…
       if(android_atomic_dec(&mQueuedFrames) > 1) {
           mFlinger->signalLayerUpdate();
```

```
        }
        …
        Rejectr(mDrawingState, currentState(), recomputeVisibleRegions);
        if (mSurfaceTexture->updateTexImage(&r) < NO_ERROR) {//加载新的纹理
           recomputeVisibleRegions = true;
            return;
        }
        mActiveBuffer = mSurfaceTexture->getCurrentBuffer();//当前活跃的缓冲区
        …//其它内部变量更新
        glTexParameterx(GL_TEXTURE_EXTERNAL_OES,GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);//配置参数
        glTexParameterx(GL_TEXTURE_EXTERNAL_OES,GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    }
}
```

SurfaceTexture设置了一个FrameQueuedListener，当BufferQueue中的一个BufferSlot被 queued后，它首先通知这个Listener，进而调用所属Layer的onFrameQueued。这个函数会增加mQueuedFrames的计 数，并且向SurfaceFlinger发出一个invalidate信号(signalLayerUpdate)。

因而如果当前没有任何queuedbuffer的话，lockPageFlip()什么都不用做。假如当前有多个mQueuedFrames的话，除了正常处理外，我们还需要另外调用signalLayerUpdate来发出一个新的event。

Layer中持有一个SurfaceTexture对象(成员变量mSurfaceTexture),用于管理BufferQueue(成员变量mBufferQueue)。一旦SurfaceFlinger需要对某个buffer进行操作时，根据前几个小节对BufferQueue状态迁移的分 析，我们知道它首先要accquire它，这个动作被封装在SurfaceTexture::updateTexImage中。除此之外，这个函数还需要 根据Buffer中的内容来更新Texture，稍后我们做详细分析。

在lockPageFlip内部，定义了一个Reject结构体，并做为函数参数传递给updateTexImage。后者在acquire到buffer后，调用Reject::reject()来判断这个缓冲区是否符合要求。如果updateTexImage成功的话，Layer就可以更新 mActiveBuffer，也就是当前活跃的缓冲区(每个layer对应32个BufferSlot);以及其它一系列相关内部成员变量，比如 mCurrentCrop、mCurrentTransform等等，这部分源码省略。

纹理贴图还涉及到很多细节的配置，比如说纹理图像与目标的尺寸大小有可能不一致，这种情况下怎么处理？或者当纹理坐标超过[0.0,1.0]范围 时，应该怎么解决。这些具体的处理方式都可以通过调用glTexParameterx(GLenumtarget, GLenum pname, GLfixed param)来配置。函数的第二个参数是需要配置的对象类型(比如GL_TEXTURE_WRAP_S和GL_TEXTURE_WRAP_T分别代表两个 坐标维);第三个就是具体的配置。

```
status_t SurfaceTexture::updateTexImage(BufferRejecter* rejecter) {…
    Mutex::Autolocklock(mMutex);
    status_t err = NO_ERROR;
```

```
…
EGLDisplay dpy =eglGetCurrentDisplay();//当前Display
EGLContext ctx =eglGetCurrentContext();//当前环境Context
…
BufferQueue::BufferItemitem;
err = mBufferQueue->acquireBuffer(&item);//acquire当前需要处理的buffer,封装在item中
if (err == NO_ERROR) {//acquire成功
    int buf = item.mBuf; //此Buffer在Slot中的序号
if(item.mGraphicBuffer != NULL) {…
      mEGLSlots[buf].mGraphicBuffer = item.mGraphicBuffer;//后面会用到
    }
    …
    EGLImageKHR image =mEGLSlots[buf].mEglImage;
    if (image ==EGL_NO_IMAGE_KHR) {/*假如image不为空的话，前面会先把它destroy掉，代码
省略*/
        if(mEGLSlots[buf].mGraphicBuffer == NULL) {
            …//异常处理
        } else {
            image = createImage(dpy, mEGLSlots[buf].mGraphicBuffer);//生成一个纹理图像
           mEGLSlots[buf].mEglImage = image;
            …
        }
    }

    if (err == NO_ERROR) {…
        glBindTexture(mTexTarget, mTexName);
        glEGLImageTargetTexture2DOES(mTexTarget, (GLeglImageOES)image);
        …
    }

    if (err != NO_ERROR) {//将acquired的buffer释放
```

```
            mBufferQueue->releaseBuffer(buf, dpy, mEGLSlots[buf].mFence);
            mEGLSlots[buf].mFence = EGL_NO_SYNC_KHR;
            return err;
        }
        …
        /*最后更新SurfaceTexture中各状态*/
        mCurrentTexture = buf;
        mCurrentTextureBuf =mEGLSlots[buf].mGraphicBuffer;
        …
    } else {…
    }
    return err;
}
```

这个函数比较长，我们只留下了最核心的部分。它的目标是更新Layer的纹理(Texture)，分为如下几步：

①Acquire一个需要处理的Buffer。根据前面对Buffer状态迁移的分析，当消费者想处理一块buffer时，它首先要向BufferQueue做acquire申请。那么BufferQueue怎么知道当前要处理哪一个Buffer呢？这是因为其内部维护有一个Fifo先 入先出队列。一旦有buffer被enqueue后，就会压入队尾;每次acquire就从队头取最前面的元素进行处理，完成之后就将其从队列移除

②Acquire到的buffer封装在BufferItem中，item.mBuf代表它在BufferSlot中的序号。正常情况下item.mGraphicBuffer都不为空，我们将它记录到mEGLSlots[buf].mGraphicBuffer中，以便后续操作。假如mEGLSlots[buf].mEglImage当前不为空(EGL_NO_IMAGE_KHR)的话，需要先把旧的image销毁(eglDestroyImageKHR)

③SurfaceFlinger有权决定SurfaceTexture得到的Buffer是否有效合法(比如说size是否正确)，这是通过updateTexImage(BufferRejecter* rejecter)中的rejecter对象来完成的。BufferRejecter实现了一个reject接口，用于SurfaceTexture调用 来验证前面acquire到的buffer是否符合要求

④接下来就生成Texture了。需要分别调用glBindTexture和 glEGLImageTargetTexture2DOES。后一个函数是opengl es中对glTexImage2D的扩展，因为在嵌入式环境下如果直接采用这个API的话，当图片很大时会严重影响到速度，而经过扩展后的 glEGLImageTargetTexture2DOES可以解决这个问题。这个接口是和eglCreateImageKHR配套使用的,封装在前面 createImage中。不了解这些函数用法的读者请务必参考opengles技术文档

⑤消费者一旦处理完Buffer后，就可以将其release了。此后这个buffer就又恢复FREE状态，以供生产者再次dequeue使用

⑥最后，我们需要更新SurfaceTexture中的各成员变量，包括mCurrentTexture、mCurrentTextureBuf、mCurrentCrop等等

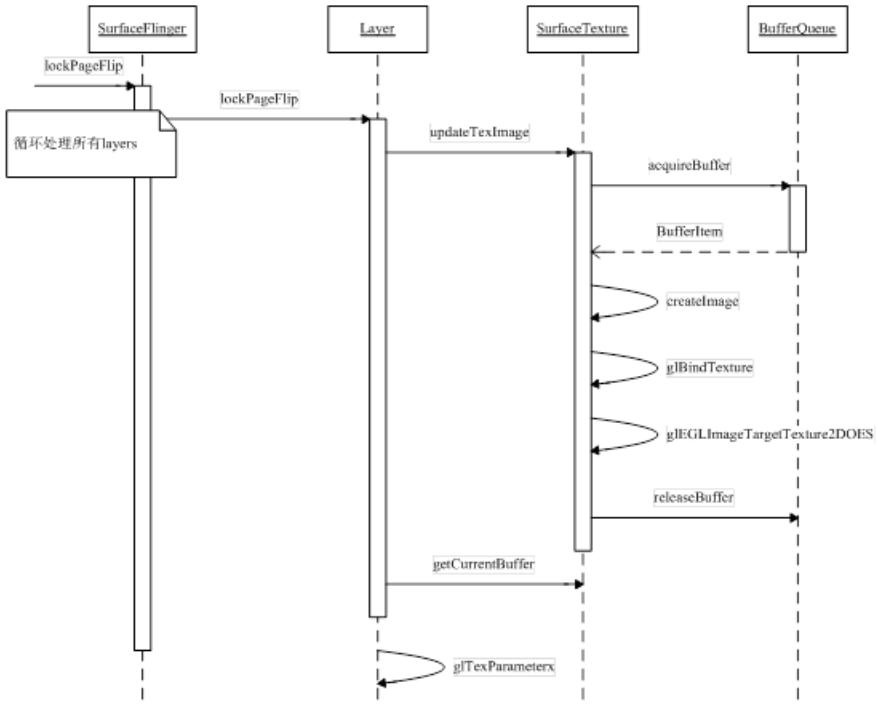

图 11　38 lockPageFlip流程图

Step2@SurfaceFlinger::handlePageFlip，计算所有layer的可见区域。在分析源码前，我们自己先来想一下，图层中什么样的区域是可见的呢？

l Z-order

各layer的z-order无疑是第一考虑的要素。因为排在越前面的图层，其获得曝光的机率越大，可见的区域也可能越大，如下图所示：

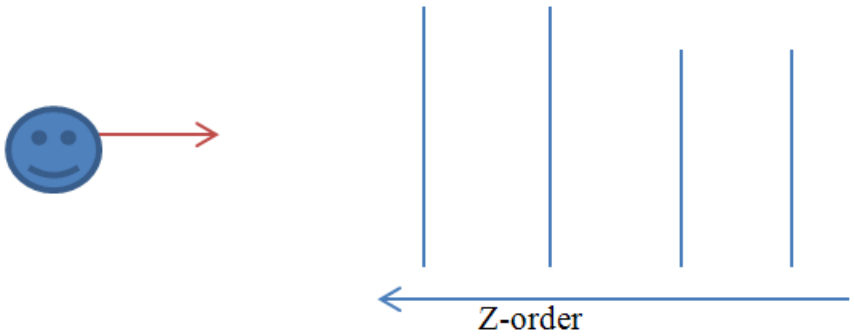

图 11　39 后面的图层有可能被遮挡而不可见

所以在计算可见性时，是按照Z-order由上而下进行的。假如一个layer的某个区域被确定为可见，那么与之相对应的它下面的所有图层区域都会被遮盖而不可见

l 透明度

虽然越前面的layer优先级越高，但这并不代表后面的图层完全没有机会。只要前一个layer不是完全不透明的，那么从理论上来讲用户就应该能“透过”这部分区域看到后面的内容

l 图层大小

与透明度一样，图层大小也直接影响到其可见区域。因为每个layer都是有大有小的，即便前一个layer是完全不透明的，但只要它的尺寸没有达到“满屏”，那么比它z-order小的图层还是有机会暴露出来的。这也是我们需要考虑的因素之一

综合上面的这几点分析，我们能大概制定出计算layer可见区域的逻辑步骤：

Ø 按照Z-order逐个计算各layer的可见区域，结果记录在LayerBase::visibleRegionScreen中

Ø 对于Z-order值最大的layer，显然没有其它图层会遮盖它。所以它的可见区域(visibleRegion)应该是(当然，前提是这个layer 没有超过屏幕区域)自身的大小再减去完全透明的部分(transparentRegionScreen)，由此计算出来的结果我们把它称为 aboveCoveredLayers。这个变量应该是全局的，因为它需要被传递到后面的layers中，然后不断地累积运算，直到覆盖整个屏幕区域

图 11　40 Z-order最大的layer可见区域示意图

如上图所示，外围加深部分是这个图层的尺寸大小，中间挖空区域则是完全透明的，因而需要被剔除。半透明区域比较特殊，它既属于上一个图层的可见区域，又不被列为遮盖区域

Ø 对于Z-order不是最大的layer，它首先要计算自身所占区域扣除aboveCoveredLayers后所剩的空间。然后才能像上一步一样再去掉完全透明的区域，这样得到的结果就是它最终的可见区域

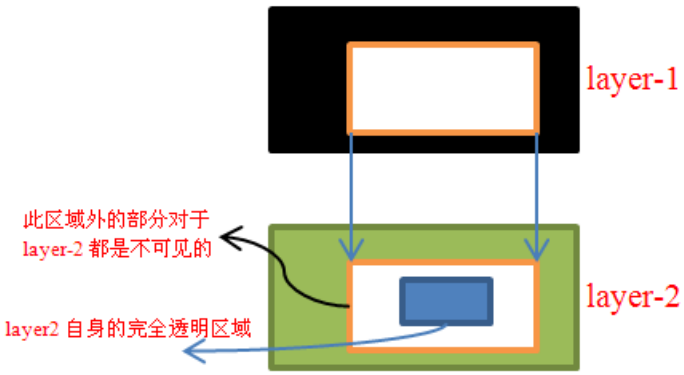

图 11　41 其它layer的可见区域

接下来看源码实现：

```
void SurfaceFlinger::computeVisibleRegions(const LayerVector&currentLayers,
                                           Region& dirtyRegion, Region& opaqueRegion)
{...
   const GraphicPlane&plane(graphicPlane(0));
   const Transform&planeTransform(plane.transform());
   const DisplayHardware&hw(plane.displayHardware());
   const RegionscreenRegion(hw.bounds());//整个屏幕区域

   Region aboveOpaqueLayers;
   Region aboveCoveredLayers;//全局的，用于描述当前已经被覆盖的区域
   Region dirty;

   bool secureFrameBuffer =false;
   size_t i =currentLayers.size();//所有layer数量
   while (i--) {/Step1. 注意计算的方向，按照Z-order由大到小的顺序
      constsp<LayerBase>& layer = currentLayers[i];//取得这一layer
     layer->validateVisibility(planeTransform);//验证当前layer的可见性，大家可以自行分析
      constLayer::State& s(layer->drawingState());//layer的状态
      Region opaqueRegion;//完全不透明的区域
      Region visibleRegion;//可见区域
```

```
Region coveredRegion;//被遮盖的区域。以上三个变量都是局部的，用于描述各个layer
if(CC_LIKELY(!(s.flags & ISurfaceComposer::eLayerHidden) && s.alpha)){
    /*Step2. 这部分计算可见区域visibleRegion和完全不透明区域opaqueRegion*/
    const booltranslucent = !layer->isOpaque(); //layer可见，又不是完全不透明，那就是半透明
    const Rectbounds(layer->visibleBounds());
   visibleRegion.set(bounds);//可见区域的初始值
   visibleRegion.andSelf(screenRegion);//和屏幕大小先进行与运算
    if(!visibleRegion.isEmpty()) {//如果经过上述运算，可见区域还不为空的 话
        if(translucent) { //将完全透明区域从可见区域中移除
           visibleRegion.subtractSelf(layer->transparentRegionScreen);
        }
        const int32_tlayerOrientation = layer->getOrientation();//旋转角度
        if(s.alpha==255 && !translucent &&
              ((layerOrientation & Transform::ROT_INVALID) == false)) {
           // theopaque region is the layer's footprint
           opaqueRegion = visibleRegion;//完全一致
        }
    }
}

/*Step3. 考虑被上层layer覆盖区域*/
coveredRegion = aboveCoveredLayers.intersect(visibleRegion);//计算会被遮盖的区域
aboveCoveredLayers.orSelf(visibleRegion);//累加计算，然后传给后面的layer
visibleRegion.subtractSelf(aboveOpaqueLayers);//扣除被遮盖的区域

/*Step4. 计算“脏”区域*/
if(layer->contentDirty) {//当前layer有脏内容
    dirty =visibleRegion;//不光要包括本次的可见区域
   dirty.orSelf(layer->visibleRegionScreen);//还应考虑上一次没处理的可见区域
   layer->contentDirty = false;//处理完成
} else {//当前layer没有脏内容
```

```
            const RegionnewExposed = visibleRegion - coveredRegion;
            const RegionoldVisibleRegion = layer->visibleRegionScreen;
            const RegionoldCoveredRegion = layer->coveredRegionScreen;
            const RegionoldExposed = oldVisibleRegion - oldCoveredRegion;
            dirty =(visibleRegion&oldCoveredRegion) | (newExposed-oldExposed);
        }
         /*Step5. 更新各变量*/
        dirty.subtractSelf(aboveOpaqueLayers);//扣除不透明区域
       dirtyRegion.orSelf(dirty);//累积计算脏区域
       aboveOpaqueLayers.orSelf(opaqueRegion);//累积不透明区域
       layer->setVisibleRegion(visibleRegion);//设置layer内部的可见区域
       layer->setCoveredRegion(coveredRegion);//设置layer内部的被遮盖区域

        /*Step6. 屏幕截图保护*/
        if(layer->isSecure() && !visibleRegion.isEmpty()) {
            secureFrameBuffer= true;
        }
    }//while循环结束

/*Step7. 考虑被移除layer占据的区域*/
dirtyRegion.orSelf(mDirtyRegionRemovedLayer);
    mDirtyRegionRemovedLayer.clear();
    mSecureFrameBuffer =secureFrameBuffer;
    opaqueRegion =aboveOpaqueLayers;
}
```

Step1@ SurfaceFlinger::computeVisibleRegions，循环处理每个layer。注意计算方向(i--)采取Z-order由大 而小的顺序，这和我们之前的分析是一致的。这个函数中定义了很多变量，包括对各layer全局的aboveOpaqueLayers、 aboveCoveredLayers、dirty，以及局部的opaqueRegion、visibleRegion、coveredRegion，但 基本逻辑还是和我们前面的推测相符合的。

Step2@ SurfaceFlinger::computeVisibleRegions，计算visibleRegion和opaqueRegion。不过有两个 情况下不需要这么做。其一就是当前layer被隐藏了，这可以从s.flags & ISurfaceComposer::eLayerHidden中获得;其二就是它的alpha值为0，也就是说当前layer是完全透明的。

如果不是这两种情况的话，就可以继续计算visibleRegion。它的初始值是layer->visibleBounds()，相当于前 面我们说的该layer所占据的区域。这一初始值与屏幕区域进行与运算，排除屏幕显示区域外的部分。此时如果visibleRegion还有剩余空间的 话，就继续减掉透明区域。

变量opaqueRegion在某些情况下和visibleRegion是一样的。即当前图层完全不透明(alpha==255&&!translucent)且旋转角度值为ROT_INVALID时，两者可认为是同一个区域。

Step3@ SurfaceFlinger::computeVisibleRegions，考虑被上层layer遮盖的情况。其中coveredRegion计算得 到会被遮盖的区域，即visibleRegion需要剔除的部分。并且累加更新全局的aboveCoveredLayers，以便后面的layer可以继 续使用。

Step4@ SurfaceFlinger::computeVisibleRegions，在前面的基础上进一步计算dirty区域，也就是需要渲染的部分。涉及到 两个变量，dirty是一个全部变量，它表示每个layer中的脏区域;而dirtyRegion是函数的出参，属于全局性的，用于计算所有layer脏 区域的集合(采用“或”运算)。可能有人会问，需要渲染的区域不就是可见区域吗？这可不一定。比如说当前界面内容没有任何改变，那么为什么还要浪费时间再 重新渲染一次呢？

如果有“脏”内容(layer->contentDirty为true)，dirty要同时覆盖当前可见区域(visibleRegion)，以及上次还未考虑的可见区域(layer->visibleRegionScreen)。

如果没有“脏”内容，那么我们只要渲染这次新“暴露”出来的区域就可以了。因为上一次暴露出来的区域已经被渲染过了，而且内部又没有变化，当然不需要再次渲染。

Step5@ SurfaceFlinger::computeVisibleRegions，更新所有相关变量，包括layer内部的 visibleRegionScreen(通过setVisibleRegion)、coveredRegionScreen(通过 setCoveredRegion)，以及需要累积计算的aboveOpaqueLayers，和dirtyRegion等等。

Step6@ SurfaceFlinger::computeVisibleRegions，判断当前窗口内容是否受保护(变量 secureFrameBuffer)。判断的标准就是当前可见区域不为空(!visibleRegion.isEmpty()),且需要安全保护 (isSecure()),这将影响到截屏功能是否可以正常执行。

Step7@ SurfaceFlinger::computeVisibleRegions，除了上述的步骤外，我们还要考虑那些被移除的layer所占据的区域，这 样得到的才是最终的结果。函数出参opaqueRegion也就是通过各layer计算后的累加值aboveOpaqueLayers。

分析完可见区域的计算后，我们回到本小节的handlePageFlip函数中。

Step3@SurfaceFlinger::handlePageFlip. 通过前面的computeVisibleRegions，现在所有layer的可见区域都已经记录在其内部的visibleRegionScreen中 了。接下来 mVisibleLayersSortedByZ用于对所有可见的图层进行z-order排序。

Step4@SurfaceFlinger::handlePageFlip. 在computeVisibleRegions中，opaqueRegion做为所有layer的遮挡区域(对应aboveOpaqueLayers)的 累积结果，正常情况下应该是和屏幕大小是一致的。反之，如果它们不一样的话，就产生了wormhole区域：

mWormholeRegion = screenRegion.subtract(opaqueRegion);

这部分区域比较特殊，因而在后续的图层合成时需要做额外处理(具体调用drawWormhole来实现)。

Step5@SurfaceFlinger::handlePageFlip，调用unlockPageFlip做最后的准备工作。实现上和 lockPageFlip有相似之处，也是通过循环调用各layer自己的unlockPageFlip来完成。因为图层“脏”区域的坐标空间和屏幕坐标 不是同一个，在这里要对其做下坐标变换。

1.1.1 handleRefresh

从名称上看，这个函数处理“刷新”。经过比对新老版本的差异，我们觉得这个地方应该是老版本遗留下来的“废料”。换句话说，它对MessageQueue::REFRESH事件的处理没有起到作用。

```
void SurfaceFlinger::handleRefresh()
{
    bool needInvalidate =false;
    const LayerVector&currentLayers(mDrawingState.layersSortedByZ);
    const size_t count =currentLayers.size();
    for (size_t i=0 ;i<count ; i++) {
        constsp<LayerBase>& layer(currentLayers[i]);
        if (layer->onPreComposition()) {
            needInvalidate =true;
        }
    }
    if (needInvalidate) {
        signalLayerUpdate();
    }
}
```

这个函数很简单，首先调用每个layer的onPreComposition，判断是否需要needInvalidate，条件就是Layer::mQueuedFrames>0。

在函数的最后，它调用signalLayerUpdate：

```
void SurfaceFlinger::signalLayerUpdate() {
  mEventQueue.invalidate();
}
```

可见它只是直接使用了MessageQueue::invalidate()：

```
void MessageQueue::invalidate() {
    mEvents->requestNextVsync();
}
```

最后的requestNextVsync用于安排下一次的VSYNC事件，这个函数只在vsync rate为0时才有效。也就是说只有当我们关闭了VSYNC的主动上报后才有作用，这和之前的分析是不符合的。因而我们可以忽略 handleRefresh，

相信在下一个版本中它会得到清理。

1.1.1 handleRepaint

经过handleTransaction和handlePageFlip等步骤的准备工作后，现在可以合成各图层数据了。

```
void SurfaceFlinger::handleRepaint()
{...
mSwapRegion.orSelf(mDirtyRegion);
   const DisplayHardware&hw(graphicPlane(0).displayHardware());
   …
   uint32_t flags =hw.getFlags();//系统支持的渲染方式
   if (flags &DisplayHardware::SWAP_RECTANGLE) {
      mDirtyRegion.set(mSwapRegion.bounds());
   } else {
       if (flags & DisplayHardware::PARTIAL_UPDATES) {
          mDirtyRegion.set(mSwapRegion.bounds());
       } else {
          mDirtyRegion.set(hw.bounds());
           mSwapRegion =mDirtyRegion;
       }
   }
   setupHardwareComposer();
   composeSurfaces(mDirtyRegion);//合成各图层数据
  mSwapRegion.orSelf(mDirtyRegion);
   mDirtyRegion.clear();
}
```

系统支持多种类型的渲染方式，可以从DisplayHardware获取到：

l SWAP_RECTANGLE

我们只需要渲染“脏”区域，或者说系统在软件层面上支持部分区域更新，但更新区域必须是长方形规则的。由于这个限制，mDirtyRegion应该是覆盖所有“脏”区域的最小矩形

l PARTIAL_UPDATES

系统支持硬件层面部分区域更新，同样也需要是矩形区域

l 其它

其它情况下，我们需要重绘整个屏幕，即hw.bounds()

接着setupHardwareComposer()对所有mVisibleLayersSortedByZ中的layers进行数据初始化，比如 合成类型(compositionType)。包括HWC_FRAMEBUFFER和HWC_OVERLAY两种。如果是前者的话，那么图层合成将由 SurfaceFlinger通过OpenGL ES来完成;否则就由一种叫overlay的硬件来执行。Overlay通常被用于处理视频回放和录制时的预览，这种硬件合成可以加快处理，减小CPU的 负担。不过从Android 4.0开始，源码中关于Overlay部分的实现被移除了。

HWComposer内部维持着一个各layer属性的list:

```
hwc_layer_list_t*      mList;
```

这个结构体定义如下:

```
typedef struct hwc_layer_list {
    uint32_t flags;//各标志
    size_tnumHwLayers;//图层数量
    hwc_layer_t  hwLayers[0];//各图层属性详细描述
} hwc_layer_list_t;
```

当SurfaceFlinger进行handleWorkList时，它根据当前可见图层的数量调用HWComposer::createWorkList(size_tnumLayers)来创建这一mList。SurfaceFlinger::setupHardwareComposer()中，首先将mList里所有的hwc_layer对象的 compositionType都设置为HWC_FRAMEBUFFER，然后调用HWComposer::prepare。这个函数会通过hwc设备 (即前面章节中讲过的HWC_HARDWARE_MODULE_ID)提供的prepare接口来为每个layer判断HWC支持的合成类型，大家可以自己挑一个具体的硬件平台来分析prepare细节。另外，它还统计使用不同合成类型的layer的数量，分别由mNumOVLayers (HWC_OVERLAY)和mNumFBLayers (HWC_FRAMEBUFFER)来记录。

图层合成最关键的一步就是composeSurfaces()，我们来详细分析它的实现:

```
void SurfaceFlinger::composeSurfaces(const Region& dirty)
{
    const DisplayHardware&hw(graphicPlane(0).displayHardware());
    HWComposer&hwc(hw.getHwComposer());//注意hw,hwc和graphicPlane的关系
    hwc_layer_t* constcur(hwc.getLayers());//前面讲到的mList
```

```
    const size_t fbLayerCount= hwc.getLayerCount(HWC_FRAMEBUFFER);// mNumFBLayers
    if (!cur || fbLayerCount){// HWC_FRAMEBUFFER类型数量必须要大于0才处理
        if(hwc.getLayerCount(HWC_OVERLAY)) {// HWC_OVERLAY数量大于0的情况
            //我们不考虑这种情况
        } else {
            if(!mWormholeRegion.isEmpty()) {
                drawWormhole();//特殊区域需要特别处理
            }
        }
        /*以下开始真正的渲染工作*/
        const Vector<sp<LayerBase> >& layers(mVisibleLayersSortedByZ);//所有可见图层
        const size_t count =layers.size();
        for (size_t i=0 ;i<count ; i++) {
            constsp<LayerBase>& layer(layers[i]);
            const Regionclip(dirty.intersect(layer->visibleRegionScreen));//判断各个layer是否需要渲染
            if(!clip.isEmpty()) {
                …
                layer->draw(clip);//调用各layer进行绘图
            }
        }
    }
}
```

在SurfaceFlinger中，显示屏用GraphicPlane表示。它内部的成员数组变量mGraphicPlanes[1]用于存储支持 的屏幕，目前只实现了第0号Display。GraphicPlane内部包含了DisplayHardware，用于封装宽、高、格式等一系列帧缓冲区 信息;OpenGL的本地窗口FramebufferNativeWindow;EGL相关结构，如配置、Context、Surface;以及HWComposer，即mHwc成员变量。

HWComposer简单来说，它管理一个名称为HWC_HARDWARE_MODULE_ID的设备，比如硬件是否支持产生VSYNC信号就需要由它来判断。大家可以理一下这几个对象的关系。

得到hwc后，进一步获取worklist，也就是我们前面讲过的mList。这时列表中所有的layer属性都已经更新完毕，包括 compositionType。而且也分别统计出了HWC_FRAMEBUFFER和HWC_OVERLAY两种类型的layer数量。所以 fbLayerCount实际上就是mNumFBLayers。假如这个数小于0，就什么都不做。

这个函数中的“脏”区域是由handleRepaint传下来的，也就是mDirtyRegion。对于每个layer，我们都使用

dirty.intersect(layer->visibleRegionScreen)来判断它的可见区域与整个“脏”区域是否有交集。有的话 才需要进行渲染，否则就不用多此一举了。

最后，通过各Layer自身的draw()来执行具体的绘图工作。

LayerBase::draw()直接调用了onDraw()，这个接口在LayerBase中虚函数，由它的子类Layer来实现。

```
void Layer::onDraw(const Region& clip) const
{…
  if(CC_UNLIKELY(mActiveBuffer == 0)) {//当前还没有活跃的Buffer，有可能发生
      Region under;//此layer下面被遮盖的区域。如果没有的话，就把它“涂黑”
      const SurfaceFlinger::LayerVector&drawingLayers(mFlinger->mDrawingState.layersSortedByZ);
      const size_t count =drawingLayers.size();
      for (size_t i=0 ;i<count ; ++i) {
          constsp<LayerBase>& layer(drawingLayers[i]);
          if (layer.get() ==static_cast<LayerBase const*>(this))//layer自己
              break;
         under.orSelf(layer->visibleRegionScreen);
      }
      Regionholes(clip.subtract(under));//哪些区域不会被遮盖
      if (!holes.isEmpty()){
          clearWithOpenGL(holes, 0, 0, 0, 1);
      }
      return;
  }
  …/*配置opengl 参数，源码省略*/
  drawWithOpenGL(clip);
  …
}
```

变量mActiveBuffer在前面lockPageFlip中已经设置成最新的缓冲区了，但它有可能是空的。比如应用程序还没有开始做绘制工 作，在此之前都可能发生这种情况。按照目前版本的实现，先找出所有在这个layer之下的可见区域，即利用 mFlinger->mDrawingState.layersSortedByZ不断累加visibleRegionScreen;再由此计算出 不会被遮盖的区域：

Region holes(clip.subtract(under));

除holes之外的所有区域，都会被“涂黑”，这是由clearWithOpenGL(holes,0, 0, 0, 1)完成的。

接下来根据当前情况调用opengl的各API接口进行必要配置，为drawWithOpenGL做最后准备工作：

```
void LayerBase::drawWithOpenGL(const Region& clip) const
{
    const DisplayHardware&hw(graphicPlane(0).displayHardware());//这句代码我们见过几次了
    const uint32_t fbHeight =hw.getHeight();//framebuffer高度
    const State&s(drawingState());//即mDrawingState

    GLenum src = mPremultipliedAlpha ? GL_ONE : GL_SRC_ALPHA;
    if (CC_UNLIKELY(s.alpha< 0xFF)) {//非完全不透明
        const GLfloat alpha =s.alpha * (1.0f/255.0f);//归一化
        if(mPremultipliedAlpha) {//预乘alpha打开
            glColor4f(alpha,alpha, alpha, alpha);//alpha表达方式不同
        } else {
            glColor4f(1, 1, 1,alpha);
        }
        glEnable(GL_BLEND);//需要混合
        glBlendFunc(src, GL_ONE_MINUS_SRC_ALPHA);//制定混合算法
        glTexEnvx(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE, GL_MODULATE);
    } else {
        glColor4f(1, 1, 1, 1);//完全不透明，alpha为1
        glTexEnvx(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE, GL_REPLACE);
        if (!isOpaque()) {//不是opaque，需要blend
            glEnable(GL_BLEND);
            glBlendFunc(src, GL_ONE_MINUS_SRC_ALPHA);//同前面s.alpha < 0xFF情况一样
        } else {
            glDisable(GL_BLEND);//不需要blend
        }
    }
    /*顶点结构体*/
    struct TexCoords {
```

```
    GLfloat u;
    GLfloat v;
};

Rect crop(s.active.w,s.active.h);
if(!s.active.crop.isEmpty()) {
    crop = s.active.crop;
}
GLfloat left = GLfloat(crop.left)/ GLfloat(s.active.w);//左边缘
GLfloat top =GLfloat(crop.top) / GLfloat(s.active.h);//上边缘
GLfloat right =GLfloat(crop.right) / GLfloat(s.active.w);//右边缘
GLfloat bottom =GLfloat(crop.bottom) / GLfloat(s.active.h);//下边缘

TexCoords texCoords[4];//定义四个点
texCoords[0].u = left;/*texCoords[4]，分别表示*/
texCoords[0].v = top;/*crop框的左上、左下、右下、和右上的4个顶点*/
texCoords[1].u = left;
texCoords[1].v = bottom;
texCoords[2].u = right;
texCoords[2].v = bottom;
texCoords[3].u = right;
texCoords[3].v = top;
for (int i = 0; i < 4;i++) {
    texCoords[i].v = 1.0f- texCoords[i].v;
}

glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glVertexPointer(2,GL_FLOAT, 0, mVertices);
glTexCoordPointer(2,GL_FLOAT, 0, texCoords);
glDrawArrays(GL_TRIANGLE_FAN,0, mNumVertices);
glDisableClientState(GL_TEXTURE_COORD_ARRAY);
```

```
    glDisable(GL_BLEND);
}
```

对DisplayHardware的获取，我们已经讲过了，不再赘述。变量mPremultipliedAlpha表示“预乘alpha通道”，它是RGBA的另一种表达方式。比如说传统的RGBA就是(r,g,b,a)，而premultiplied alpha就是(ra,ga,ba,a)。在某些场景下采用后一种方式能达到更好的效果，大家可以自行查阅相关资料了解详情。

因而当mPremultipliedAlpha为true时(这个变量的初始值就是true，当LayerBase:: initStates时，再根据eNonPremultiplied标志来判断是否启用)设置颜色：

glColor4f(alpha, alpha, alpha, alpha);

否则就是：

glColor4f(1, 1, 1, alpha);

假如当前不是完全不透明的，或者isOpaque()返回为false，那么需要开启BLEND功能。OpenglES中对应的API是 glEnable(GLenum cap)，参数为GL_BLEND。BLEND的目的就是通过源色和目标色的混合计算来产生透明特效，有多种混合算法可供选择，由glBlendFunc 来配置。

glBlendFunc(GLenum sfactor, GLenum dfactor);

第一个参数就是源因子，后一个为目标因子。

可选值如下表所示：

表格 18glBlendFunc可选因子

VALUE	Description
GL_ZERO	使用0.0作为混合因子
GL_ONE	使用1.0作为混合因子
GL_SRC_COLOR	根据源颜色的各分量计算出混合因子
GL_ONE_MINUS_SRC_COLOR	根据(1-源颜色各分量)计算出混合因子
GL_SRC_ALPHA	根据源颜色的alpha计算出混合因子
GL_ONE_MINUS_SRC_ALPHA	根据(1.0-源颜色alpha)计算出混合因子
GL_DST_ALPHA	根据目标颜色的alpha计算出混合因子
GL_ONE_MINUS_DST_ALPHA	根据(1.0-目标颜色alpha)计算出混合因子
GL_DST_COLOR	根据目标颜色的各分量计算出混合因子

GL_ONE_MINUS_DST_COLOR	根据(1-目标颜色各分量)计算出混合因子

每种情况下混合因子的具体计算方法，以及blend的计算公式，可以参考官方文档描述。这里我们只要明白它的使用方法就行了。

接下来计算纹理区域的各坐标点，结果以texCoords[4]数组来表示。这样才能保证绘图结果体现在正确的区域中。一切准备就绪，最后就可以调用openglapi进行绘制了：

其中glEnableClientState(GL_TEXTURE_COORD_ARRAY)说明要处理的是纹理坐标数 组;glVertexPointer(2, GL_FLOAT, 0, mVertices)指明是二维坐标，float数据类型，紧凑方式排列，顶点数组描述是mVertices(这个数组值在 validateVisibility中计算，个数为4); glTexCoordPointer(2,GL_FLOAT, 0, texCoords)也是二维坐标系，float数据类型，紧凑排列，纹理的顶点数组由前面计算得到的texCoords表示。

最后，glDrawArrays(GL_TRIANGLE_FAN,0, mNumVertices)来绘制三角形。OpenGLES取消了对QUAD的支持，有GL_TRIANGLE_FAN与GL_TRIANGLE_STRIP两种三角形绘制方式。

GL_TRIANGLE_FAN：假设有N个顶点。那么第n个三角形的顶点就是(1,n+1,n+2),总共有N-2个三角形。

GL_TRIANGLE_STRIP：假设有N个顶点,也是有N-2个三角形

Ø N为奇数(odd)。那么第n个三角形的顶点是(n,n+1,n+2)

Ø N为偶数(even)。那么第n个三解形的顶点就是(n+1,n,n+2)

以这个场景为例，因为mVertices[0]-mVertices[3]分别表示左上、左下、右下、右上四个顶点(可以参见validateVisibility中的实现)，那么采用这两种模式的结果如下：

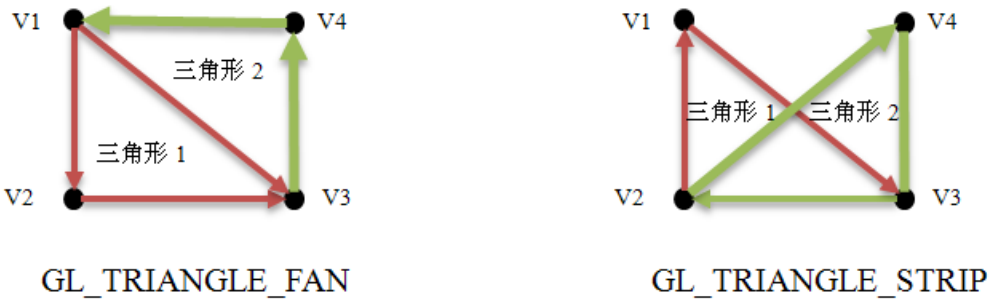

图 11　42 三角形绘制的两种模式

1.1.1 postFramebuffer

在多缓冲区机制中，只有把显示数据写入framebuffer才能真正在物理屏幕上显示。前面几个小节的输出都是backbuffers，我们还需要最后一步——postFramebuffer。

```
void SurfaceFlinger::postFramebuffer()

{…
```

```
    const DisplayHardware&hw(graphicPlane(0).displayHardware());
    …
    hw.flip(mSwapRegion);//交换前后台buffer

    size_t numLayers =mVisibleLayersSortedByZ.size();
    for (size_t i = 0; i <numLayers; i++) {
        mVisibleLayersSortedByZ[i]->onLayerDisplayed();
    }
    …
}
```

先从opengl本地窗口的角度来想一下：

Ø queueBuffer

一旦“生产者”完成生产后，它需要把当前的buffer重新入队，以使“消费者”可以做接下来的处理

Ø dequeueBuffer

为了“生产者”可以继续下一轮的工作，它会重新deque

基本的思路就是这样子，不过Android系统将一些步骤封装到了DisplayHardware中，我们稍后会看到。

DisplayHardware::flip完成后，分别通知各可见Layer它们的内容已经显示出来了。

```
void DisplayHardware::flip(const Region& dirty) const
{…
    mPageFlipCount++;//flip计数

    if (mHwc->initCheck()== NO_ERROR) {
        mHwc->commit();
    } else {
        eglSwapBuffers(dpy, surface);
    }
    …
}
```

分为两条路径：

(1)commit

成员变量mHwc是在DisplayHardware::init中生成的一个HWComposer对象。只要HWC_HARDWARE_MODULE_ID模块可以正常加载，且hwc_open能打开hwc_composer_device设备，那么initCheck()就返回NO_ERROR，否则就是NO_INIT。

此时我们通过HWComposer::commit来执行flip，这个函数直接调用如下硬件接口：

mHwc->set(mHwc,mDpy, mSur, mList);

set()和后面的eglSwapBuffers是基本等价的，原型如下：

```
int (*set)(struct hwc_composer_device *dev,hwc_display_t dpy,
        hwc_surface_t sur,hwc_layer_list_t* list);
```

其中最后一个list必须与最近一次的prepare()所用列表完全一致。假如list为空或者列表数量为0的话，说明SurfaceFlinger已经利用OpenGL ES做了composition，此时set就和eglSwapBuffers一样。当list不为空，且layer的compositionType == HWC_OVERLAY，那么HWComposer需要进行硬件合成。

如果成功执行的话，set返回0，否则就是HWC_EGL_ERROR。可以通过eglGetError()来获得具体的error。感兴趣的读者可以自己挑选一个具体的平台实现来分析，比如三星的crespo(路径是/device/Samsung/crespo/libhwcomposer)。

(2)eglSwapBuffers

以libagl为例，这个函数又调用了如下的swapBuffers：

```
/*frameworks/native/opengl/libagl/Egl.cpp*/
EGLBoolean egl_window_surface_v2_t::swapBuffers()
{...
    nativeWindow->queueBuffer(nativeWindow, buffer);
    …
    // dequeue一个新的buffer
    if (nativeWindow->dequeueBuffer(nativeWindow, &buffer) == NO_ERROR) {
        …
} else {
        returnsetError(EGL_BAD_CURRENT_SURFACE, EGL_FALSE);
    }

    return EGL_TRUE;
}
```

这和我们一开始的推测是一致的——通过queueBuffer来入队，然后通过dequeueBuffer重新申请一个buffer以用于下一轮的刷新。关于SurfaceFlinger中所使用的这一OpenGL本地窗口，即FramebufferNativeWindow的缓冲区

管理，我们在前几个小节已经分析过了，大家可以结合这里的场景再看一遍。

分享到：

上一篇：SurfaceFlinger Layer Clip and Draw---大密度注释

下一篇：android performance trace使能cpufreq_gov

查看评论

暂无评论

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

公司简介 | 招贤纳士 | 广告服务 | 银行汇款帐号 | 联系方式 | 版权声明 | 法律顾问 | 问题报告

QQ客服　微博客服　论坛反馈　联系邮箱：webmaster@csdn.net　服务热线：400-600-2320

京 ICP 证 070598 号

北京创新乐知信息技术有限公司 版权所有

世纪乐知(北京)网络技术有限公司 提供技术支持