**Hardware render**

1. **系统中三个关于ui库的介绍**

**GUI：**用来和surfaceflinger进行通信

**HWUI：**App用来进行hardrender

**UI：**Surfaceflinger 使用的工具

**2.RenderProxy类有三个重要的成员变量:**

mRenderThread、mContext和mDrawFrameTask，它们的类型分别为RenderThread、CanvasContext和DrawFrameTask。其中，mRenderThread描述的就是Render Thread，mContext描述的是一个画布上下文，mDrawFrameTask描述的是一个用来执行渲染任务的Task。

RenderProxy::RenderProxy(bool translucent, RenderNode\* rootRenderNode, IContextFactory\* contextFactory)

: mRenderThread(RenderThread::getInstance())

, mContext(nullptr) {

SETUP\_TASK(createContext);

args->translucent = translucent;

args->rootRenderNode = rootRenderNode;

args->thread = &mRenderThread;

args->contextFactory = contextFactory;

mContext = (CanvasContext\*) postAndWait(task);

mDrawFrameTask.setContext(&mRenderThread, mContext);

}

(1)**.ThreadedRenderer对象的创建：**

mRenderThread创建：RenderThread::getInstance()。

1. 调用ThreadedRenderer类的成员函数nCreateRootRenderNode在Native层创建了一个Render Node，并且通过Java层的RenderNode类的静态成员函数adopt将其封装在一个Java层的Render Node中。这个Render Node即为窗口的Root Render Node。

2. 调用ThreadedRenderer类的成员函数nCreateProxy在Native层创建了一个Render Proxy对象。该Render Proxy对象以后将负责从Main Thread向Render Thread发送命令。

3. 调用AtlasInitializer类的成员函数init初始化一个系统预加载资源的地图集。通过这个地图集，可以优化资源的内存使用。

**(2).MethodInvokeRenderTask 来创建CanvasContext。**

**1.CREATE\_BRIDGE4宏的解析：**

CREATE\_BRIDGE4(createContext, RenderThread\* thread, bool translucent,

RenderNode\* rootRenderNode, IContextFactory\* contextFactory) {

return new CanvasContext(\*args->thread, args->translucent,

args->rootRenderNode, args->contextFactory);

}

转化：

**CREATE\_BRIDGE4：**

typedef struct {

RenderThread\* thread;

bool translucent;

RenderNode\* rootRenderNode;

IContextFactory\* contextFactory;

} createContextArgs;

static void\* Bridge\_createContext(createContextArgs\* args){

return new CanvasContext(\*args->thread, args->translucent,

args->rootRenderNode, args->contextFactory);

}

**2.SETUP\_TASK**

**SETUP\_TASK(createContext)；**

MethodInvokeRenderTask\* task = new MethodInvokeRenderTask((RunnableMethod) Bridge\_createContext);

createContextArgs\* args = createContextArgs\* task->payload();

**3.postAndWait函数的解析：**

创建一个包含MethodInvokeRenderTask的SignalingRenderTask，将该task 加入到RenderThread的TaskQueue中，同时唤醒renderthread 线程对该RenderTask进行处理。最终会调用MethodInvokeRenderTask的Bridge\_createContext函数将创建的CanvasContext对象返回给mContext成员变量。

**4.DrawFrameTask ：**

setContext，该对象的作用是app通过nSyncAndDrawFrame函数来请求renderthread进行绘制的动作。

1. **Surface绑定到renderthread**

1.最终在renderthread线程中调用CanvasContext类的initialize函数，在该函数中使用ANativeWindow来创建EGLSurface。

CREATE\_BRIDGE2(initialize, CanvasContext\* context, ANativeWindow\* window) {

return (void\*) args->context->initialize(args->window);

}

bool RenderProxy::initialize(const sp<ANativeWindow>& window) {

SETUP\_TASK(initialize);

args->context = mContext;

args->window = window.get();

return (bool) postAndWait(task);

}

1. 关于ANativeWindow的数据结构：

struct ANativeWindow

{

struct android\_native\_base\_t common;

/\* flags describing some attributes of this surface or its updater \*/

const uint32\_t flags;

/\* min swap interval supported by this updated \*/

const int minSwapInterval;

/\* max swap interval supported by this updated \*/

const int maxSwapInterval;

/\* horizontal and vertical resolution in DPI \*/

const float xdpi;

const float ydpi;

/\* Some storage reserved for the OEM's driver. \*/

intptr\_t oem[4];

int (\*setSwapInterval)(struct ANativeWindow\* window,

int interval);

int (\*dequeueBuffer\_DEPRECATED)(struct ANativeWindow\* window,

struct ANativeWindowBuffer\*\* buffer);

int (\*lockBuffer\_DEPRECATED)(struct ANativeWindow\* window,

struct ANativeWindowBuffer\* buffer);

int (\*queueBuffer\_DEPRECATED)(struct ANativeWindow\* window,

struct ANativeWindowBuffer\* buffer);

int (\*query)(const struct ANativeWindow\* window,

int what, int\* value);

int (\*perform)(struct ANativeWindow\* window,

int operation, ... );

int (\*cancelBuffer\_DEPRECATED)(struct ANativeWindow\* window,

struct ANativeWindowBuffer\* buffer);

int (\*dequeueBuffer)(struct ANativeWindow\* window,

struct ANativeWindowBuffer\*\* buffer, int\* fenceFd);

int (\*queueBuffer)(struct ANativeWindow\* window,

struct ANativeWindowBuffer\* buffer, int fenceFd);

int (\*cancelBuffer)(struct ANativeWindow\* window,

struct ANativeWindowBuffer\* buffer, int fenceFd);

}

3.初始化时创建的openGL的对象：

EGLDisplay mEglDisplay;

EGLConfig mEglConfig;

EGLContext mEglContext;

EGLSurface mPBufferSurface;

1. **AssetAtlas的创建（AssetAtlasService）：**

**1.获取从zygot中PreloadedDrawables的image图片资源。**

1. **GraphicBuffer的创建：**

Java层使用从native层创建的nativeobj来创建对象。

public static GraphicBuffer create(int width, int height, int format, int usage) {

long nativeObject = nCreateGraphicBuffer(width, height, format, usage);

if (nativeObject != 0) {

return new GraphicBuffer(width, height, format, usage, nativeObject);

}

return null;

}

使用surfaceflinger创建一个GraphicBufferAlloc对象，使用该对象创建GraphicBuffer。

static jlong android\_view\_GraphiceBuffer\_create(JNIEnv\* env, jobject clazz,

jint width, jint height, jint format, jint usage) {

sp<ISurfaceComposer> composer(ComposerService::getComposerService());

sp<IGraphicBufferAlloc> alloc(composer->createGraphicBufferAlloc());

status\_t error;

sp<GraphicBuffer> buffer(alloc->createGraphicBuffer(width, height, format, usage, &error));

GraphicBufferWrapper\* wrapper = new GraphicBufferWrapper(buffer);

return reinterpret\_cast<jlong>(wrapper);

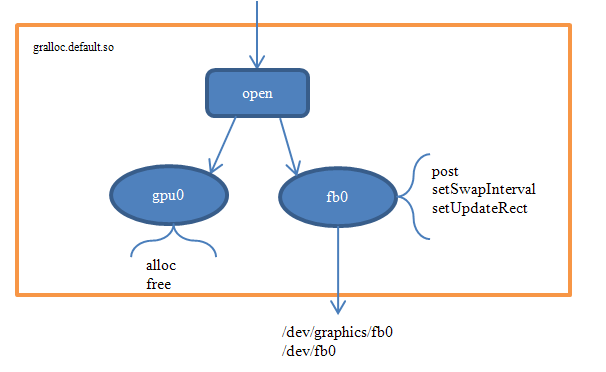
}

**3.gralloc简图：**

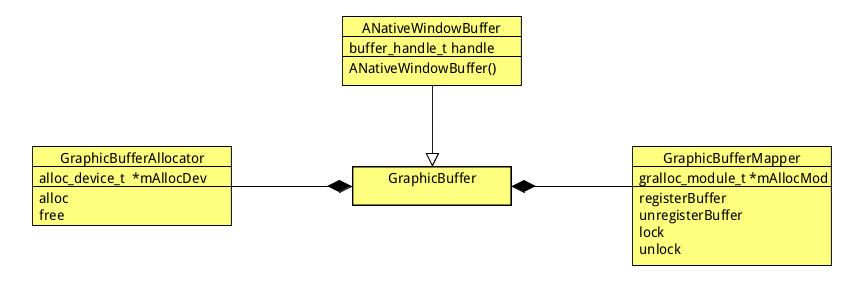
Gralloc 中的两个device：

1.GRALLOC\_HARDWARE\_GPU0 图形缓冲设备。

2.GRALLOC\_HARDWARE\_FB0，帧缓冲设备。



**1.图形缓冲设备GraphicBuffer结构：**



Galloc 中private\_module\_t赋值过程：

struct private\_module\_t HAL\_MODULE\_INFO\_SYM = {

.base = {

.common = {

.methods = &gralloc\_module\_methods

},

.registerBuffer = gralloc\_register\_buffer,

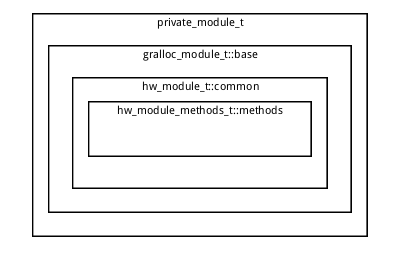
.unregisterBuffer = gralloc\_unregister\_buffer,

.lock = gralloc\_lock,

.unlock = gralloc\_unlock,

},

};



GraphicBufferMapper 就是使用gralloc\_module\_t中的方法对buffer\_handle\_t进行操作.

GraphicBufferMapper::GraphicBufferMapper(): mAllocMod(0)

{

hw\_module\_t const\* module;

int err = hw\_get\_module(GRALLOC\_HARDWARE\_MODULE\_ID, &module);

mAllocMod = reinterpret\_cast<gralloc\_module\_t const \*>(module);

}

GraphicBufferAllocator使用函数gralloc\_open来打开alloc\_device\_t.

gralloc\_open最终调用private\_module\_t.common.methods.open函数，也就是gralloc\_device\_open。

alloc\_device\_t \*mAllocDev;

**GraphicBufferAllocator::GraphicBufferAllocator(): mAllocDev(0)**

{

hw\_module\_t const\* module;

int err = hw\_get\_module(GRALLOC\_HARDWARE\_MODULE\_ID, &module);

gralloc\_open(module, &mAllocDev);

}

**static inline int gralloc\_open(const struct hw\_module\_t\* module, struct alloc\_device\_t\*\* device) {**

return module->methods->open(module,GRALLOC\_HARDWARE\_GPU0,(structhw\_device\_t\*\*)device);

}

**int gralloc\_device\_open(const hw\_module\_t\* module, const char\* name,hw\_device\_t\*\* device)**

{

if (!strcmp(name, GRALLOC\_HARDWARE\_GPU0)) {

gralloc\_context\_t \*dev;

dev = (gralloc\_context\_t\*)malloc(sizeof(\*dev));

dev->device.common.module = const\_cast<hw\_module\_t\*>(module);

dev->device.common.close = gralloc\_close;

dev->device.alloc = gralloc\_alloc;

dev->device.free = gralloc\_free;

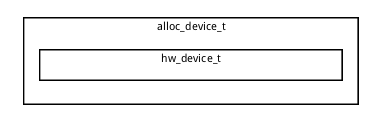
\*device = &dev->device.common;

} else {

status = fb\_device\_open(module, name, device);

}

}



1. **帧缓冲设备：**

**Open：**

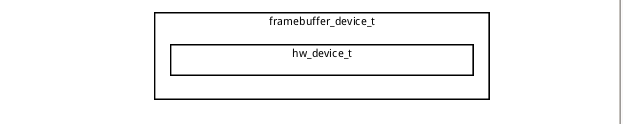
**static inline int framebuffer\_open(const struct hw\_module\_t\* module,**

**struct framebuffer\_device\_t\*\* device) {**

return module->methods->open(module,

GRALLOC\_HARDWARE\_FB0, (struct hw\_device\_t\*\*)device);

}

****

**int fb\_device\_open(hw\_module\_t const\* module, const char\* name,**

hw\_device\_t\*\* device)

{

int status = -EINVAL;

if (!strcmp(name, GRALLOC\_HARDWARE\_FB0)) {

fb\_context\_t \*dev = (fb\_context\_t\*)malloc(sizeof(\*dev));

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

dev->device.common.tag = HARDWARE\_DEVICE\_TAG;

dev->device.common.version = 0;

dev->device.common.module = const\_cast<hw\_module\_t\*>(module);

dev->device.common.close = fb\_close;

dev->device.setSwapInterval = fb\_setSwapInterval;

dev->device.post = fb\_post;

dev->device.setUpdateRect = 0;

private\_module\_t\* m = (private\_module\_t\*)module;

status = mapFrameBuffer(m);

}

return status;

}

**static int mapFrameBuffer(struct private\_module\_t\* module)**

{

int err = mapFrameBufferLocked(module);

return err;

}

mapFrameBufferLocked 函数的主要行为是：

1. 打开帧缓冲设备
2. 获取帧缓冲设备的信息
3. 将帧缓冲设备的地址空间映射到用户进程空间为显示做好准备。

帧缓冲设备是在surfaceflinger中打开进行管理的，对应的管理类为HWComposer。

**int mapFrameBufferLocked(struct private\_module\_t\* module)**

{

char const \* const device\_template[] = {

"/dev/graphics/fb%u",

"/dev/fb%u",

0 };

fd = open(name, O\_RDWR, 0);

struct fb\_fix\_screeninfo finfo;

if (ioctl(fd, FBIOGET\_FSCREENINFO, &finfo) == -1)

struct fb\_var\_screeninfo info;

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

size\_t fbSize = roundUpToPageSize(finfo.line\_length \* info.yres\_virtual);

module->framebuffer = new private\_handle\_t(dup(fd), fbSize, 0);

module->numBuffers = info.yres\_virtual / info.yres;

module->bufferMask = 0;

void\* vaddr = mmap(0, fbSize, PROT\_READ|PROT\_WRITE, MAP\_SHARED, fd, 0);

if (vaddr == MAP\_FAILED) {

ALOGE("Error mapping the framebuffer (%s)", strerror(errno));

return -errno;

}

module->framebuffer->base = intptr\_t(vaddr);

}

GraphicBuffer在创建的过程中会使用图形缓冲设备来alloc一块buffer保存在buffer\_handle\_t中。

typedef const native\_handle\_t\* buffer\_handle\_t;

typedef struct native\_handle

{

int version; /\* sizeof(native\_handle\_t) \*/

int numFds; /\* number of file-descriptors at &data[0] \*/

int numInts; /\* number of ints at &data[numFds] \*/

int data[0]; /\* numFds + numInts ints \*/

} native\_handle\_t;

将所有的Bitmap画在创建的Canvas上，通过nUploadAtlas函数上传到GPU中。

至此服务端地图集的创建完成。

**Client使用地图集：**

在ThreadedRenderer对象的创建的第三步中，ProcessInitializer对象的初始化中获取地图集。

private static class ProcessInitializer {

synchronized void init(Context context, long renderProxy) {

if (mInitialized) return;

mInitialized = true;

initGraphicsStats(context, renderProxy);

initAssetAtlas(context, renderProxy);

}

}

private static void initAssetAtlas(Context context, long renderProxy) {

IBinder binder = ServiceManager.getService("assetatlas");

IAssetAtlas atlas = IAssetAtlas.Stub.asInterface(binder);

GraphicBuffer buffer = atlas.getBuffer();

if (buffer != null) {

long[] map = atlas.getMap();

if (map != null) {

nSetAtlas(renderProxy, buffer, map);

}

}

}

通过GraphicBuffer和map数组，将地图集中的资源保存在AssetAtlas的数组mEntries中：

private static void initAssetAtlas(Context context, long renderProxy) {

IBinder binder = ServiceManager.getService("assetatlas");

IAssetAtlas atlas = IAssetAtlas.Stub.asInterface(binder);

GraphicBuffer buffer = atlas.getBuffer();

if (buffer != null) {

long[] map = atlas.getMap();

if (map != null) {

nSetAtlas(renderProxy, buffer, map);

}

}

}

void CanvasContext::setTextureAtlas(RenderThread& thread,

const sp<GraphicBuffer>& buffer, int64\_t\* map, size\_t mapSize) {

thread.eglManager().setTextureAtlas(buffer, map, mapSize);

}

void EglManager::setTextureAtlas(const sp<GraphicBuffer>& buffer,

int64\_t\* map, size\_t mapSize) {

mAtlasBuffer = buffer;

mAtlasMap = map;

mAtlasMapSize = mapSize;

if (hasEglContext()) {

initAtlas();

}

}

void AssetAtlas::init(sp<GraphicBuffer> buffer, int64\_t\* map, int count) {

mImage = new Image(buffer);

Caches& caches = Caches::getInstance();

mTexture = new Texture(caches);

mTexture->width = buffer->getWidth();

mTexture->height = buffer->getHeight();

createEntries(caches, map, count);

}

**5.构建displaylist**

1.DecorView的mRenderNode树的构建过程：