内容目录

[一 显示系统框架 1](#__RefHeading___Toc248_2054555503)

[a. 显示驱动framebuffer(fb)的原理及改进 1](#__RefHeading___Toc250_2054555503)

[b. 多任务系统的显示: 必定有一个显示管理者 2](#__RefHeading___Toc252_2054555503)

[二 修改源码禁用hwc和GPU 2](#__RefHeading___Toc254_2054555503)

[2.1 tiny4412 2](#__RefHeading___Toc256_2054555503)

[2.2 qcom 4](#__RefHeading___Toc258_2054555503)

[三 最简单的Surface测试程序 5](#__RefHeading___Toc840_1903101977)

[四 SurfaceFlinger内部机制分析 6](#__RefHeading___Toc1263_1273687975)

[4.1 APP跟SurfaceFlinger之间的重要数据结构 6](#__RefHeading___Toc1265_1273687975)

[4.2 APP创建SurfaceFlinger客户端(client)的过程 6](#__RefHeading___Toc1267_1273687975)

[4.3 APP申请创建Surface的过程 6](#__RefHeading___Toc1269_1273687975)

[4.4 APP申请(lock)Buffer的过程\_框架 + 7](#__RefHeading___Toc1271_1273687975)

[4.5 APP申请(lock)Buffer的过程\_分配buffer + 7](#__RefHeading___Toc1273_1273687975)

[4.6 APP申请(lock)Buffer的过程\_获得buffer信息 7](#__RefHeading___Toc1275_1273687975)

[4.7 APP提交(unlockAndPost)Buffer的过程\_框架 + 7](#__RefHeading___Toc1277_1273687975)

[4.8 APP提交(unlockAndPost)Buffer的过程\_消费者创建过程 + 7](#__RefHeading___Toc1279_1273687975)

[4.9 APP提交(unlockAndPost)Buffer的过程\_提交过程 7](#__RefHeading___Toc1281_1273687975)

[五. Vsync机制 7](#__RefHeading___Toc1283_1273687975)

[5.1 黄油计划\_三个方法改进显示系统 7](#__RefHeading___Toc1285_1273687975)

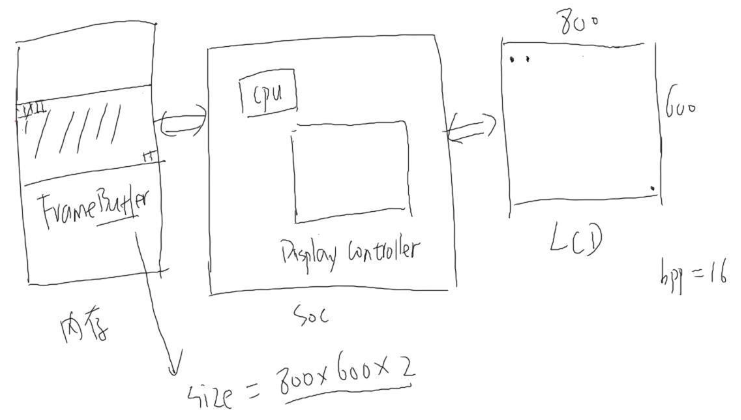
[5.2 Vsync框架 7](#__RefHeading___Toc1764_1519259852)

[5.3 初始化代码分析 7](#__RefHeading___Toc1766_1519259852)

# 一 显示系统框架

显示系统001\_框架.jpg

## a. 显示驱动framebuffer(fb)的原理及改进

只有1个fb的缺点

(1) 如果APP写fb速度慢，LCD图像变化慢

(2) 如果APP写fb速度不快不慢，闪烁

(3) APP写fb速度快--OK

改进：使用多个FB，循环 while(1) {

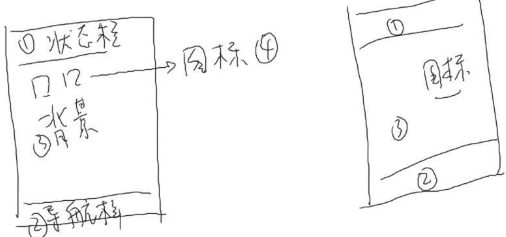
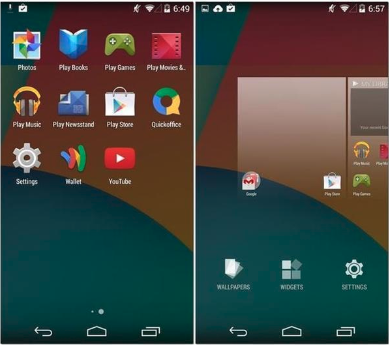
(1) Display Controller使用FB0

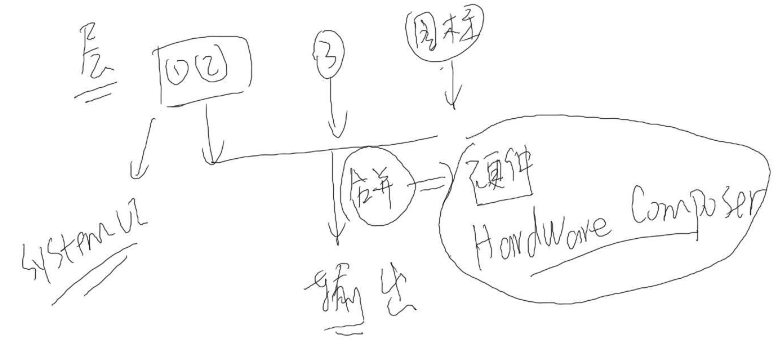
(2) APP写fb1

(3) Display Controller使用FB1

(4) APP写fb0

}

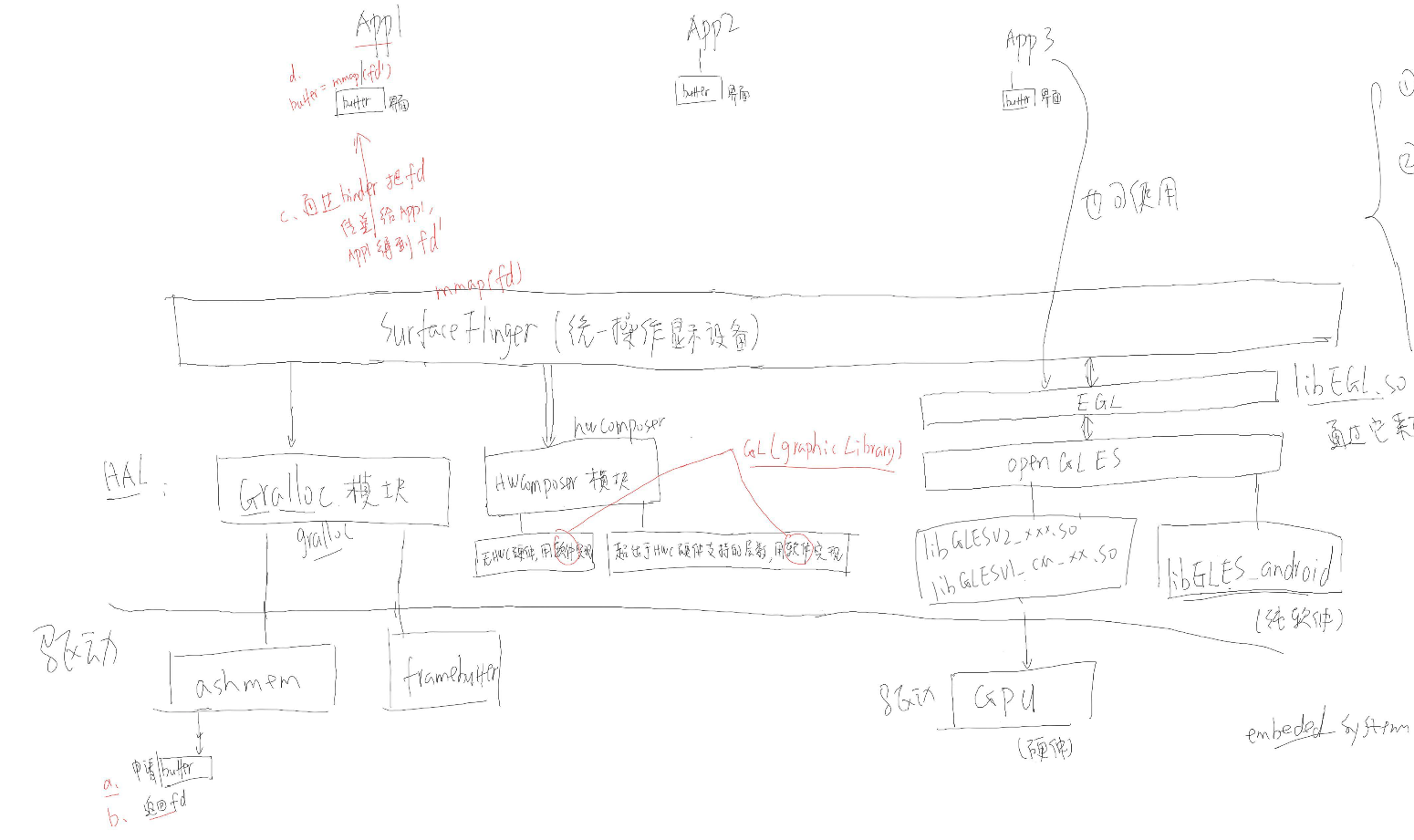


 重复工作1、2、3，其中1和2是systemUI进程，3是进程2，图标是进程3，如果芯片支持合成HardwareComposer那么可以提升性能。

驱动支持HWC:

每一层对应一个驱动/dev/fbx，APP操作某层，直接写对应的framebuffer，硬件自动合成他们。

## b. 多任务系统的显示: 必定有一个显示管理者

 跟高通文档的架构很相近，只不过高通底层是SDM，这边是通用的framebuffer

sufraceflinger:

(1)给APP提供buffer

a.通过gralloc模块向ashmem申请内存

b.得到一个fd

c.通过binder把fd传给某个APP，APP得到fd’

d.APP再mmap(fd’)

(2)APP1、2、3把各自界面发给它，它根据层次、大小进行合成、显示

a.根据各个界面的Z值决定前后顺序，由WindowManagerService确定

b.把这些排序后的buffer传给HWComposer

(3)当HWC不能处理(无HWC硬件/超出HWC层数)buffer时，使用GL(GraphicLibrary)来处理

libEGL硬件GL库、软件GL库

# 二 修改源码禁用hwc和GPU

厂家一般不会提供硬件合成器和GPU源代码，无法分析

## 2.1 tiny4412

git clone https://github.com/weidongshan/SYS\_0003\_Patch\_Disable\_HWC\_GPU\_tiny4412.git

git checkout v1

android-5.0.2\_no\_hwc\_no\_gpu.patch

这个补丁做了3件事:

**a. 去掉厂家提供的gralloc, hwcompser HAL模块**

hardware/libhardware/hardware.c

#if defined(\_\_LP64\_\_)

#define HAL\_LIBRARY\_PATH1 "/system/lib64/hw"

#define HAL\_LIBRARY\_PATH2 "/vendor/lib64/hw"

#define HAL\_LIBRARY\_PATH3 "/odm/lib64/hw"

#else

#define HAL\_LIBRARY\_PATH1 "/system/lib/hw"

#define HAL\_LIBRARY\_PATH2 "/vendor/lib/hw"

#define HAL\_LIBRARY\_PATH3 "/odm/lib/hw"

#endif

static const char \*variant\_keys[] = {

"ro.hardware", /\* This goes first so that it can pick up a different

file on the emulator. \*/

"ro.product.board",

"ro.board.platform",

"ro.arch"

};

hw\_get\_module\_by\_class //查找，load so库

文件名gralloc.属性值.so: gralloc.tiny4412.so, gralloc.exynos4.so，最后 gralloc.default.so

**a.1 删除单板上/system/lib/hw**

gralloc.tiny4412.so

hwcomposer.exynos4.so

adb reboot之后，黑屏一片

查看错误信息

logcat \*:E

hwcomposer module not found

invalid buffer handle given //这个错误视频中查找不到源代码，是厂家另外一个库文件，把相关的都可以干掉/system/lib/egl/，不干掉也没事，因为后续要修改属性

**a.2 修改源码使得编译结果中不含上述文件 (修改vendor/friendly-arm/tiny4412/device-tiny4412.mk, 参考补丁)**

-

**b. 添加属性让android系统认为自己运行于"没有GPU的模拟器"**

修改libagl/Android.mk，给系统添加libGLES\_android.so (软件实现的GL)

**b.1 添加软件GPU库**

开发板: su, mount -o remount /system

在服务器编译软件GPU库: mmm frameworks/native/opengl/libagl

把得到的libGLES\_android.so复制到单板/system/lib/eg/，并添加读属性

adb reboot之后

couldn’t find an OpenGL ES implementation

搜索之后，错误文件

frameworks/native/opengl/libs/EGL/Loader.cpp

找到加载libGLES\_android.so的代码，发现需要修改属性值

**b.2 修改frameworks/native/opengl/libagl/Android.mk (参考补丁)**

-

**b.3 修改属性文件 , 单板 adb shell 进去修改 /system/build.prop，添加:**

ro.kernel.qemu=1

ro.kernel.qemu.gles=0

或修改源码 device/friendly-arm/tiny4412/system.prop 同样添加上述属性, 然后重新编译系统

adb reboot之后

no suitable EGLConfig found, giving up

**c. 修改系统自带的gralloc模块的BUG**

**c.1 查找错误文件**

frameworks/native/services/surfaceflinger/RenderEngine/RenderEngine.cpp

EGLConfig RenderEngine::chooseEglConfig(EGLDisplay display, int format, bool logConfig) {

尝试获得ES2的配置，如果失败尝试获得ES1的配置(硬线相关的)，否则使用简单查询方式来获得配置

}

经过debug，怀疑是配置问题

hardware/libhardware/modules/gralloc/Framebuffer.cpp

HAL\_PIXEL\_FORMAT\_BGRA\_8888

改为：

HAL\_PIXEL\_FORMAT\_RGBA\_8888

mmm hardware/libhardware/modules/gralloc/

把gralloc.default.so复制到单板/system/lib/hw

adb reboot之后，出错:

E/BufferQueueProducer( 2320): [FramebufferSurface] dequeueBuffer: createGraphicBuffer failed

W/GraphicBufferAllocator( 1918): alloc(800, 480, 1, 00001a33, ...) failed -12 (Out of memory)

**c.2 继续修改 hardware\libhardware\modules\gralloc\Framebuffer.cpp**

frameworks/native/libs/ui/GraphicBufferAllocator.cpp

const std::unique\_ptr<const Gralloc2::Allocator> mAllocator;

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)

{

。。。

Gralloc2::Error error = mAllocator->allocate(info, stride, handle);//打开Gralloc模块涉及的结构体

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;

}

}

经过code flow追查

hardware/libhardware/modules/gralloc/gralloc.cpp

dev->device.alloc = gralloc\_alloc;

static int gralloc\_alloc(alloc\_device\_t\* dev,

int width, int height, int format, int usage,

buffer\_handle\_t\* pHandle, int\* pStride)

{

。。。

int err;

if (usage & GRALLOC\_USAGE\_HW\_FB) {

err = gralloc\_alloc\_framebuffer(dev, size, usage, pHandle);//从framebuffer里面分配内存，错误信息usage=1a33

if (bufferMask >= ((1LU<<numBuffers)-1)) {

// We ran out of buffers.

return -ENOMEM;//-12

}

} else {

err = gralloc\_alloc\_buffer(dev, size, usage, pHandle);//从ashmem里面分配内存

}

if (err < 0) {

return err;

}

\*pStride = stride;

return 0;

}

hardware/libhardware/modules/gralloc/framebuffer.cpp

+#if 0//不需要再申请framebuffer

/\*

\* Request NUM\_BUFFERS screens (at lest 2 for page flipping)

\*/

info.yres\_virtual = info.yres \* NUM\_BUFFERS;

- uint32\_t flags = PAGE\_FLIP;

#if USE\_PAN\_DISPLAY

if (ioctl(fd, FBIOPAN\_DISPLAY, &info) == -1) {

ALOGW("FBIOPAN\_DISPLAY failed, page flipping not supported");

@@ -195,6 +206,16 @@

info.yres\_virtual = info.yres;

flags &= ~PAGE\_FLIP;

}

+#endif//直接获取即可

+

+ if (ioctl(fd, FBIOGET\_VSCREENINFO, &info) == -1)

+ return -errno;

+ info.yres\_virtual = info.yres\_virtual;//虚拟分辨率，比如800×600，虚拟y可以分配3个800，而yres为800

+ if (info.yres\_virtual > info.yres)

+ flags |= PAGE\_FLIP;

+ else

+ flags &= ~PAGE\_FLIP;

+

hardware/libhardware/modules/gralloc/framebuffer.cpp

+#if 0

if (ioctl(m->framebuffer->fd, FBIOPUT\_VSCREENINFO, &m->info) == -1) {

ALOGE("FBIOPUT\_VSCREENINFO failed");

m->base.unlock(&m->base, buffer);

return -errno;

}

+#else

+ if (ioctl(m->framebuffer->fd, FBIOPAN\_DISPLAY, &m->info) == -1) {//通过FBIOPAN\_DISPLAY 来确定使用那个buffer

+ ALOGE("FBIOPAN\_DISPLAY failed");

+ m->base.unlock(&m->base, buffer);

+ return -errno;

+ }

+#endif

+

**重启之后成功!**

## **2.2 qcom**

**gralloc**

源码位置

hardware/qcom/display/gralloc

LOCAL\_MODULE := gralloc.$(TARGET\_BOARD\_PLATFORM)

库位置

sm6150\_au:/vendor/lib64/hw # ls gralloc.\*.so

gralloc.default.so gralloc.sm6150.so

sm6150\_au:/ # getprop ro.hardware

qcom

sm6150\_au:/ # getprop ro.product.board

sm6150

sm6150\_au:/ # getprop ro.board.platform

sm6150

sm6150\_au:/ # getprop ro.arch

属性值

ro.kernel.qemu

qemu.gles

# 三 最简单的Surface测试程序

git clone https://github.com/weidongshan/APP\_0010\_SurfaceTest.git

Display/APP\_0010\_SurfaceTest

参考demo

frameworks/native/services/surfaceflinger/tests/resize

错误fix

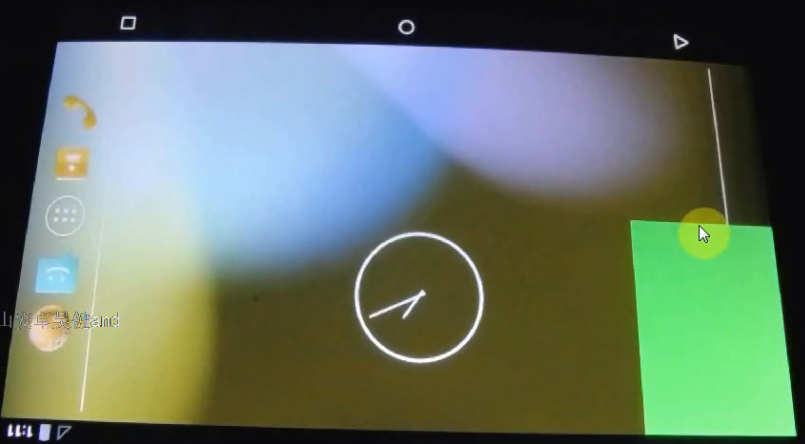
<http://www.aichengxu.com/android/8815305.htm>

取出指定版本:

git checkout v1 // v1, correct the bugs from frameworks/native/services/surfaceflinger/tests/resize

git checkout v2 // v2, display B,G,R color

git checkout v3 // v3, print the buffer address



v1版本编译后多了一个绿色的长方块

int main(int argc, char\*\* argv)

{

// set up the thread-pool

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

ProcessState::self()->startThreadPool();

// create a client to surfaceflinger

sp<SurfaceComposerClient> client = new SurfaceComposerClient();//获得surface服务

sp<SurfaceControl> surfaceControl = client->createSurface(String8("resize"),//创建surface

160, 240, PIXEL\_FORMAT\_RGB\_565, 0);

sp<Surface> surface = surfaceControl->getSurface();//获得surface

SurfaceComposerClient::openGlobalTransaction();

surfaceControl->setLayer(100000);//设置Z轴，为了覆盖用，dumpsys SurfaceFlinger可以查看到z轴的大小

SurfaceComposerClient::closeGlobalTransaction();

ANativeWindow\_Buffer outBuffer;

surface->lock(&outBuffer, NULL);//获得surface的一个buffer

ssize\_t bpr = outBuffer.stride \* bytesPerPixel(outBuffer.format);

android\_memset16((uint16\_t\*)outBuffer.bits, 0xF800, bpr\*outBuffer.height);//填充buffer，0xF800一种颜色

surface->unlockAndPost();//把buffer提交给surfaceflinger让它显示出来

+ sleep(3);//加入休眠，为了看出变化(v2版本主要改动，也加入了其他颜色)

surface->lock(&outBuffer, NULL);//获得surface的另一个buffer

android\_memset16((uint16\_t\*)outBuffer.bits, 0x07E0, bpr\*outBuffer.height);//填充buffer，另外一种颜色

surface->unlockAndPost();//再次提交

+ sleep(3);

SurfaceComposerClient::openGlobalTransaction();

surfaceControl->setSize(320, 240);

SurfaceComposerClient::closeGlobalTransaction();

+ for (int i = 0; i < 100; i++) {//v3

+ surface->lock(&outBuffer, NULL);

+ printf("%03d buff addr = 0x%x\n", i, (unsigned int)outBuffer.bits);

+ surface->unlockAndPost();

+ }

输出结果，应用程序对于一个surface，分配了三个buffer

000 buff addr = 0x4003e000

001 buff addr = 0x40083000

002 buff addr = 0x403be000

003 buff addr = 0x4003e000

004 buff addr = 0x40083000

005 buff addr = 0x403be000

...

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

return 0;

}

# 四 SurfaceFlinger内部机制分析

调用关系uml工程: uml\_tmp\_file\uml\_tmp\_file\surface\_uml\surface\_uml.prj

## 4.1 APP跟SurfaceFlinger之间的重要数据结构

一个应用程序可以构造多个surface，一般来说只有一个surface，一个surface里面可以有多个buffer

这些buffer需要向SurfaceFlinger来申请

先写出结论:

用client表示APP，多个APP那就有多个client，

client中用Layer，用来表示APP的SurfaceControl

frameworks/native/services/surfaceflinger/Layer.h

Layer{

生产者 -- sp<IGraphicBufferProducer> mProducer;

消费者 -- sp<SurfaceFlingerConsumer> mSurfaceFlingerConsumer;

}//生产者从buffer中放入data，消费者从buffer中取出data

从本文来说，Layer里面的生产者和消费者拥有的是同一个BufferQueueCore

mProducer和mSurfaceFlingerConsumer有同一个mCore(sp<BufferQueueCore> mCore;)

frameworks/native/include/gui/BufferQueueCore.h

BufferQueueCore有一个mSlots(BufferSlot[64]数组)，这意味着APP里每一个surface里面最多可以有64个buffer，这个BufferSlot每一项中有一个mGraphicBuffer，用来表示buffer

BufferQueueDefs::SlotsType mSlots;//SlotsType是一个数组

frameworks/native/libs/gui/include/gui/BufferQueueDefs.h

typedef BufferSlot SlotsType[NUM\_BUFFER\_SLOTS];

frameworks/native/include/gui/BufferSlot.h

sp<GraphicBuffer> mGraphicBuffer;//用来表示一个buffer

打开之前的SurfaceFlinger测试程序，

sp<SurfaceComposerClient> client = new SurfaceComposerClient();//每个APP跟SurfaceFlinger都有一个连接，表示这个连接

native/include/gui/SurfaceComposerClient.h

sp<ISurfaceComposerClient> mClient;//有一个mClient(APP里)，它指向的就是SurfaceFlinger的client

sp<SurfaceContorl> surfaceControl = client->createSurface(String8("resize"), 160, 240, PIXEL\_FORMAT\_RGB\_565, 0)//每一个Surface对应有一个SurfaceControl

sp<Surface> surface = surfaceControl->getSurface();//再从SurfaceContorl里面得到我们的surface，所以说surface是用SurfaceContorl来管理的，SurfaceControl对应SurfaceFlinger的Layer

surface->lock(&outBuffer, NULL);//从surface里面取出buffer

frameworks/native/include/gui/Surface.h

BufferSlot mSlots[NUM\_BUFFER\_SLOTS];//一个surface里有一个数组对应SurfaceFlinger的mSlots BufferSlot[64]数组，每一个数组项含有一个mGraphicBuffer，表示buffer，双方的buffer指向同一个物理内存

小结:

APP发起请求，SurfaceFlinger接收到请求之后，使用Gralloc模块从匿名共享内存中分配出一块内存，然后记录在mGraphicBuffer的buffer里面，将这个信息返回给APP，APP再根据这个信息构造出对应的mGraphicBuffer

## 4.2 APP创建SurfaceFlinger客户端(client)的过程

//创建client

sp<SurfaceComposerClient> client = new SurfaceComposerClient();//获得surfaceflinger服务, class Client : public BnSurfaceComposerClient

uml\_tmp\_file\uml\_tmp\_file\surface\_uml\surface\_uml.prj -- 001\_create\_client A1.

## 4.3 APP申请创建Surface的过程

//1 创建surfaceControl

sp<SurfaceControl> surfaceControl = client->createSurface(String8("resize"), 160, 240, PIXEL\_FORMAT\_RGB\_565, 0);

sp<Surface> surface = surfaceControl->getSurface();

binder proxy端发起createSurface{

mClient->createSurface(..., &handle, &gpb)//主要就是这个代理类gpb最终指向的是surfaceflinger端的BufferQueueProducer，他们都遵守了IGraphicBufferProducer

}

生产者(代理类){

1. APP - surfaceflinger与之对应一个client

2. SurfaceControl - surfaceflinger与之对应一个Layer(本例是createNormalLayer-> new Layer){.mProducer(MonitorProducer); .mProducer里面还有一个mProducer(BufferQueueProducer)}

3. .gpb指向mProducer(BufferQueueProducer)，

}

//2 获得surface，surface里面封装有一个生产者，还有一个mSlot[64]，后续的获得buffer、提交buffer肯定都跟他有关

sp<Surface> surface = surfaceControl->getSurface();

-> mSurfaceData = new Surface(mGraphicBufferProducer, false);//mGraphicBufferProducer就是gpb，这个就是后续操作显存的核心

## 4.4 APP申请(lock)Buffer的过程\_框架 +

## 4.5 APP申请(lock)Buffer的过程\_分配buffer +

## 4.6 APP申请(lock)Buffer的过程\_获得buffer信息

surface->lock(&outBuffer, NULL);

APP获得Surface中的Buffer，猜测一下：

a. 查看mSlots中有无空余项(代码里没有这一项)

b. 若无，向生产者申请，SurfaceFlinger进程的动作

b.1 查看mSlots中有无空余项

b.2 若无，向Gralloc HAL申请

b.3 返回fd给APP

c. 获得fd'，mmap获得地址，通过Gralloc HAL来mmap

//以上是一系列复杂调用，最终导致APP获得了SurfaceFlinger端的分配的虚拟地址和显存信息

## 4.7 APP提交(unlockAndPost)Buffer的过程\_框架 +

## 4.8 APP提交(unlockAndPost)Buffer的过程\_消费者创建过程 +

## 4.9 APP提交(unlockAndPost)Buffer的过程\_提交过程

surface->unlockAndPost();

猜测unlockAndPost会做什么事情: 入队列，通知消费者，消费者通知Layer，Layer通知client，client通知SurfaceFlinger，SurfaceFlinger是消费者

# 五. Vsync机制

## 5.1 黄油计划\_三个方法改进显示系统

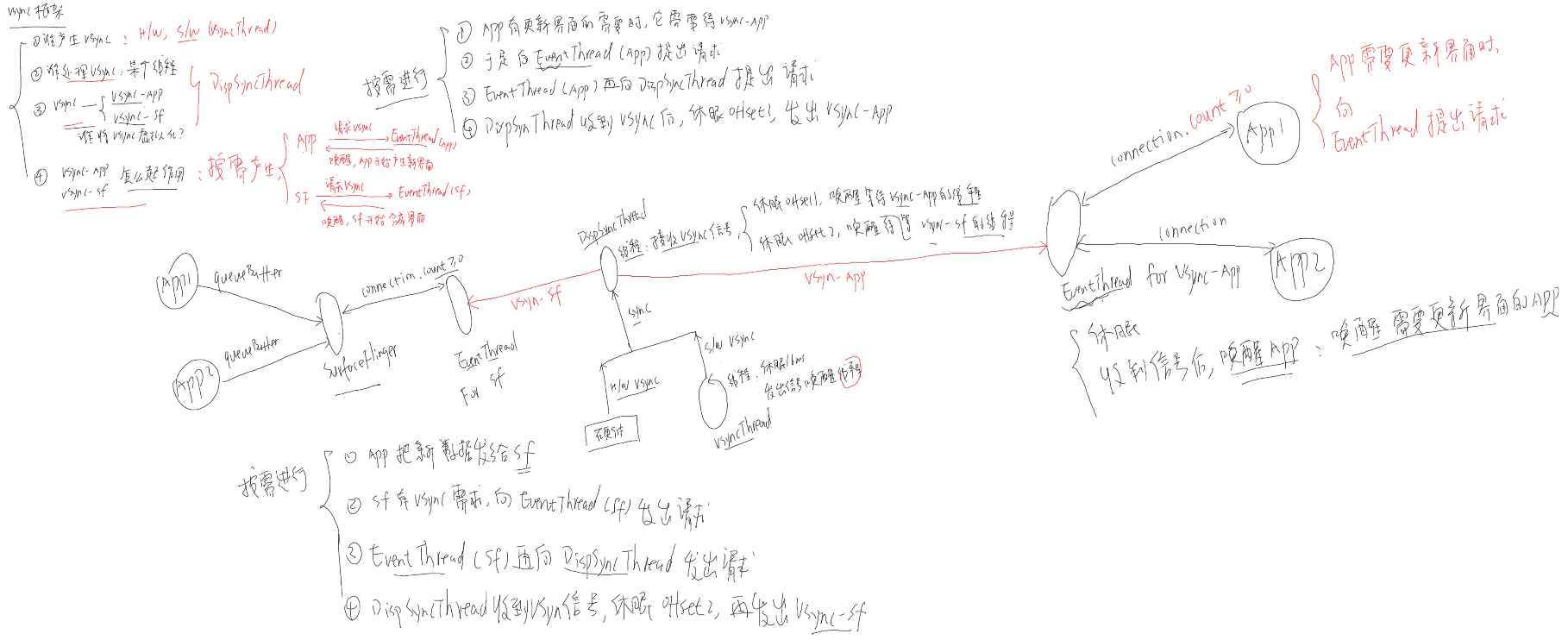
高清电影或者玩游戏的时候，帧率要达到60fps以上，才比较舒服比较流畅，即1秒要显示60幅图像，也就是16毫秒要显示一幅图像

利用vsync机制，vysnc到来时各单位必须及时工作；三个buffer；vsync分为两路，一路是app(vsync+offset1)，一路是sf(sync+offset2)，这样使得app构造好的新画面，sf紧接着合成

## 5.2 Vsync框架

## 5.3 初始化代码分析

显示系统009\_VSYNC机制\_框架.jpg，图中s/w vsync线程、DispSyncThread、eventThread-for-vsync-app、eventThread-for-vsync-sf、surfaceflinger线程，都在surfaceflinger进程中。这节主要目的就是看看以上这五个线程的创建。



frameworks/native/services/surfaceflinger/main\_surfaceflinger.cpp

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

signal(SIGPIPE, SIG\_IGN);

hardware::configureRpcThreadpool(1 /\* maxThreads \*/,

false /\* callerWillJoin \*/);

startGraphicsAllocatorService();

// 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，创建surfaceflinger对象

sp<SurfaceFlinger> flinger = DisplayUtils::getInstance()->getSFInstance();

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

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

// Put most SurfaceFlinger threads in the system-background cpuset

// Keeps us from unnecessarily using big cores

// Do this after the binder thread pool init

if (cpusets\_enabled()) set\_cpuset\_policy(0, SP\_SYSTEM);

// initialize before clients can connect

flinger->init();// DispSyncThread创建出来，找到第一个

// publish surface flinger

sp<IServiceManager> sm(defaultServiceManager());

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

// publish GpuService

sp<GpuService> gpuservice = new GpuService();

sm->addService(String16(GpuService::SERVICE\_NAME), gpuservice, false);

startDisplayService(); // dependency on SF getting registered above

struct sched\_param param = {0};

param.sched\_priority = 2;

if (sched\_setscheduler(0, SCHED\_FIFO, &param) != 0) {

ALOGE("Couldn't set SCHED\_FIFO");

}

// run surface flinger in this thread

flinger->run();//找到第五个

return 0;

}

frameworks/native/services/surfaceflinger/SurfaceFlinger.h

class SurfaceFlinger

{

DispSync mPrimaryDispSync;

mutable MessageQueue mEventQueue;

}

frameworks/native/services/surfaceflinger/DispSync.cpp

DispSync::DispSync(const char\* name) :

mName(name),

mRefreshSkipCount(0),

mThread(new DispSyncThread(name)) {

}

void DispSync::init(bool hasSyncFramework, int64\_t dispSyncPresentTimeOffset) {

mIgnorePresentFences = !hasSyncFramework;

mPresentTimeOffset = dispSyncPresentTimeOffset;

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

…

}

frameworks/native/services/surfaceflinger/SurfaceFlinger.cpp

void SurfaceFlinger::onFirstRef()

{

mEventQueue.init(this);//sp被首次引用时，初始化一个消息队列

}

void SurfaceFlinger::init() {//我们只看vsync相关的

// start the EventThread，offset和名字不一样，找到第三个，第四个

sp<VSyncSource> vsyncSrc = new DispSyncSource(&mPrimaryDispSync,

vsyncPhaseOffsetNs, true, "app");

mEventThread = new EventThread(vsyncSrc, \*this, false);

sp<VSyncSource> sfVsyncSrc = new DispSyncSource(&mPrimaryDispSync,

sfVsyncPhaseOffsetNs, true, "sf");

mSFEventThread = new EventThread(sfVsyncSrc, \*this, true);

mEventQueue.setEventThread(mSFEventThread);

mHwc.reset(new HWComposer(false));

}

void SurfaceFlinger::run() {

do {

waitForEvent();

} while (true);

}

frameworks/native/services/surfaceflinger/DisplayHardware/HWComposer\_hwc1.cpp – 版本比较旧了

HWComposer::HWComposer(…) {

...

if (needVSyncThread) {

// we don't have VSYNC support, we need to fake it

mVSyncThread = new VsyncThread(\*this);//软件模拟产生vsync信号，找到第二个

}

}

void HWComposer::VSyncThread::onFirstRef() {

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

}

frameworks/native/services/surfaceflinger/MessageQueue.cpp

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

{

if (mEventThread == eventThread) {

return;

}

if (mEventTube.getFd() >= 0) {

mLooper->removeFd(mEventTube.getFd());

}

mEventThread = eventThread;

mEvents = eventThread->createEventConnection();

mEvents->stealReceiveChannel(&mEventTube);

mLooper->addFd(mEventTube.getFd(), 0, Looper::EVENT\_INPUT,//sf和EventThread之间通过文件句柄进行数据count的传递

MessageQueue::cb\_eventReceiver, this);

}

聂神给的一块代码

if (!timestamp && !eventPending) {

// wait for something to happen

if (waitForVSync) {

bool softwareSync = mUseSoftwareVSync;//这里只考虑硬件vsync的情况,软件模拟的暂时不考虑

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

//如注释所说的，如果是driver的bug，如果硬件一直不上报vsync事件怎么办？？难道就一直等下去？？那client不就饿死了么？

//所以这里如果driver不报vsync，那么就软件模拟一个vsync事件，这里的timeout是1000ms，发一个

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

if (!softwareSync) {

ALOGW("Timed out waiting for hw vsync; faking it");

}

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

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

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

mVSyncEvent[0].vsync.count++;

}

} else {

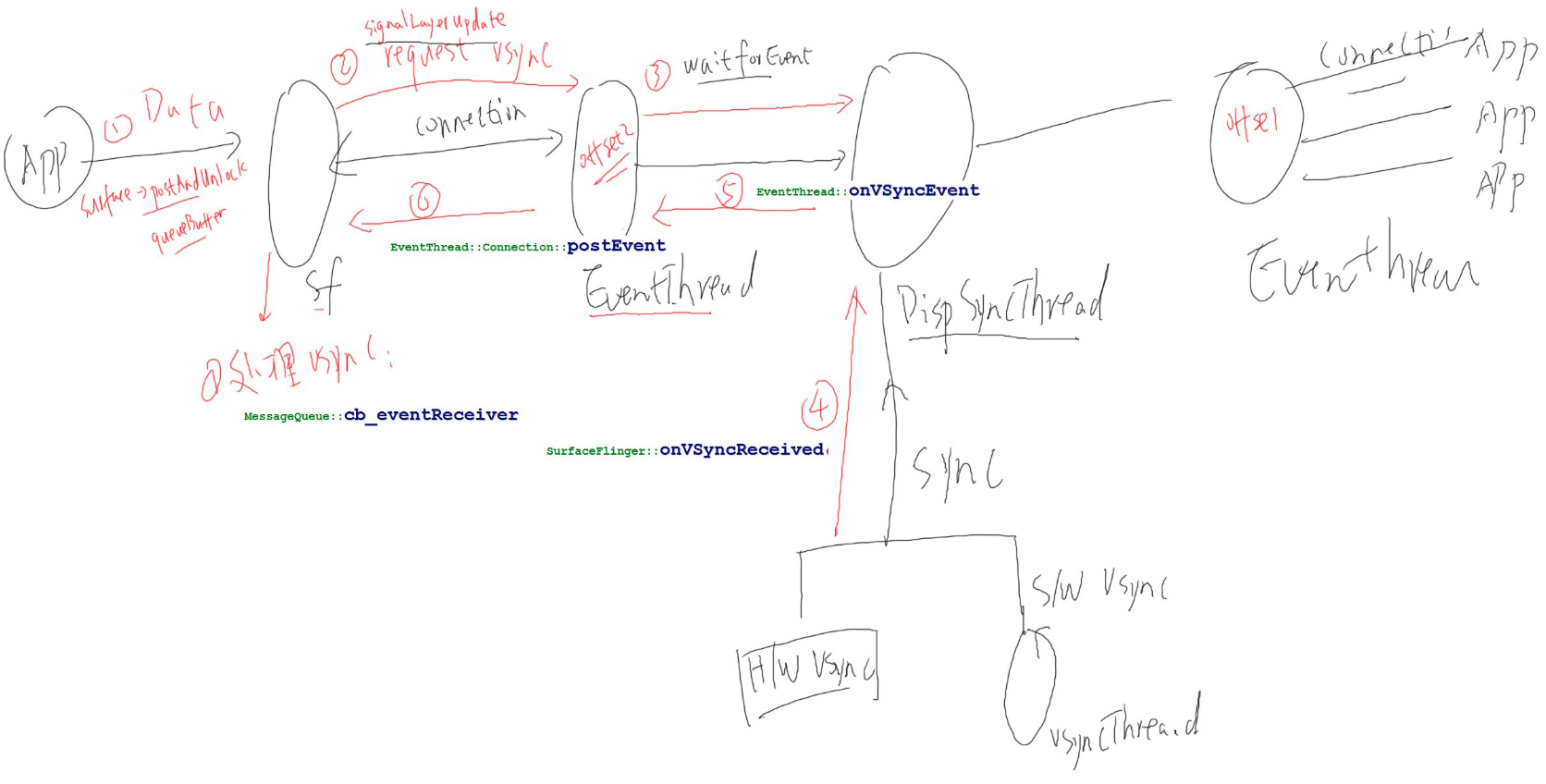
//既没有client, 又没有硬件vsync事件，那么就死等下去了。

mCondition.wait(mLock);

}

}

## 5.4 surfaceflinger使用vsync过程代码分析



### 5.4.1 APP->SF

步骤一，app是怎么提交buf到sf的，在《显示系统007\_APP提交buffer的过程.jpg》已经分析完了，不再赘述

### 5.4.2 SF->EventThread

步骤二，发送vsync请求，从mFlinger->singnalLayerUpdate()开始看，我们尽量看HWC2版本的代码。

frameworks/native/services/surfaceflinger/SurfaceFlinger.cpp

void SurfaceFlinger::signalLayerUpdate() {

mEventQueue.invalidate();

}

frameworks/native/services/surfaceflinger/MessageQueue.cpp

void MessageQueue::invalidate() {

mEvents->requestNextVsync();//请求得到下一个vsync信号

}

frameworks/native/services/surfaceflinger/EventThread.cpp

void EventThread::Connection::requestNextVsync() {

mEventThread->requestNextVsync(this);

}

void EventThread::requestNextVsync(

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

Mutex::Autolock \_l(mLock);

mFlinger.resyncWithRateLimit();

if (connection->count < 0) {

connection->count = 0;//大于等于0时，表示需要vsync信号

mCondition.broadcast();//目的是唤醒EventThread线程

}

}

### 5.4.3 EventThread->DispSyncThread

步骤三，发送vsync请求

frameworks/native/services/surfaceflinger/EventThread.cpp

bool EventThread::threadLoop() {

//1. 向DispSyncThread发出vsync请求

//2. 等待vsync信号

signalConnections = waitForEvent(&event);

status\_t err = conn->postEvent(event);//步骤6

}

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

DisplayEventReceiver::Event\* event)

{

do {

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

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

if (connection != NULL) {

bool added = false;

if (connection->count >= 0) {//判断每一个connection→count是否大于等于0，标明它需要得到下一个vsync信号，注册回调

// we need vsync events because at least

// one connection is waiting for it

waitForVSync = true;

。。。

} else if (!timestamp && waitForVSync) {

enableVSyncLocked();

}

。。。

mCondition.wait(mLock);//休眠，等待DispSyncThread来唤醒

} while (signalConnections.isEmpty());

}

void EventThread::enableVSyncLocked() {

if (!mUseSoftwareVSync) {

// never enable h/w VSYNC when screen is off

if (!mVsyncEnabled) {

mVsyncEnabled = true;

mVSyncSource->setCallback(static\_cast<VSyncSource::Callback\*>(this));//给DispSyncThread设置callback

mVSyncSource->setVSyncEnabled(true);

}

}

mDebugVsyncEnabled = true;

sendVsyncHintOnLocked();

}

### 5.4.4 H/W vsync->DispSyncThread

软件产生

bool HWComposer::VSyncThread::threadLoop() {

。。。

int err;

do {

err = clock\_nanosleep(CLOCK\_MONOTONIC, TIMER\_ABSTIME, &spec, NULL);//休眠一段时间

} while (err<0 && errno == EINTR);

if (err == 0) {

//发出vsync信号，无论是HW还是SW vsync，最终调用到void SurfaceFlinger::onVSyncReceived，Android 8.1查看code flow，硬件vsync是从hw binder传上来的

mHwc.mEventHandler.onVSyncReceived(&mHwc, 0, next\_vsync);

}

return true;

}

SurfaceFlinger.cpp

void SurfaceFlinger::onVsyncReceived(int32\_t sequenceId, hwc2\_display\_t displayId, int64\_t timestamp)

→ needsHwVsync = mPrimaryDispSync.addResyncSample(timestamp);

→ mThread->updateModel(mPeriod, mPhase, mReferenceTime);

→ void updateModel(nsecs\_t period, nsecs\_t phase, nsecs\_t referenceTime)//DispSync.cpp

→ mCond.signal();//mCond 是class DispSyncThread的成员，来唤醒DispSyncThread

### 5.4.5 + 5.4.6 DispSyncThread->EventThread->SF

DispSync.cpp

virtual bool threadLoop() {

targetTime = computeNextEventTimeLocked(now);//计算最近的EventThread{Listener}的时间，EventThread发现connection->count大于等于0时，向DispSyncThread注册callback，变成listener

if (now < targetTime) {

err = mCond.waitRelative(mMutex, targetTime – now);//休眠，被mCond.signal()唤醒

callbackInvocations = gatherCallbackInvocationsLocked(now);//收集callback

fireCallbackInvocations(callbackInvocations);//调用callback，最终会导致EventThread中onVsyncEvent函数被调用

}

EventThread.cpp

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

Mutex::Autolock \_l(mLock);

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

mVSyncEvent[0].header.id = 0;

mVSyncEvent[0].header.timestamp = timestamp;

mVSyncEvent[0].vsync.count++;

mCondition.broadcast();//还是老套路，条件发送信号，唤醒 EventThread::threadLoop

}

//该到步骤6了

bool EventThread::threadLoop() {

//1. 向DispSyncThread发出vsync请求

//2. 等待vsync信号

signalConnections = waitForEvent(&event);

status\_t err = conn->postEvent(event);//步骤6，向sf发出信号，或者向app发出信号

}

status\_t EventThread::Connection::postEvent(

const DisplayEventReceiver::Event& event) {

ssize\_t size = DisplayEventReceiver::sendEvents(&mChannel, &event, 1);//通过connection里面的fd写给sf，BitTube，封装了unix套接字

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

}

### 5.4.7 sf对vsync的处理

在5.3节中 SurfaceFlinger::init会注册cb\_eventReceiver这个回调，

MessageQueue.cpp

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

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

return queue->eventReceiver(fd, events);

→ mHandler->dispatchInvalidate();

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

}

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

switch (message.what) {

case INVALIDATE:

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

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

break;

case REFRESH:

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

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

break;

}

}

SurfaceFlinger.cpp

void SurfaceFlinger::onMessageReceived(int32\_t what) {//后续章节再分析

ATRACE\_CALL();

switch (what) {

case MessageQueue::INVALIDATE: {

bool refreshNeeded = handleMessageTransaction();

refreshNeeded |= handleMessageInvalidate();

refreshNeeded |= mRepaintEverything;

signalRefresh();

}

break;

}

case MessageQueue::REFRESH: {

handleMessageRefresh();

break;

}

}

}

## 5.5 surfaceflinger对vsync的处理

SurfaceFlinger.h

class State {

public:

explicit State(LayerVector::StateSet set) : stateSet(set) {}

State& operator=(const State& other) {

layersSortedByZ = other.layersSortedByZ;

displays = other.displays;

return \*this;

}

const LayerVector::StateSet stateSet = LayerVector::StateSet::Invalid;

LayerVector layersSortedByZ;//比较它，即知Layer的增/减

DefaultKeyedVector< wp<IBinder>, DisplayDeviceState> displays;//比较它，即知Display的增/减，add/remove

void traverseInZOrder(const LayerVector::Visitor& visitor) const;

void traverseInReverseZOrder(const LayerVector::Visitor& visitor) const;

};

State mDrawingState{LayerVector::StateSet::Drawing};//正在 / 上次使用的状态，SurfaceFlinger下一帧的状态

State mCurrentState{LayerVector::StateSet::Current};//当前的 / 被修改的状态，当前正在绘制的状态，这个是SurfaceFlinger处理完事务后更新出来的状态，是最终的状态

Layer.h

struct State {

Geometry active;

Geometry requested;

int32\_t z;

uint32\_t layerStack;//若和DisplayDeviceState ::layerStack 相等，该Layer可以在这个Display显示

#ifdef USE\_HWC2

float alpha;

#else

uint8\_t alpha;

#endif

uint8\_t flags;

uint8\_t mask;

uint8\_t reserved[2];

int32\_t sequence; // changes when visible regions can change //比较它，即知属性有无变化

bool modified;

Rect crop;

Rect requestedCrop;

Rect finalCrop;

Rect requestedFinalCrop;

wp<Layer> barrierLayer;

uint64\_t frameNumber;

Region activeTransparentRegion;

Region requestedTransparentRegion;

android\_dataspace dataSpace;

uint32\_t appId;

uint32\_t type;

wp<Layer> zOrderRelativeOf;

SortedVector<wp<Layer>> zOrderRelatives;

};

State mCurrentState;//正在 / 上次使用的状态

State mDrawingState;//当前的 / 被修改的状态

DisplayDevice.h

struct DisplayDeviceState {

DisplayDeviceState() = default;

DisplayDeviceState(DisplayDevice::DisplayType type, bool isSecure);

bool isValid() const { return type >= 0; }

bool isMainDisplay() const { return type == DisplayDevice::DISPLAY\_PRIMARY; }

bool isVirtualDisplay() const { return type >= DisplayDevice::DISPLAY\_VIRTUAL; }

static std::atomic<int32\_t> nextDisplayId;

int32\_t displayId = nextDisplayId++;

DisplayDevice::DisplayType type = DisplayDevice::DISPLAY\_ID\_INVALID;

sp<IGraphicBufferProducer> surface;

uint32\_t layerStack = DisplayDevice::NO\_LAYER\_STACK;

Rect viewport;

Rect frame;

uint8\_t orientation = 0;

uint32\_t width = 0;

uint32\_t height = 0;

String8 displayName;

bool isSecure = false;

};

SurfaceFlinger.cpp

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

ATRACE\_CALL();

switch (what) {

case MessageQueue::INVALIDATE: {

bool refreshNeeded = handleMessageTransaction();//处理事务

refreshNeeded |= handleMessageInvalidate();//处理各Layer的Buffer更换

refreshNeeded |= mRepaintEverything;

signalRefresh();//发出刷新信号，MessageQueue::REFRESH

break;

}

case MessageQueue::REFRESH: {

handleMessageRefresh();

break;

}

}

}

■ handleMessageTransaction，设置了flag，并未实际操作各个buffer

bool SurfaceFlinger::handleMessageTransaction() {

uint32\_t transactionFlags = peekTransactionFlags();

if (transactionFlags) {

handleTransaction(transactionFlags);

return true;

}

return false;

}

void SurfaceFlinger::handleTransaction(uint32\_t transactionFlags) {

…

handleTransactionLocked(transactionFlags);

...

}

void SurfaceFlinger::handleTransactionLocked(uint32\_t transactionFlags) {

mCurrentState.traverseInZOrder([](Layer\* layer) {//通过Z轴排好序

layer->notifyAvailableFrames();

});

if (transactionFlags & eTraversalNeeded) {

mCurrentState.traverseInZOrder([&](Layer\* layer) {//遍历每个Layer，执行layer->doTransaction()，根据返回值设置mVisibleRegionsDirty

uint32\_t trFlags = layer->getTransactionFlags(eTransactionNeeded);//trFlags == 0表示这个layer没有什么变化

if (!trFlags) return;

const uint32\_t flags = layer->doTransaction(0);//trFlags != 0的话

if (flags & Layer::eVisibleRegion)

mVisibleRegionsDirty = true;//得知确实变化了，标明这块区域脏了，以后要来更新它

});

}

if (transactionFlags & eDisplayTransactionNeeded) {//处理Display事务

add/remove/change

}

if (transactionFlags & (eTraversalNeeded|eDisplayTransactionNeeded)) {//Layer的角度发生了变化

}

if (mLayersAdded) {//处理sf本身的事务，Layer的增/减

mLayersAdded = false;

// Layers have been added.

mVisibleRegionsDirty = true;

}

if (mLayersRemoved) {

mLayersRemoved = false;

。。。

invalidateLayerStack(layer, visibleReg);

}

}

void SurfaceFlinger::invalidateLayerStack(const sp<const Layer>& layer, const Region& dirty) {

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

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

if (layer->belongsToDisplay(hw->getLayerStack(), hw->isPrimary())) {

hw->dirtyRegion.orSelf(dirty);//这个显示器上某一个层被删除之后，我们需要更新它的dirtyRegion

}

}

}

Layer.cpp

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

ATRACE\_CALL();

pushPendingState();

Layer::State c = getCurrentState();

if (!applyPendingStates(&c)) {

return 0;

}

const Layer::State& s(getDrawingState());

if (sizeChanged) {//比较getCurrentState和getDrawingState的变化，比如尺寸

}

。。。

return flags;//如果发生了变化设置flag返回给调用者

}

■ handleMessageInvalidate 使原来的界面数据无效：准备好新数据，用来更换

bool SurfaceFlinger::handleMessageInvalidate() {

ATRACE\_CALL();

return handlePageFlip();

}

bool SurfaceFlinger::handlePageFlip()

{

ALOGV("handlePageFlip");

nsecs\_t latchTime = systemTime();

bool visibleRegions = false;

bool frameQueued = false;

bool newDataLatched = false;

mDrawingState.traverseInZOrder([&](Layer\* layer) {

if (layer->hasQueuedFrame()) {

frameQueued = true;

if (layer->shouldPresentNow(mPrimaryDispSync)) {//收集一下这些需要更新的layer，如果某个layer有新书据要显示的话，里面有已经放在队列里的frame，

mLayersWithQueuedFrames.push\_back(layer);//如果条件成立，会把这个layer push到 mLayersWithQueuedFrames

} else {

layer->useEmptyDamage();

}

} else {

layer->useEmptyDamage();

}

});

for (auto& layer : mLayersWithQueuedFrames) {

const Region dirty(layer->latchBuffer(visibleRegions, latchTime));//锁定这个buffer

layer->useSurfaceDamage();

invalidateLayerStack(layer, dirty);

if (layer->isBufferLatched()) {

newDataLatched = true;

}

}

mVisibleRegionsDirty |= visibleRegions;

if (frameQueued && (mLayersWithQueuedFrames.empty() || !newDataLatched)) {

signalLayerUpdate();

}

return !mLayersWithQueuedFrames.empty() && newDataLatched;

}

Layer.cpp

Region Layer::latchBuffer(bool& recomputeVisibleRegions, nsecs\_t latchTime) {

。。。

status\_t updateResult = mSurfaceFlingerConsumer->updateTexImage(&r,

mFlinger->mPrimaryDispSync, &mAutoRefresh, &queuedBuffer,

mLastFrameNumberReceived);

。。。

}

SurfaceFlingerConsumer.cpp

status\_t SurfaceFlingerConsumer::updateTexImage(BufferRejecter\* rejecter,

const DispSync& dispSync, bool\* autoRefresh, bool\* queuedBuffer,

uint64\_t maxFrameNumber)

{

err = acquireBufferLocked(&item, computeExpectedPresent(dispSync),

maxFrameNumber);

err = updateAndReleaseLocked(item, &mPendingRelease);

err = bindTextureImageLocked();//Texture—纹理，图形界面处理相关术语，最终会调用到GL 函数:

glEGLImageTargetTexture2DOES(texTarget, static\_cast<GLeglImageOES>(mEglImage));

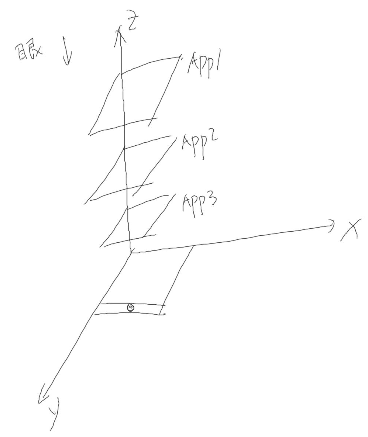
}

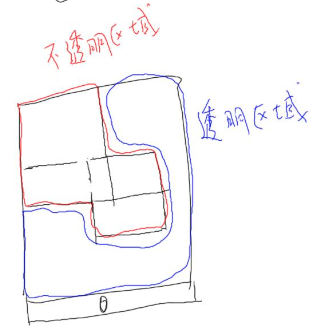


## 5.6 Region的定义及操作

一个界面的透光度，不透明(遮盖)、全透明(不遮盖，可以不绘画)、半透明(不遮盖，需要绘画下)

### 5.6.1 Region定义

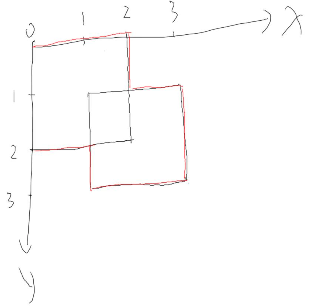




Region由一系列的Rect矩形构成，并且:

1)从上到下，从左到右排序

2)互不遮挡：各矩形的Y轴坐标相同或无交叉，相同解释=》对于Y轴坐标相同的矩形，X轴坐标无交叉

 用Rect来正确表示上图的Region

Rect(0, 0, 2, 2) + Rect(1, 1, 3, 3) – 错误，因为这两个Rect Y轴有交叉

Rect(0, 0, 2, 1) + Rect(0, 1, 2, 2) + Rect(1, 1, 3, 2) + Rect(1, 2, 3, 3) – 错误，第二个和第三个Rect X轴有交叉

Rect(0, 0, 2, 1) + Rect(0, 1, 2, 2) + Rect(2, 1, 3, 2) + Rect(1, 2, 3, 3) – 正确

Rect(0, 0, 2, 1) + Rect(0, 1, 1, 2) + Rect(1, 1, 2, 2) + Rect(2, 1, 3, 2) + Rect(1, 2, 3, 3) – 正确

引入一个概念Span：一组Rect，Y轴一样，X坐标无交叉

Region由一组“Y轴无交叉”的span组成

Region(0, 0, 2, 1) + Region(0, 1, 3, 2) + Region(1, 2, 3, 3)

frameworks/native/libs/ui/include/ui/Region.h

class Region : public LightFlattenable<Region>

{

Vector<Rect> mStorage;//存放有一个个矩形，其最后一项表示边界，

如果Region只有一个Rect，那么 mStorage只有1项，即只有1个Rect，既表示Rect本身也可以表示它的边界

如果Region有2个Rect，那么mStorage含3个Rect，最后一项表示边界

这意味着 mStorage不会有两个Rect，比如上图就是Rect(0, 0, 3, 3)

};

### 5.6.1 Region操作

显示系统013\_基础知识\_Region操作.jpg

r1(0, 0, 2, 2)

r2(1, 1, 3, 3)

(1)andSelf 共同部分r1 = (1, 1, 2, 2)，结果会影响到自己

(2)orSelf 合并r1 (0, 0, 2, 1) (0, 1, 3, 2) (1, 2, 3, 3)

(3)xorSelf 相异的部分(0, 0, 2, 1) (0, 1, 1, 2) (2, 1, 3, 2) (1, 2, 3, 3) – 就是中间的小方块扣除了

(4)intersect 交集，共同部分 new r3(1, 1, 2, 2)，结果不影响r1

(5)subtractSelf r1等于减去r1和r2共同的部分就是中间的那个小方块