Android8.1 Framework--SurfaceFlinger

Android 启动前，会在本地新启动一个服务，即native/services/surfaceflinger服务，这个目录包含的内容包括如下：

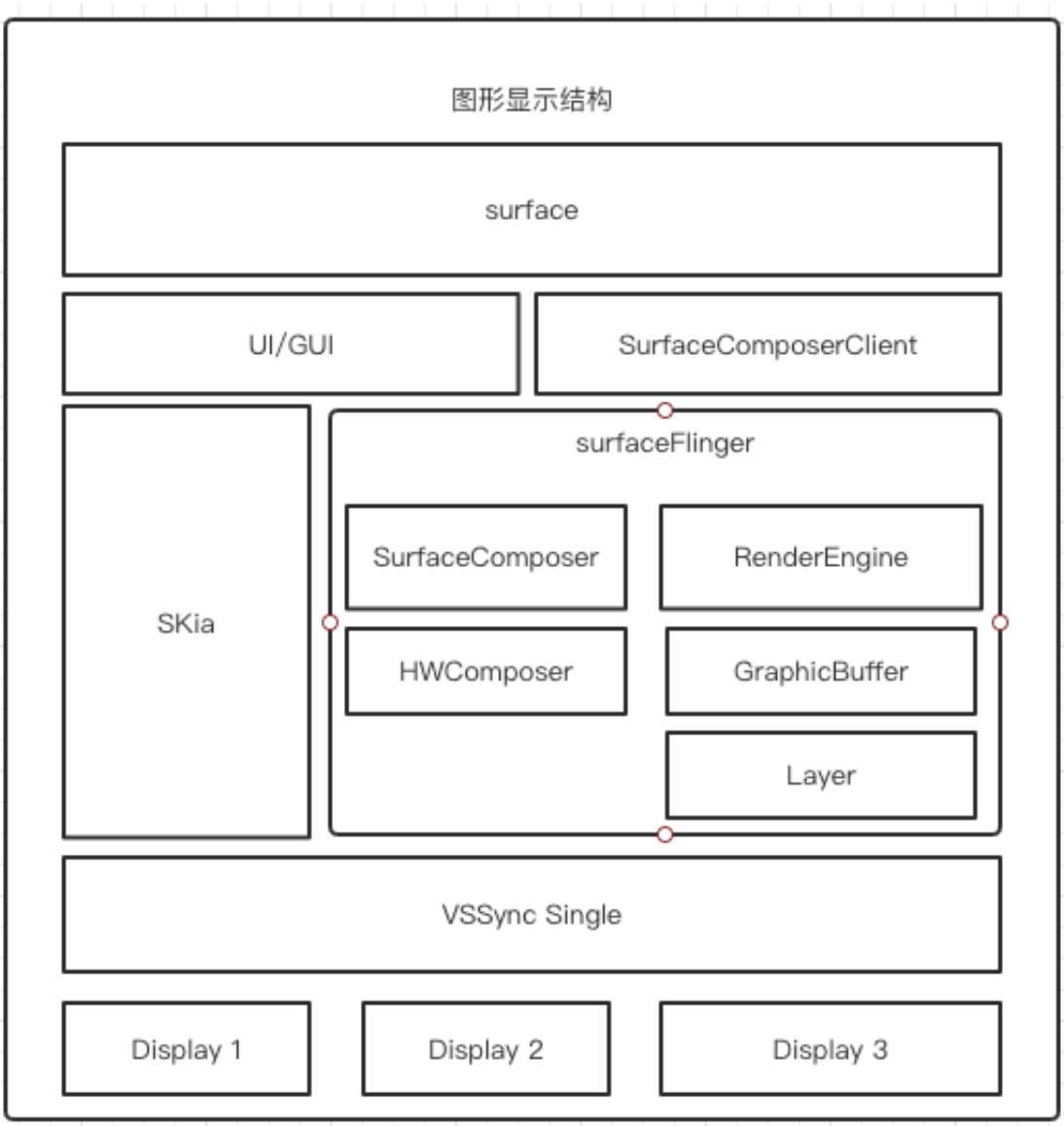
- DisplayHardware
- Effectss
- EventLog
- RenderEngine
- tests
- Other

这几个子模块组成。首先看到有一个main_surfaceflinger.cpp的文件，服务从main函数进入，可以看到主要做了有以下几项工作：

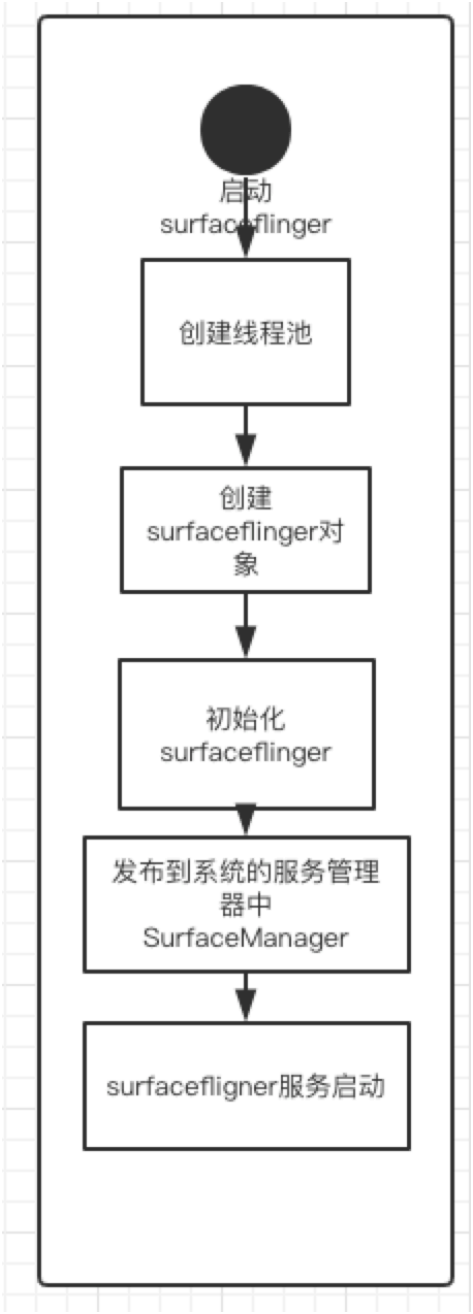
```
#include <sys/resource.h>
#include <cutils/sched_policy.h>
#include <binder/IServiceManager.h>
#include <binder/IPCThreadState.h>
#include <binder/ProcessState.h>
#include <binder/IServiceManager.h>
#include "SurfaceFlinger.h"

using namespace android;

int main(int, char**) {
    signal(SIGPIPE, SIG_IGN);
    // When SF is launched in its own process, limit the number of
    // binder threads to 4.
    ProcessState::self()->setThreadPoolMaxThreadCount(4);

    // start the thread pool
    sp<ProcessState> ps(ProcessState::self());
    ps->startThreadPool();

    // instantiate surfaceflinger
    sp<SurfaceFlinger> flinger = new SurfaceFlinger();

    setpriority(PRIO_PROCESS, 0, PRIORITY_URGENT_DISPLAY);

    set_sched_policy(0, SP_FOREGROUND);

    // initialize before clients can connect
    flinger->init();

    // publish surface flinger
    sp<IServiceManager> sm(defaultServiceManager());
```

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

    // run in this thread
    flinger->run();

    return 0;
}
```

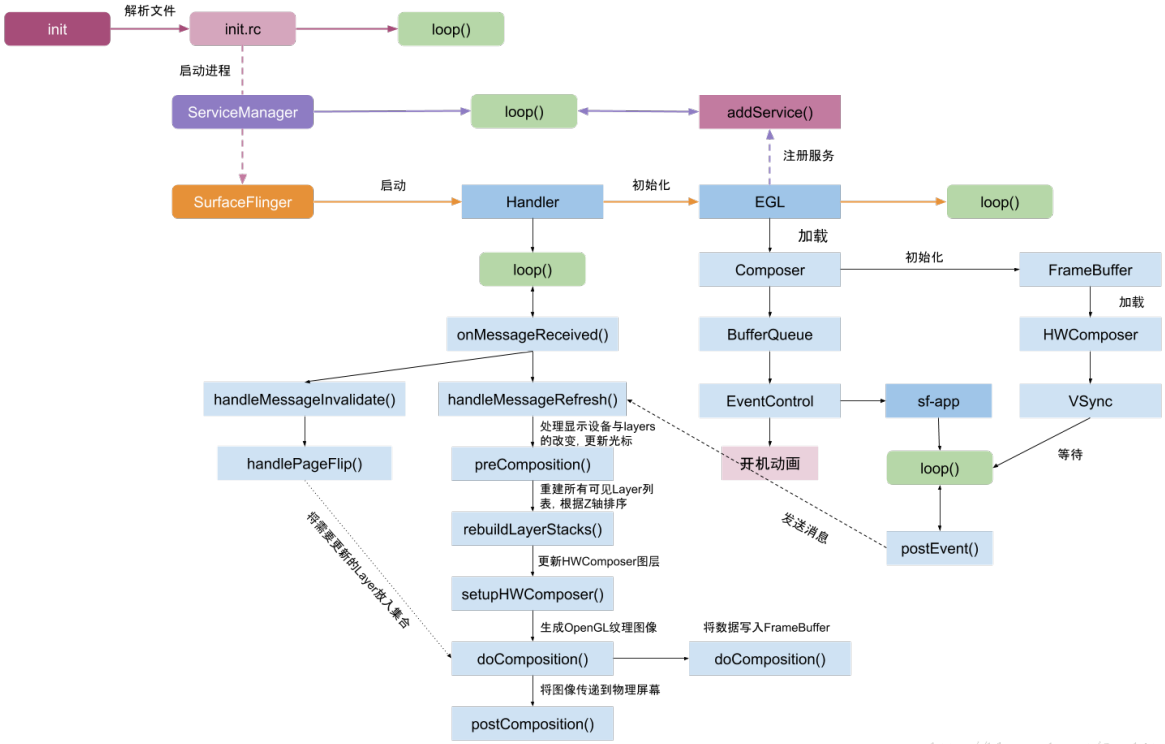
大概的显示结构如下：



从代码上看，首先surfaceFlinger先创建了一个线程池，线程池的大小为4。并且创建了一个surfaceFlinger对象，对系统配置进行了初始化值写入操作，再初始化好surfaceflinger，待初始化完成后，将创建的surfaceflinger发布到系统serviceManager中去，最后，启动flinger线程工作。具体的启动流程可以用以下一个简单的图表表示：



参考网上的流程，大致看了一下surfaceFlinger的工作流程：



实际在SurfaceFlinger打印的输出：

```
2018-12-24 16:53:22.498 1388-1388/? W/SurfaceFlinger:
SurfaceFlinger::setCompositorTimingSnapped
2018-12-24 16:53:22.499 1388-1412/? W/SurfaceFlinger:
SurfaceFlinger::setTransactionState(
2018-12-24 16:53:22.499 1388-1412/? W/SurfaceFlinger:
SurfaceFlinger::setTransactionFlags
2018-12-24 16:53:22.499 1388-1412/? W/SurfaceFlinger:
SurfaceFlinger::setTransactionState(
2018-12-24 16:53:22.499 1388-1412/?
W/SurfaceFlinger:  SurfaceFlinger::setClientStateLocked(
2018-12-24 16:53:22.499 1388-1412/? W/SurfaceFlinger:
SurfaceFlinger::onLayerRemoved
2018-12-24 16:53:22.499 1388-1412/? W/SurfaceFlinger:
SurfaceFlinger::removeLayer
2018-12-24 16:53:22.499 1388-1412/? W/SurfaceFlinger:
SurfaceFlinger::setTransactionFlags
2018-12-24 16:53:22.502 1388-1412/?
W/SurfaceFlinger:  SurfaceFlinger::onLayerDestroyed
2018-12-24 16:53:22.502 1388-1412/? W/SurfaceFlinger:
SurfaceFlinger::removeLayer
2018-12-24 16:53:22.516 1388-1388/? W/SurfaceFlinger:
SurfaceFlinger::invalidateLayerStack
2018-12-24 16:53:22.516 1388-1388/? W/SurfaceFlinger:
SurfaceFlinger::commitTransaction
2018-12-24 16:53:22.516 1388-1388/?
W/SurfaceFlinger:  SurfaceFlinger::recordBufferingStats(const char*
```

```
layerName,
2018-12-24 16:53:22.516 1388-1388/? W/SurfaceFlinger:
SurfaceFlinger::updateCursorAsync
2018-12-24 16:53:22.516 1388-1388/? W/SurfaceFlinger:
SurfaceFlinger::handlePageFlip
2018-12-24 16:53:22.516 1388-1388/? W/SurfaceFlinger:
SurfaceFlinger::preComposition
2018-12-24 16:53:22.516 1388-1388/? W/SurfaceFlinger:
SurfaceFlinger::rebuildLayerStacks
2018-12-24 16:53:22.516 1388-1388/? W/SurfaceFlinger:
SurfaceFlinger::computeVisibleRegions
2018-12-24 16:53:22.516 1388-1388/? W/SurfaceFlinger:
SurfaceFlinger::setUpHWComposer
2018-12-24 16:53:22.516 1388-1388/? W/SurfaceFlinger:
SurfaceFlinger::computeSaturationMatrix
2018-12-24 16:53:22.517 1388-1388/? W/SurfaceFlinger:
SurfaceFlinger::doDebugFlashRegions
2018-12-24 16:53:22.517 1388-1388/? W/SurfaceFlinger:
SurfaceFlinger::doComposition
2018-12-24 16:53:22.517 1388-1388/? W/SurfaceFlinger:
SurfaceFlinger::doDisplayComposition
2018-12-24 16:53:22.517 1388-1388/? W/SurfaceFlinger:
SurfaceFlinger::doComposeSurfaces
2018-12-24 16:53:22.525 1388-1388/? W/SurfaceFlinger:
SurfaceFlinger::postFramebuffer
2018-12-24 16:53:22.526 1388-1388/? W/SurfaceFlinger:
SurfaceFlinger::postComposition
2018-12-24 16:53:22.526 1388-1388/? W/SurfaceFlinger:
SurfaceFlinger::updateCompositorTiming
2018-12-24 16:53:22.526 1388-1388/? W/SurfaceFlinger:
SurfaceFlinger::setCompositorTimingSnapped
```

从上述可以看到具体的启动过程，接下去说下surfacefligner类主要的内容。 首先是surfaceflinger的构造方法来看。

```
SurfaceFlinger::SurfaceFlinger()
    :   BnSurfaceComposer(),
        mTransactionFlags(0),
        mTransactionPending(false),
        mAnimTransactionPending(false),
        mLayersRemoved(false),
        mRepaintEverything(0),
        mRenderEngine(NULL),
        mBootTime(systemTime()),
        mVisibleRegionsDirty(false),
        mHwWorkListDirty(false),
        mAnimCompositionPending(false),
        mDebugRegion(0),
        mDebugDDMS(0),
        mDebugDisableHWC(0),
```

```
        mDebugDisableTransformHint(0),
        mDebugInSwapBuffers(0),
        mLastSwapBufferTime(0),
        mDebugInTransaction(0),
        mLastTransactionTime(0),
        mBootFinished(false),
        mForceFullDamage(false),
        mPrimaryHWVsyncEnabled(false),
        mHWVsyncAvailable(false),
        mDaltonize(false),
        mHasColorMatrix(false),
        mHasPoweredOff(false),
        mFrameBuckets(),
        mTotalTime(0),
        mLastSwapTime(0)
{
    ALOGI("SurfaceFlinger is starting");

    // debugging stuff...
    char value[PROPERTY_VALUE_MAX];

    property_get("ro.bq.gpu_to_cpu_unsupported", value, "0");
    mGpuToCpuSupported = !atoi(value);

    property_get("debug.sf.drop_missed_frames", value, "0");
    mDropMissedFrames = atoi(value);

    property_get("debug.sf.showupdates", value, "0");
    mDebugRegion = atoi(value);

    property_get("debug.sf.ddms", value, "0");
    mDebugDDMS = atoi(value);
    if (mDebugDDMS) {
        if (!startDdmConnection()) {
            // start failed, and DDMS debugging not enabled
            mDebugDDMS = 0;
        }
    }
    ALOGI_IF(mDebugRegion, "showupdates enabled");
    ALOGI_IF(mDebugDDMS, "DDMS debugging enabled");
}
```

从代码上看，初始化构造函数只是对内部各项进行初始化，并且从系统配置中读取相关的配置参数，最后在开起debug的后门。熟悉屏幕驱动的同学或者了解过嵌入式GUI的同学，理解起来应该不是很陌生。通常我们在处理大数据量传输的时候，会采用DMA加速的方式，得到更加快速的总线传输数据，并且，在屏幕刷新驱动的过程中，大部分屏幕采用了逐行扫描的时序逻辑。可以用比较形象的一个例子便于了解和消化。

我们将一块屏幕，比作是一个RGBA四通道的一个二维矩阵，每个像素点用一个uchar类型的8Bit存储单元存储通道值，并且设置了逐行扫描的过程，假如，现在需要在第一个点写入0xff00ff00,第二个点写入0xff00ffff，那么实际上屏幕塞入的数据时序类似：

0xff 0x00 0xff 0x00 0xff 0x00 0xff 0xff 0x00 0x00 0x00 ……

按照一帧完整的数据塞入到屏幕的驱动芯片中，就可以展现出我们想要得到的渲染效果。笔者之前的学习经验是基于ARM 2440开发平台的裸板驱动调试中发现，实际上，可以抽象的想象，把屏幕当做一个buffer缓存，按照时序，塞入相应的行数数据到屏幕驱动的缓存芯片中，最后屏幕会把缓存的数据显示到屏幕。理解了这个概念，也就便于理解接下去Android的屏幕显示部分。

原生部分将surfaceFlinger的工作已经备注的较为完备了。

```
void SurfaceFlinger::init() {
    ALOGI(  "SurfaceFlinger's main thread ready to run. "
            "Initializing graphics H/W...");

    Mutex::Autolock _l(mStateLock);

    // initialize EGL for the default display
    mEGLDisplay = eglGetDisplay(EGL_DEFAULT_DISPLAY);
    eglInitialize(mEGLDisplay, NULL, NULL);

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

    // Initialize the H/W composer object.  There may or may not
be an
    // actual hardware composer underneath.
    mHwc = new HWComposer(this,
            *static_cast<HWComposer::EventHandler *>(this));

    // get a RenderEngine for the given display / config (can't
fail)
    mRenderEngine = RenderEngine::create(mEGLDisplay, mHwc-
>getVisualID());

    // retrieve the EGL context that was selected/created
    mEGLContext = mRenderEngine->getEGLContext();

    LOG_ALWAYS_FATAL_IF(mEGLContext == EGL_NO_CONTEXT,
            "couldn't create EGLContext");

    // initialize our non-virtual displays
    for (size_t i=0 ; i<DisplayDevice::NUM_BUILTIN_DISPLAY_TYPES ;
```

```
i++) {
        DisplayDevice::DisplayType
type((DisplayDevice::DisplayType)i);
        // set-up the displays that are already connected
        if (mHwc->isConnected(i) ||
type==DisplayDevice::DISPLAY_PRIMARY) {
            // All non-virtual displays are currently considered
secure.
            bool isSecure = true;
            createBuiltinDisplayLocked(type);
            wp<IBinder> token = mBuiltinDisplays[i];

            sp<IGraphicBufferProducer> producer;
            sp<IGraphicBufferConsumer> consumer;
            BufferQueue::createBufferQueue(&producer, &consumer,
                    new GraphicBufferAlloc());

            sp<FramebufferSurface> fbs = new
FramebufferSurface(*mHwc, i,
                    consumer);
            int32_t hwcId = allocateHwcDisplayId(type);
            sp<DisplayDevice> hw = new DisplayDevice(this,
                    type, hwcId, mHwc->getFormat(hwcId), isSecure,
token,
                    fbs, producer,
                    mRenderEngine->getEGLConfig());
            if (i > DisplayDevice::DISPLAY_PRIMARY) {
                // FIXME: currently we don't get blank/unblank
requests
                // for displays other than the main display, so we
always
                // assume a connected display is unblanked.
                ALOGD("marking display %zu as acquired/unblanked",
i);
                hw->setPowerMode(HWC_POWER_MODE_NORMAL);
            }
            mDisplays.add(token, hw);
        }
    }

    // make the GLContext current so that we can create textures
when creating Layers
    // (which may happens before we render something)
    getDefaultDisplayDevice()->makeCurrent(mEGLDisplay,
mEGLContext);

    mEventControlThread = new EventControlThread(this);
    mEventControlThread->run("EventControl",
PRIORITY_URGENT_DISPLAY);

    // set a fake vsync period if there is no HWComposer
```

```
    if (mHwc->initCheck() != NO_ERROR) {
        mPrimaryDispSync.setPeriod(16666667);
    }

    // initialize our drawing state
    mDrawingState = mCurrentState;

    // set initial conditions (e.g. unblank default device)
    initializeDisplays();

    // start boot animation
    startBootAnim();
}
```

从代码上讲，初始化的过程。surfaceflinger首先做的，是获取了显示的容器。并且对这个容器进行了初始化。

```
// initialize EGL for the default display
mEGLDisplay = eglGetDisplay(EGL_DEFAULT_DISPLAY);
eglInitialize(mEGLDisplay, NULL, NULL);
```

接着，创建了两个线程，在Android 4.1之后引入的VSSync刷新机制，如果硬件支持VS，显示效果会更好，如果不支持，则提供软件模拟信号的方式显示。

至于VSSync刷新方式的具体原理，后续在来讨论，详细代码部分请参阅：

https://blog.csdn.net/u013928208/article/details/82744147

https://blog.csdn.net/marshal_zsx/article/details/84141204

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

紧接着，会创建一个HWComposer对象，这个可以形象的理解为一个硬件的抽离层，具体的还得仔细去琢磨，后面会详细介绍。

之后，创建一个渲染单元，并且获取EGL context对象，至此，初始化创建对象部分的启动流程大致上如此。

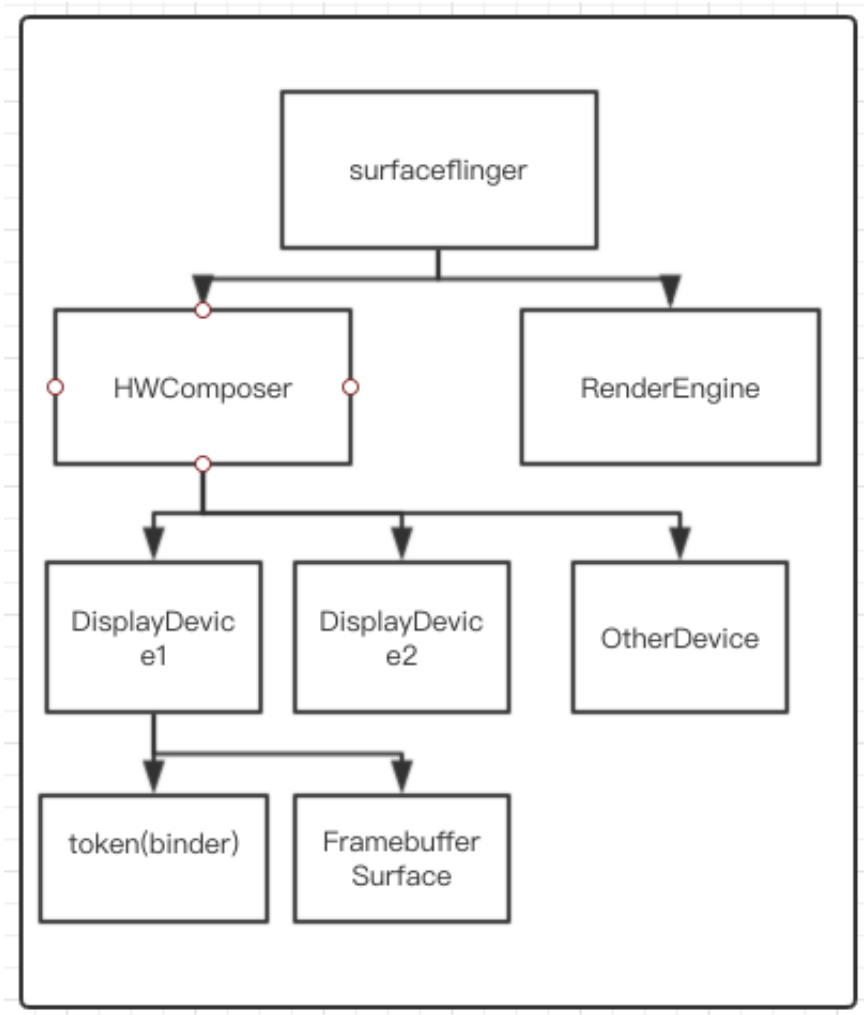
```
// get a RenderEngine for the given display / config (can't fail)
mRenderEngine = RenderEngine::create(mEGLDisplay, mHwc->getVisualID());

// retrieve the EGL context that was selected/created
mEGLContext = mRenderEngine->getEGLContext();

LOG_ALWAYS_FATAL_IF(mEGLContext == EGL_NO_CONTEXT,
        "couldn't create EGLContext");
```

前面主要是初始化过程中创建的一些对象或者组件，接下去就是初始化所需要做的事

情，首先surfaceflinger会对每个屏幕进行初始化，并且创建一个对应的屏幕缓存buffer。具体请看下面：

```
// initialize our non-virtual displays
for (size_t i=0 ; i<DisplayDevice::NUM_BUILTIN_DISPLAY_TYPES ; i++) {
    DisplayDevice::DisplayType type((DisplayDevice::DisplayType)i);
    // set-up the displays that are already connected
    if (mHwc->isConnected(i) || type==DisplayDevice::DISPLAY_PRIMARY) {
        // All non-virtual displays are currently considered secure.
        bool isSecure = true;
        createBuiltinDisplayLocked(type);
        wp<IBinder> token = mBuiltinDisplays[i];

        sp<IGraphicBufferProducer> producer;
        sp<IGraphicBufferConsumer> consumer;
        //创建buffer
        BufferQueue::createBufferQueue(&producer, &consumer,
                new GraphicBufferAlloc());

        sp<FramebufferSurface> fbs = new FramebufferSurface(*mHwc, i,
                consumer);
        int32_t hwcId = allocateHwcDisplayId(type);
        sp<DisplayDevice> hw = new DisplayDevice(this,
                type, hwcId, mHwc->getFormat(hwcId), isSecure, token,
                fbs, producer,
                mRenderEngine->getEGLConfig());
        if (i > DisplayDevice::DISPLAY_PRIMARY) {
            // FIXME: currently we don't get blank/unblank requests
            // for displays other than the main display, so we always
            // assume a connected display is unblanked.
            ALOGD("marking display %zu as acquired/unblanked", i);
            hw->setPowerMode(HWC_POWER_MODE_NORMAL);
        }
        //关联display和hwComposer
        mDisplays.add(token, hw);
    }
}
```

从结构上理解并且猜想一下，大致上可以把HWComposer抽象的理解为一个屏幕管理器模块，对应的displaydevice就是所述的对应一块屏幕，每一个DisplayDevice中都存有一个FreebufferSurface，最后，RenderEngline做渲染，每一块屏幕都bind到HwComposer中。可以参考下图：



按照目前的结构看，猜想大致上SurfaceFlinger的结构可以按照上图的形式。接下去也就是默认显示屏，屏幕状态和初始化，在初始化的最后，开始开机动画。

```
// make the GLContext current so that we can create textures when creating
Layers
// (which may happens before we render something)
getDefaultDisplayDevice()->makeCurrent(mEGLDisplay, mEGLContext);

mEventControlThread = new EventControlThread(this);
mEventControlThread->run("EventControl", PRIORITY_URGENT_DISPLAY);

// set a fake vsync period if there is no HWComposer
if (mHwc->initCheck() != NO_ERROR) {
    mPrimaryDispSync.setPeriod(16666667);
}

// initialize our drawing state
mDrawingState = mCurrentState;

// set initial conditions (e.g. unblank default device)
```

```
initializeDisplays();

// start boot animation
startBootAnim();
```

最后可以看下定义的头文件，可以了解一下surfacefling具体的函数定义。

```
/*
* Copyright (C) 2007 The Android Open Source Project
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*      http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*/

#ifndef ANDROID_SURFACE_FLINGER_H
#define ANDROID_SURFACE_FLINGER_H

#include <stdint.h>
#include <sys/types.h>

#include <EGL/egl.h>

/*
* NOTE: Make sure this file doesn't include  anything from <gl/ > or <gl2/
>
*/

#include <cutils/compiler.h>

#include <utils/Atomic.h>
#include <utils/Errors.h>
#include <utils/KeyedVector.h>
#include <utils/RefBase.h>
#include <utils/SortedVector.h>
#include <utils/threads.h>

#include <binder/IMemory.h>

#include <ui/PixelFormat.h>
#include <ui/mat4.h>

#include <gui/ISurfaceComposer.h>
#include <gui/ISurfaceComposerClient.h>
```

```
#include <hardware/hwcomposer_defs.h>

#include <private/gui/LayerState.h>

#include "Barrier.h"
#include "DisplayDevice.h"
#include "DispSync.h"
#include "FrameTracker.h"
#include "MessageQueue.h"

#include "DisplayHardware/HWComposer.h"
#include "Effects/Daltonizer.h"

namespace android {

// ---------------------------------------------------------------------------

class Client;
class DisplayEventConnection;
class EventThread;
class IGraphicBufferAlloc;
class Layer;
class LayerDim;
class Surface;
class RenderEngine;
class EventControlThread;

// ---------------------------------------------------------------------------

enum {
    eTransactionNeeded        = 0x01,
    eTraversalNeeded          = 0x02,
    eDisplayTransactionNeeded = 0x04,
    eTransactionMask          = 0x07
};

class SurfaceFlinger : public BnSurfaceComposer,
                       private IBinder::DeathRecipient,
                       private HWComposer::EventHandler
{
public:
    static char const* getServiceName() ANDROID_API {
        return "SurfaceFlinger";
    }

    SurfaceFlinger() ANDROID_API;

    // must be called before clients can connect
```

```
    void init() ANDROID_API;

    // starts SurfaceFlinger main loop in the current thread
    void run() ANDROID_API;

    enum {
        EVENT_VSYNC = HWC_EVENT_VSYNC
    };

    // post an asynchronous message to the main thread
    status_t postMessageAsync(const sp<MessageBase>& msg, nsecs_t reltime =
0, uint32_t flags = 0);

    // post a synchronous message to the main thread
    status_t postMessageSync(const sp<MessageBase>& msg, nsecs_t reltime =
0, uint32_t flags = 0);

    // force full composition on all displays
    void repaintEverything();

    // returns the default Display
    sp<const DisplayDevice> getDefaultDisplayDevice() const {
        return
getDisplayDevice(mBuiltinDisplays[DisplayDevice::DISPLAY_PRIMARY]);
    }

    // utility function to delete a texture on the main thread
    void deleteTextureAsync(uint32_t texture);

    // enable/disable h/w composer event
    // TODO: this should be made accessible only to EventThread
    void eventControl(int disp, int event, int enabled);

    // called on the main thread by MessageQueue when an internal message
    // is received
    // TODO: this should be made accessible only to MessageQueue
    void onMessageReceived(int32_t what);

    // for debugging only
    // TODO: this should be made accessible only to HWComposer
    const Vector< sp<Layer> >& getLayerSortedByZForHwcDisplay(int id);

    RenderEngine& getRenderEngine() const {
        return *mRenderEngine;
    }

private:
    friend class Client;
    friend class DisplayEventConnection;
    friend class Layer;
    friend class MonitoredProducer;
```

```
    // This value is specified in number of frames.  Log frame stats at
most
    // every half hour.
    enum { LOG_FRAME_STATS_PERIOD =  30*60*60 };

    static const size_t MAX_LAYERS = 4096;

    // We're reference counted, never destroy SurfaceFlinger directly
    virtual ~SurfaceFlinger();

    /* ------------------------------------------------------------------
----
     * Internal data structures
     */

    class LayerVector : public SortedVector< sp<Layer> > {
    public:
        LayerVector();
        LayerVector(const LayerVector& rhs);
        virtual int do_compare(const void* lhs, const void* rhs) const;
    };

    struct DisplayDeviceState {
        DisplayDeviceState();
        DisplayDeviceState(DisplayDevice::DisplayType type);
        bool isValid() const { return type >= 0; }
        bool isMainDisplay() const { return type ==
DisplayDevice::DISPLAY_PRIMARY; }
        bool isVirtualDisplay() const { return type >=
DisplayDevice::DISPLAY_VIRTUAL; }
        DisplayDevice::DisplayType type;
        sp<IGraphicBufferProducer> surface;
        uint32_t layerStack;
        Rect viewport;
        Rect frame;
        uint8_t orientation;
        uint32_t width, height;
        String8 displayName;
        bool isSecure;
    };

    struct State {
        LayerVector layersSortedByZ;
        DefaultKeyedVector< wp<IBinder>, DisplayDeviceState> displays;
    };

    /* ------------------------------------------------------------------
----
     * IBinder interface
     */
```

```
    virtual status_t onTransact(uint32_t code, const Parcel& data,
        Parcel* reply, uint32_t flags);
    virtual status_t dump(int fd, const Vector<String16>& args);

    /* ------------------------------------------------------------------
----
     * ISurfaceComposer interface
     */
    virtual sp<ISurfaceComposerClient> createConnection();
    virtual sp<IGraphicBufferAlloc> createGraphicBufferAlloc();
    virtual sp<IBinder> createDisplay(const String8& displayName, bool
secure);
    virtual void destroyDisplay(const sp<IBinder>& display);
    virtual sp<IBinder> getBuiltInDisplay(int32_t id);
    virtual void setTransactionState(const Vector<ComposerState>& state,
            const Vector<DisplayState>& displays, uint32_t flags);
    virtual void bootFinished();
    virtual bool authenticateSurfaceTexture(
        const sp<IGraphicBufferProducer>& bufferProducer) const;
    virtual sp<IDisplayEventConnection> createDisplayEventConnection();
    virtual status_t captureScreen(const sp<IBinder>& display,
            const sp<IGraphicBufferProducer>& producer,
            Rect sourceCrop, uint32_t reqWidth, uint32_t reqHeight,
            uint32_t minLayerZ, uint32_t maxLayerZ,
            bool useIdentityTransform, ISurfaceComposer::Rotation
rotation);
    virtual status_t getDisplayStats(const sp<IBinder>& display,
            DisplayStatInfo* stats);
    virtual status_t getDisplayConfigs(const sp<IBinder>& display,
            Vector<DisplayInfo>* configs);
    virtual int getActiveConfig(const sp<IBinder>& display);
    virtual void setPowerMode(const sp<IBinder>& display, int mode);
    virtual status_t setActiveConfig(const sp<IBinder>& display, int id);
    virtual status_t clearAnimationFrameStats();
    virtual status_t getAnimationFrameStats(FrameStats* outStats) const;

    /* ------------------------------------------------------------------
----
     * DeathRecipient interface
     */
    virtual void binderDied(const wp<IBinder>& who);

    /* ------------------------------------------------------------------
----
     * RefBase interface
     */
    virtual void onFirstRef();

    /* ------------------------------------------------------------------
----
     * HWComposer::EventHandler interface
```

```
     */
    virtual void onVSyncReceived(int type, nsecs_t timestamp);
    virtual void onHotplugReceived(int disp, bool connected);

    /* ------------------------------------------------------------------------
     * Message handling
     */
    void waitForEvent();
    void signalTransaction();
    void signalLayerUpdate();
    void signalRefresh();

    // called on the main thread in response to initializeDisplays()
    void onInitializeDisplays();
    // called on the main thread in response to setActiveConfig()
    void setActiveConfigInternal(const sp<DisplayDevice>& hw, int mode);
    // called on the main thread in response to setPowerMode()
    void setPowerModeInternal(const sp<DisplayDevice>& hw, int mode);

    // Returns whether the transaction actually modified any state
    bool handleMessageTransaction();

    // Returns whether a new buffer has been latched (see handlePageFlip())
    bool handleMessageInvalidate();

    void handleMessageRefresh();

    void handleTransaction(uint32_t transactionFlags);
    void handleTransactionLocked(uint32_t transactionFlags);

    void updateCursorAsync();

    /* handlePageFlip - latch a new buffer if available and compute the dirty
     * region. Returns whether a new buffer has been latched, i.e., whether it
     * is necessary to perform a refresh during this vsync.
     */
    bool handlePageFlip();

    /* ------------------------------------------------------------------------
     * Transactions
     */
    uint32_t getTransactionFlags(uint32_t flags);
    uint32_t peekTransactionFlags(uint32_t flags);
    uint32_t setTransactionFlags(uint32_t flags);
    void commitTransaction();
    uint32_t setClientStateLocked(const sp<Client>& client, const layer_state_t& s);
```

```
    uint32_t setDisplayStateLocked(const DisplayState& s);

    /* ------------------------------------------------------------------
----
     * Layer management
     */
    status_t createLayer(const String8& name, const sp<Client>& client,
            uint32_t w, uint32_t h, PixelFormat format, uint32_t flags,
            sp<IBinder>* handle, sp<IGraphicBufferProducer>* gbp);

    status_t createNormalLayer(const sp<Client>& client, const String8&
name,
            uint32_t w, uint32_t h, uint32_t flags, PixelFormat& format,
            sp<IBinder>* outHandle, sp<IGraphicBufferProducer>* outGbp,
            sp<Layer>* outLayer);

    status_t createDimLayer(const sp<Client>& client, const String8& name,
            uint32_t w, uint32_t h, uint32_t flags, sp<IBinder>* outHandle,
            sp<IGraphicBufferProducer>* outGbp, sp<Layer>* outLayer);

    // called in response to the window-manager calling
    // ISurfaceComposerClient::destroySurface()
    status_t onLayerRemoved(const sp<Client>& client, const sp<IBinder>&
handle);

    // called when all clients have released all their references to
    // this layer meaning it is entirely safe to destroy all
    // resources associated to this layer.
    status_t onLayerDestroyed(const wp<Layer>& layer);

    // remove a layer from SurfaceFlinger immediately
    status_t removeLayer(const sp<Layer>& layer);

    // add a layer to SurfaceFlinger
    status_t addClientLayer(const sp<Client>& client,
            const sp<IBinder>& handle,
            const sp<IGraphicBufferProducer>& gbc,
            const sp<Layer>& lbc);

    /* ------------------------------------------------------------------
----
     * Boot animation, on/off animations and screen capture
     */

    void startBootAnim();

    void renderScreenImplLocked(
            const sp<const DisplayDevice>& hw,
            Rect sourceCrop, uint32_t reqWidth, uint32_t reqHeight,
            uint32_t minLayerZ, uint32_t maxLayerZ,
            bool yswap, bool useIdentityTransform,
```

```
Transform::orientation_flags rotation);

    status_t captureScreenImplLocked(
            const sp<const DisplayDevice>& hw,
            const sp<IGraphicBufferProducer>& producer,
            Rect sourceCrop, uint32_t reqWidth, uint32_t reqHeight,
            uint32_t minLayerZ, uint32_t maxLayerZ,
            bool useIdentityTransform, Transform::orientation_flags
rotation);

    /* ------------------------------------------------------------------------
     * EGL
     */
    size_t getMaxTextureSize() const;
    size_t getMaxViewportDims() const;

    /* ------------------------------------------------------------------------
     * Display and layer stack management
     */
    // called when starting, or restarting after system_server death
    void initializeDisplays();

    // Create an IBinder for a builtin display and add it to current state
    void createBuiltinDisplayLocked(DisplayDevice::DisplayType type);

    // NOTE: can only be called from the main thread or with mStateLock
held
    sp<const DisplayDevice> getDisplayDevice(const wp<IBinder>& dpy) const
{
        return mDisplays.valueFor(dpy);
    }

    // NOTE: can only be called from the main thread or with mStateLock
held
    sp<DisplayDevice> getDisplayDevice(const wp<IBinder>& dpy) {
        return mDisplays.valueFor(dpy);
    }

    // mark a region of a layer stack dirty. this updates the dirty
    // region of all screens presenting this layer stack.
    void invalidateLayerStack(uint32_t layerStack, const Region& dirty);

    // allocate a h/w composer display id
    int32_t allocateHwcDisplayId(DisplayDevice::DisplayType type);

    /* ------------------------------------------------------------------------
     * H/W composer
     */
```

```
    HWComposer& getHwComposer() const { return *mHwc; }

    /* ------------------------------------------------------------------------
     * Compositing
     */
    void invalidateHwcGeometry();
    static void computeVisibleRegions(
            const LayerVector& currentLayers, uint32_t layerStack,
            Region& dirtyRegion, Region& opaqueRegion);

    void preComposition();
    void postComposition();
    void rebuildLayerStacks();
    void setUpHWComposer();
    void doComposition();
    void doDebugFlashRegions();
    void doDisplayComposition(const sp<const DisplayDevice>& hw, const
Region& dirtyRegion);

    // compose surfaces for display hw. this fails if using GL and the
surface
    // has been destroyed and is no longer valid.
    bool doComposeSurfaces(const sp<const DisplayDevice>& hw, const Region&
dirty);

    void postFramebuffer();
    void drawWormhole(const sp<const DisplayDevice>& hw, const Region&
region) const;

    /* ------------------------------------------------------------------------
     * Display management
     */

    /* ------------------------------------------------------------------------
     * VSync
     */
     void enableHardwareVsync();
     void disableHardwareVsync(bool makeUnavailable);
     void resyncToHardwareVsync(bool makeAvailable);

    /* ------------------------------------------------------------------------
     * Debugging & dumpsys
     */
    void listLayersLocked(const Vector<String16>& args, size_t& index,
String8& result) const;
    void dumpStatsLocked(const Vector<String16>& args, size_t& index,
```

```
String8& result) const;
    void clearStatsLocked(const Vector<String16>& args, size_t& index,
String8& result);
    void dumpAllLocked(const Vector<String16>& args, size_t& index,
String8& result) const;
    bool startDdmConnection();
    static void appendSfConfigString(String8& result);
    void checkScreenshot(size_t w, size_t s, size_t h, void const* vaddr,
            const sp<const DisplayDevice>& hw,
            uint32_t minLayerZ, uint32_t maxLayerZ);

    void logFrameStats();

    void dumpStaticScreenStats(String8& result) const;

    /* ------------------------------------------------------------------
----
     * Attributes
     */

    // access must be protected by mStateLock
    mutable Mutex mStateLock;
    State mCurrentState;
    volatile int32_t mTransactionFlags;
    Condition mTransactionCV;
    bool mTransactionPending;
    bool mAnimTransactionPending;
    Vector< sp<Layer> > mLayersPendingRemoval;
    SortedVector< wp<IBinder> > mGraphicBufferProducerList;

    // protected by mStateLock (but we could use another lock)
    bool mLayersRemoved;

    // access must be protected by mInvalidateLock
    volatile int32_t mRepaintEverything;

    // constant members (no synchronization needed for access)
    HWComposer* mHwc;
    RenderEngine* mRenderEngine;
    nsecs_t mBootTime;
    bool mGpuToCpuSupported;
    bool mDropMissedFrames;
    sp<EventThread> mEventThread;
    sp<EventThread> mSFEventThread;
    sp<EventControlThread> mEventControlThread;
    EGLContext mEGLContext;
    EGLDisplay mEGLDisplay;
    sp<IBinder> mBuiltinDisplays[DisplayDevice::NUM_BUILTIN_DISPLAY_TYPES];

    // Can only accessed from the main thread, these members
    // don't need synchronization
```

```
    State mDrawingState;
    bool mVisibleRegionsDirty;
    bool mHwWorkListDirty;
    bool mAnimCompositionPending;

    // this may only be written from the main thread with mStateLock held
    // it may be read from other threads with mStateLock held
    DefaultKeyedVector< wp<IBinder>, sp<DisplayDevice> > mDisplays;

    // don't use a lock for these, we don't care
    int mDebugRegion;
    int mDebugDDMS;
    int mDebugDisableHWC;
    int mDebugDisableTransformHint;
    volatile nsecs_t mDebugInSwapBuffers;
    nsecs_t mLastSwapBufferTime;
    volatile nsecs_t mDebugInTransaction;
    nsecs_t mLastTransactionTime;
    bool mBootFinished;
    bool mForceFullDamage;

    // these are thread safe
    mutable MessageQueue mEventQueue;
    FrameTracker mAnimFrameTracker;
    DispSync mPrimaryDispSync;

    // protected by mDestroyedLayerLock;
    mutable Mutex mDestroyedLayerLock;
    Vector<Layer const *> mDestroyedLayers;

    // protected by mHWVsyncLock
    Mutex mHWVsyncLock;
    bool mPrimaryHWVsyncEnabled;
    bool mHWVsyncAvailable;

    /* ------------------------------------------------------------------------
----
     * Feature prototyping
     */

    Daltonizer mDaltonizer;
    bool mDaltonize;

    mat4 mColorMatrix;
    bool mHasColorMatrix;

    // Static screen stats
    bool mHasPoweredOff;
    static const size_t NUM_BUCKETS = 8; // < 1-7, 7+
    nsecs_t mFrameBuckets[NUM_BUCKETS];
    nsecs_t mTotalTime;
```

```
    nsecs_t mLastSwapTime;
};

}; // namespace android

#endif // ANDROID_SURFACE_FLINGER_H
```

在framework对应的onTransact 方法中加入打印Log,具体内容如下：

```
status_t SurfaceFlinger::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    status_t credentialCheck = CheckTransactCodeCredentials(code);
    if (credentialCheck != OK) {
        return credentialCheck;
    }
    ALOGE("SurfaceFlinger" " SurfaceFlinger::onTransact code: %d,flags:
%d",(int)code,flags);
    status_t err = BnSurfaceComposer::onTransact(code, data, reply, flags);
    if (err == UNKNOWN_TRANSACTION || err == PERMISSION_DENIED) {
        CHECK_INTERFACE(ISurfaceCompo
```

可以看到，系统在刷新页面时会出现以下几个log项目：

```
2018-12-11 19:42:40.128 1389-1438/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 19:42:40.153 1389-1438/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 4,flags: 16
2018-12-11 19:42:40.153 1389-1887/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 4,flags: 16
2018-12-11 19:42:40.155 1389-1887/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 7,flags: 16
2018-12-11 19:42:40.155 1389-1887/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 11,flags: 16
2018-12-11 19:42:40.155 1389-1887/? E/SurfaceFlinger: ro.sf.lcd_density
must be defined as a build property
2018-12-11 19:42:40.156 1389-1887/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 12,flags: 16
2018-12-11 19:42:40.157 1389-1887/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 4,flags: 16
2018-12-11 19:42:40.160 1389-1438/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 2,flags: 16
2018-12-11 19:42:40.188 1389-1438/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 19:42:40.193 1389-1438/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 19:42:40.195 1389-1887/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 19:42:40.195 1389-1887/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 19:42:40.195 1389-1887/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
```

```
2018-12-11 19:42:40.195 2945-2962/android:ui I/zygote:
android::hardware::configstore::V1_0::ISurfaceFlingerConfigs::hasWideColorD
isplay retrieved: 0
2018-12-11 19:42:40.197 1389-1759/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 19:42:40.197 1389-1759/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 19:42:40.208 1389-1438/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 19:42:40.212 1389-1887/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 7,flags: 16
2018-12-11 19:42:40.212 1389-1887/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 11,flags: 16
2018-12-11 19:42:40.212 1389-1887/? E/SurfaceFlinger: ro.sf.lcd_density
must be defined as a build property
2018-12-11 19:42:40.212 1389-1887/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 12,flags: 16
2018-12-11 19:42:40.213 1389-1438/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 19:42:40.218 1389-1438/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 19:42:40.231 1389-1887/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
```

在页面发生变化时，会向surfaceflinger发送一个命令，对应的code枚举类型如下：

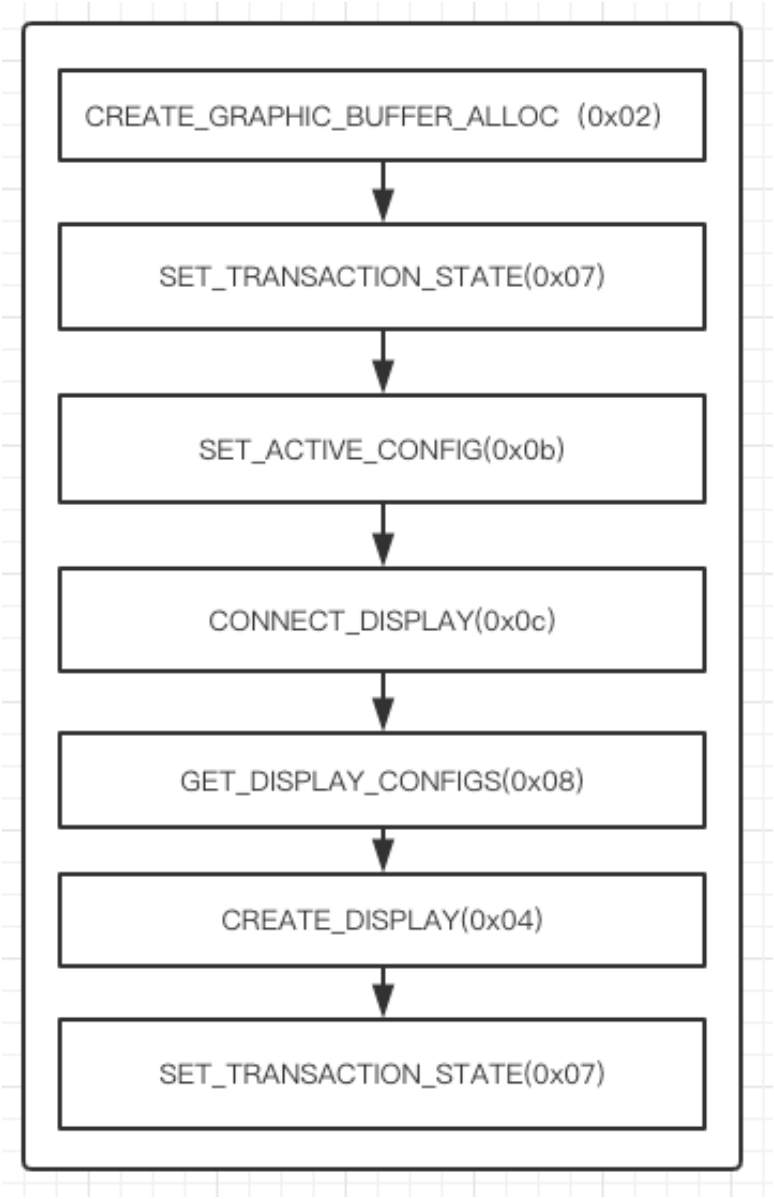
```
enum {
    // Note: BOOT_FINISHED must remain this value, it is called
from
    // Java by ActivityManagerService.
    BOOT_FINISHED = IBinder::FIRST_CALL_TRANSACTION      (0x00),
    CREATE_CONNECTION                                    (0x01),
    CREATE_GRAPHIC_BUFFER_ALLOC                          (0x02),
    CREATE_DISPLAY_EVENT_CONNECTION                      (0x03),
    CREATE_DISPLAY                                       (0x04),
    DESTROY_DISPLAY                                      (0x05),
    GET_BUILT_IN_DISPLAY                                 (0x06),
    SET_TRANSACTION_STATE                                (0x07),
    AUTHENTICATE_SURFACE                                 (0x08),
    GET_DISPLAY_CONFIGS                                  (0x09),
    GET_ACTIVE_CONFIG                                    (0x0A),
    SET_ACTIVE_CONFIG                                    (0x0b),
    CONNECT_DISPLAY                                      (0x0c),
    CAPTURE_SCREEN                                       (0x0d),
    CLEAR_ANIMATION_FRAME_STATS                          (0x0e),
    GET_ANIMATION_FRAME_STATS                            (0x0f),
    SET_POWER_MODE                                       (0x10),
    GET_DISPLAY_STATS                                    (0x11),
};
```

完整虚拟机开机过程log。

```
2018-12-11 20:20:50.213 1388-1388/? I/SurfaceFlinger: Enabling HWC virtual
displays
2018-12-11 20:20:50.213 1388-1388/? I/SurfaceFlinger: Disabling Triple
Buffering
2018-12-11 20:20:50.213 1388-1388/? I/SurfaceFlinger: SurfaceFlinger's main
thread ready to run. Initializing graphics H/W...
2018-12-11 20:20:50.213 1388-1388/? I/SurfaceFlinger: Phase offest NS:
1000000
2018-12-11 20:20:50.226 1388-1388/? I//system/bin/surfaceflinger:
android::hardware::configstore::V1_0::ISurfaceFlingerConfigs::hasWideColorD
isplay retrieved: 0
2018-12-11 20:20:50.227 1388-1388/? W/SurfaceFlinger: no suitable EGLConfig
found, trying a simpler query
2018-12-11 20:20:50.227 1388-1388/? I/SurfaceFlinger: EGL information:
2018-12-11 20:20:50.227 1388-1388/? I/SurfaceFlinger: vendor    : Android
2018-12-11 20:20:50.227 1388-1388/? I/SurfaceFlinger: version   : 1.4
Android META-EGL
2018-12-11 20:20:50.227 1388-1388/? I/SurfaceFlinger: extensions:
EGL_KHR_get_all_proc_addresses EGL_ANDROID_presentation_time
EGL_KHR_swap_buffers_with_damage EGL_ANDROID_get_native_client_buffer
EGL_ANDROID_front_buffer_auto_refresh EGL_ANDROID_get_frame_timestamps
EGL_KHR_image_base EGL_KHR_gl_texture_2D_image
EGL_KHR_gl_texture_cubemap_image EGL_KHR_gl_renderbuffer_image
EGL_KHR_fence_sync EGL_KHR_create_context EGL_ANDROID_image_native_buffer
EGL_ANDROID_recordable
2018-12-11 20:20:50.227 1388-1388/? I/SurfaceFlinger: Client API: OpenGL_ES
2018-12-11 20:20:50.227 1388-1388/? I/SurfaceFlinger: EGLSurface: 8-8-8-8,
config=0x2d
2018-12-11 20:20:50.231 1388-1388/? I/SurfaceFlinger: OpenGL ES
informations:
2018-12-11 20:20:50.231 1388-1388/? I/SurfaceFlinger: vendor    : Google
Inc.
2018-12-11 20:20:50.231 1388-1388/? I/SurfaceFlinger: renderer  : Google
SwiftShader
2018-12-11 20:20:50.231 1388-1388/? I/SurfaceFlinger: version   : OpenGL ES
3.0 SwiftShader 3.3.0.2
2018-12-11 20:20:50.231 1388-1388/? I/SurfaceFlinger: extensions:
GL_EXT_debug_marker GL_OES_compressed_ETC1_RGB8_texture GL_OES_depth24
GL_OES_depth32 GL_OES_depth_texture GL_OES_depth_texture_cube_map
GL_OES_EGL_image GL_OES_EGL_image_external GL_OES_EGL_sync
GL_OES_element_index_uint GL_OES_framebuffer_object
GL_OES_packed_depth_stencil GL_OES_rgb8_rgba8 GL_OES_standard_derivatives
GL_OES_texture_float GL_OES_texture_float_linear GL_OES_texture_half_float
GL_OES_texture_half_float_linear GL_OES_texture_npot GL_OES_texture_3D
GL_EXT_blend_minmax GL_EXT_color_buffer_half_float GL_EXT_draw_buffers
GL_EXT_instanced_arrays GL_EXT_occlusion_query_boolean
GL_EXT_read_format_bgra GL_EXT_texture_filter_anisotropic
GL_EXT_texture_format_BGRA8888 GL_ANGLE_framebuffer_blit
GL_ANGLE_framebuffer_multisample GL_ANGLE_instanced_arrays GL_NV_fence
```

```
GL_NV_framebuffer_blit GL_NV_read_depth ANDROID_EMU_CHECKSUM_HELPER_v1
ANDROID_EMU_dma_v1 ANDROID_EMU_gles_max_version_3_0
GL_OES_vertex_array_object
2018-12-11 20:20:50.231 1388-1388/? I/SurfaceFlinger: GL_MAX_TEXTURE_SIZE =
8192
2018-12-11 20:20:50.231 1388-1388/? I/SurfaceFlinger: GL_MAX_VIEWPORT_DIMS
= 4096
2018-12-11 20:20:50.321 1388-1388/? D/SurfaceFlinger: shader cache
generated - 24 shaders in 74.296722 ms
2018-12-11 20:20:50.322 1388-1388/? D/SurfaceFlinger: Set power mode=2,
type=0 flinger=0xb28ca000
2018-12-11 20:20:50.501 1388-1423/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 2,flags: 16
2018-12-11 20:20:50.503 1388-1423/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 7,flags: 16
2018-12-11 20:20:50.503 1388-1423/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 11,flags: 16
2018-12-11 20:20:50.503 1388-1423/? E/SurfaceFlinger: ro.sf.lcd_density
must be defined as a build property
2018-12-11 20:20:50.503 1388-1423/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 12,flags: 16
2018-12-11 20:20:50.504 1388-1423/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 20:20:50.521 1440-1454/? I//system/bin/bootanimation:
android::hardware::configstore::V1_0::ISurfaceFlingerConfigs::hasWideColorD
isplay retrieved: 0
2018-12-11 20:21:48.773 1388-1423/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 4,flags: 16
2018-12-11 20:21:48.773 1388-1423/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 4,flags: 16
2018-12-11 20:21:48.773 1388-1423/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 7,flags: 16
2018-12-11 20:21:48.773 1388-1423/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 11,flags: 16
2018-12-11 20:21:48.773 1388-1423/? E/SurfaceFlinger: ro.sf.lcd_density
must be defined as a build property
2018-12-11 20:21:48.773 1388-1423/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 12,flags: 16
2018-12-11 20:21:48.773 1388-1423/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 22,flags: 16
2018-12-11 20:21:48.773 1388-1423/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 21,flags: 16
2018-12-11 20:21:48.773 1388-1423/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 20,flags: 16
2018-12-11 20:21:48.773 1388-1423/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 7,flags: 16
2018-12-11 20:21:48.774 1388-1423/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 18,flags: 16
2018-12-11 20:21:48.774 1388-1388/? D/SurfaceFlinger: Set power mode=2,
type=0 flinger=0xb28ca000
2018-12-11 20:21:49.110 1388-1423/? E/SurfaceFlinger: SurfaceFlinger
```

```
SurfaceFlinger::onTransact code: 2,flags: 16
2018-12-11 20:21:49.121 1388-1458/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 4,flags: 16
2018-12-11 20:21:49.126 1388-1458/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 4,flags: 16
2018-12-11 20:21:49.126 1388-1421/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 4,flags: 16
2018-12-11 20:21:49.135 1388-1606/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 20:21:49.155 1388-1606/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 20:21:49.171 1388-1458/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 20:21:49.177 1388-1606/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 20:21:49.179 1388-1606/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 20:21:49.188 1388-1606/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 20:21:49.188 1388-1606/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 20:21:49.188 1388-1458/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 20:21:49.189 1388-1458/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 20:21:49.378 1388-1458/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 20:21:49.379 1388-1458/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 20:21:49.390 1388-1606/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 20:21:49.390 1388-1606/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-11 20:21:49.420 1550-1550/system_process I/zygote: Looking for
service android.hardware.configstore@1.0::ISurfaceFlingerConfigs/default
2018-12-11 20:21:49.427 1388-1606/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 4,flags: 16
2018-12-11 20:21:49.427 1388-1606/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 4,flags: 16
2018-12-11 20:21:49.433 1388-1606/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
```



整个过程打印流程。

处理业务类，在/native/libs/gui/ISurfaceComposer.cpp文件中，具体处理代码如下所示：

```
status_t BnSurfaceComposer::onTransact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
    switch(code) {
        case CREATE_CONNECTION: {
            CHECK_INTERFACE(ISurfaceComposer, data, reply);
            sp<IBinder> b = IInterface::asBinder(createConnection());
            reply->writeStrongBinder(b);
            return NO_ERROR;
        }
        case CREATE_GRAPHIC_BUFFER_ALLOC: {
            CHECK_INTERFACE(ISurfaceComposer, data, reply);
```

```
            sp<IBinder> b =
IInterface::asBinder(createGraphicBufferAlloc());
            reply->writeStrongBinder(b);
            return NO_ERROR;
        }
        case SET_TRANSACTION_STATE: {
            CHECK_INTERFACE(ISurfaceComposer, data, reply);

            size_t count = data.readUint32();
            if (count > data.dataSize()) {
                return BAD_VALUE;
            }
            ComposerState s;
            Vector<ComposerState> state;
            state.setCapacity(count);
            for (size_t i = 0; i < count; i++) {
                if (s.read(data) == BAD_VALUE) {
                    return BAD_VALUE;
                }
                state.add(s);
            }

            count = data.readUint32();
            if (count > data.dataSize()) {
                return BAD_VALUE;
            }
            DisplayState d;
            Vector<DisplayState> displays;
            displays.setCapacity(count);
            for (size_t i = 0; i < count; i++) {
                if (d.read(data) == BAD_VALUE) {
                    return BAD_VALUE;
                }
                displays.add(d);
            }

            uint32_t stateFlags = data.readUint32();
            setTransactionState(state, displays, stateFlags);
            return NO_ERROR;
        }
        case BOOT_FINISHED: {
            CHECK_INTERFACE(ISurfaceComposer, data, reply);
            bootFinished();
            return NO_ERROR;
        }
        case CAPTURE_SCREEN: {
            CHECK_INTERFACE(ISurfaceComposer, data, reply);
            sp<IBinder> display = data.readStrongBinder();
            sp<IGraphicBufferProducer> producer =
                    interface_cast<IGraphicBufferProducer>
(data.readStrongBinder());
```

```
            Rect sourceCrop;
            data.read(sourceCrop);
            uint32_t reqWidth = data.readUint32();
            uint32_t reqHeight = data.readUint32();
            uint32_t minLayerZ = data.readUint32();
            uint32_t maxLayerZ = data.readUint32();
            bool useIdentityTransform = static_cast<bool>
(data.readInt32());
            int32_t rotation = data.readInt32();

            status_t res = captureScreen(display, producer,
                    sourceCrop, reqWidth, reqHeight, minLayerZ, maxLayerZ,
                    useIdentityTransform,
                    static_cast<ISurfaceComposer::Rotation>(rotation));
            reply->writeInt32(res);
            return NO_ERROR;
        }
        case AUTHENTICATE_SURFACE: {
            CHECK_INTERFACE(ISurfaceComposer, data, reply);
            sp<IGraphicBufferProducer> bufferProducer =
                    interface_cast<IGraphicBufferProducer>
(data.readStrongBinder());
            int32_t result = authenticateSurfaceTexture(bufferProducer) ? 1
: 0;
            reply->writeInt32(result);
            return NO_ERROR;
        }
        case CREATE_DISPLAY_EVENT_CONNECTION: {
            CHECK_INTERFACE(ISurfaceComposer, data, reply);
            sp<IDisplayEventConnection>
connection(createDisplayEventConnection());
            reply->writeStrongBinder(IInterface::asBinder(connection));
            return NO_ERROR;
        }
        case CREATE_DISPLAY: {
            CHECK_INTERFACE(ISurfaceComposer, data, reply);
            String8 displayName = data.readString8();
            bool secure = bool(data.readInt32());
            sp<IBinder> display(createDisplay(displayName, secure));
            reply->writeStrongBinder(display);
            return NO_ERROR;
        }
        case DESTROY_DISPLAY: {
            CHECK_INTERFACE(ISurfaceComposer, data, reply);
            sp<IBinder> display = data.readStrongBinder();
            destroyDisplay(display);
            return NO_ERROR;
        }
        case GET_BUILT_IN_DISPLAY: {
            CHECK_INTERFACE(ISurfaceComposer, data, reply);
            int32_t id = data.readInt32();
```

```
            sp<IBinder> display(getBuiltInDisplay(id));
            reply->writeStrongBinder(display);
            return NO_ERROR;
        }
        case GET_DISPLAY_CONFIGS: {
            CHECK_INTERFACE(ISurfaceComposer, data, reply);
            Vector<DisplayInfo> configs;
            sp<IBinder> display = data.readStrongBinder();
            status_t result = getDisplayConfigs(display, &configs);
            reply->writeInt32(result);
            if (result == NO_ERROR) {
                reply->writeUint32(static_cast<uint32_t>(configs.size()));
                for (size_t c = 0; c < configs.size(); ++c) {
                    memcpy(reply->writeInplace(sizeof(DisplayInfo)),
                            &configs[c], sizeof(DisplayInfo));
                }
            }
            return NO_ERROR;
        }
        case GET_DISPLAY_STATS: {
            CHECK_INTERFACE(ISurfaceComposer, data, reply);
            DisplayStatInfo stats;
            sp<IBinder> display = data.readStrongBinder();
            status_t result = getDisplayStats(display, &stats);
            reply->writeInt32(result);
            if (result == NO_ERROR) {
                memcpy(reply->writeInplace(sizeof(DisplayStatInfo)),
                        &stats, sizeof(DisplayStatInfo));
            }
            return NO_ERROR;
        }
        case GET_ACTIVE_CONFIG: {
            CHECK_INTERFACE(ISurfaceComposer, data, reply);
            sp<IBinder> display = data.readStrongBinder();
            int id = getActiveConfig(display);
            reply->writeInt32(id);
            return NO_ERROR;
        }
        case SET_ACTIVE_CONFIG: {
            CHECK_INTERFACE(ISurfaceComposer, data, reply);
            so<IBinder> display = data.readStrongBinder();
            int id = data.readInt32();
            status_t result = setActiveConfig(display, id);
            reply->writeInt32(result);
            return NO_ERROR;
        }
        case CLEAR_ANIMATION_FRAME_STATS: {
            CHECK_INTERFACE(ISurfaceComposer, data, reply);
            status_t result = clearAnimationFrameStats();
            reply->writeInt32(result);
            return NO_ERROR;
```

```
        }
        case GET_ANIMATION_FRAME_STATS: {
            CHECK_INTERFACE(ISurfaceComposer, data, reply);
            FrameStats stats;
            status_t result = getAnimationFrameStats(&stats);
            reply->write(stats);
            reply->writeInt32(result);
            return NO_ERROR;
        }
        case SET_POWER_MODE: {
            CHECK_INTERFACE(ISurfaceComposer, data, reply);
            sp<IBinder> display = data.readStrongBinder();
            int32_t mode = data.readInt32();
            setPowerMode(display, mode);
            return NO_ERROR;
        }
        default: {
            return BBinder::onTransact(code, data, reply, flags);
        }
    }
}
```

在surfaceFlinger里面有个doDebugFlashRegions方法，可以看到刷新的步骤和具体的叠层方法，实际打开后会发现每次刷新屏幕都十分闪烁，具体代码在：

```
void SurfaceFlinger::doDebugFlashRegions()
{

    ALOGW("SurfaceFlinger::doDebugFlashRegions");
       //注释代码段
     // is debugging enabled
    // if (CC_LIKELY(!mDebugRegion))
    //     return;

    const bool repaintEverything = mRepaintEverything;
    for (size_t dpy=0 ; dpy<mDisplays.size() ; dpy++) {
        const sp<DisplayDevice>& hw(mDisplays[dpy]);
        if (hw->isDisplayOn()) {
            // transform the dirty region into this screen's coordinate
space
            const Region dirtyRegion(hw-
>getDirtyRegion(repaintEverything));
            if (!dirtyRegion.isEmpty()) {
                // redraw the whole screen
                doComposeSurfaces(hw, Region(hw->bounds()));

                // and draw the dirty region
                const int32_t height = hw->getHeight();
                //指定着色器和着色范围
```

```
                RenderEngine& engine(getRenderEngine());
                engine.fillRegionWithColor(dirtyRegion, height, 1, 0, 1,
1);
                //渲染到Buffer
                hw->swapBuffers(getHwComposer());
            }
        }
    }
    //输出到frameBuffer
    postFramebuffer();

    if (mDebugRegion > 1) {
        usleep(mDebugRegion * 1000);
    }
    //恢复
    for (size_t displayId = 0; displayId < mDisplays.size(); ++displayId) {
        auto& displayDevice = mDisplays[displayId];
        if (!displayDevice->isDisplayOn()) {
            continue;
        }

        status_t result = displayDevice->prepareFrame(*mHwc);
        ALOGE_IF(result != NO_ERROR, "prepareFrame for display %zd failed:"
                " %d (%s)", displayId, result, strerror(-result));
    }
}
```

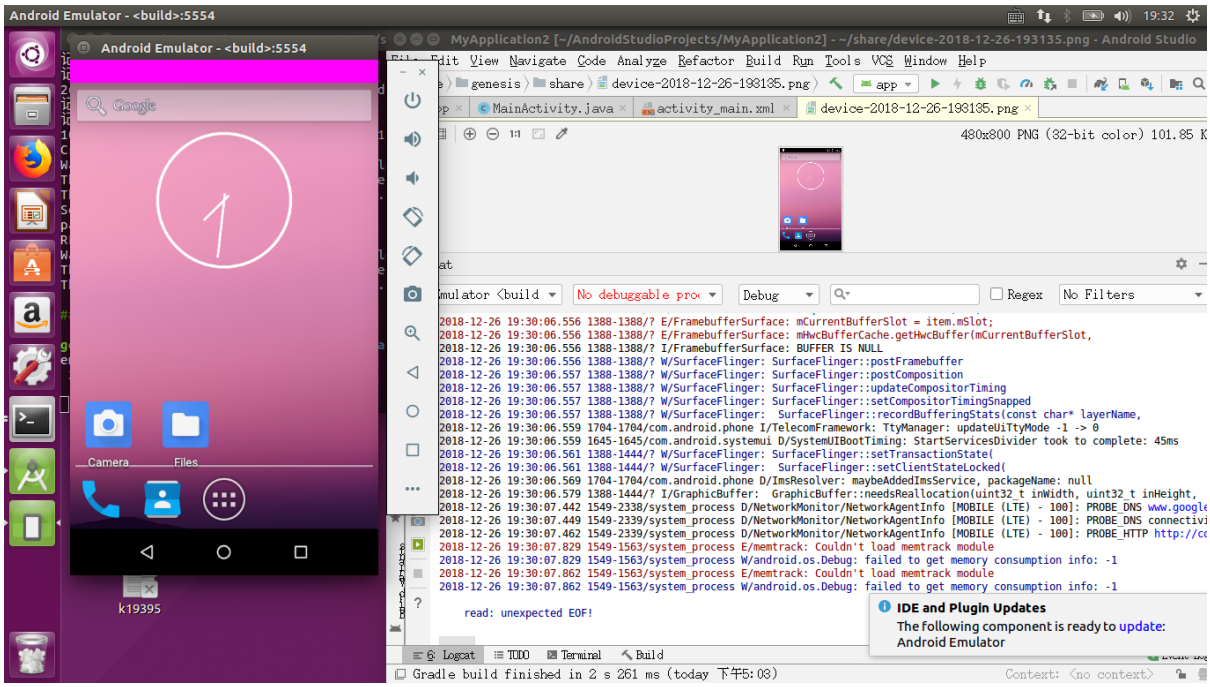
在着色器的代码中，实际上可以看到是绘制了一个Mask层。

```
void RenderEngine::fillRegionWithColor(const Region& region,
uint32_t height,
        float red, float green, float blue, float alpha) {
    size_t c;
    Rect const* r = region.getArray(&c);
    Mesh mesh(Mesh::TRIANGLES, c*6, 2);
    Mesh::VertexArray<vec2> position(mesh.getPositionArray<vec2>
());
    for (size_t i=0 ; i<c ; i++, r++) {
        position[i*6 + 0].x = r->left;
        position[i*6 + 0].y = height - r->top;
        position[i*6 + 1].x = r->left;
        position[i*6 + 1].y = height - r->bottom;
        position[i*6 + 2].x = r->right;
        position[i*6 + 2].y = height - r->bottom;
        position[i*6 + 3].x = r->left;
        position[i*6 + 3].y = height - r->top;
        position[i*6 + 4].x = r->right;
        position[i*6 + 4].y = height - r->bottom;
        position[i*6 + 5].x = r->right;
        position[i*6 + 5].y = height - r->top;
    }
```

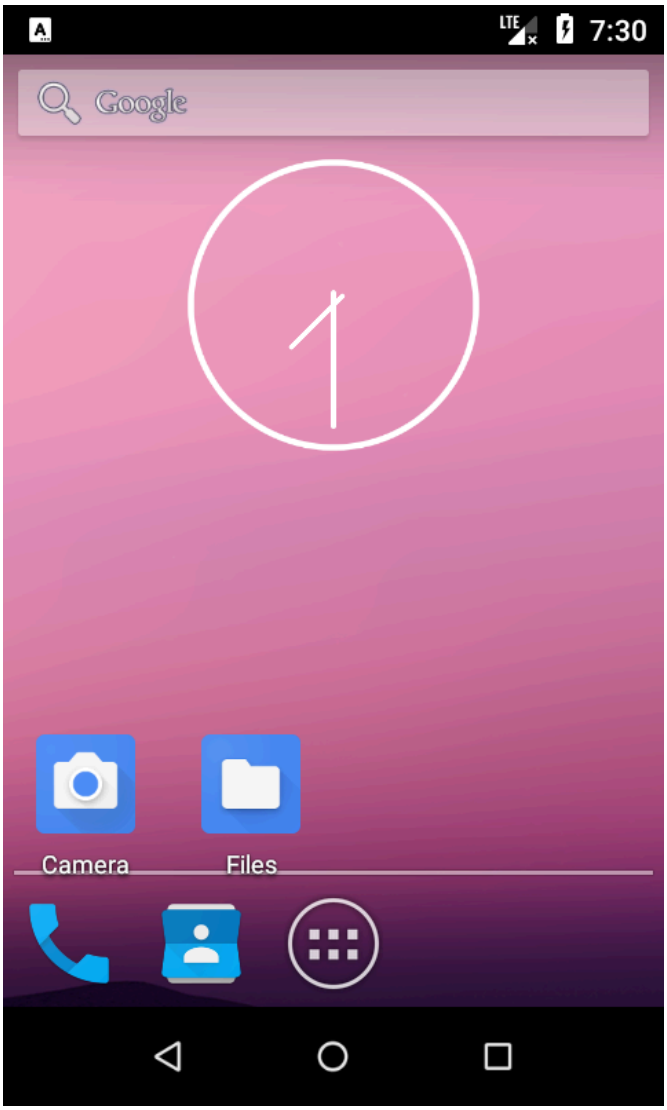
```
    setupFillWithColor(red, green, blue, alpha);
    drawMesh(mesh);
}
```

我们做一个相关性实验，如果把着色器的alpha替换为0.1f，并且将恢复的代码干掉，具体如下所示：

```
        const Region dirtyRegion(hw->getDirtyRegion(repaintEverything));
            if (!dirtyRegion.isEmpty()) {
                // redraw the whole screen
                doComposeSurfaces(hw, Region(hw->bounds()));
                // and draw the dirty region
                const int32_t height = hw->getHeight();
                RenderEngine& engine(getRenderEngine());
                engine.fillRegionWithColor(dirtyRegion, height, 1, 0, 1,
0.1f);

                hw->swapBuffers(getHwComposer());
            }
        }
    }

    postFramebuffer();
    //
    // if (mDebugRegion > 1) {
    //     usleep(mDebugRegion * 100);
    // }
    //
    // for (size_t displayId = 0; displayId < mDisplays.size();
++displayId) {
    //     auto& displayDevice = mDisplays[displayId];
    //     if (!displayDevice->isDisplayOn()) {
    //         continue;
    //     }
    //
    //     status_t result = displayDevice->prepareFrame(*mHwc);
    //     ALOGE_IF(result != NO_ERROR, "prepareFrame for display %zd
failed:"
    //             " %d (%s)", displayId, result, strerror(-result));
    // }
```



图层显示如上所示截图，但是实际ADB从IO管道流读取到的图片如下所示：



这个实验可以验证图层显示的遮罩，doDebug只是在上面刷了一层，实际上Androidframebuffer做的动画叠层，极有可能推断出是在surfaceFlinger上实现的，Android实际上只是对局部做了刷新。底下只有一个图层framebuffer，这也可以说明一个侧边的说明为什么要alloc 3个framebuffer作为缓存。至于为什么显示的图层没有透明度?

举下一个例子好了，在surfaceView创建一个，不设置pixelType，在native层解析这个surface，可以发现一点，实际上默认的编码格式是RGB565。具体代码如下：

```
//fixme 默认采用了RGB 565的输出格式，我这边使用了ARGB的格式做的例子
JNIEXPORT void JNICALL
Java_com_genesis_frameworktest_NativeFunc_nDealSurface(JNIEnv*
env, jclass type, jobject surface)
{

    ANativeWindow* nativeWindow = ANativeWindow_fromSurface(env,
surface);
    ARect        * rect         = (ARect*) malloc(sizeof(ARect));
```

```
    ANativeWindow_Buffer buffer;

    int count = 0x00;
//    while (1)
    {
        count++;
        ANativeWindow_lock(nativeWindow, &buffer, rect);
        LOGI("RECT bottom %d,left : %d top : %d right :%d", rect-
>bottom, rect->left, rect->top,
             rect->right);
        LOGI("buffer width %d,height : %d format : %d, sizeof
buffers  %d,%p", buffer.width,
             buffer.height, buffer.format, sizeof(buffer.bits),
buffer.bits);

        int height = buffer.height;
        int width  = buffer.width;
        ANativeWindow_setBuffersGeometry(nativeWindow, width,
height, buffer.format);
        char* data = (char*) (buffer.bits);


        for (int i = 0; i < height; ++i)
        {
            for (int j = 0; j < width; ++j)
            {
                data[(i * width + j) * 4 + 0] = count % 256;
                data[(i * width + j) * 4 + 1] = count % 256;
                data[(i * width + j) * 4 + 2] = count % 256;
                count++;
            }
        }
//        memccpy(buffer.bits, potemp, sizeof(char), width *
height);

        ANativeWindow_unlockAndPost(nativeWindow);
        ANativeWindow_release(nativeWindow);
    }
}
```

- Layer

Android 在每个视图层创建的时候，都会create一个layer。传入对应的窗口大小。

```
Layer::Layer(SurfaceFlinger* flinger, const sp<Client>& client,
        const String8& name, uint32_t w, uint32_t h, uint32_t flags)
```

Layer 的和上层的layout有关，可以加入chiild layout，setPosition可以设置对应的位

置，长宽等等。可以说一下，Android系统中，任意一个页面，dialog都实质上是一个Layer。

Layer实际上就是一个图层缓存buffer和管理，在layer.cpp的ondraw方法打印，例如下面：

```
/*
* onDraw will draw the current layer onto the presentable buffer
*/
void Layer::onDraw(const sp<const DisplayDevice>& hw, const Region& clip,
        bool useIdentityTransform) const
{
    ATRACE_CALL();
    // ALOGW("Layer::onDraw");
    if (CC_UNLIKELY(mActiveBuffer == 0)) {
        // the texture has not been created yet, this Layer has
        // in fact never been drawn into. This happens frequently with
        // SurfaceView because the WindowManager can't know when the client
        // has drawn the first time.

        // If there is nothing under us, we paint the screen in black,
otherwise
        // we just skip this update.

        // figure out if there is something below us
        Region under;
        bool finished = false;
        mFlinger->mDrawingState.traverseInZOrder([&](Layer* layer) {
            if (finished || layer == static_cast<Layer const*>(this)) {
                finished = true;
                 ALOGW("Layer::onDmFlinger-
>mDrawingState.traverseInZOrder([&](Layer* layer) raw");
                return;
            }
            ALOGI("genesis Layer::onDraw  layer size %d ", layer-
>visibleRegion.isRect());
            under.orSelf( hw->getTransform().transform(layer-
>visibleRegion) );
        });
        // if not everything below us is covered, we plug the holes!
        Region holes(clip.subtract(under));
        if (!holes.isEmpty()) {
          ALOGW("Layer::holes.isEmpty()");
            clearWithOpenGL(hw, 0, 0, 0, 1);
        }
        return;
    }

    // Bind the current buffer to the GL texture, and wait for it to be
    // ready for us to draw into.
    status_t err = mSurfaceFlingerConsumer->bindTextureImage();
    if (err != NO_ERROR) {
```

```
        ALOGW("onDraw: bindTextureImage failed (err=%d)", err);
        // Go ahead and draw the buffer anyway; no matter what we do the
screen
        // is probably going to have something visibly wrong.
    }

    bool blackOutLayer = isProtected() || (isSecure() && !hw->isSecure());

    RenderEngine& engine(mFlinger->getRenderEngine());

    if (!blackOutLayer) {
        // TODO: we could be more subtle with isFixedSize()
        const bool useFiltering = getFiltering() || needsFiltering(hw) ||
isFixedSize();

        // Query the texture matrix given our current filtering mode.
        float textureMatrix[16];
        mSurfaceFlingerConsumer->setFilteringEnabled(useFiltering);
        mSurfaceFlingerConsumer->getTransformMatrix(textureMatrix);

        if (getTransformToDisplayInverse()) {

            /*
             * the code below applies the primary display's inverse
transform to
             * the texture transform
             */
            uint32_t transform =
                    DisplayDevice::getPrimaryDisplayOrientationTransform();
            mat4 tr = inverseOrientation(transform);

            /**
             * TODO(b/36727915): This is basically a hack.
             *
             * Ensure that regardless of the parent transformation,
             * this buffer is always transformed from native display
             * orientation to display orientation. For example, in the case
             * of a camera where the buffer remains in native orientation,
             * we want the pixels to always be upright.
             */
            sp<Layer> p = mDrawingParent.promote();
            if (p != nullptr) {
                const auto parentTransform = p->getTransform();
                tr = tr *
inverseOrientation(parentTransform.getOrientation());
            }

            // and finally apply it to the original texture matrix
            const mat4 texTransform(mat4(static_cast<const float*>
(textureMatrix)) * tr);
            memcpy(textureMatrix, texTransform.asArray(),
```

```
sizeof(textureMatrix));
        }

        // Set things up for texturing.
        mTexture.setDimensions(mActiveBuffer->getWidth(), mActiveBuffer-
>getHeight());
        ALOGW("mTexture.setDimensions: width = %d ,height = %d ,layercount
%d, z: %d layerStack %d", mActiveBuffer->getWidth(), mActiveBuffer-
>getHeight(), mActiveBuffer-
>getLayerCount(),mCurrentState.z,mCurrentState.layerStack);
        mTexture.setFiltering(useFiltering);
        mTexture.setMatrix(textureMatrix);
        engine.setupLayerTexturing(mTexture);
    } else {
        engine.setupLayerBlackedOut();
    }
    drawWithOpenGL(hw, useIdentityTransform);
    engine.disableTexturing();
}
```

启动一个简单的Activity，如图页面：

实际上，整个页面layer可以分为三个部分，第一部分，是总体的一个图层layer，第二部分是红色的Actionbar，最后是下面navi这部分。故，在渲染方面，会存在一个实际一点的问题，就是会先渲染rootview，在渲染childview，实际上，在这里，我们需要明白一个概念，是layerstack。实际上一层layer，可以嵌套多个子layer，然后这些layer，都具有对应的一个GraphyiaBuffer作为缓存区。再将这些buffer集体写入到OpenGL渲染引擎，做硬件图层叠加c狐狸渲染。具体的流程打印可以看到：

```
2019-01-10 16:41:44.713 1387-1387/? W/Layer: gmTexture.setDimensions: width
= 480 ,height = 728 ,layercount 1, z: 21000 drawStatus 0 mCurrentState : 0,
name com.android.launcher/com.android.launcher2.Launcher#0
2019-01-10 16:41:44.725 1387-1387/? W/Layer: gmTexture.setDimensions: width
= 480 ,height = 36 ,layercount 1, z: 181000 drawStatus 0 mCurrentState : 0,
name StatusBar#0
2019-01-10 16:41:44.726 1387-1387/? W/Layer: gmTexture.setDimensions: width
= 480 ,height = 72 ,layercount 1, z: 231000 drawStatus 0 mCurrentState : 0,
name NavigationBar#0
```

```
2019-01-10 16:41:44.737 1387-1387/? W/Layer: gmTexture.setDimensions: width
= 1243 ,height = 1024 ,layercount 1, z: 11000 drawStatus 2 mCurrentState :
2, name com.android.systemui.ImageWallpaper#0
```

其中，z打印的是对应的type，权重。

Layer只有长和宽，还有一个是Region，从表面看起来应该是整个Layer的相对位置和范围，实际详细可以看源码UI下面的Region.cpp。

还有一点需要注意，Layer的长宽，不一定最大是屏幕的限制，如果你申请的是一个scrollview，那么系统会依据你的传入，自动构建好Layer的长宽，显示范围等等。实际在图层表现，可能只是屏幕的一角，实际在内存中，可能是一整副图层。

其次，注意，在Android9.0的源码发现，改变了这个设计，有可能是一个frame的设计。

以下测试需要打开调试框。编写一个最简单的Activity应用，如下代码所示：

```
package com.example.myapplication;

import android.content.Intent;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        findViewById(R.id.btntest).setOnClickListener(new
View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new
Intent(MainActivity.this,MainActivity.class);
                intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
                startActivity(intent);
            }
        });
    }
}
```

功能很简单，就是每次点击Activity的Btn就启动自生，观测图层打印信息：

```
// RecordingCanvas做的缓存OP记录
2018-12-18 17:11:09.348 2772-2772/com.example.myapplication
D/OpenGLRenderer: RecordingCanvas::addOp insertIndex :0
2018-12-18 17:11:09.350 2772-2772/com.example.myapplication
D/OpenGLRenderer: RecordingCanvas::addOp insertIndex :0
2018-12-18 17:11:09.352 2772-2772/com.example.myapplication
```

```
D/OpenGLRenderer: RecordingCanvas::addOp insertIndex :1
2018-12-18 17:11:09.353 2772-2789/com.example.myapplication
D/EGL_emulation: eglMakeCurrent: 0xa56383e0: ver 3 0 (tinfo 0xaceeefa0)
2018-12-18 17:11:09.362 2772-2772/com.example.myapplication
D/OpenGLRenderer: RecordingCanvas::addOp insertIndex :0
2018-12-18 17:11:09.363 2772-2772/com.example.myapplication
D/OpenGLRenderer: RecordingCanvas::addOp insertIndex :0
2018-12-18 17:11:09.363 2772-2772/com.example.myapplication
D/OpenGLRenderer: RecordingCanvas::addOp insertIndex :1
2018-12-18 17:11:09.380 2772-2789/com.example.myapplication
D/OpenGLRenderer: Current memory usage / total memory usage (bytes):
      TextureCache           103680 /  9216000
      Layers total          0 (numLayers = 0)
      RenderBufferCache           0 /   768000
      GradientCache               0 /  1048576
      PathCache                   0 /  1536000
      TessellationCache         456 /  1048576
      TextDropShadowCache         0 /   768000
      PatchCache                  0 /   131072
      FontRenderer A8          8623 /   409600
        A8   texture 0         8623 /   409600
      FontRenderer RGBA           0 /        0
      FontRenderer total       8623 /   409600
    Other:
      FboCache                    0 /        0
    Total memory usage:
      513736 bytes, 0.49 MB
//之后Layer做的内存申请
2018-12-18 17:11:09.680 1389-1509/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-18 17:11:09.681 1389-1389/? I/Layer: Genesis Layer w : 480 h :800
2018-12-18 17:11:09.681 1389-1389/? I/Layer:  genesis Layer::setBuffers w =
480 ,h =800
2018-12-18 17:11:09.681 1389-1509/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-18 17:11:09.682 1389-1509/? I/chatty: uid=1000(system)
Binder:1389_3 identical 3 lines
2018-12-18 17:11:09.683 1389-1509/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16

//图层等级，长宽等信息
2018-12-18 17:11:09.683 1389-1389/? W/Layer: mTexture.setDimensions: width
= 480 ,height = 800 ,layercount 1, z: 21020

//GraphicBufferAllocator  Alloc 内存空间
12-18 17:11:09.685 1389-1509/? E/GraphicBufferAllocator:  allocate (480 x
800) layerCount 1 format 1 usage 900
2018-12-18 17:11:09.685 1381-1454/? D/gralloc_ranchu: gralloc_alloc:
Creating ashmem region of size 1540096
2018-12-18 17:11:09.686 1389-1389/? W/Layer: mTexture.setDimensions: width
= 480 ,height = 36 ,layercount 1, z: 181000 layerStack 0
```

```
2018-12-18 17:11:09.686 1389-1389/? W/Layer: mTexture.setDimensions: width
= 480 ,height = 72 ,layercount 1, z: 231000 layerStack 0
2018-12-18 17:11:09.686 1389-1509/? E/GraphicBufferAllocator:  allocate
(480 x 800) layerCount 1 format 1 usage 900
2018-12-18 17:11:09.686 1381-1381/? D/gralloc_ranchu: gralloc_alloc:
Creating ashmem region of size 1540096
2018-12-18 17:11:09.693 1389-1509/? E/GraphicBufferAllocator:  allocate
(480 x 800) layerCount 1 format 1 usage 900
2018-12-18 17:11:09.693 1381-1381/? D/gralloc_ranchu: gralloc_alloc:
Creating ashmem region of size 1540096
2018-12-18 17:11:09.700 1389-1509/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-18 17:11:09.700 1389-1389/? W/Layer: mTexture.setDimensions: width
= 480 ,height = 800 ,layercount 1, z: 21020

//释放Caches
2018-12-18 17:11:09.903 1389-1442/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-18 17:11:09.908 2772-2789/com.example.myapplication
D/OpenGLRenderer: Flushing caches (mode 0)
2018-12-18 17:11:09.909 1389-1442/? E/SurfaceFlinger: SurfaceFlinger
SurfaceFlinger::onTransact code: 8,flags: 16
2018-12-18 17:11:09.909 1389-1442/? I/Layer:  genesis Layer::setPosition x
= 0.000000 ,y =4.000000 name
com.example.myapplication/com.example.myapplication.MainActivity#0
2018-12-18 17:11:09.910 1389-1389/? W/Layer: mTexture.setDimensions: width
```

测试点函数LOG：

```
status_t GraphicBufferAllocator::allocate(uint32_t width, uint32_t height,
        PixelFormat format, uint32_t layerCount, uint64_t usage,
        buffer_handle_t* handle, uint32_t* stride,
        uint64_t /*graphicBufferId*/, std::string requestorName)
{
    ATRACE_CALL();

    // make sure to not allocate a N x 0 or 0 x N buffer, since this is
    // allowed from an API stand-point allocate a 1x1 buffer instead.
    if (!width || !height)
        width = height = 1;

    // Ensure that layerCount is valid.
    if (layerCount < 1)
        layerCount = 1;

    Gralloc2::IMapper::BufferDescriptorInfo info = {};
    info.width = width;
    info.height = height;
    info.layerCount = layerCount;
    info.format = static_cast<Gralloc2::PixelFormat>(format);
    info.usage = usage;
```

```
    ALOGE(" allocate (%u x %u) layerCount %u format %d "
            "usage %" PRIx64 "",
            width, height, layerCount, format, usage
            );
    //Alloc FrameBuffer 缓存Cached width*height*4 channel
    Gralloc2::Error error = mAllocator->allocate(info, stride, handle);
    if (error == Gralloc2::Error::NONE) {
        Mutex::Autolock _l(sLock);
        KeyedVector<buffer_handle_t, alloc_rec_t>& list(sAllocList);
        uint32_t bpp = bytesPerPixel(format);
        alloc_rec_t rec;
        rec.width = width;
        rec.height = height;
        rec.stride = *stride;
        rec.format = format;
        rec.layerCount = layerCount;
        rec.usage = usage;
        rec.size = static_cast<size_t>(height * (*stride) * bpp);
        rec.requestorName = std::move(requestorName);
        list.add(*handle, rec);

        return NO_ERROR;
    } else {
        ALOGE("Failed to allocate (%u x %u) layerCount %u format %d "
                "usage %" PRIx64 ": %d",
                width, height, layerCount, format, usage,
                error);
        return NO_MEMORY;
    }
}
```

首先我们看Alloc的调用周期，如下log：

```
//开机调用surfaceflinger的Alloc buffer 创建一个
2018-12-18 19:25:51.715 1389-1389/? I//system/bin/surfaceflinger:
android::hardware::configstore::V1_0::ISurfaceFlingerConfigs::startGraphics
AllocatorService retrieved: 0
2018-12-18 19:25:51.874 1389-1389/? D/GraphicBufferAllocator:
GraphicBufferAllocator::GraphicBufferAllocator()

//开机动画图层，Alloc内存映射
2018-12-18 19:25:51.874 1389-1389/? D/GraphicBufferAllocator: status_t
GraphicBufferAllocator::allocate(uint32_t width, uint32_t
height,PixelFormat format, uint32_t layerCount, uint64_t
usage,buffer_handle_t* handle, uint32_t* stride,uint64_t
/*graphicBufferId*/, std::string requestorName)
2018-12-18 19:25:51.874 1389-1389/? E/GraphicBufferAllocator:  allocate
(480 x 800) layerCount 1 format 1 usage 1a00

//FramebufferSurface Alloc内存
2018-12-18 19:25:51.877 1389-1389/? D/GraphicBufferAllocator: status_t
GraphicBufferAllocator::allocate(uint32_t width, uint32_t
```

```
height,PixelFormat format, uint32_t layerCount, uint64_t
usage,buffer_handle_t* handle, uint32_t* stride,uint64_t
/*graphicBufferId*/, std::string requestorName)
2018-12-18 19:25:51.877 1389-1389/? I/GraphicBufferAllocator: GraphicBuffer
Alloc (480,800),layerCount 1, usage 6656,  request :FramebufferSurface
2018-12-18 19:25:51.877 1389-1389/? E/GraphicBufferAllocator:  allocate
(480 x 800) layerCount 1 format 1 usage 1a00
2018-12-18 19:25:51.879 1389-1389/? D/GraphicBufferAllocator:  the list
size 2

2018-12-18 19:25:51.879 1389-1389/? D/GraphicBufferAllocator: status_t
GraphicBufferAllocator::allocate(uint32_t width, uint32_t
height,PixelFormat format, uint32_t layerCount, uint64_t
usage,buffer_handle_t* handle, uint32_t* stride,uint64_t
/*graphicBufferId*/, std::string requestorName)

2018-12-18 19:25:51.879 1389-1389/? I/GraphicBufferAllocator: GraphicBuffer
Alloc (480,800),layerCount 1, usage 6656,  request :FramebufferSurface
2018-12-18 19:25:51.879 1389-1389/? E/GraphicBufferAllocator:  allocate
(480 x 800) layerCount 1 format 1 usage 1a00
2018-12-18 19:25:51.881 1389-1389/? D/GraphicBufferAllocator:  the list
size 3
2018-12-18 19:25:57.914 1389-1440/? D/GraphicBufferAllocator: status_t
GraphicBufferAllocator::allocate(uint32_t width, uint32_t
height,PixelFormat format, uint32_t layerCount, uint64_t
usage,buffer_handle_t* handle, uint32_t* stride,uint64_t
/*graphicBufferId*/, std::string requestorName)
```

这里的using，对应了渲染角色，如果整图是一个Bitmap,则是0x1a00,对应的是OpenGL的相关策略枚举，详细请看gl.xml.这里摘取相关部分代码：

```
<enum value="0x1904" name="GL_GREEN_NV"/>
<enum value="0x1905" name="GL_BLUE"/>
<enum value="0x1905" name="GL_BLUE_NV"/>
<enum value="0x1906" name="GL_ALPHA"/>
<enum value="0x1907" name="GL_RGB"/>
<enum value="0x1908" name="GL_RGBA"/>
<enum value="0x1909" name="GL_LUMINANCE"/>
<enum value="0x190A" name="GL_LUMINANCE_ALPHA"/>
    <unused start="0x1910" end="0x19FF" comment="Unused for PixelFormat"/>
<enum value="0x1A00" name="GL_BITMAP"/>
    <unused start="0x1A01" end="0x1AFF" comment="Unused for PixelType"/>
<enum value="0x1B00" name="GL_POINT"/>
<enum value="0x1B01" name="GL_LINE"/>
<enum value="0x1B02" name="GL_FILL"/>
    <unused start="0x1B03" end="0x1BFF" comment="Unused for PolygonMode"/>
<enum value="0x1C00" name="GL_RENDER"/>
<enum value="0x1C01" name="GL_FEEDBACK"/>
<enum value="0x1C02" name="GL_SELECT"/>
    <unused start="0x1C03" end="0x1CFF" comment="Unused for
RenderingMode"/>
<enum value="0x1D00" name="GL_FLAT"/>
```

```
<enum value="0x1D01" name="GL_SMOOTH"/>
    <unused start="0x1D02" end="0x1DFF" comment="Unused for ShadingModel"/>
<enum value="0x1E00" name="GL_KEEP"/>
<enum value="0x1E01" name="GL_REPLACE"/>
```

那么实际启动的时候，GraphicBuffer是如何分配如何申请的呢？做一个简单的实验：如上的Activity，在xml布局文件中，修改对应代码：

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <TextView
        android:layout_width="200sp"
        android:layout_height="300sp"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
    <Button
        android:layout_width="100sp"
        android:id="@+id/btntest"
        android:text="button click"
        android:layout_height="100sp" />

</android.support.constraint.ConstraintLayout>
```

在启动Activity实际看到的情况是如下打印：

```
2018-12-19 11:27:16.403 1389-1539/? D/GraphicBufferAllocator: status_t
GraphicBufferAllocator::allocate(uint32_t width, uint32_t
height,PixelFormat format, uint32_t layerCount, uint64_t
usage,buffer_handle_t* handle, uint32_t* stride,uint64_t
/*graphicBufferId*/, std::string requestorName)
2018-12-19 11:27:16.403 1389-1539/? I/GraphicBufferAllocator: GraphicBuffer
Alloc (480,800),layerCount 1, usage 2304,  request
:com.example.myapplication/com.example.myapplication.MainActivity#0
2018-12-19 11:27:16.403 1389-1539/? E/GraphicBufferAllocator:  allocate
(480 x 800) layerCount 1 format 1 usage 900
2018-12-19 11:27:16.404 1389-1539/? D/GraphicBufferAllocator:  the list
size 17,buffer size : 1536000
```

从上面LOG，我们可以看到，启动Activity加载View时，GraphicBufferAllocator会申请一块内存，即是整个FrameBuffer大小的内存空间。其次，Android为了便于在内存上管理，使用了IPC的方式去申请内存空间。可以理解是一个抽象的单例模式。并且这些内存Buffer

都保存在一个List当中。实际对上面能看到的只是一个地址指针。GraphicBufferAllocator实际上是一个单例的类，可以通过GraphicBufferAllocator::get()获取对象。

在调用上，surfaceFlinger执行postBuffer方法，会逐层去合计layer的buffer数据。

```
void SurfaceFlinger::postFramebuffer()
{
    ALOGW("SurfaceFlinger::postFramebuffer");
    ATRACE_CALL();
    ALOGV("postFramebuffer");

    const nsecs_t now = systemTime();
    mDebugInSwapBuffers = now;

    for (size_t displayId = 0; displayId < mDisplays.size(); ++displayId) {
        auto& displayDevice = mDisplays[displayId];
        if (!displayDevice->isDisplayOn()) {
            continue;
        }
        const auto hwcId = displayDevice->getHwcDisplayId();
        if (hwcId >= 0) {
            mHwc->presentAndGetReleaseFences(hwcId);
        }
        displayDevice->onSwapBuffersCompleted();
        displayDevice->makeCurrent(mEGLDisplay, mEGLContext);
        for (auto& layer : displayDevice->getVis·ibleLayersSortedByZ()) {
            //fixme GraphicBuffer Layer 缓存的显示
            const sp<GraphicBuffer>& buffer(
                                layer->getActiveBuffer());
            int32_t format = -1;
            String8 name("unknown");
            if (buffer != NULL) {
               format = buffer->getPixelFormat();
               name = layer->getName();
               ALOGE("surface flinger genesis test %d
%s",format,name.string());
            }
            else
            {
               ALOGE("surface flinger genesis test buffer is null ");
            }
            sp<Fence> releaseFence = Fence::NO_FENCE;
            if (layer->getCompositionType(hwcId) ==
HWC2::Composition::Client) {
                releaseFence = displayDevice-
>getClientTargetAcquireFence();
            } else {
                auto hwcLayer = layer->getHwcLayer(hwcId);
                releaseFence = mHwc->getLayerReleaseFence(hwcId, hwcLayer);
            }
            layer->onLayerDisplayed(releaseFence);
```

```
        }
        if (hwcId >= 0) {
            mHwc->clearReleaseFences(hwcId);
        }
    }

    mLastSwapBufferTime = systemTime() - now;
    mDebugInSwapBuffers = 0;

    // |mStateLock| not needed as we are on the main thread
    uint32_t flipCount = getDefaultDisplayDeviceLocked()-
>getPageFlipCount();
    if (flipCount % LOG_FRAME_STATS_PERIOD == 0) {
        logFrameStats();
    }
}
```

layer的概念中，还有一个概念是layerStacks的概念，这个概念实际上是图层的显示层级权值。每次渲染图层都会依据对应的layer type 实际上就是layer的z值大小，进行排序，最后混合成一个图层输出。我们关联一下surfaceflinger的 doComposeSurface方法中一段函数：

```
   const Transform& displayTransform = displayDevice->getTransform();
    if (hwcId >= 0) {
        // we're using h/w composer
        bool firstLayer = true;
        for (auto& layer : displayDevice->getVisibleLayersSortedByZ()) {
            const Region clip(dirty.intersect(
                    displayTransform.transform(layer->visibleRegion)));
            ALOGV("Layer: %s", layer->getName().string());
            ALOGV("  Composition type: %s",
                    to_string(layer->getCompositionType(hwcId)).c_str());
            if (!clip.isEmpty()) {
                switch (layer->getCompositionType(hwcId)) {
                    case HWC2::Composition::Cursor:
                    case HWC2::Composition::Device:
                    case HWC2::Composition::Sideband:
                    case HWC2::Composition::SolidColor: {
                        const Layer::State& state(layer-
>getDrawingState());
                        if (layer->getClearClientTarget(hwcId) &&
!firstLayer &&
                                layer->isOpaque(state) && (state.alpha ==
1.0f)
                                && hasClientComposition) {
                            // never clear the very first layer since we're
                            // guaranteed the FB is already cleared
                            layer->clearWithOpenGL(displayDevice);
                        }
                        break;
                    }
```

```
                    case HWC2::Composition::Client: {
                        layer->draw(displayDevice, clip);
                        break;
                    }
                    default:
                        break;
                }
            } else {
                ALOGV("  Skipping for empty clip");
            }
            firstLayer = false;
        }
    } else {
        // we're not using h/w composer
        for (auto& layer : displayDevice->getVisibleLayersSortedByZ()) {
            const Region clip(dirty.intersect(
                    displayTransform.transform(layer->visibleRegion)));
            if (!clip.isEmpty()) {
                layer->draw(displayDevice, clip);
            }
        }
    }
```

我们看到实际的这个layer显示默认会依据z值权重进行排序，然后一层一层的显示出来，这里再去执行layer的draw方法。这个我们大致可以得到的设想是，layer的渲染策略和surface的策略有关联。由于Android图层渲染是一个复杂的模块，存在软渲染和硬件渲染的区别，以现在的理解，hwCompose会依据不同的环境，分配不同的渲染策略。可能我们上层View在执行某些draw方法的时候，或者定义某些渲染策略的时候，需要选择适合的rootLayer来包含，才能发挥最好的机器性能。

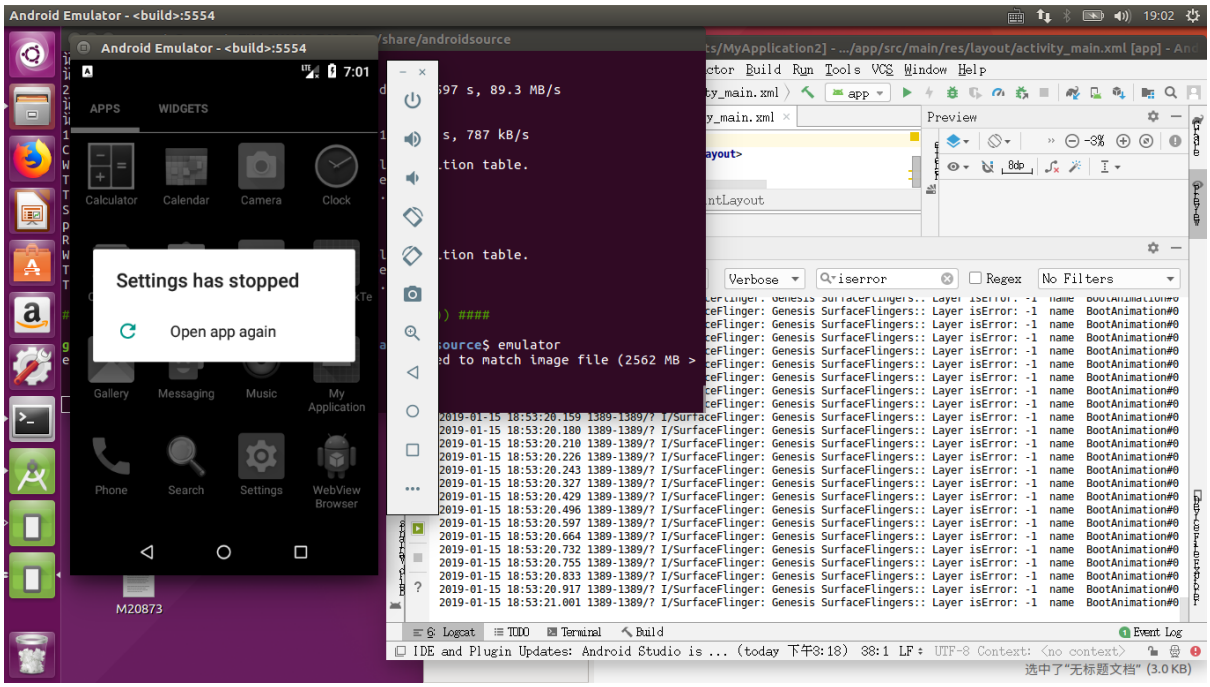
那么，surfaceFlinger和layer的关系是什么?

我们看下layer在surfaceFlinger中调用的一段代码：

```
bool SurfaceFlinger::doComposeSurfaces(
        const sp<const DisplayDevice>& displayDevice, const Region& dirty)
{
    ALOGW("SurfaceFlinger::doComposeSurfaces");
    ALOGV("doComposeSurfaces");

    const auto hwcId = displayDevice->getHwcDisplayId();

    mat4 oldColorMatrix;
    const bool applyColorMatrix = !mHwc->hasDeviceComposition(hwcId) &&
            !mHwc-
>hasCapability(HWC2::Capability::SkipClientColorTransform);
    if (applyColorMatrix) {
        mat4 colorMatrix = mColorMatrix * mDaltonizer();
        oldColorMatrix =
```

```
getRenderEngine().setupColorTransform(colorMatrix);
    }
    ....

        for (auto& layer : displayDevice->getVisibleLayersSortedByZ()) {
            const Region clip(dirty.intersect(
                    displayTransform.transform(layer->visibleRegion)));
            ALOGI("SurfaceFlinger:: genesis Layer: %s  size : %d", layer-
>getName().string(),defcount);
            ALOGI("SurfaceFlinger::genesis Composition type: %s",
                    to_string(layer->getCompositionType(hwcId)).c_str());
            if (!clip.isEmpty()) {
                switch (layer->getCompositionType(hwcId)) {
                    case HWC2::Composition::Cursor:
                    case HWC2::Composition::Device:
                    case HWC2::Composition::Sideband:
                    case HWC2::Composition::SolidColor: {
                        const Layer::State& state(layer-
>getDrawingState());
                        if (layer->getClearClientTarget(hwcId) &&
!firstLayer &&
                                layer->isOpaque(state) && (state.alpha ==
1.0f)
                                && hasClientComposition) {
                            // never clear the very first layer since we're
                            // guaranteed the FB is already cleared
                            layer->clearWithOpenGL(displayDevice);
                        }
                        break;
                    }
                    case HWC2::Composition::Client: {
                        layer->draw(displayDevice, clip);
                          layerZ = layer->getCurrentState().z;
                        bool isDebug = true;
            …..
```

如上代码段中是对每一个图层进行绘制的代码段，surfaceFlinger会对每一层layer进行绘图，然后生成一帧图片，在通过layer合成到framebuffer上输出。如果在这里对每一层图进行处理，我们如果以launcher和dialog框为例，出现dialog框，则背景制成灰度图输出。则如下的效果：



注意：这里的layer可能不仅仅是3层图，可能包括了堆栈的缓存cache，动画过场过场中，都会存在这一图层，并且，有些效果图层，他的GraphicBuffer可能为空，如果随意使用，可能会导致系统渲染模块重新启动。

- GraphicBuffer

在GraphicBufferAllocator上层是一个GraphicBuffer，每次Activity启动，都会调用一下

```
GraphicBuffer::GraphicBuffer()
    : BASE(), mOwner(ownData), mBufferMapper(GraphicBufferMapper::get()),
      mInitCheck(NO_ERROR), mId(getUniqueId()), mGenerationNumber(0)
```

这个构造方法，然后执行init这个初始化方法。

```
status_t GraphicBuffer::initWithSize(uint32_t inWidth, uint32_t inHeight,
        PixelFormat inFormat, uint32_t inLayerCount, uint64_t inUsage,
        std::string requestorName)
```

这个方法再去AllocGraphicBuffer，申请相对整屏幕的buffer。GraphicBuffer提供了一个getNativeBuffer的方法，用以获取整个屏幕的buffer缓存区，缓存的buffer也封装成了一个ANativeWindowBuffer的结构体。定义如下：

```
//获取缓存方法定义
ANativeWindowBuffer* GraphicBuffer::getNativeBuffer() const

//ANativeWindowBuffer定义
typedef struct ANativeWindowBuffer
{
    #ifdef __cplusplus
```

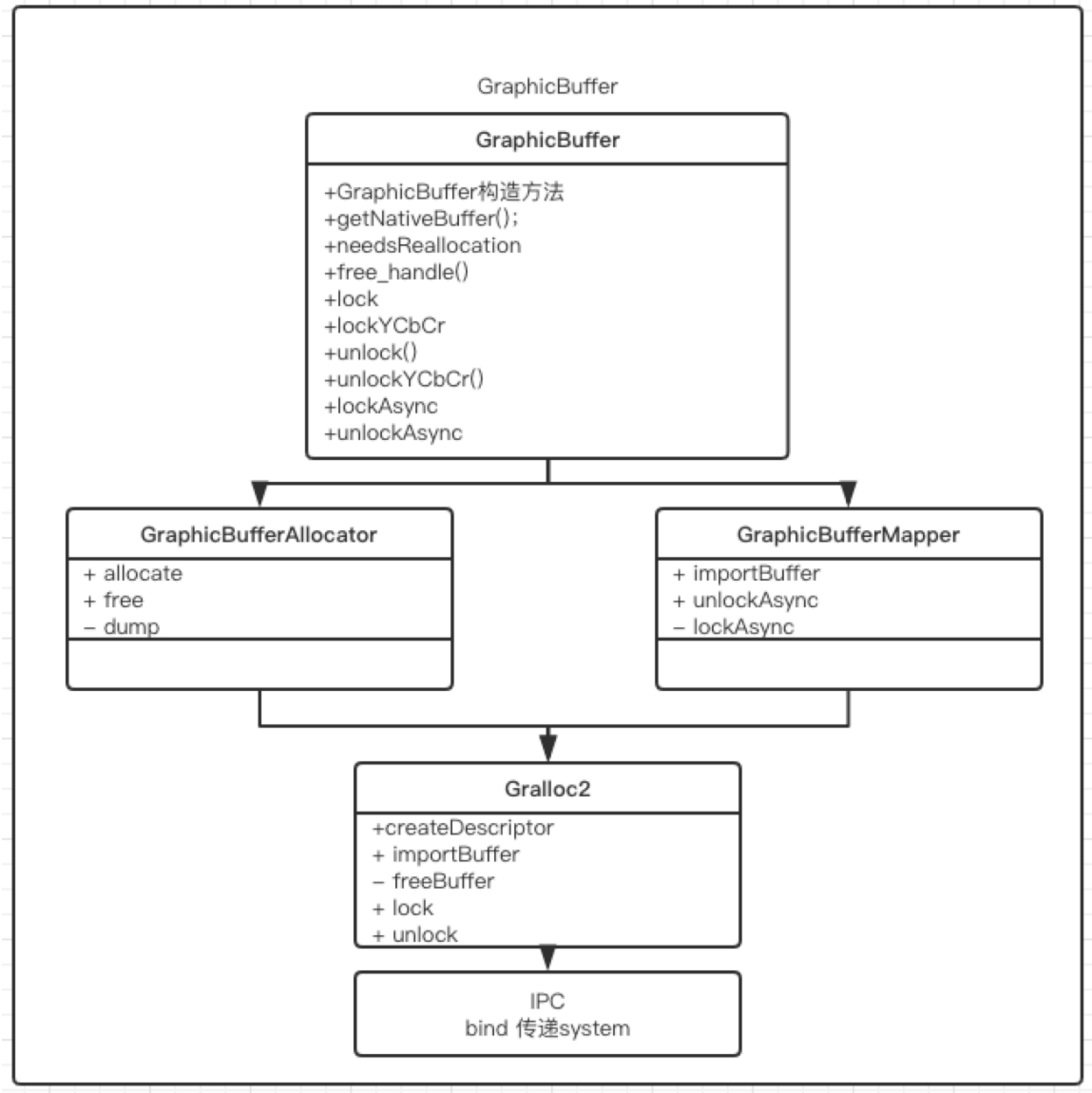
```
    ANativeWindowBuffer() {
        common.magic = ANDROID_NATIVE_BUFFER_MAGIC;
        common.version = sizeof(ANativeWindowBuffer);
        memset(common.reserved, 0, sizeof(common.reserved));
    }

    // Implement the methods that sp<ANativeWindowBuffer> expects so that
it
    // can be used to automatically refcount ANativeWindowBuffer's.
    void incStrong(const void* /*id*/) const {
        common.incRef(const_cast<android_native_base_t*>(&common));
    }
    void decStrong(const void* /*id*/) const {
        common.decRef(const_cast<android_native_base_t*>(&common));
    }
#endif

    struct android_native_base_t common;

    int width;
    int height;
    int stride;
    int format;
    int usage_deprecated;
    uintptr_t layerCount;

    void* reserved[1];

    const native_handle_t* handle;
    uint64_t usage;

    // we needed extra space for storing the 64-bits usage flags
    // the number of slots to use from reserved_proc depends on the
    // architecture.
    void* reserved_proc[8 - (sizeof(uint64_t) / sizeof(void*))];
} ANativeWindowBuffer_t;
```

在GraphicBufferAllocator下面，还有一层Gralloc2的封装，里面提供了IPC和系统服务进行操作的接口方法。在几个方法中加入LOG，打印调度信息：

```
2018-12-19 15:15:33.305 1388-1430/? I/Gralloc2:
Graphic  Mapper::createDescriptor(
2018-12-19 15:15:33.305 1388-1430/? I/Gralloc2:
Graphic  Allocator::allocate
2018-12-19 15:15:33.307 1388-1430/? I/Gralloc2:
Graphic  Mapper::importBuffer(
2018-12-19 15:15:33.307 1388-1430/? I/Gralloc2:
Graphic  Mapper::createDescriptor(
2018-12-19 15:15:33.307 1388-1430/? I/Gralloc2:
Graphic  Allocator::allocate
2018-12-19 15:15:33.308 1388-1430/? I/Gralloc2:
```

```
Graphic  Mapper::importBuffer(
2018-12-19 15:15:33.308 1388-1430/? I/Gralloc2:
Graphic  Mapper::createDescriptor(
2018-12-19 15:15:33.308 1388-1430/? I/Gralloc2:
Graphic  Allocator::allocate
2018-12-19 15:15:33.311 1388-1430/? I/Gralloc2:
Graphic  Mapper::importBuffer(
2018-12-19 15:15:33.312 2335-2356/com.example.myapplication I/Gralloc2:
Graphic  Mapper::importBuffer(
2018-12-19 15:15:33.354 2335-2356/com.example.myapplication I/Gralloc2:
Graphic  Mapper::importBuffer(
2018-12-19 15:15:33.432 2335-2356/com.example.myapplication I/Gralloc2:
Graphic  Mapper::importBuffer(
```

可以看到，每次new一个GraphicBuffer,都会create一个Mapper的映射，然后申请相关的内存空间，最后如果填充，则调用importBuffer方法。从代码结构上可以看到GraphicBuffer模块的结构大体上如下：



每次启动一个页面，都会申请一块GraphicBuffer的内存区块，GraphicBuffer通过GraphicBufferAllocator去申请，GraphicBufferMapper想对应，最底下的是Gralloc2做同步、申请等操作。并且，每一块都拥有一个操作符来限定对共享内存操作的用途。

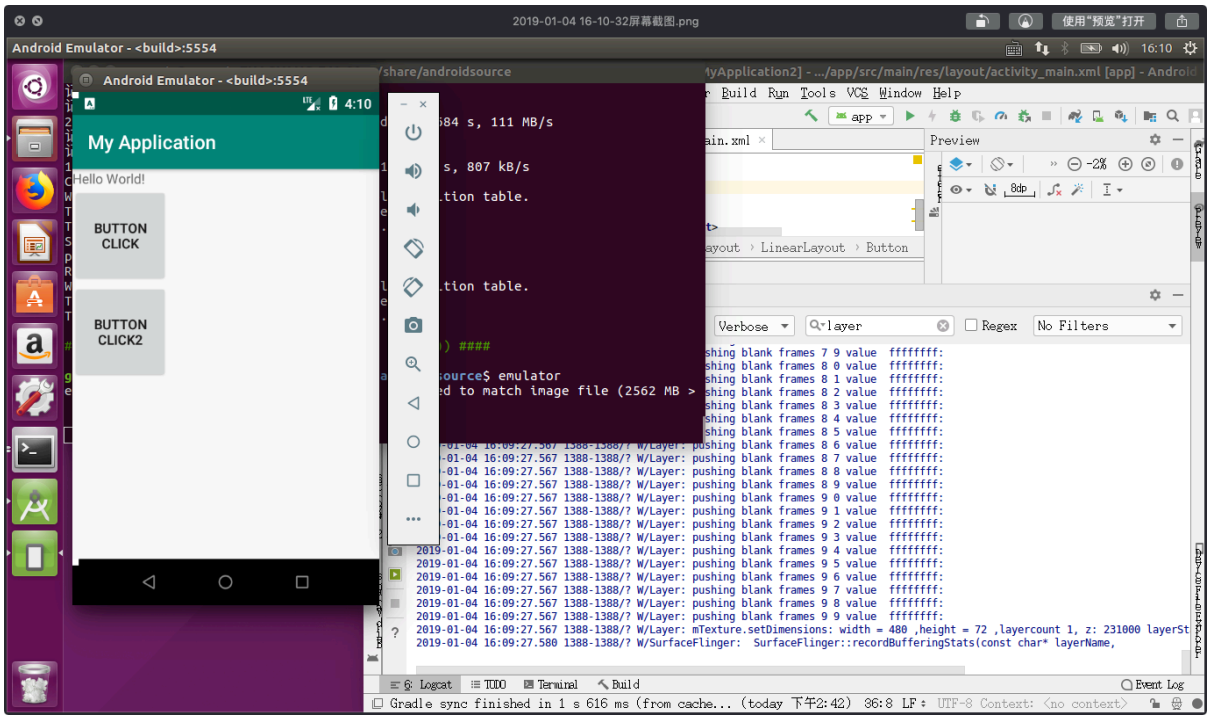
```
enum {
    USAGE_SW_READ_NEVER     = GRALLOC_USAGE_SW_READ_NEVER,
    USAGE_SW_READ_RARELY    = GRALLOC_USAGE_SW_READ_RARELY,
    USAGE_SW_READ_OFTEN     = GRALLOC_USAGE_SW_READ_OFTEN,
    USAGE_SW_READ_MASK      = GRALLOC_USAGE_SW_READ_MASK,

    USAGE_SW_WRITE_NEVER    = GRALLOC_USAGE_SW_WRITE_NEVER,
    USAGE_SW_WRITE_RARELY   = GRALLOC_USAGE_SW_WRITE_RARELY,
    USAGE_SW_WRITE_OFTEN    = GRALLOC_USAGE_SW_WRITE_OFTEN,
    USAGE_SW_WRITE_MASK     = GRALLOC_USAGE_SW_WRITE_MASK,

    USAGE_SOFTWARE_MASK     =
USAGE_SW_READ_MASK|USAGE_SW_WRITE_MASK,

    USAGE_PROTECTED         = GRALLOC_USAGE_PROTECTED,

    USAGE_HW_TEXTURE        = GRALLOC_USAGE_HW_TEXTURE,
    USAGE_HW_RENDER         = GRALLOC_USAGE_HW_RENDER,
    USAGE_HW_2D             = GRALLOC_USAGE_HW_2D,
    USAGE_HW_COMPOSER       = GRALLOC_USAGE_HW_COMPOSER,
    USAGE_HW_VIDEO_ENCODER  = GRALLOC_USAGE_HW_VIDEO_ENCODER,
    USAGE_HW_MASK           = GRALLOC_USAGE_HW_MASK,

    USAGE_CURSOR            = GRALLOC_USAGE_CURSOR,
};

GraphicBuffer();
```

GraphicBuffer的概念和位图的概念有点像。这里的GraphicBuffer实际上提供了图片的长宽，编解码格式等。由于这个内存是通过IPC去申请的共享内存段，因此，每次去更新这个图的时候，都会需要去lock和unlock一下，才能修改整个内存区域的数据。并且，在Android系统的设计里面，每一层layer，每一层surface都对应了一个GraphicBuffer，这个是SKia引擎绘图最终的输出容器。在layer做一个实验，添加代码如下：

```
        int dWidth =  mActiveBuffer->getWidth();
        int dHeight =  mActiveBuffer->getHeight();
        if(dWidth>10&&dHeight>10)
        {
          uint32_t *img = NULL;
          err = mActiveBuffer-
>lock(GRALLOC_USAGE_SW_WRITE_OFTEN|GRALLOC_USAGE_SW_READ_OFTEN, (void**)
(&img));
          if (err != NO_ERROR)  {
              ALOGE("error pushing blank frames: lock failed: %s (%d)",
```

```
strerror(-err), -err);
          }
          else
          {
            // ALOGW("pushing blank frames:");
            for (int i = 0; i < 10; i++) {
              for (int j= 0;j<10;j++ ) {
                //img实际上对应的是每一层layer成像的像素值，这里渲染成形后默认的是
ARGB格式，和nativeWindow不同
                ALOGW("pushing blank frames %d %d
value  %x:",i,j,img[i*dWidth +j ]);
                img[i*dWidth +j ]= 0xFFFFFFFF;
              }
            }
          }
        }
        //同步到GraphicBuffer实际的内存地址
        mActiveBuffer->unlock();
        // Set things up for texturing.
        mTexture.setDimensions(mActiveBuffer->getWidth()/2, mActiveBuffer-
>getHeight());
        ALOGW("mTexture.setDimensions: width = %d ,height = %d ,layercount
%d, z: %d layerStack %d", mActiveBuffer->getWidth(), mActiveBuffer-
>getHeight(), mActiveBuffer-
>getLayerCount(),mCurrentState.z,mCurrentState.layerStack);
        mTexture.setFiltering(useFiltering);
        mTexture.setMatrix(textureMatrix);
        engine.setupLayerTexturing(mTexture);
```

如上所示的代码，原则上输出会在每一层layer输出的时候，把首位10*10像素的值变成白色。我们运行模拟器后，会发现和实际预期的结果类似：



我们可以看到图上对应的首位10*10和代码的描述一致，变成了0xffffffff的像素值。也即是白色，如此所述的一个设计，可以在对应的图层上，做任何动画，用CL在GPU运算，可以做任何的图层变换效果。

值得注意的是，layer局部刷新，可能不会重新去调Skia渲染图形，如果是全局刷新，则可以执行，如果是局部，会直接copy内存段合成整张SKbitmap。因为GLConsumer并不会每次都会去绘制一张图片，具体看调用情况，如下：

在SurfaceFlingerConsumer中的

```
status_t SurfaceFlingerConsumer::bindTextureImage()
{
    Mutex::Autolock lock(mMutex);

    return bindTextureImageLocked();
}
```

其调用的是GLConsumer中的方法

```
status_t GLConsumer::bindTextureImageLocked() {
    if (mEglDisplay == EGL_NO_DISPLAY) {
        ALOGE("bindTextureImage: invalid display");
        return INVALID_OPERATION;
    }

    GLenum error;
    while ((error = glGetError()) != GL_NO_ERROR) {
        GLC_LOGW("bindTextureImage: clearing GL error: %#04x", error);
    }
```

```
    glBindTexture(mTexTarget, mTexName);
    if (mCurrentTexture == BufferQueue::INVALID_BUFFER_SLOT &&
            mCurrentTextureImage == NULL) {
        GLC_LOGE("bindTextureImage: no currently-bound texture");
        return NO_INIT;
    }

    status_t err = mCurrentTextureImage->createIfNeeded(mEglDisplay,
                                                        mCurrentCrop);
    if (err != NO_ERROR) {
        GLC_LOGW("bindTextureImage: can't cre

    ……
```

这里的createIfNeeded方法中，会对之前的create状态进行判断，如果没有缓存图，才会重新绘图，否则不会：

```
status_t GLConsumer::EglImage::createIfNeeded(EGLDisplay eglDisplay,
                                              const Rect& cropRect,
                                              bool forceCreation) {
    // If there's an image and it's no longer valid, destroy it.
    bool haveImage = mEglImage != EGL_NO_IMAGE_KHR;
    bool displayInvalid = mEglDisplay != eglDisplay;
    bool cropInvalid = hasEglAndroidImageCrop() && mCropRect != cropRect;
    if (haveImage && (displayInvalid || cropInvalid || forceCreation)) {
        if (!eglDestroyImageKHR(mEglDisplay, mEglImage)) {
           ALOGE("createIfNeeded: eglDestroyImageKHR failed");
        }
        eglTerminate(mEglDisplay);
        mEglImage = EGL_NO_IMAGE_KHR;
        mEglDisplay = EGL_NO_DISPLAY;
    }

    // If there's no image, create one.
    if (mEglImage == EGL_NO_IMAGE_KHR) {
        mEglDisplay = eglDisplay;
        mCropRect = cropRect;
        mEglImage = createImage(mEglDisplay, mGraphicBuffer, mCropRect);
    }

    // Fail if we can't create a valid image.
    if (mEglImage == EGL_NO_IMAGE_KHR) {
        mEglDisplay = EGL_NO_DISPLAY;
        mCropRect.makeInvalid();
        const sp<GraphicBuffer>& buffer = mGraphicBuffer;
        ALOGE("Failed to create image. size=%ux%u st=%u usage=0x%x fmt=%d",
            buffer->getWidth(), buffer->getHeight(), buffer->getStride(),
            buffer->getUsage(), buffer->getPixelFormat());
        return UNKNOWN_ERROR;
    }
```

```
    return OK;
}
```

- DisplayDevice

这个模块好比是页面生成器，就是最后生成一个SKBitmap，这个模块可以想象是一个页面构建层级的东西。着重注意几个方法：

```
// We pass in mustRecompose so we can keep VirtualDisplaySurface's state
// machine happy without actually queueing a buffer if nothing has changed

status_t beginFrame(bool mustRecompose) const;
status_t prepareFrame(const HWComposer& hwc) const;

void swapBuffers(HWComposer& hwc) const;
status_t compositionComplete() const;

// called after h/w composer has completed its set() call
void onSwapBuffersCompleted(HWComposer& hwc) const;

Rect getBounds() const {
    return Rect(mDisplayWidth, mDisplayHeight);
}
inline Rect bounds() const { return getBounds(); }

void setDisplayName(const String8& displayName);
const String8& getDisplayName() const { return mDisplayName; }

EGLBoolean makeCurrent(EGLDisplay dpy, EGLContext ctx) const;
void setViewportAndProjection() const;
```

在实际的渲染最后，都会走到onSwapBuffers这个方法，这个方法是在HWComposer中的渲染策略，生成一个GraphicBuffer。然后再将数据流输出到Hardware上去。在相关代码上加入探针Log:

```
void DisplayDevice::swapBuffers(HWComposer& hwc) const {
    ALOGE("DisplayDevice::swapBuffers(HWComposer& hwc)");
    // hwc2
#ifdef USE_HWC2
    if (hwc.hasClientComposition(mHwcDisplayId)) {
#else
    // We need to call eglSwapBuffers() if:
    //  (1) we don't have a hardware composer, or
    //  (2) we did GLES composition this frame, and either
    //    (a) we have framebuffer target support (not present on legacy
```

```
    //        devices, where HWComposer::commit() handles things); or
    //    (b) this is a virtual display
    if (hwc.initCheck() != NO_ERROR ||
            (hwc.hasGlesComposition(mHwcDisplayId) &&
             (hwc.supportsFramebufferTarget() || mType >=
DISPLAY_VIRTUAL))) {
#endif
        //对应的surface和display做渲染。
        EGLBoolean success = eglSwapBuffers(mDisplay, mSurface);
        if (!success) {
            EGLint error = eglGetError();
            if (error == EGL_CONTEXT_LOST ||
                    mType == DisplayDevice::DISPLAY_PRIMARY) {
                LOG_ALWAYS_FATAL("eglSwapBuffers(%p, %p) failed with
0x%08x",
                        mDisplay, mSurface, error);
            } else {
                ALOGE("eglSwapBuffers(%p, %p) failed with 0x%08x",
                        mDisplay, mSurface, error);
            }
        }
    }
    ALOGE("DisplayDevice::swapBuffers(HWComposer& hwc) mDisplaySurface-
>advanceFrame()”);
    //配置frame或者输出frame
    status_t result = mDisplaySurface->advanceFrame();
    if (result != NO_ERROR) {
        ALOGE("[%s] failed pushing new frame to HWC: %d",
                mDisplayName.string(), result);
    }
}
```

再看下advanceFrame方法，我们找到frameBufferSurface方法，具体定义如下：

```
status_t FramebufferSurface::advanceFrame() {
    ALOGE("status_t FramebufferSurface::advanceFrame() #in");
#ifdef USE_HWC2
    ALOGE("status_t FramebufferSurface::advanceFrame() #in 2");
    uint32_t slot = 0;
    sp<GraphicBuffer> buf;
    sp<Fence> acquireFence(Fence::NO_FENCE);
    android_dataspace_t dataspace = HAL_DATASPACE_UNKNOWN;
    //实际处理方法
    status_t result = nextBuffer(slot, buf, acquireFence, dataspace);
    if (result != NO_ERROR) {
        ALOGE("error latching next FramebufferSurface buffer: %s (%d)",
                strerror(-result), result);
    }
    return result;
#else
    // Once we remove FB HAL support, we can call nextBuffer() from here
    // instead of using onFrameAvailable(). No real benefit, except it'll
```

```
be
    // more like VirtualDisplaySurface.
    return NO_ERROR;
#endif
}

//处理方法
#ifdef USE_HWC2
status_t FramebufferSurface::nextBuffer(uint32_t& outSlot,
        sp<GraphicBuffer>& outBuffer, sp<Fence>& outFence,
        android_dataspace_t& outDataspace) {
#else
status_t FramebufferSurface::nextBuffer(sp<GraphicBuffer>& outBuffer,
sp<Fence>& outFence) {
#endif
    Mutex::Autolock lock(mMutex);

    BufferItem item;
    status_t err = acquireBufferLocked(&item, 0);

    ALOGW("acquireBufferLocked(&item, 0);");
    if (err == BufferQueue::NO_BUFFER_AVAILABLE) {
#ifdef USE_HWC2
        //获取当前图层
        mHwcBufferCache.getHwcBuffer(mCurrentBufferSlot, mCurrentBuffer,
                &outSlot, &outBuffer);
        ALOGW("mHwcBufferCache.getHwcBuffer(mCurrentBufferSlot,
mCurrentBuffer,&outSlot, &outBuffer);");
#else
        ALOGW("outBuffer = mCurrentBuffer;");
        outBuffer = mCurrentBuffer;
#endif
        return NO_ERROR;
    } else if (err != NO_ERROR) {
        ALOGE("error acquiring buffer: %s (%d)", strerror(-err), err);
        return err;
    }

    // If the BufferQueue has freed and reallocated a buffer in
mCurrentSlot
    // then we may have acquired the slot we already own.  If we had
released
    // our current buffer before we call acquireBuffer then that release
call
    // would have returned STALE_BUFFER_SLOT, and we would have called
    // freeBufferLocked on that slot.  Because the buffer slot has already
    // been overwritten with the new buffer all we have to do is skip the
    // releaseBuffer call and we should be in the same state we'd be in if
we
    // had released the old buffer first.
    if (mCurrentBufferSlot != BufferQueue::INVALID_BUFFER_SLOT &&
```

```
        item.mSlot != mCurrentBufferSlot) {
#ifdef USE_HWC2
        mHasPendingRelease = true;
        mPreviousBufferSlot = mCurrentBufferSlot;
        mPreviousBuffer = mCurrentBuffer;
#else
        // Release the previous buffer.
        err = releaseBufferLocked(mCurrentBufferSlot, mCurrentBuffer,
                EGL_NO_DISPLAY, EGL_NO_SYNC_KHR);
        if (err < NO_ERROR) {
            ALOGE("error releasing buffer: %s (%d)", strerror(-err), err);
            return err;
        }
#endif
    }
    ALOGE("mCurrentBufferSlot = item.mSlot;");
    mCurrentBufferSlot = item.mSlot;
    mCurrentBuffer = mSlots[mCurrentBufferSlot].mGraphicBuffer;
    mCurrentFence = item.mFence;

    outFence = item.mFence;


#ifdef USE_HWC2
    ALOGE("mHwcBufferCache.getHwcBuffer(mCurrentBufferSlot,");

    mHwcBufferCache.getHwcBuffer(mCurrentBufferSlot, mCurrentBuffer,
            &outSlot, &outBuffer);
    //测试代码，有坑，outBuffer可能为空
    if(outBuffer!=NULL)
    {
        //获取NativeWindowBuffer
       ANativeWindowBuffer* out = outBuffer->getNativeBuffer();
       ALOGI("BUFFER IS NULL %d",out==NULL);
    }
    else
    {
         ALOGI("BUFFER IS NULL");
    }

    outDataspace = item.mDataSpace;
    status_t result =
            mHwc.setClientTarget(mDisplayType, outSlot, outFence,
outBuffer, outDataspace);
    if (result != NO_ERROR) {
        ALOGE("error posting framebuffer: %d", result);
        return result;
    }
#else
    outBuffer = mCurrentBuffer;
#endif
```

```
    return NO_ERROR;
}
```

注意一点是，有些版本Android可能不再这个文件中，而且，这个文件中的调用方式有点类似孔和槽的概念。熟悉操作系统、玩过QT的同学应该不是很陌生。

- 关于页面刷新

假定surface的某个layer发生变更，假定有三个layer，naviBar、statusBar和lunchBar三个，如果刷新了statusBar，在surfaceFlinger会发生怎么样的动作呢?

```
2018-12-28 14:23:18.865 1465-1465/? I/zygote: option[47]=-
Xfingerprint:Android/aosp_x86/generic_x86:8.1.0/OPM7.181205.001/genesi12171
521:eng/test-keys
2018-12-28 14:23:18.892 1476-1484/? I/ServiceManager: Waiting for service
package_native...
2018-12-28 14:23:19.454 1465-1465/? W/zygote: Could not reserve sentinel
fault page
2018-12-28 14:23:19.893 1476-1484/? I/ServiceManager: Waiting for service
package_native...
2018-12-28 14:23:19.998 1456-1505/? I/Surface: Surface::queueBuffer
```

```
2018-12-28 14:23:19.999 1456-1505/? I/Surface: Surface::dequeueBuffer
2018-12-28 14:23:19.999 1389-1442/?
I/GraphicBuffer:  GraphicBuffer::GraphicBuffer() BASE(), mOwner(ownData),
2018-12-28 14:23:19.999 1389-1442/?
I/GraphicBuffer:  GraphicBuffer::iinitWithSize(uint32_t inWidth, uint32_t
inHeight,
2018-12-28 14:23:19.999 1389-1442/? D/GraphicBufferAllocator: status_t
GraphicBufferAllocator::allocate(uint32_t width, uint32_t
height,PixelFormat format, uint32_t layerCount, uint64_t
usage,buffer_handle_t* handle, uint32_t* stride,uint64_t
/*graphicBufferId*/, std::string requestorName)
2018-12-28 14:23:19.999 1389-1442/? I/GraphicBufferAllocator: GraphicBuffer
Alloc (480,800),layerCount 1, usage 2304,  request :BootAnimation#0
2018-12-28 14:23:19.999 1389-1442/? E/GraphicBufferAllocator:  allocate
(480 x 800) layerCount 1 format 2 usage 900
2018-12-28 14:23:19.999 1389-1442/? I/Gralloc2:
Graphic  Mapper::createDescriptor(
2018-12-28 14:23:19.999 1389-1442/? I/Gralloc2:
Graphic  Allocator::allocate
2018-12-28 14:23:19.999 1381-1453/? D/gralloc_ranchu: gralloc_alloc:
Creating ashmem region of size 1540096
2018-12-28 14:23:20.001 1389-1442/? I/Gralloc2:
Graphic  Mapper::importBuffer(
2018-12-28 14:23:20.002 1389-1389/? W/SurfaceFlinger:
SurfaceFlinger::handlePageFlip
2018-12-28 14:23:20.002 1389-1389/?
I/GraphicBuffer:  GraphicBuffer::getNativeBuffer()
2018-12-28 14:23:20.002 1389-1389/? W/SurfaceFlinger:
SurfaceFlinger::invalidateLayerStack
2018-12-28 14:23:20.002 1389-1389/? W/SurfaceFlinger:
SurfaceFlinger::preComposition
2018-12-28 14:23:20.002 1389-1389/? W/SurfaceFlinger:
SurfaceFlinger::rebuildLayerStacks
2018-12-28 14:23:20.002 1389-1389/? W/SurfaceFlinger:
SurfaceFlinger::computeVisibleRegions
2018-12-28 14:23:20.002 1389-1389/? W/SurfaceFlinger:
SurfaceFlinger::setUpHWComposer
2018-12-28 14:23:20.002 1389-1442/? D/GraphicBufferAllocator:  the list
size 5,buffer size : 1536000
2018-12-28 14:23:20.002 1389-1389/? W/SurfaceFlinger:
SurfaceFlinger::computeSaturationMatrix
2018-12-28 14:23:20.002 1389-1389/?
I/GraphicBuffer:  GraphicBuffer::getNativeBuffer()
2018-12-28 14:23:20.003 1389-1508/? I/GraphicBuffer: GraphicBuffer::flatten
2018-12-28 14:23:20.003 1456-1505/?
I/GraphicBuffer:  GraphicBuffer::GraphicBuffer() BASE(), mOwner(ownData),
2018-12-28 14:23:20.003 1456-1505/? I/GraphicBuffer:
GraphicBuffer::unflatten
2018-12-28 14:23:20.003 1456-1505/? I/Gralloc2:
Graphic  Mapper::importBuffer(
2018-12-28 14:23:20.005 1389-1389/? W/SurfaceFlinger:
```

```
SurfaceFlinger::doDebugFlashRegions
2018-12-28 14:23:20.005 1389-1389/? W/SurfaceFlinger:
SurfaceFlinger::doComposition
2018-12-28 14:23:20.005 1389-1389/? E/SurfaceFlinger:
SurfaceFlinger::doComposition 0
2018-12-28 14:23:20.005 1389-1389/? W/SurfaceFlinger:
SurfaceFlinger::doDisplayComposition
2018-12-28 14:23:20.005 1389-1389/? W/SurfaceFlinger:
SurfaceFlinger::doComposeSurfaces
2018-12-28 14:23:20.006 1389-1389/?
I/GraphicBuffer:  GraphicBuffer::getNativeBuffer()
2018-12-28 14:23:20.006 1389-1389/? W/Layer: mTexture.setDimensions: width
= 480 ,height = 800 ,layercount 1, z: 1073741824 layerStack 0
2018-12-28 14:23:20.006 1389-1389/? E/DisplayDevice:
DisplayDevice::swapBuffers(HWComposer& hwc)
2018-12-28 14:23:20.055 1389-1389/? I/Surface: Surface::queueBuffer
2018-12-28 14:23:20.055 1389-1389/? I/Surface: Surface::dequeueBuffer
2018-12-28 14:23:20.055 1389-1389/?
I/GraphicBuffer:  GraphicBuffer::needsReallocation(uint32_t inWidth,
uint32_t inHeight,
2018-12-28 14:23:20.055 1389-1389/? E/DisplayDevice:
DisplayDevice::swapBuffers(HWComposer& hwc) mDisplaySurface->advanceFrame()
2018-12-28 14:23:20.055 1389-1389/? E/FramebufferSurface: status_t
FramebufferSurface::advanceFrame() #in
2018-12-28 14:23:20.055 1389-1389/? E/FramebufferSurface: status_t
FramebufferSurface::advanceFrame() #in 2
2018-12-28 14:23:20.055 1389-1389/? W/FramebufferSurface:
acquireBufferLocked(&item, 0);
2018-12-28 14:23:20.055 1389-1389/? E/FramebufferSurface:
mCurrentBufferSlot = item.mSlot;
2018-12-28 14:23:20.055 1389-1389/? E/FramebufferSurface:
mHwcBufferCache.getHwcBuffer(mCurrentBufferSlot,
2018-12-28 14:23:20.055 1389-1389/?
I/GraphicBuffer:  GraphicBuffer::getNativeBuffer()
2018-12-28 14:23:20.055 1389-1389/? I/FramebufferSurface: BUFFER IS NULL 0
2018-12-28 14:23:20.055 1389-1389/?
I/GraphicBuffer:  GraphicBuffer::getNativeBuffer()
2018-12-28 14:23:20.055 1389-1389/? W/SurfaceFlinger:
SurfaceFlinger::postFramebuffer
2018-12-28 14:23:20.056 1389-1389/? E/SurfaceFlinger: surface flinger
genesis test 2 BootAnimation#0
2018-12-28 14:23:20.056 1389-1389/? W/SurfaceFlinger:
SurfaceFlinger::postComposition
2018-12-28 14:23:20.056 1389-1389/? W/SurfaceFlinger:
SurfaceFlinger::updateCompositorTiming
2018-12-28 14:23:20.056 1389-1389/? W/SurfaceFlinger:
SurfaceFlinger::setCompositorTimingSnapped
2018-12-28 14:23:20.056 1389-1389/?
W/SurfaceFlinger:  SurfaceFlinger::recordBufferingStats(const char*
layerName,
2018-12-28 14:23:20.119 1456-1505/? I/Surface: Surface::queueBuffer
```

```
2018-12-28 14:23:20.119 1456-1505/? I/Surface: Surface::dequeueBuffer
```