

安卓图形架构中的Anbox的HAL模块

在安卓官方文档当中，安卓的系统架构被分为五层架构，其五层架构分别为

表1 安卓系统结构

架构	主要负责的内容
应用框架 APP FRAME	应用框架主要为开发者使用，开发者API提供了很多映射至底层的HAL接口，提供实现驱动程序的相关信息
Binder IPC	主要负责进程间的通讯，不仅仅是顶层进程之间的交互，更重要的是支持顶层进程和安卓系统服务间甚至是HAL的交互
系统服务	安卓把各种各样的系统服务包装为各种模块化组件，可以方便后续升级。其中包括有DNS解析模块，媒体模块等。
硬件抽象层 HAL	HAL可以定义一个接口给硬件供应商实现，主要用于实现系统服务和底层驱动程序的通讯和运作。其被封装成一个模块被安卓系统随时调用
Linux内核	Android使用的linux内核与一般的内核不同，如：必须支持Binder、包含一些特殊功能（内存守护）

安卓图形部分架构以及anbox图形模块

其中在anbox的图形模块中，最主要关心的就是HAL层以及LINUX KERNEL。因为本项目中安卓是运行在容器内的，安卓不能像原生一样顺利地让HAL层直接调用其LINUX KERNEL内的驱动和所需资源，而是要跟容器外部正在运行的LINUX环境对接。在本项目中的这一目标就是努力**让容器内的安卓与容器外本不兼容的LINUX环境对接**，成功把图形化渲染的工作交给LINUX环境当中，实现渲染效率比软件模拟成倍提升的效果。

在安卓主要的图形模块当中，所有的渲染信息流大致如下：

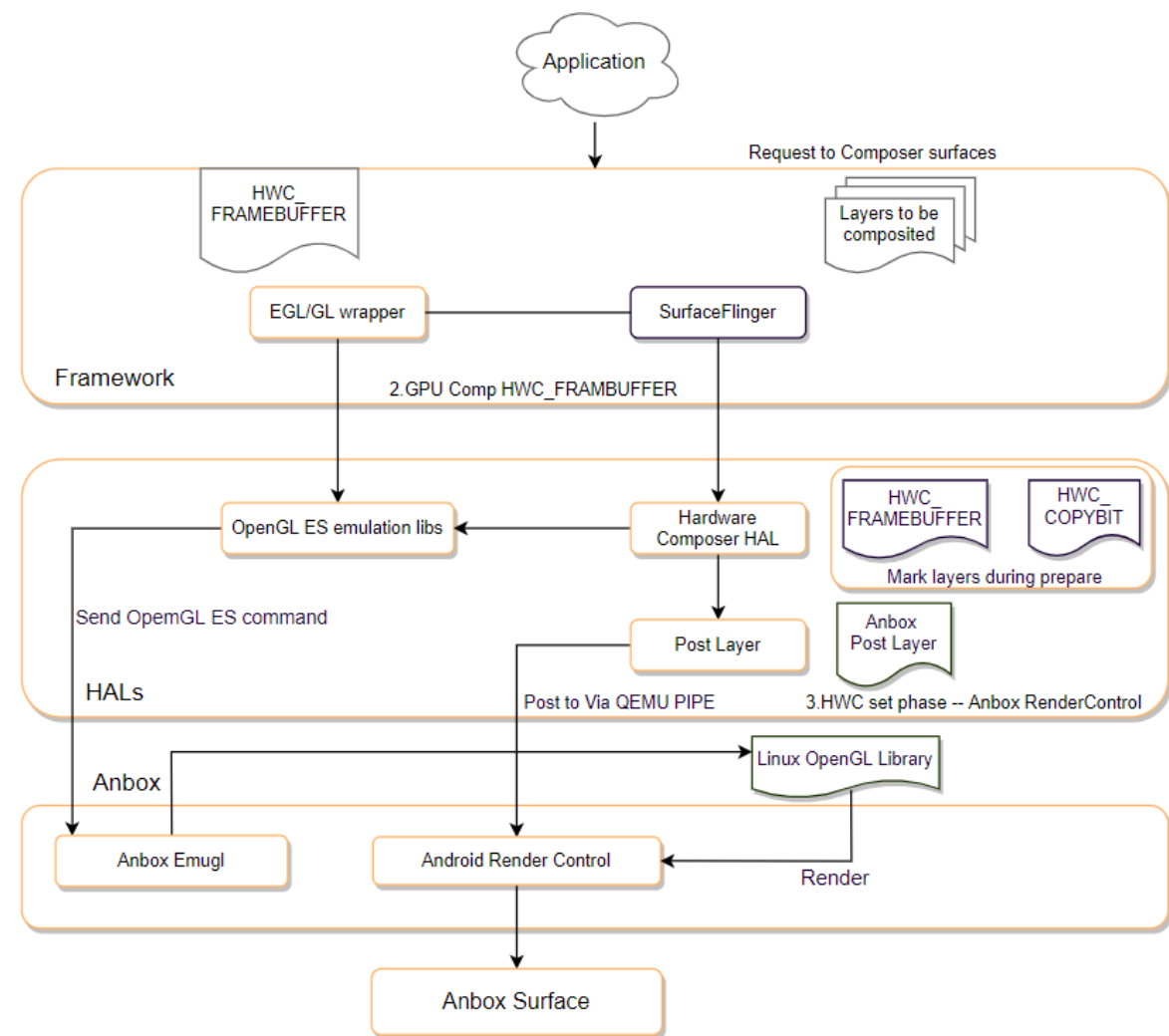


图1 安卓显示相关实现流程图

1. HAL层

在本文档中，主要聚焦于HAL层anbox的实现。在安卓系统中HAL层包括两个组件，HWC(Hardware Composer)与Gralloc。

1.1 HWC(Hardware Composer)模块

1.1.1 初步介绍

无论开发者使用什么渲染API，一切的内容都将会渲染到**surface**上，surface可以被理解为一个生产缓冲区，整个被渲染的surface队列都会被**surfaceflinger**所消耗而准备合成到屏幕上。

Android Graphics Pipeline

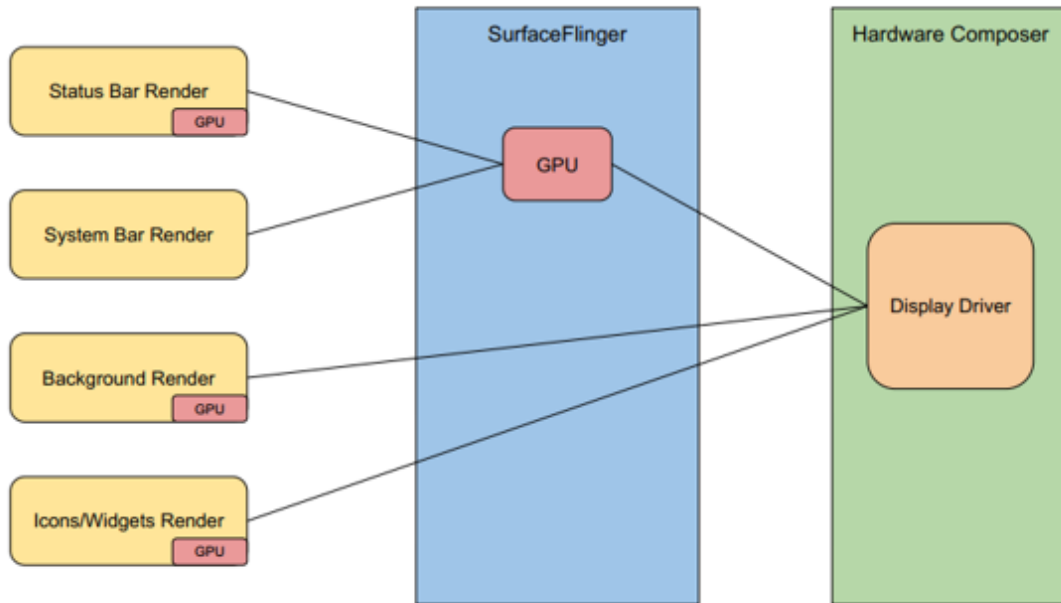


图1.1 安卓显示流水线

合成是一种将缓冲区队列内所有元素有规律有层级的覆盖为一块画面的操作。在安卓系统中，surface队列内的元素通常是大小不一且杂乱的，因为来自多个不同源的缓冲区，例如有些是出于顶部的状态栏，有些是出于底部的导航栏，而中间的应用内容也是一层缓冲区，直接送出来的画面完全不是屏幕在显示的画面。怎么解决这个问题？在安卓中选择了效率最高的方式来处理合成问题——将三个缓冲区全部传送到显示硬件，并指示它从不同的缓冲区读取屏幕不同部分的数据。这个操作的承担者就是硬件混合渲染器HAL层。

合成到屏幕这一步就需要调用我们的HAL层中的HWC了。HWC主要跟屏幕显示组件通讯，获得屏幕相应的参数，计算出最常用的四个不同的部分——状态栏、系统栏、应用以及壁纸/背景所覆盖的画面，再接受 `surfaceflinger` 需要合成的队列所拥有的内容，把这些内容分别合成到状态栏、系统栏、应用以及壁纸/背景所覆盖的画面后再与屏幕沟通，最终显示在真正现实的手机屏幕上。

当然，HWC也会负责虚拟屏幕合成，同时会向硬件设备注册三个回调，以便应对这些事件的发生：屏幕热插拔，刷新和VSync信号。

在anbox当中，HWC还有一个比较重要的功能，它可以把单个安卓里面的应用程序映射到桌面环境的单个窗口中，因为在桌面环境下我们不需要再像现实中让HWC把所有内容合成到一块屏幕上。anbox会通过其 `hwcomposer` 实现告诉 `SurfaceFlinger` 为每个应用程序获取一个层，并将其与从 `Android WindowManager` 收到的额外信息结合起来，将单个层映射到应用程序。

Android Graphics Pipeline

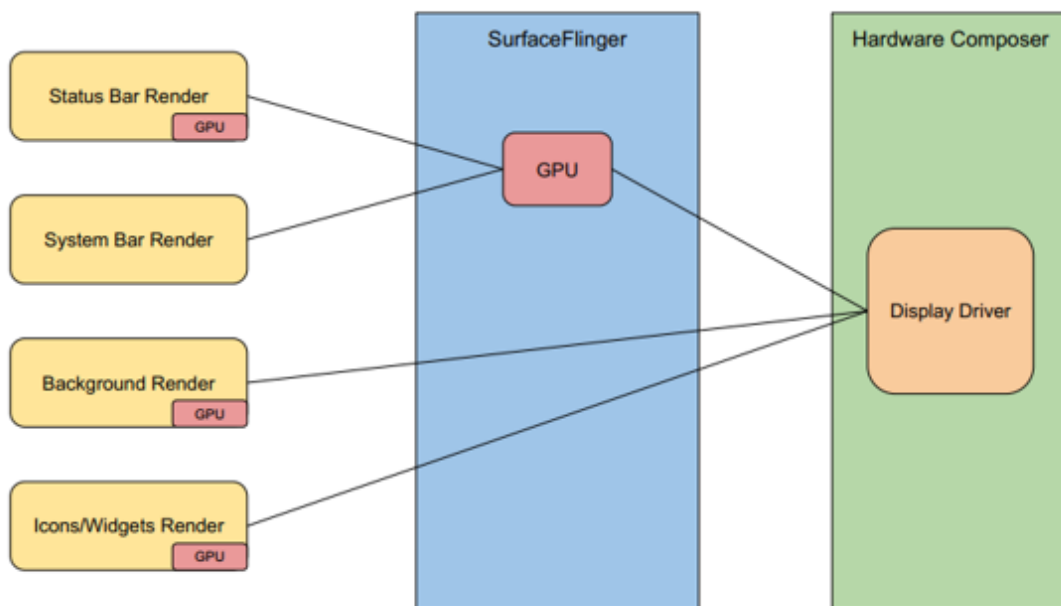


图1.2 HWC显示合成概念图1

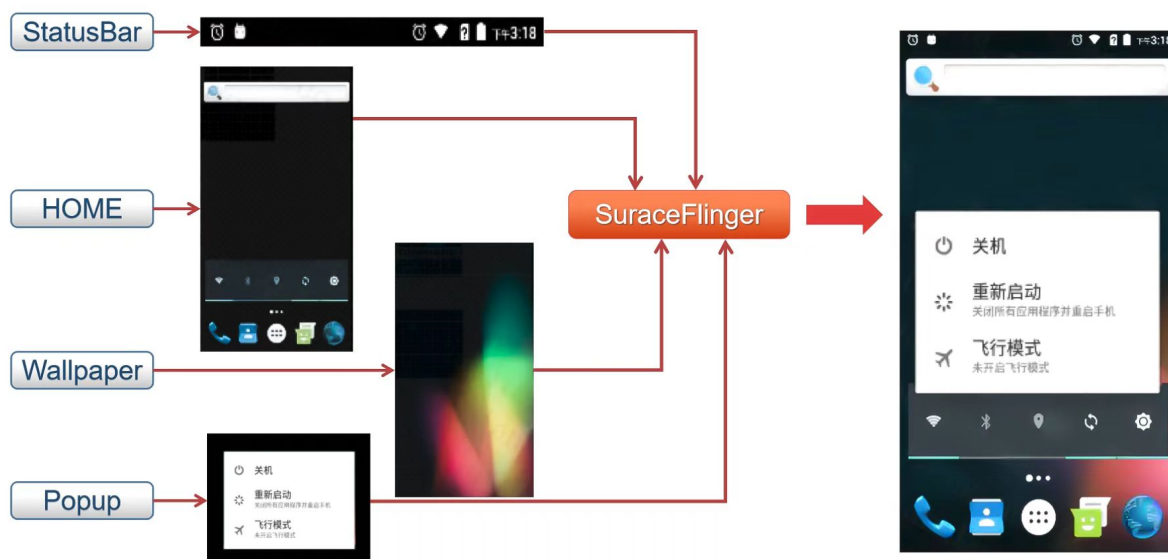


图1.3 HWC显示合成概念图2

1.1.2 HWC模块重要实现

(1) 安卓中HWC部分的实现

首先 `surfaceflinger` 创建 `HWComposer`。

```
//节选自main_surfaceflinger.cpp
void SurfaceFlinger::init() {
    // 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)); //调用构造函数
}
```

`HWComposer` 构造函数。

```
//节选自DisplayHardware/HwComposer.cpp
HwComposer::HwComposer(const sp<SurfaceFlinger>& flinger)
    : mFlinger(flinger),
      mAdapter(),
      mHwcDevice(),
      mDisplayData(2),
      mFreeDisplaySlots(),
      mHwcDisplaySlots(),
      mCBContext(),
      mEventHandler(nullptr),
      mVSyncCounts(),
      mRemainingHwcVirtualDisplays(0)
{
    for (size_t i=0 ; i<HWC_NUM_PHYSICAL_DISPLAY_TYPES ; i++) {
        mLastHwVSync[i] = 0;
        mVSyncCounts[i] = 0;
    }
    loadHwcModule(); //调用HWC模块
}
```

HwComposer 构造函数一般使用 loadHwcModule 方法加载来 HwComposer 模块。

```
//节选自DisplayHardware/HwComposer.cpp
void HwComposer::loadHwcModule()
{
    ALOGV("loadHwcModule");
    // 其定义在hardware.h中，表示一个硬件模块
    hw_module_t const* module;
    // 加载硬件厂商提供的hwcomposer模块，HWC_HARDWARE_MODULE_ID定义在hwcomposer_defs.h
    // 中，表示"hwcomposer"
    if (hw_get_module(HWC_HARDWARE_MODULE_ID, &module) != 0) {
        ALOGE("%s module not found, aborting", HWC_HARDWARE_MODULE_ID);
        abort();
    }

    hw_device_t* device = nullptr;
    // 通过硬件厂商提供的open函数打开一个"composer"硬件设备，HWC_HARDWARE_COMPOSER也定义
    // 在hwcomposer_defs.h中，表示"composer"
    int error = module->methods->open(module, HWC_HARDWARE_COMPOSER, &device);
    if (error != 0) {
        ALOGE("Failed to open HWC device (%s), aborting", strerror(-error));
        abort();
    }

    uint32_t majorVersion = (device->version >> 24) & 0xF;
    // mHwcDevice是HWC2.h中定义的HWC2::Device，所有与HWC的交互都通过mHwcDevice
    if (majorVersion == 2) { // HWC2, hwc2_device_t是hwcomposer2.h中的结构体
        mHwcDevice = std::make_unique<HWC2::Device>(
            reinterpret_cast<hwc2_device_t*>(device));
    } else { // 设备是基于HWC1，这里用HWC2去适配，Android7.0及以前默认都是HWC1，
    // hwc_composer_device_1_t是hwcomposer.h中的结构体
        mAdapter = std::make_unique<HWC2On1Adapter>(
            reinterpret_cast<hwc_composer_device_1_t*>(device));
        uint8_t minorVersion = mAdapter->getHwc1MinorVersion();
        if (minorVersion < 1) {
```

```

        ALOGE("Cannot adapt to HWC version %d.%d",
              static_cast<int32_t>((minorVersion >> 8) & 0xF),
              static_cast<int32_t>(minorVersion & 0xF));
        abort();
    }
    mHwcDevice = std::make_unique<HWC2::Device>(
        static_cast<hwc2_device_t*>(mAdapter.get()));
}
// 获取硬件支持的最大虚拟屏幕数量，virtualDisplay主要是用来用于录屏
mRemainingHwcVirtualDisplays = mHwcDevice->getMaxVirtualDisplayCount();
}

```

先加载 `hwcomposer` 模块得到 `hw_module_t`，再打开 `composer` 设备得到 `hw_device_t`。所以一般情况下是先有HAL模块，再有实现此模块的硬件设备。

```

//节选自hardware/libhardware/include/hardware/hardware.h
typedef struct hw_module_t {
//每一个HAL库都会提供的一个方法methods，
    struct hw_module_methods_t* methods;
}hw_module_t;

typedef struct hw_module_methods_t {
//这个 methods的数据结构中只有一个函数指针变量open，用来打开指定的硬件设备。
    int (*open)(const struct hw_module_t* module, const char* id,
                struct hw_device_t** device);
} hw_module_methods_t;

```

在每个HAL层模块实现都要定义一个 `HAL_MODULE_INFO_SYM` 数据结构，并且该结构的第一个字段必须是 `hw_module_t`，下面是anbox当中 `hwcomposer` 模块的定义。

```

//节选自vendor/anbox/android/hwcomposer/hwcomposer.cpp
hwc_module_t HAL_MODULE_INFO_SYM = {
    .common = {
        .tag = HARDWARE_MODULE_TAG,
        .version_major = 1,
        .version_minor = 0,
        .id = HWC_HARDWARE_MODULE_ID,
        .name = "Hardware Composer Module",
        .author = "Anbox Developers",
        .methods = &hwc_module_methods,
    }
};

```

这里Anbox中对应method如下，实现了打开模块的接口。

```

//节选自vendor/anbox/android/hwcomposer/hwcomposer.cpp
//methods数据结构中只有一个函数指针变量open，用来打开指定的硬件设备（Anbox中是软件实现）
static hw_module_methods_t hwc_module_methods = {
    .open = hwc_device_open
};

//Anbox中这里注册了自己实现的HWC设备的各种接口对接函数

```

```

static int hwc_device_open(const hw_module_t* module, const char* name,
hw_device_t** device) {
    ALOGD("%s", __PRETTY_FUNCTION__);

    auto dev = new HwcContext;
    dev->device.common.tag = HARDWARE_DEVICE_TAG; // 标记这是一个硬件设备的模块
    dev->device.common.version = HWC_DEVICE_API_VERSION_1_1; // HWC版本, Anbox还是用的HWC1

    dev->device.common.module = const_cast<hw_module_t*>(module); // 对应上面模块信息结构体定义
    dev->device.common.close = hwc_device_close; // HWC设备关闭调用的操作
    dev->device.prepare = hwc_prepare; // 图层的配置, surfaceflinger把要显示的layers放在displays参数里, 针对其参数对应Display硬件配置图层的参数类型等。
    dev->device.set = hwc_set; // hwc_set方法, 在SurfaceFlinger要求HWC发送图层数据, 该方法进行发送相关图层数据给Anbox在Linux的前端实现。
    dev->device.eventControl = hwc_event_control; // 使能/禁止vsync, Anbox未实现
    dev->device.blank = hwc_blank; // 老的hwc(1.3以前)用blank控制display on/off, 最新的hwc里用setPowerMode。实现的功能差不多, 但setPowerMode的参数更丰富, 不像blank就0/1。
    dev->device.query = hwc_query;
    dev->device.getDisplayConfigs = hwc_get_display_configs; //获取显示硬件的配置, 一般是多显示屏等配置的获取返回, Anbox只有一个主显示器, 因此这里只是配置0让HWC进入获取属性的调用。
    dev->device.getDisplayAttributes = hwc_get_display_attributes; //获取显示硬件的各个属性
    dev->device.registerProcs = hwc_register_procs;
    dev->device.dump = nullptr;

    *device = &dev->device.common;

    return 0;
}

```

可以看出这一阶段已经完成了Anbox的HWC具体实现的模块打开并注册接口, 下面将针对hwc_get_display_attributes和hwc_set进行讲解, 这两个是Anbox在HWC这一部分显示输出的精髓。

(2) Anbox部分重要实现

在Anbox的HWC模块中, 其中最主要的就是hwc_get_display_attributes和hwc_set的实现, 这里把一些重要的部分进行讲解。

hwc_get_display_attributes函数部分实现

```

//节选自vendor/anbox/android/hwcomposer/hwcomposer.cpp
static int hwc_get_display_attributes(hwc_composer_device_1* dev,
int disp, uint32_t config,
const uint32_t* attributes,
int32_t* values) {

    if (disp != 0 || config != 0) {
        return -EINVAL;
    }

    //建立与Anbox前端的QEMU_PIPE连接
    DEFINE_AND_VALIDATE_HOST_CONNECTION();

    // 下面各个属性都是通过与Anbox连接获取返回的值
    while (*attributes != HWC_DISPLAY_NO_ATTRIBUTE) {

```

```

//针对各种属性返回对应的值，其中HWC_DISPLAY_NO_ATTRIBUTE是这个attributes数组的结束
switch (*attributes) {
    //获取屏幕VSYNC垂直同步信号的周期值
    case HWC_DISPLAY_VSYNC_PERIOD:
        *values = rcEnc->rcGetDisplayVsyncPeriod(rcEnc, disp);
        break;
    //获取屏幕宽高和DPI参数
    case HWC_DISPLAY_WIDTH:
        *values = rcEnc->rcGetDisplaywidth(rcEnc, disp);
        break;
    case HWC_DISPLAY_HEIGHT:
        *values = rcEnc->rcGetDisplayHeight(rcEnc, disp);
        break;
    case HWC_DISPLAY_DPI_X:
        *values = 1000 * rcEnc->rcGetDisplayDpiX(rcEnc, disp);
        break;
    case HWC_DISPLAY_DPI_Y:
        *values = 1000 * rcEnc->rcGetDisplayDpiY(rcEnc, disp);
        break;
    default:
        ALOGE("Unknown attribute value 0x%02x", *attributes);
}
++attributes;
++values;
}
return 0;
}

```

可以看得出hwc_get_display_attributes是通过Anbox获取屏幕的参数返回给安卓HWC模块，方便给SurfaceFlinger和HWC进行屏幕输出。

接下来查看其中rcEnc调用的方法的来源：

```

//节选自vendor/anbox/android/hwcomposer/hwcomposer.cpp
//其来自于建立QEMU_PIPE的地方
#define DEFINE_AND_VALIDATE_HOST_CONNECTION() \
    //获取Anbox的QEMU_PIPE连接，这部分不再赘述
    HostConnection *hostCon = HostConnection::get(); \
    if (!hostCon) { \
        ALOGE("hwcomposer.anbox: Failed to get host connection\n"); \
        return -EIO; \
    } \
    //获取Host端的rcEncoder方法返回的远程调用对象
    renderControl_encoder_context_t *rcEnc = hostCon->rcEncoder(); \
    if (!rcEnc) { \
        ALOGE("hwcomposer.anbox: Failed to get renderControl encoder context\n"); \
        return -EIO; \
    }

```

其来自于建立QEMU_PIPE的地方，通过与外部Anbox建立连接，利用Anbox 基于 Protobuf 设计的RPC 进行通信，实现远程方法调用。

这里以rcEnc->rcGetDisplayWidth(rcEnc, disp)为例，远程能调用到Anbox在Host端的GL库函数返回的值：


```
//节选自vendor/anbox/src/anbox/graphics/emugl/RenderControl.cpp
int rcGetDisplayWidth(uint32_t display_id) {
    (void)display_id;
    // 调用到实际的GL库的函数，获取实际的垂直分辨率大小
    return static_cast<int>(anbox::graphics::emugl::DisplayInfo::get()-
>vertical_resolution());
}
```

最终能正确通信获得rcGetDisplayWidth返回值。

以上的获取DisplayWidth的实现，其流程从自下向上来看，其流程如下：

① hwc_get_display_attributes中的rcEnc->rcGetDisplayWidth远程调用获得了Host端Anbox的窗口垂直分辨率的值。

② dev->device.getDisplayAttributes = hwc_get_display_attributes是该HWC注册的方法，被populateConfigs函数调用该模块获取信息。

③ 接着被HWC2On1Adapter::populatePrimary()、HWC2On1Adapter::hwc1Hotplug、HWC2On1Adapter::createVirtualDisplay等方法调用，主要用于在显示设备创建、热插拔时获取显示设备信息返回给SurfaceFlinger及HWC。

hwc_set函数部分实现

dev->device.set是SurfaceFlinger要求HWC发送图层数据，在这里Anbox对应注册的函数是hwc_set，其函数实现如下：

```
//节选自vendor/anbox/android/hwcomposer/hwcomposer.cpp
static int hwc_set(hwc_composer_device_1_t* dev, size_t numDisplays,
                  hwc_display_contents_1_t** displays) {
    // HWC上下文
    auto context = reinterpret_cast<HwcContext*>(dev);

    if (displays == NULL || displays[0] == NULL)
        return -EFAULT;
    //通过QEMU_PIPE与Anbox连接，原理同上一步hwc_get_display_attributes中的实现
    DEFINE_AND_VALIDATE_HOST_CONNECTION();

    // 循环处理当前display的所有层
    for (size_t i = 0 ; i < displays[0]->numHwLayers ; i++) {
        const auto layer = &displays[0]->hwLayers[i];

        //如果是指针或者需要跳过而不用刷新的层，跳过
        if (layer->flags & HWC_SKIP_LAYER ||
            layer->flags & HWC_IS_CURSOR_LAYER)
            continue;

        // Anbox下，HWC注册的compositionType是HWC_FRAMEBUFFER_TARGET
        if(layer->compositionType == HWC_FRAMEBUFFER_TARGET) {
            // 通过getprop去读取“anbox.layer_name”这一系统配置，获取当前要传输的Layer名称
            std::string layer_name_temp =
            android::base::GetProperty("anbox.layer_name", "");
            std::string layer_name = layer_name_temp.substr(0,
            layer_name_temp.find('#'));
            strncpy(layer->name, layer_name.c_str(), layer_name.size());
        }
    }
}
```

```

    }

    // 根据图层的三组必要参数，发送Layer
    rcEnc->rcPostLayer(rcEnc,
                      layer->name,
                      cb->hostHandle,
                      layer->planeAlpha / 255,
                      layer->sourceCrop.left,
                      layer->sourceCrop.top,
                      layer->sourceCrop.right,
                      layer->sourceCrop.bottom,
                      layer->displayFrame.left,
                      layer->displayFrame.top,
                      layer->displayFrame.right,
                      layer->displayFrame.bottom);

    hostCon->flush(); //刷新缓冲区
}
// 通知Anbox所有发送完毕
rcEnc->rcPostAllLayersDone(rcEnc);

check_sync_fds(numDisplays, displays);

return 0;
}

```

该函数中，第一个系统配置“anbox.layer_name”是Anbox修改了SurfaceFlinger，在准备输出图层前把图层名称保存成这个配置中，具体实现如下：

```

//节选自: frameworks/native/services/surfaceflinger/BufferLayer.cpp
//SurfaceFlinger
void BufferLayer::setPerFrameData(const sp<const DisplayDevice>& displayDevice,
                                  const ui::Transform& transform, const Rect&
viewport,
                                  int32_t supportedPerFrameMetadata,
                                  const ui::Dataspace targetDataspace) {
    RETURN_IF_NO_HWC_LAYER(displayDevice);

    // Apply this display's projection's viewport to the visible region
    // before giving it to the HWC HAL.
    //获取可见区域
    Region visible = transform.transform(visibleRegion.intersect(viewport));

    // 寻找当前显示器的输出图层
    const auto outputLayer = findOutputLayerForDisplay(displayDevice);
    LOG_FATAL_IF(!outputLayer || !outputLayer->getState().hwc);
    //获取HWC
    auto& hwcLayer = (*outputLayer->getState().hwc).hwcLayer;
    //保存当前图层名称到anbox.layer_name系统配置中
    android::base::SetProperty("anbox.layer_name", mName.string());
    //设置刚刚获取的可见区域到HWC
    auto error = hwcLayer->setVisibleRegion(visible);
    if (error != HWC2::Error::None) {
        ALOGE("[%s] Failed to set visible region: %s (%d)", mName.string(),
              to_string(error).c_str(), static_cast<int32_t>(error));
        visible.dump(LOG_TAG);
    }
}

```

```
}
```

而rcEnc->rcPostLayer中planeAlpha、sourceCrop、displayFrame就是下图中各个区域。

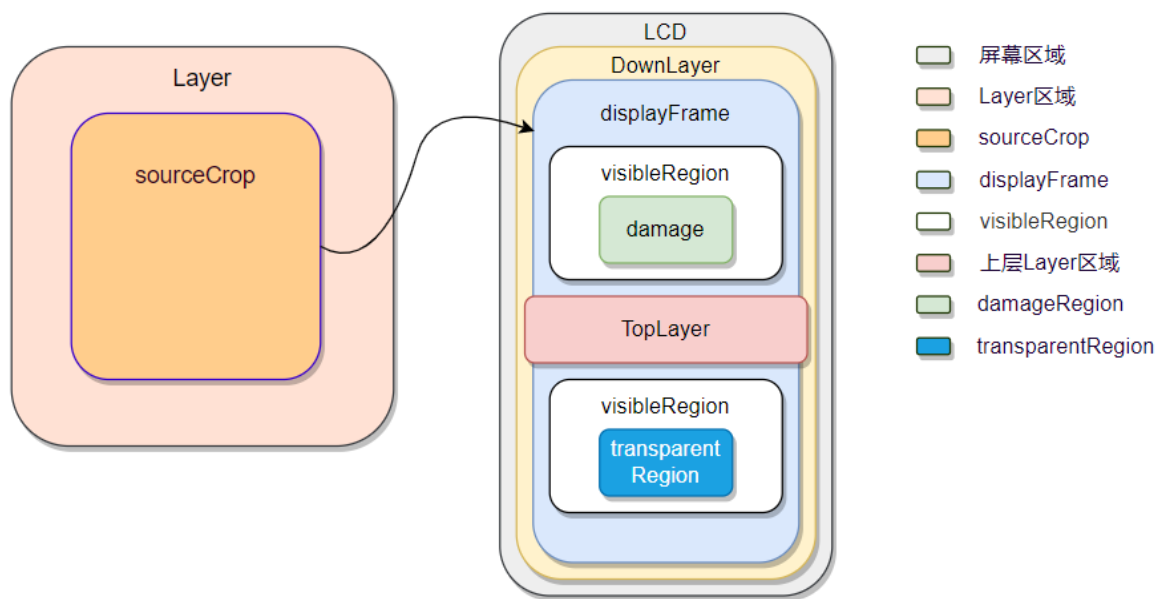


图1.4 安卓显示区域间的关系图

① sourceCrop是对Layer进行剪切的，值截取部分Layer的内容进行显示；sourceCrop不超过Layer的大小，超过没有意义。

② displayFrame表示Layer在屏幕上的显示区域，具体说来，是sourceCrop区域在显示屏上的显示区域。displayFrame一般来说，小于屏幕的区域。而displayFrame可能比sourceCrop大，可能小，这都是正常的，只是需要做缩放，这就是合成时需要处理的。

而在外部Linux的Anbox中，上面rcEnc->rcPostLayer、rcEnc->rcPostAllLayersDone也是属于Anbox的RPC实现，这里只截取最终在Anbox中调用的地方，不多赘述。

```
//节选自vendor/anbox/src/anbox/graphics/emugl/RenderControl.cpp
void rcPostLayer(const char *name, uint32_t color_buffer, float alpha,
                 int32_t sourceCropLeft, int32_t sourceCropTop,
                 int32_t sourceCropRight, int32_t sourceCropBottom,
                 int32_t displayFrameLeft, int32_t displayFrameTop,
                 int32_t displayFrameRight, int32_t displayFrameBottom) {
    //建立渲染结构体
    Renderable r{
        name,
        color_buffer,
        alpha,
        {displayFrameLeft, displayFrameTop, displayFrameRight,
        displayFrameBottom},
        {sourceCropLeft, sourceCropTop, sourceCropRight, sourceCropBottom}};
    //发送给Anbox的Surface处理
    frame_layers.push_back(r);
}

void rcPostAllLayersDone() {
    if (composer) composer->submit_layers(frame_layers);
}
```

```
//提交Layer后刷新缓冲区
frame_layers.clear();
}
```

1.2 Gralloc模块

1.2.1 初步介绍

在android图形架构中，Gralloc属于低级别组件，用于给图形缓冲队列进行缓冲区分配，是一个内存分配器。其通过 `用法标志` 执行缓冲区分配。用法标志包括以下属性：

- 从软件 (CPU) 访问内存的频率
- 从硬件 (GPU) 访问内存的频率
- 是否将内存用作 OpenGL ES (GLES) 纹理
- 视频编码器是否会使用内存

例如，如果生产方的缓冲区格式指定 `RGBA_8888` 像素，并且生产方指明将从软件访问缓冲区，Gralloc将以R-G-B-A的顺序为每个像素创建一个4字节的缓冲区。如果情况相反，生产方指明仅从硬件访问其缓冲区且缓冲区作为 GLES 纹理，那么Gralloc可以做任何GLES驱动想要做的事情，比如BGRA排序、非线性swizzled布局和其他颜色格式。允许硬件使用其首选格式可以提高性能。

Gralloc 返回的句柄可以通过 Binder 在进程之间进行传递。

1.2.2 Gralloc模块的重要实现

(1) Gralloc在安卓中的实现

Gralloc主要定义了以 `HAL_MODULE_INFO_SYM` 为符号的类型为 `private_module_t` 的结构体

```
//节选自gralloc.cpp
static struct hw_module_methods_t gralloc_module_methods = {
    .open = gralloc_device_open
};

struct private_module_t HAL_MODULE_INFO_SYM = {
    .base = {
        .common = {
            .tag = HARDWARE_MODULE_TAG,
            .version_major = 1,
            .version_minor = 0,
            .id = GRALLOC_HARDWARE_MODULE_ID,
            .name = "Graphics Memory Allocator Module",
            .author = "The Android Open Source Project",
            .methods = &gralloc_module_methods
        },
        .registerBuffer = gralloc_register_buffer,
        .unregisterBuffer = gralloc_unregister_buffer,
        .lock = gralloc_lock,
        .unlock = gralloc_unlock,
    },
    .framebuffer = 0,
    .flags = 0,
```

```
.numBuffers = 0,
.bufferMask = 0,
.lock = PTHREAD_MUTEX_INITIALIZER,
.currentBuffer = 0,
};
```

`private_module_t` 用于描述Gralloc模块下的系统帧缓冲区信息,主要作用是将图形缓冲区渲染到帧缓冲区。

```
//节选自gralloc/gralloc_priv.h
struct private_module_t {
    gralloc_module_t base;

    private_handle_t* framebuffer; //指向系统帧缓冲区的句柄
    uint32_t flags; //用来标志系统帧缓冲区是否支持双缓冲
    uint32_t numBuffers; //表示系统帧缓冲区包含有多少个图形缓冲区
    uint32_t bufferMask; //记录系统帧缓冲区中的图形缓冲区的使用情况
    pthread_mutex_t lock; //一个互斥锁,用来保护结构体private_module_t的并行访问
    buffer_handle_t currentBuffer; //用来描述当前正在被渲染的图形缓冲区
    int pmem_master;
    void* pmem_master_base;

    struct fb_var_screeninfo info; //保存设备显示屏的动态属性信息
    struct fb_fix_screeninfo finfo; //保存设备显示屏的固定属性信息
    float xdpi; //描述设备显示屏在宽度
    float ydpi; //描述设备显示屏在高度
    float fps; //用来描述显示屏的刷新频率
};
```

`gralloc_module_t` 用于描述gralloc模块信息,主要用于分配或者释放图形缓冲区。

```
//节选自hardware/gralloc.h
typedef struct gralloc_module_t {
    struct hw_module_t common;
    int (*registerBuffer)(struct gralloc_module_t const* module,
        buffer_handle_t handle); //映射一块图形缓冲区到一个进程的地址空间去
    int (*unregisterBuffer)(struct gralloc_module_t const* module,
        buffer_handle_t handle); //取消映射一块图形缓冲区到一个进程的地址空间去
    int (*lock)(struct gralloc_module_t const* module,
        buffer_handle_t handle, int usage,
        int l, int t, int w, int h,
        void** vaddr); //锁定一个指定的图形缓冲区
    int (*unlock)(struct gralloc_module_t const* module,
        buffer_handle_t handle); //解锁一个指定的图形缓冲区
    int (*perform)(struct gralloc_module_t const* module,
        int operation, ... );
    int (*lockAsync)(struct gralloc_module_t const* module,
        buffer_handle_t handle, int usage,
        int l, int t, int w, int h,
        void** vaddr, int fenceFd);
    int (*unlockAsync)(struct gralloc_module_t const* module,
        buffer_handle_t handle, int* fenceFd);
    int (*lockAsync_ycbcr)(struct gralloc_module_t const* module,
        buffer_handle_t handle, int usage,
```

```

        int l, int t, int w, int h,
        struct android_ycbcr *ycbcr, int fenceFd);
void* reserved_proc[3];
} gralloc_module_t;

```

`alloc_device_t` 用于描述gralloc设备的信息。

```

//节选自hardware/gralloc.h
typedef struct alloc_device_t {
    struct hw_device_t common;
    int (*alloc)(struct alloc_device_t* dev,
        int w, int h, int format, int usage,
        buffer_handle_t* handle, int* stride); //用于分配一块图形缓冲区
    int (*free)(struct alloc_device_t* dev,
        buffer_handle_t handle); //用于释放指定的图形缓冲区
    void (*dump)(struct alloc_device_t *dev, char *buff, int buff_len);
    void* reserved_proc[7];
} alloc_device_t;

```

还有 `hw_module_t` 主要用于关联模块和设备，其在hardware.h被定义。

在 `gralloc.h` 中，还定义了 `GRALLOC_HARDWARE_GPU0` 设备,其主要是用于分配图形缓冲区，`hw_module_t`用于描述硬件抽象层Gralloc模块，而`hw_device_t`则用于描述硬件抽象层Gralloc设备，通过硬件抽象层设备可以找到对应的硬件抽象层模块。

图形缓冲区的结构则用 `private_handle_t` 来描述

```

//节选自gralloc/gralloc_priv.h
struct private_handle_t : public native_handle {
#else
struct private_handle_t {
    struct native_handle nativeHandle;
#endif
    enum {
        PRIV_FLAGS_FRAMEBUFFER = 0x00000001
    };
    // file-descriptors
    int fd; //指向一个文件描述符，这个文件描述符要么指向帧缓冲区设备，要么指向一块匿名共享内存
    // ints
    int magic;
    int flags; //用来描述一个缓冲区的标志，当一个缓冲区的标志值等于
    PRIV_FLAGS_FRAMEBUFFER的时候，就表示它是在帧缓冲区中分配的。
    int size; //用来描述一个缓冲区的大小
    int offset; //用来描述一个缓冲区的偏移地址
    // FIXME: the attributes below should be out-of-line
    uint64_t base __attribute__((aligned(8))); //用来描述一个缓冲区的实际地址
    int pid; //用来描述一个缓冲区的创建者的PID

```

`GRALLOC_HARDWARE_GPU0` 设备使用结构体 `alloc_device_t` 来描述。结构体 `alloc_device_t` 有两个成员函数`alloc`和`free`，在上文当中[`alloc_device_t`用于描述gralloc设备的信息]已经提及。

```
//节选自hardware/gralloc.h
static inline int gralloc_open(const struct hw_module_t* module,
                               struct alloc_device_t** device) {
    return module->methods->open(module,
                                  GRALLOC_HARDWARE_GPU0, (struct hw_device_t**)device);
}
```

module指向的是一个用来描述Gralloc模块的 `hw_module_t` 结构体，它的成员变量methods所指向的一个 `hw_module_methods_t` 结构体的成员函数open指向了Gralloc模块中的函数 `gralloc_device_open`。这里传入的设备名为 `GRALLOC_HARDWARE_GPU0`，表示当前打开的是gpu设备。

下面这个函数主要是用来创建一个 `gralloc_context_t` 结构体，并且对它的成员变量device进行初始化。结构体 `gralloc_context_t` 的成员变量device的类型为 `gralloc_device_t`，它用来描述一个gralloc设备。前面提到，gralloc设备是用来分配和释放图形缓冲区的，这是通过调用它的成员函数alloc和free来实现的。从这里可以看出，函数 `gralloc_device_open` 所打开的gralloc设备的成员函数alloc和free分别被设置为Gralloc模块中的函数 `gralloc_alloc` 和 `gralloc_free`。

Gralloc主要就是通过上面两大函数来进行缓冲区的分配和释放。

```
//节选自gralloc.cpp
int gralloc_device_open(const hw_module_t* module, const char* name,
                        hw_device_t** device)
{
    int status = -EINVAL;
    if (!strcmp(name, GRALLOC_HARDWARE_GPU0)) {
        gralloc_context_t *dev;
        dev = (gralloc_context_t*)malloc(sizeof(*dev));
        /* initialize our state here */
        memset(dev, 0, sizeof(*dev));
        /* initialize the procs */
        dev->device.common.tag = HARDWARE_DEVICE_TAG; // 这是一个硬件模块标记
        dev->device.common.version = 0;
        dev->device.common.module = const_cast<hw_module_t*>(module);
        dev->device.common.close = gralloc_close; // gralloc关闭
        dev->device.alloc = gralloc_alloc; //分配图形缓冲区
        dev->device.free = gralloc_free; // 释放图形缓冲区
        *device = &dev->device.common;
        status = 0;
    } else {
        status = fb_device_open(module, name, device);
    }
    return status;
}
```

(2) anbox部分重要实现

上面已经针对Gralloc模块基础进行介绍，为了了解这一部分的实现原理，这里只关注整个Gralloc的HAL模块中Anbox注册的主要函数gralloc_alloc和gralloc_free进行介绍，而其中用于帧缓冲刷新新的fb_post，Anbox并未实现。

gralloc_alloc及gralloc_free函数的实现

首先这里介绍gralloc_alloc以及gralloc_free函数，这里本质就是实现图形缓冲区的申请与释放，由于代码量较多，这里截取主要部分。

```
//节选自: vendor/anbox/android/opengl/system/gralloc/gralloc.cpp
//申请图形缓冲区
static int gralloc_alloc(alloc_device_t* dev,
                        int w, int h, int format, int usage,
                        buffer_handle_t* pHandle, int* pStride)
{
    //当前gralloc设备
    gralloc_device_t *grdev = (gralloc_device_t *)dev;

    //一些标记，不同标记表面当前Gralloc申请的缓冲区的用途，比如用于camera的缓冲区
    //
    // Note: in screen capture mode, both sw_write and hw_write will be on
    // and this is a valid usage
    //
    bool sw_write = (0 != (usage & GRALLOC_USAGE_SW_WRITE_MASK));
    bool hw_write = (usage & GRALLOC_USAGE_HW_RENDER);
    bool sw_read = (0 != (usage & GRALLOC_USAGE_SW_READ_MASK));
    bool hw_cam_write = usage & GRALLOC_USAGE_HW_CAMERA_WRITE;
    bool hw_cam_read = usage & GRALLOC_USAGE_HW_CAMERA_READ;
    bool hw_vid_enc_read = usage & GRALLOC_USAGE_HW_VIDEO_ENCODER;

    // 缓冲区大小，Anbox的这一模块是采用了ashmem这一共享内存来申请缓冲区
    int ashmem_size = 0;
    int stride = w;

    GLenum glFormat = 0;
    GLenum glType = 0;

    int bpp = 0;
    int align = 1;
    // 针对各种图像格式选择不同的像素位数(bpp)大小、格式(glFormat)等配置信息
    switch (format) {
        case HAL_PIXEL_FORMAT_RGBA_8888:
        case HAL_PIXEL_FORMAT_RGBX_8888:
        case HAL_PIXEL_FORMAT_BGRA_8888:
            bpp = 4;
            glFormat = GL_RGBA;
            glType = GL_UNSIGNED_BYTE;
            break;
        case HAL_PIXEL_FORMAT_RGB_888:
            bpp = 3;
            glFormat = GL_RGB;
            glType = GL_UNSIGNED_BYTE;
            break;
        default:
            ALOGE("gralloc_alloc: Unknown format %d", format);
            return -EINVAL;
    }

    //针对不同用途给ashmem_size增加大小
    if (sw_read || sw_write || hw_cam_write || hw_vid_enc_read) {
        // keep space for image on guest memory if SW access is needed
    }
}
```



```

// or if the camera is doing writing
if (yuv_format) {
    size_t yStride = (w*bpp + (align - 1)) & ~(align-1);
    size_t uvStride = (yStride / 2 + (align - 1)) & ~(align-1);
    size_t uvHeight = h / 2;
    ashmem_size += yStride * h + 2 * (uvHeight * uvStride);
    stride = yStride / bpp;
} else {
    size_t bpr = (w*bpp + (align-1)) & ~(align-1);
    ashmem_size += (bpr * h);
    stride = bpr / bpp;
}
}

```

//因为Anbox的实现没有实际图形硬件设备，从ashmem（Android 匿名共享内存，基于 mmap系统调用）申请图形缓冲区

```

//
// Allocate space in ashmem if needed
//
int fd = -1;
if (ashmem_size > 0) {
    // 对齐page size;
    ashmem_size = (ashmem_size + (PAGE_SIZE-1)) & ~(PAGE_SIZE-1);
    // 申请gralloc-buffer区域的共享内存
    fd = ashmem_create_region("gralloc-buffer", ashmem_size);
    if (fd < 0) {
        ALOGE("gralloc_alloc failed to create ashmem region: %s\n",
            strerror(errno));
        return -errno;
    }
}
// 建立给申请方的回调
cb_handle_t *cb = new cb_handle_t(fd, ashmem_size, usage, w, h,
frameworkFormat, format, glFormat, glType);

if (ashmem_size > 0) {
    // 申请到了就map这片区域
    //
    // map ashmem region if exist
    //
    void *vaddr;
    int err = map_buffer(cb, &vaddr);
    if (err) {
        close(fd);
        delete cb;
        return err;
    }
}

// map成功就更新fd给回调
cb->setFd(fd);
}

```

// 这里是如果有一些情况需要回传Anbox在主机侧Surface的数据，比如需要录屏的时候就需要这种情况

```

//

```

```

// Allocate ColorBuffer handle on the host (only if h/w access is allowed)
// Only do this for some h/w usages, not all.
// Also do this if we need to read from the surface, in this case the
// rendering will still happen on the host but we also need to be able to
// read back from the color buffer, which requires that there is a buffer
//
if (usage & (GRALLOC_USAGE_HW_TEXTURE | GRALLOC_USAGE_HW_RENDER |
            GRALLOC_USAGE_HW_2D | GRALLOC_USAGE_HW_COMPOSER |
            GRALLOC_USAGE_HW_FB | GRALLOC_USAGE_SW_READ_MASK) ) {
    DEFINE_HOST_CONNECTION;
    // 建立QEMU_PIPE连接后获取Anbox端的数据，这里是建立Anbox端的Buffer，否则Anbox那边
    // 会丢掉。创建了就会把Handle返回给相关接口以便调用。
    if (hostCon && rcEnc) {
        cb->hostHandle = rcEnc->rcCreateColorBuffer(rcEnc, w, h, glFormat);
        D("Created host ColorBuffer 0x%x\n", cb->hostHandle);
    }

    if (!cb->hostHandle) {
        // Could not create colorbuffer on host !!!
        close(fd);
        delete cb;
        return -EIO;
    }
}

// 申请成功，把相关东西返回给对应接口，把申请的handle插入到已申请的列表（链表）
//
// alloc succeeded - insert the allocated handle to the allocated list
//
AllocListNode *node = new AllocListNode();
pthread_mutex_lock(&grdev->lock); // 互斥锁
// 临界区
node->handle = cb;
node->next = grdev->allocListHead;
node->prev = NULL; //单向链表
// 链表头插法
if (grdev->allocListHead) {
    grdev->allocListHead->prev = node;
}
grdev->allocListHead = node;
pthread_mutex_unlock(&grdev->lock); //退出临界区，取消互斥锁

*pHandle = cb;
return 0;
}

//释放图形缓冲区
static int gralloc_free(alloc_device_t* dev,
                        buffer_handle_t handle)
{
    const cb_handle_t *cb = (const cb_handle_t *)handle;
    if (!cb_handle_t::validate((cb_handle_t*)cb)) {
        ERR("gralloc_free: invalid handle");
        return -EINVAL;
    }
}

```

```

// 如果该Handle有在外部Anbox上的ColorBuffer，连接QEMU_PIPE并释放它。
if (cb->hostHandle != 0) {
    DEFINE_AND_VALIDATE_HOST_CONNECTION;
    D("Closing host ColorBuffer 0x%x\n", cb->hostHandle);
    rcEnc->rcCloseColorBuffer(rcEnc, cb->hostHandle);
}

// 释放ashmem申请的区域
//
// detach and unmap ashmem area if present
//
if (cb->fd > 0) {
    if (cb->ashmemSize > 0 && cb->ashmemBase) {
        munmap((void *)cb->ashmemBase, cb->ashmemSize);
    }
    close(cb->fd);
}

// 从已申请的列表（链表实现）移除
// remove it from the allocated list
gralloc_device_t *grdev = (gralloc_device_t *)dev;
pthread_mutex_lock(&grdev->lock); // 互斥锁
// 进入临界区
AllocListNode *n = grdev->allocListHead;
// 找到链表中对应这个Handle
while( n && n->handle != cb ) {
    n = n->next;
}
if (n) {
    // buffer found on list - remove it from list
    // 把Handle移出链表
    if (n->next) {
        n->next->prev = n->prev;
    }
    if (n->prev) {
        n->prev->next = n->next;
    }
    else {
        grdev->allocListHead = n->next;
    }

    delete n;
}
pthread_mutex_unlock(&grdev->lock); //退出临界区，取消互斥锁

delete cb;

return 0;
}

```

以上gralloc_alloc实现了图形缓冲区的申请，gralloc_free实现了缓冲区释放，但是在Anbox中，由于没有实际的图形硬件设备，其图形部分均由软件实现，因此其与图形缓冲区的相关实现均也由软件实现。与之相对应的，实际手机硬件中，这里的缓冲区是来自于硬件，例如通过FrameBuffer或者DRM显示设备进行申请等。

由于Anbox的图形缓冲区是软件实现的，同时在渲染的画面均在Linux这边的Anbox上实现的，这一实现便有了缺点，例如DRM的实现可以通过硬件DMA对图形缓冲器进行处理，而Anbox的软件实现只能靠CPU进行内存拷贝，尤其在录屏的情况下，屏幕数据需要让CPU拷贝到外部进行合成显示，又需要拷贝回来再让编码器进行编码，两趟拷贝浪费了许多CPU与内存的性能。

参考资料

1. [自上而下解读Android显示流程（中上） - 知乎 \(zhihu.com\)](#)
2. [Android图形系统系统篇之HWC leontli的博客-CSDN博客_setclienttarget](#)
3. [Andorid 硬件显示系统HWC&HWC2架构详解coloriy的博客-CSDN博客android display_hwc](#)