Android 图形系统及 GPU 加速分析

---西安交大智能交互与软件实验室

2013-5-7

一、Android 图形系统简介

1.1 Surface 与 SurfaceFlinger

对于现代的操作系统而言，流畅、精致的图形用户界面是构建优秀用户体验的重要手段。作为图形用户界面赖以存在的基础，图形系统已经成为操作系统的重要基础设施，特别对于移动设备而言，图形系统决定着该设备的整体体验与运行效率。

Android 图形系统由两大组件构成：Surface 和 SurfaceFlinger。

Surface，又称绘图表面，代表应用程序需要显示在屏幕上的内容。Andorid 系统中同时存在多个 Surface，这些 Surface 可能会重叠、相互遮挡。

SurfaceFlinger 负责 Surface 的叠加与合成，并把合成结果输出到屏幕上。同时，SurfaceFlinger 为每一个 Surface 建立一个绘图缓冲区队列，用于存储 Surface 绘制的图像。

Android 图形系统的总体结构如下：

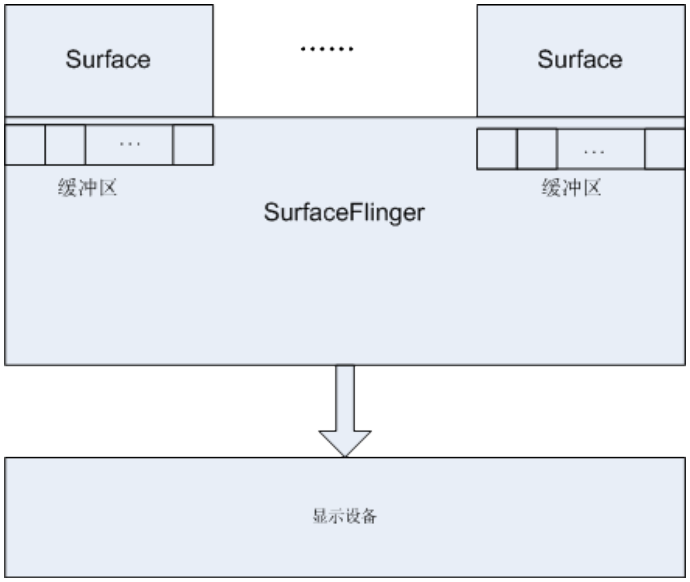

图 1 Android 图形系统总体结构

1.2 Surface 与 SurfaceFlinger 的交互过程

Surface 与 SurfaceFlinger 的交互主要集中于当 Surface 需要更新的时候。交互过程如图 2 所示：

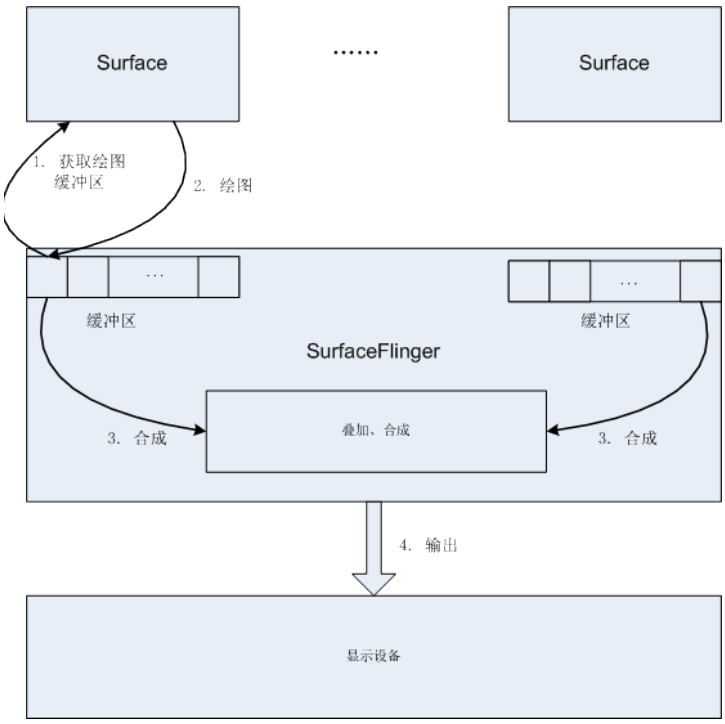

图 2　Surface 与 SurfaceFlinger 的交互过程

首先，当 Surface 需要更新的时候，向 SurfaceFlinger 申请一块绘图缓冲区；之后，Surface 就在这块缓冲区上绘图；绘图完成后，Surface 把缓冲区归还给 SurfaceFlinger，并通知 SurfaceFlinger 重新合成各个 Surface；SurfaceFlinger 接收到合成请求后，合成并重绘需要重绘的 Surface，并输出给显示设备，从而使屏幕上呈现最新的界面。

1.3 图形系统的效率提升

作为 Android 系统的关键组件，图形系统的运行效率直接影响着用户体验，而且关乎系统的整体效率。Android 系统采取如下措施提升图形系统的效率：

1. 共享缓冲区
2. 使用 GPU 加速图形绘制与叠加

下面，就分别分析 Android 图形系统对缓冲区的管理和对 GPU 的使用。

二、缓冲区管理

2.1 Gralloc

为了便于使用，并隔离底层硬件的变化，向上提供统一的接口， Android 在其 HAL 层为其图形系统提供了 gralloc（即 Graphic Alloc）模块，负责图像缓冲区的分配、映射和回收。它可以从 Andorid 匿名共享内存(以下简称 AShm 设备)和图形设备(Framebuffer 设备，以下简称 fb 设备)的存储区中分配缓冲区，并且可以把缓冲区映射到不同的进程中。

Ashm 设备的存储区位于系统内存中，它的容量只受系统内存大小的限制。

Framebuffer 设备的存储区位于显示设备的内存（即显存），或者位于系统内存的一块特定区域，这里统称为显存。显示设备可以直接访问显存，但是受技术和成本的限制，相对于系统内存来说，其容量往往较小。

注意，这两个存储区是不相交的，它们之间传递数据只能逐位复制。

Gralloc 向外提供了 3 个主要的数据结构及其接口，他们的关系如下：

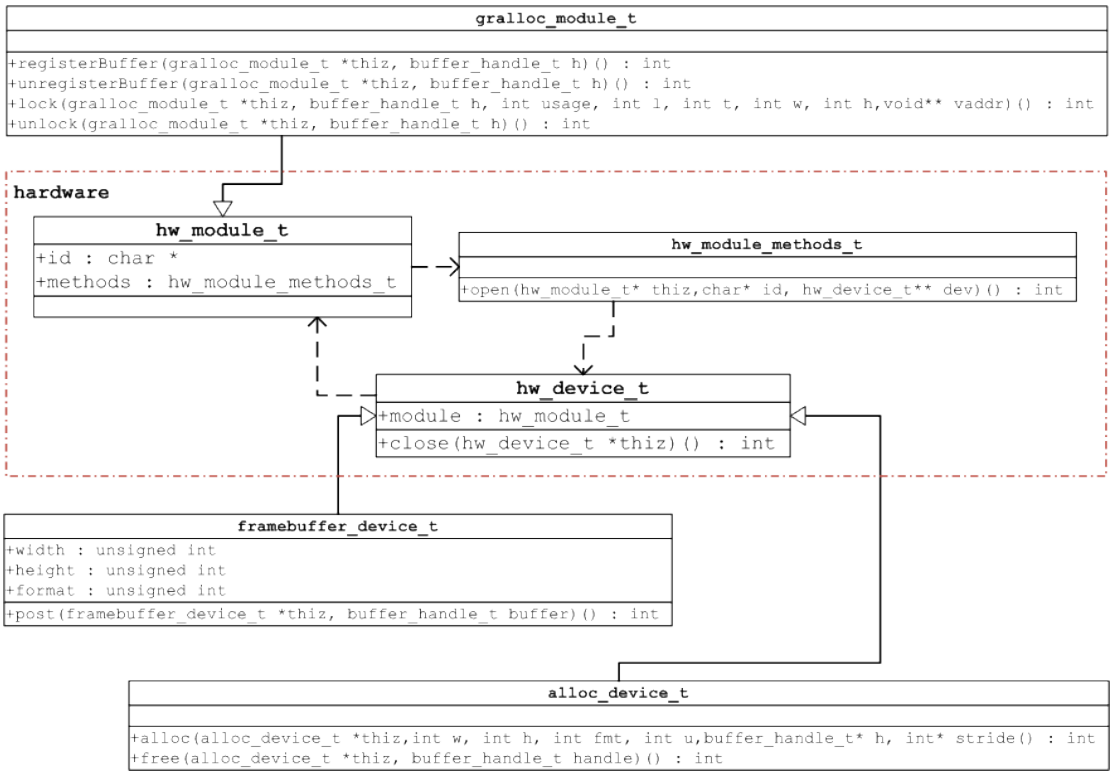

图 3　Gralloc 数据结构及其接口

Gralloc 是作为 HAL 层的一个模块运行的，所以其必须遵守 HAL 层的规范，即继承并实现代表 HAL 模块的 hw_module_t 和代表设备的 hw_device_t。有关 Android HAL 的详细信息，请参考资料 2。

作为分配图像缓冲区的模块，gralloc 应该提供缓冲区分配和回收的机制；由于图像缓冲区会被不同的进程使用，gralloc 还应该提供把缓冲区映射到特定进程的机制；为了保证多个进程的有序访问，gralloc 必须提供锁机制；同时，gralloc 还应该能够把缓冲区提交给 fb 设备，用于显示保存在其中的图像。

gralloc 模块的实现如下：

gralloc_module_t：代表 gralloc 模块，并提供了进程映射和锁机制。其中：

int registerBuffer(gralloc_module_t const* thiz, buffer_handle_t h)

会把 h 代表的缓冲区映射到调用者进程；

int unregisterBuffer(gralloc_module_t const* thiz, buffer_handle_t h)

执行相反的操作，即从调用者进程解除映射；

int lock(gralloc_module_t * thiz, buffer_handle_t h, int usage,int l, int t, int w, int h,
void** vaddr)

会锁定 h 缓冲区中的左上角为(l,t)，宽为 w，高为 h 的区域，并把该区域的起始地址保存在 vaddr 中；

int unlock(gralloc_module_t *thiz, buffer_handle_t h)

用于解锁 h 代表的缓冲区。

alloc_device_t 代表用于分配缓冲区的设备，它提供缓冲区分配与回收的功能，其中：

int alloc(alloc_device_t* thiz, int w, int h, int format, int usage,buffer_handle_t* h,
int* stride)

用于分配符合需求（宽、高、格式等）的缓冲区，并保存在 h 中；

int free(alloc_device_t* thiz, buffer_handle_t h)

用于释放 h 所代表的缓冲区。

framebuffer_device_t 代表 fb 设备，它提供提交缓冲区的功能，其中：

int post(framebuffer_device_t* thiz, buffer_handle_t buffer)

用于向 fb 设备提交 buffer 所代表的缓冲区，之后，这个缓冲区中的内容就会被显示在屏幕上。

关于 gralloc 详细的详细信息，可以参考资料 3。

2.2 缓冲区与缓冲区队列

2.2.1 缓冲区分类

Android 图形系统中，缓冲区主要用于存储绘图结果，可分为两类：

1. Surface 的绘图缓冲区，用于保存每个 Surface 的绘图结果，称之为绘图缓冲区

2. SurfaceFlinger 的合成缓冲区，用于保存经 SurfaceFlinger 叠加合成后的图像信息，称之为合成缓冲区。

理想情况下，缓冲区从 fb 设备的存储区中分配是最好的，因为这样的话，绘图和叠加完成后，显示设备就能直接把结果显示出来。但是，由于系统中会同时存在多个 Surface，如果每一个 Surface 的绘图缓冲区都从 fb 设备中分配，那么就需要 fb 设备有很大的存储，这无疑会增加成本。

所以，Android 就采取折中的办法，绘图缓冲区在系统内存中分配；由于合成缓冲区全局只需要一个（事实上，为了图像显示稳定，需要 2 个或 3 个，即便是这样，需要的存储空间也不大），所以合成缓冲区就从 fb 设备中分配。

Android 图形系统分别使用 GraphicBuffer 和 NativeWindow 类描述这两类缓冲区，他们的关系如图 4 所示：

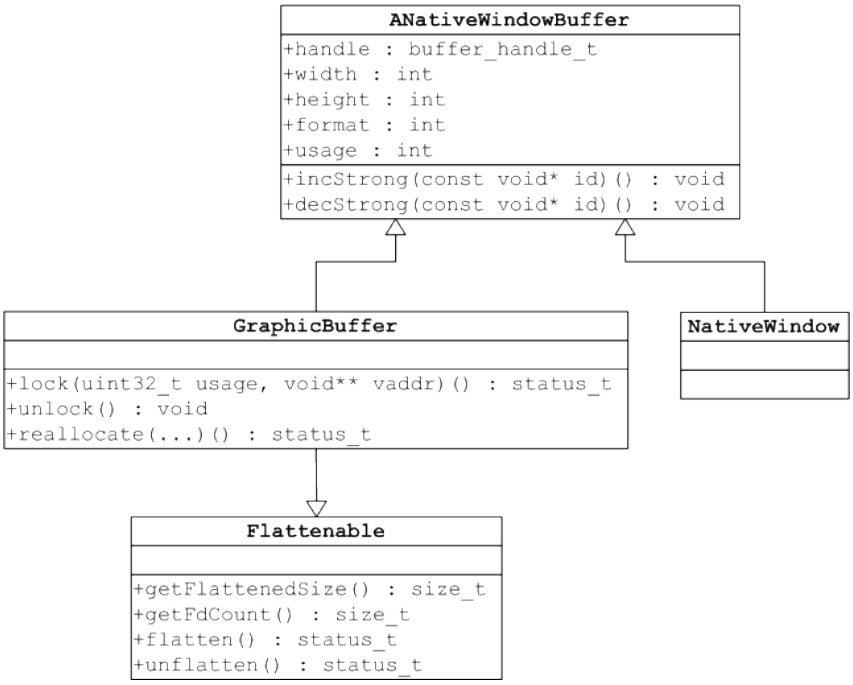

图 4　缓冲区类图

其中，基类 ANativeWindowBuffer 定义了缓冲区的基本属性，如起始地址（handle）、宽（width）、高（height）、格式（format）、用途（usage）。其中，usage 决定了缓冲区是位于 fb 设备中，还是位于 AShm 设备中。当 usage 为 1 时，其位于 fb 设备中；否则，位于 AShm 设备中。它的两个方法 incStrong(...)和 decStrong(...)用于增加和减少引用计数，所以 ANativeWindowBuffer 可以作为智能指针使用。

绘图缓冲区

GraphicBuffer 表示绘图表面 Surface 使用的绘图缓冲区，它一般都是从 AShm 设备分配的，即其存储区位于内存，并可以在进程间共享。

由于 GraphicBuffer 所代表的缓冲区需要在应用程序进程和 SurfaceFlinger 进程中共享，所以 GraphicBuffer 需要提供锁机制保证缓冲区的互斥访问，这是通过 lock 和 unlock 方法实现的；由于应用程序每次更新的区域大小不固定，每次需要的缓冲区的大小就不同，所以，GraphicBuffer 应该提供重新分配缓冲区存储空间的机制，这是通过 reallocate 方法实现的。各个方法介绍如下：

lock(uint32_t usage, void **vaddr)

用来锁定整个缓冲区，并把缓冲区的起始地址保存在 vaddr 变量中；

unlock()

用于解锁缓冲区；

reallocate(uint32_t w, uint32_t h, PixelFormat f, uint32_t usage)

用来重新分配存储。

同时，GraphicBuffer 也继承于 Flattenable，表示它可以被串行化，这样它就能通过 Binder 在不同进程之间传递，它实现了 Flattenable 的 getFlattenedSize(...)、getFdCount(...)、flatten(...)以及 unflatten(...)方法，用于处理串行化和反串行化。

合成缓冲区

NativeWindow 表示 SurfaceFlinger 保存合成结果的合成缓冲区。它没有在其父类的基础

上增加新的属性和方法。NativeWindow 所代表的缓冲区是从 Framebuffer 设备中分配的。

2.2.2 缓冲区队列

为了方便管理，Android 把上述两类缓冲区组织成队列的形式，分别用 Surface 和 FramebufferNativeWindow 描述。他们的关系如图 5 所示：

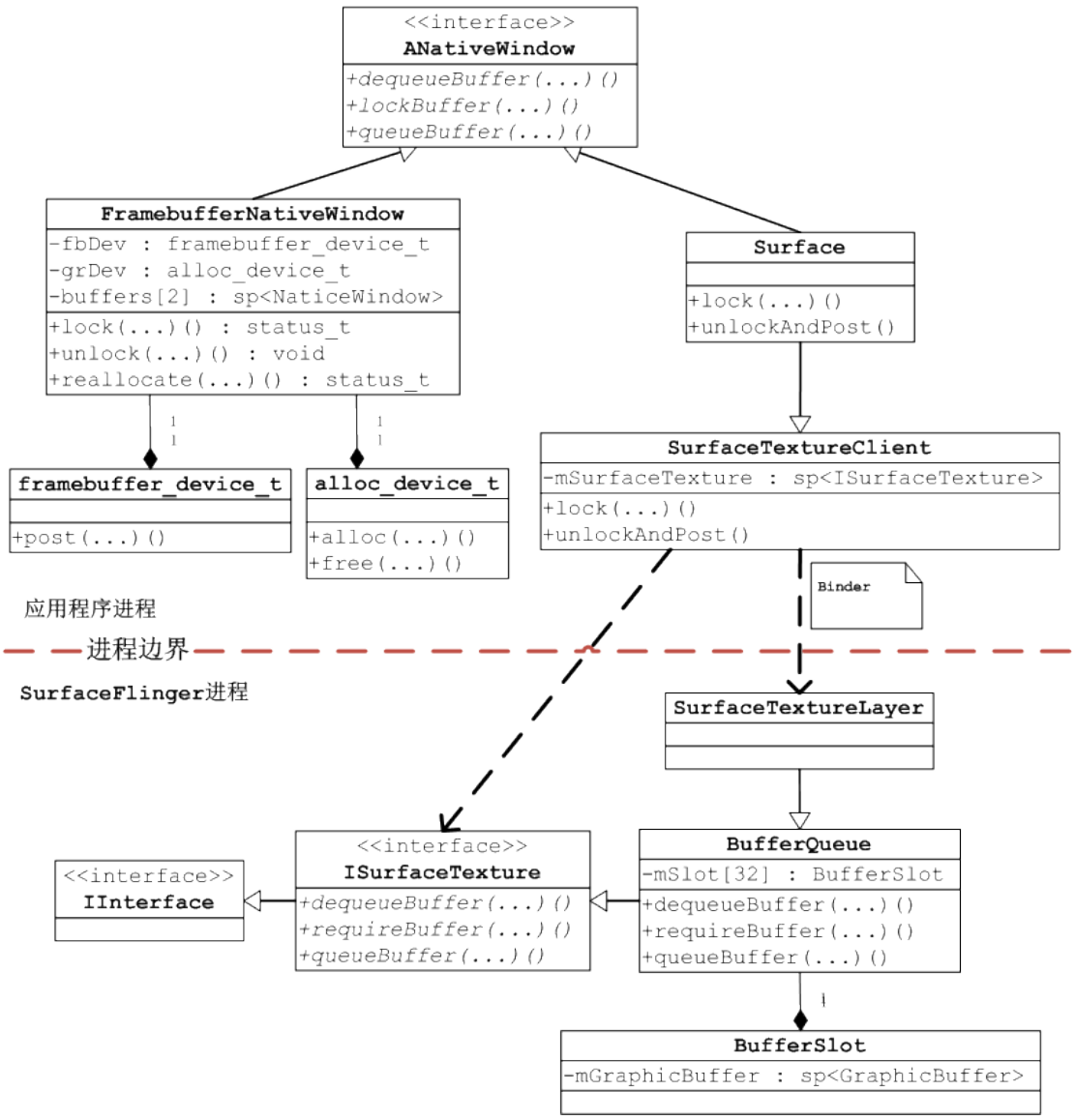

图 5　缓冲区队列类图

其中，基类 ANativeWindow 定义了缓冲区队列的基本属性和方法。对于队列而言，最重要的就是出对和入队操作。ANativeWindow 的重要方法说明如下：

dequeueBuffer(ANativeWindow* win, ANativeWindowBuffer** buf)

用于从队列中获取一个缓冲区，即出队，并保存在 buf 中。为了保持兼容性，ANativeWindow 使用 C 语言结构体定义的，dequeueBuffer 是结构体中的一个函数指针，所以，需要第一个参数指定这个函数操作的对象，相当于 C++中的 this 指针，下同。

lockBuffer(ANativeWindow* win, ANativeWindowBuffer* buf)

用于锁定一个 buffer，这是为了保证 buffer 在不同进程中能被互斥的访问。

queueBuffer(ANativeWindow* win, ANativeWindowBuffer* buf)

用于向队列中归还 buffer，即入队。

绘图缓冲区队列

Surface（及其父类 SurfaceTextureClient）用于描述一个 Surface 需要的绘图缓冲区队列，注意区分两个 Surface，前者是上图中的类，而后者是上文所说的绘图表面。

Surface 和 SurfaceTextureClient 在 ANativeWindow 的基础上，将 dequeueBuffer 和 lock 两个函数封装成一个 lock 函数；将 queueBuffer 封装成 unlockAndPost 函数。lock 和 unlockAndPost 说明如下：

lock(SurfaceInfo* info, Region* dirty)

会从缓冲区队列获取并锁定一个缓冲区，并把这个缓冲区的的相关信息存储在 info 中（包括缓冲区的长、宽、格式、用途以及起始地址），上层应用就能在这个缓冲区中绘图了。

unlockAndPost()

用于解锁并发布上面用 lock 获取的缓冲区，并通知 SurfaceFlinger 刷新界面。这两个函数都是调用 SurfaceTextureClient 的 lock 和 unlockAndPost 函数实现的。

从类名可以看出，SurfaceTextureClient 只是一个客户端，其服务端对应的是 SurfaceTextureLayer，其中包括缓冲区类型为 GraphicBuffer 的缓冲区队列（队列大小目前为 32）。SurfaceTextureLayer 是一个 Binder 对象，在 SurfaceFlinger 中被初始化，每一个绘图表面都对应一个，其中的缓冲区在需要的时候分配。

SurfaceTextureClient 和 SurfaceTextureLayer 分别位于应用程序进程和 SurfaceFlinger 进程，它们之间需要通过 Android 的进程间通信机制——Binder 来相互通信。关于 Binder 的详细信息，请参考资料 4。SurfaceTextureClient: :mSurfaceTexture 是 SurfaceTextureLayer 的 Binder 代理，SurfaceTextureClient 通过这个代理与 SurfaceTextureLayer 交互。

SurfaceTextureClient::lock(...) 方法会调用自身的 dequeueBuffer(...), 其中会通过 mSurfaceTexture 从 SurfaceTextureLayer 中获取一个缓冲区并锁定，值得注意的是，因为 SurfaceTextureLayer 中的缓冲区都是在 SurfaceFlinger 进程中分配的，所以，在把缓冲区传递给应用程序进程中的 SurfaceTextureClient 时，系统会把缓冲区映射到应用程序进程中。

SurfaceTextureClient::unlockAndPost() 方法调用自身的 queueBuffer(...), 其中会通过 mSurfaceTexture 向 SurfaceTextureLayer 归还缓冲区，之后，SurfaceTextureLayer 会通知 SurfaceFlinger 更新图形界面。

SurfaceTextureLayer 所包含的缓冲区 GraphicBuffer 最终会被会被用做 OpenGL 的纹理，这也是 SurfaceTextureLayer 和 SurfaceTextureClient 这样命名的原因。

合成缓冲区队列

FramebufferNativeWindow 用于描述合成缓冲区队列。目前，其中 buffers 是指向 NativeWindow 类型数组，表示缓冲区，目前数组大小是 2。它们的存储空间都来自于 Framebuffer 设备。

grDev 是指向 gralloc 设备描述符 alloc_device_t 的指针，FramebufferNativeWindow 使用它就能访问 gralloc 的 alloc(...)和 free(...)方法，从 gralloc 中分配存储，buffers 就是在 FramebufferNativeWindow 初始化的时候，使用 alloc(...)函数从 gralloc 中分配的。

fbDev 是指向 framebuffer 设备描述符 framebuffer_dvice_t 的指针，它也是在 gralloc 中定义的，FramebufferNativeWindow 使用它就能访问 post(...)方法，从而向 Framebuffer 提交一个缓冲区，用于显示。

dequeueBuffer(...)和 queueBuffer(...)分别用来获取和归还缓冲区。其中，dequeueBuffer 会从缓冲区 buffers 中取出空闲的 buffer；queueBuffer 最终会使用 fbDev 的 post(...)方法，把

新的缓冲区提交给 fb 设备，用于显示。

2.3 缓冲区的使用

绘图缓冲区的使用

Android 应用程序在初始化的时候，系统会为其每一个可视化界面创建一个绘图缓冲区队列 Surface，Surface 会调用 SurfaceFlinger 的 createSurface(...)方法，通知 SurfaceFlinger 建立于 Surface 对应的 Layer 对象。Layer 对象在初始化的时候，会创建 Surface 绘图需要的缓冲区队列，具体的，就是创建前文提及的 SurfaceTextureLayer 对象，其中，就会初始化缓冲区队列。

因为 Surface 每次重绘需要的缓冲区的大小、格式等可能都会不一样，所以，缓冲区队列在初始化的时候并没有实际创建缓冲区对象 GraphicBuffer。只有在 Surface 需要时，才会使用 SurfaceTextureLayer::dequeueBuffer(...)获取一个 buffer，此时，才会创建相应的 GraphicBuffer 缓冲区对象，其中，GraphicBuffer 会根据 Surface 的需要，向 gralloc 模块申请需要的存储区，这快缓冲区来自于 Ashm。因为 Surface 所在的应用程序和分配 GraphicBuffer 的 SurfaceFlinger 不在同一个进程中，所以就需要把 GraphicBuffer 中缓冲区的地址映射到应用程序进程中，这个是通过上述 gralloc_module_t 中的的 registerBuffer 实现的。

合成缓冲区的使用

SurfaceFlinger 会在初始化的时候创建 EGL 环境，EGL 是 OpenGL 与本地图形系统的接口。SurfaceFlinger 在使用 eglCreateWindowSurface(...)初始化 EGL 的绘图表面时，将 FramebufferNativeWindow 设置为 EGL 绘图表面的缓冲区队列。其中的两个缓冲区会分别作为前台缓冲区（称为 frontbuffer）和后台缓冲区（称为 backbuffer），其中，frontbuffer 是正在被显示的缓冲区，backbuffer 是用于后台绘图的缓冲区。绘图完毕后，交换 frontbuffer 和 backbuffer，之前的 frontbuffer 成为新的 backbuffer，用于下一次绘图；之前的 backbuffer 成为新的 frontbuffer，其中的图像数据被显示在屏幕上。这就实现了一个双缓冲机制，这种机制可以避免产生显示画面的抖动和残缺。

EGL 在为 OpenGL 准备绘图环境时，使用 FramebufferNativeWindow::dequeueBuffer(...)获取一个绘图缓冲区，作为 backbuffer，之后 OpenGL 所绘制的图像信息就保存在这个缓冲区上；OpenGL 绘图完成后使用 eglSwapBuffers(...)交换缓冲区时，eglSwapBuffers(...)会使用 FramebufferNativeWindow::queueBuffer()将 backbuffer 入队，在 queueBuffer(...)中，会调用 gralloc 模块中 framebuffer_device_t 的 post(…)函数将 backbuffer 与 frontbuffer 交换，这样 backbuffer 中的图像信息显示在屏幕；之后，eglSwapBuffers(...)又会使用 dequeueBuffer()获取一个新的绘图缓冲区作为新的 backbuffer，用于 OpenGL 的下一次绘制。

综上所述，SurfaceFlinger 的结构图可以进一步细化为图 6 所示：

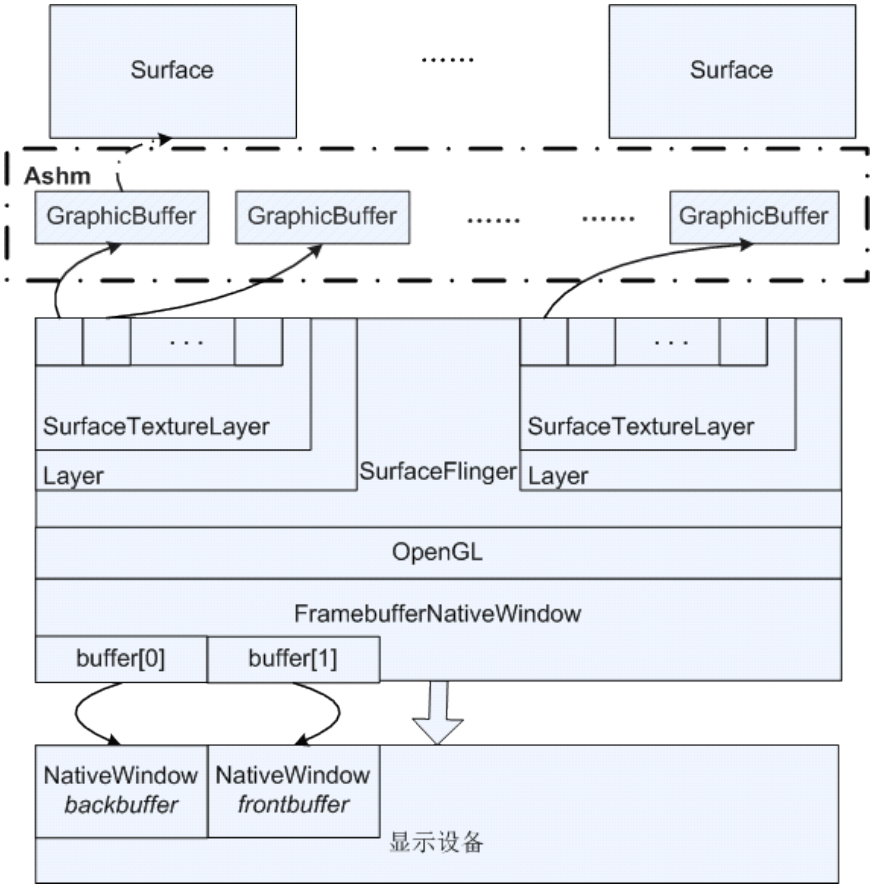

图 6　细化的结构图

三、使用 GPU 加速图形绘制与叠加

作为面向图形处理的芯片，GPU 对图形处理做了专门优化，为了简化 GPU 的使用，业界提出了一个标准化的 API——OpenGL。通过 OpenGL，应用程序就能最大限度的发挥 GPU 的强大处理能力。

Android 系统也使用 OpenGL 加速图形系统效率，这体现在两方面：使用 OpenGL 加速 Surface 的绘制；使用 OpenGL 加速 SurfaceFlinger 对各个 Surface 的叠加。下面就从这两个方面分析。

3.1 使用 OpenGL 加速图形的绘制

Android 应用程序的界面是由多个可视化组件组成的，如 Button（按钮）、文本框（TextView）等，这些可视化组件最终都会被绘制在 Surface 上。

之前，这些组件是使用 Skia 绘图库绘制到 Surface 上的。Skia 是一个 2D 图形处理函数库，使用 Skia 可以绘制基本的图元，如点、线、面等，还可以绘制字型、进行坐标转换、绘制点阵图等。Skia 是一个纯软件的解决方案，也就是说，使用它绘图时，绘图操作都是通过 CPU 执行的。

随着 Android 设备屏幕尺寸的增大和分辨率的不断提升，这种纯软件的解决方案会增加

了 CPU 的负担，导致图形界面卡顿、系统整体不流畅。

从 Android 3.0 之后，Android 开始使用 OpenGL 绘制控件，同时也保留了 Skia。这样，在保证兼容性的前提下，Android 可以发挥 GPU 的作用，加速图形系统的运行效率。

当一个组件需要重绘时，这些可视化控件的绘制都是从 ViewRootImpl.java 的 draw 方法开始的，其可以简化为：

```
void draw(boolean fullRedrawNeeded) {
    Surface surface = mSurface;
    if (mAttachInfo.mHardwareRenderer != null && mAttachInfo.mHardwareRenderer.isEnabled())
        attachInfo.mHardwareRenderer.draw(mView, attachInfo, this, mDirty);
    else
        drawSoftware(surface, attachInfo, yoff, scalingRequired, mDirty);
}
```

其中，drawSoftware 就是使用 Skia 绘制，而 attachInfo.mHardwareRenderer.draw 使用 OpenGL 绘制。

在使用 Skia 绘制时，drawSoftware(...)系统会调用 2.2.2 节中的 Surface::lock(SurfaceInfo* info, Region* dirty)从 SurfaceFlinger 获取一个 GraphicBuffer 缓冲区，该缓冲区的信息存储在 info 中；之后，将这个缓冲区作为 Skia 的画布 Canvas（具体类型为 CompatibleCanvas）的绘图设备。之后，调用 Skia 绘图，绘图结果就保存在上述的 GraphicBuffer 缓冲区中。之后，使用 Surface::unlockAndPost()，将保存有绘制结果的缓冲区重新放入 SurfaceFlinger 中的缓冲区队列中，并通知 SurfaceFlinger 重新合成界面。其中，Surface::unlockAndPost()又会使用 SurfaceTextureClient::queueBuffer(…)完成上述功能。

如果打开了硬件加速（可以在应用程序的 AndroidManifest.xml 中设置），就会使用 OpenGL，通过 GPU 加速绘图。

attachInfo.mHardwareRenderer 是 GLES20Renderer。它在初始化的时候，会初始化 EGL 并创建绘图表面，并建立画板 Canvas，和软件绘制方式不同，此处使用的画布的具体类型为 GLES20Canvas，在其中绘制图像会被 OpenGL 加速。

在创建绘图表面的时候，会调用 EGLImpl.eglCreateWindowSurface(EGLDisplay display, EGLConfig config, Object native_window, int[] attrib_list))，这个函数会检查第三个参数 native_window 的类型，只有第三个是 SurfaceView 或者 SurfaceHolder 类型的才可以，也就是说，目前，只有继承了 SurfaceView 或者 SurfaceHolder 的控件才能被 GPU 加速，一般的空间不能使用 GPU 加速。

如果类型正确，那么，该函数就会调用 native_window.getSurface()，这个函数返回的就是 2.2.2 节中的 Surface 绘图缓冲区队列，之后，就会把 Surface 作为 OpenGL 的绘图表面，OpenGL 的绘图结果就会保存在其中。

由 2.2.2 节对 Surface 的分析可知，其中的缓冲区都是 GraphicBuffer 类型的，位于 Ashm，即内存中，并受 SurfaceFlinger 的管理。

所以，最终使用 OpenGL 绘图的结果保存在内存中，并在调用 eglSwapBuffers()的时候，向 SurfaceFlinger 提交重绘请求。eglSwapBuffer()会调用 EGL 绘图表面的 queueBuffer(…)把保存有绘制结果的缓冲区放入 SurfaceFlinger 的缓冲区队列中。根据上文，此处 EGL 的绘图表面就是 Surface，所以会使用 Surface::queueBuffer(…)完成缓冲区提交，由于 Surface 继承于 SurfaceTextureClient 而且没有重写 queueBuffer(…)，所以，和使用软件绘制一样，最终都是使用 SurfaceTextureClient::queueBuffer(…)向 SurfaceFlinger 提交保存有绘制结果的缓冲区，之后，SurfaceFlinger 就会叠加并输出这些缓冲区。

而且，不论是用 Skia 软件方式绘制，还是 OpenGL 硬件加速绘制，它们最终都会把绘

制好的图像信息存储在上述的 GraphicBuffer 中，并由 SurfaceFlinger 统一管理。

GLES20Renderer 对应的画板是 GLES20Canvas。它在初始化的时候，会创建在这个画板上绘图的渲染器 OpenGLRenderer。OpenGLRenderer 内部用 OpenGL 实现了常用的图形操作，像平移、旋转、缩放等，同时也实现了常用的画图操作，像画点，画线等。

同时，每一个控件的都会调用 GLES20Renderer.createDisplayList(...)创建一个显示列表。控件需要重绘的时候，把绘制命令保存在显示列表中等待绘制。然后，在实际绘制的时候一次性把这些绘图命令传递给 GPU。所以，使用显示列表就能以批处理的方式一次性处理一批请求，提高了效率。

attachInfo.mHardwareRenderer.draw(...)实际调用的就是 GLES20Renderer.draw(...)，其中会调用 GLES20Canvas.drawDisplayList(...)，继而调用与 GLES20Canvas 关联的渲染器 OpenGLRenderer 的 drawDisplayList，其中会调用 DisplayList 的 replay 方法，从列表中取出绘图命令，最终使用 OpenGLRenderer 完成绘制工作。

3.2 使用 OpenGL 加速 Surface 的叠加

在 Surface 绘制完图形并把绘图缓冲区交还给 SurfaceFlinger 后，各个缓冲区就保存着更新后的界面信息，SurfaceFlinger 会把各个缓冲区中的图像信息叠加成一个完整的画面，并保存在合成缓冲区中，即 backbuffer 中。之后，通过交换 backbuffer 和 frontbuffer，就能把结果显示出来。

由于各个 Surface 会相互重叠，而且 Surface 可能会是半透明的，所以，需要计算各个 Surface 对应的 Layer 的可视区域，之后，只用重绘这些可视区域就行；为了提高叠加和绘制的效率，可以把绘制结果作为 Layer 的纹理，使用 OpenGL 的纹理贴图功能加速叠加和绘制过程。

叠加过程可以简略描述如下：

1. 把保存着 Surface 绘制结果的 buffer 作为该 Surface 对应的 Layer 的纹理对象；
2. 计算需要每个 Layer 的可视区域；
3. 通过 OpenGL 叠加各个 Layer 的可视区域，OpenGL 会把 Layer 对应的纹理贴到该区域上；
4. 把绘制好的界面输出到屏幕上

下面，就逐一分析。

3.2.1 把绘制结果作为纹理对象

SurfaceFlinger 会调用每个 layer 的 lockPageFlip(...)，锁定该 layer，其中又会调用与该 layer 关联的 SurfaceTexture 中的 updateTexImage(...)更新 layer 对应的纹理对象。SurfaceTexture 用来把 Surface 对应的 buffer 转化为 OpenGL 的纹理。

updateTexImage(...)其可以简化为：

```
status_t SurfaceTexture::updateTexImage(BufferRejecter* rejecter) {
    BufferQueue::BufferItem item;
    status_t err = mBufferQueue->acquireBuffer(&item);
    if( err!=NO_ERROR )
        return err;
    int buf = item.mBuf;
```

```
        GLeglImageOES image =
                createImage(eglGetCurrentDisplay(), mEGLSlots[buf].mGraphicBuffer);
        glBindTexture(mTexTarget, mTexName);
        glEGLImageTargetTexture2DOES(mTexTarget, (GLeglImageOES)image);
    }
```

其中，mBufferQueue 就是 SurfaceTextureLayer 的实例。该函数首先用 SurfaceTextureLayer::acquireBuffer(...)从 layer 对应的缓冲区队列中获取一个 buffer，如果没有需要更新的 buffer，就说明这个 Layer 没有变化，不需要更新纹理对象，那么直接返回即可。然后使用 createImage(...)创建一个 EGL 图像，并将刚才获取的 buffer 中的数据作为该图像的内容；之后，使用 glBindTexture 设置 OpenGL 当前活动的纹理对象为 mTexName，mTexName 是在 Layer 初始化的时候使用 glGenTexture 创建的，每个 Layer 都有一个并且各不相同；之后，使用 glEGLImageTargetTexture2DOES 把刚才创建的 image 作为当前的活动纹理，即与 mTexName 关联起来。这样，保存有绘制结果的 buffer 就成为该 layer 的纹理对象了。

3.2.2 计算可见区域

绑定纹理对象后，SurfaceFlinger 开始计算每一个 Layer 的可见区域。Layer 的可见区域就是能被显示在屏幕上的区域。由于同时存在多个 layer，所以 layer 之间就会出现相互重叠的现象。当两个 layer 相互重叠时，下面 layer 的某些区域可能就会因为被遮挡而不可见。如果一个 layer 的可见区域与其需要重绘的区域没有交集，即需要重绘的部分是不可见的，那么就不需要重绘了。

一个 layer 的更新有可能导致 layer 之间重叠关系发生变化，所以需要重新计算每一个 layer 的可见区域。计算原理如下：

一个区域由 3 部分组成：不透明区域 OP，透明区域 TP 以及半透明区域 TL，很明显，透明区域 TP 是不可见的。

记 layer0 的全部区域为 R，可见区域为 V，那么 V=R-TP。

当 layer0 被另外一个 layer1 覆盖时，记 layer1 覆盖在 layer0 上的区域为 C。很明显，只用区域 C 中不透明区域 OP 会影响 layer0 被覆盖区域的可见性，即会让 layer0 的该区域不可见，所以，此时，现在 layer0 的可见区域就是 layer0 之前的可见区域去除被不透明区域覆盖的部分，即 V = V - (V∩C.OP)。

所以，最终，一个 layer 的可见区域 V = R - TP - (V∩C.OP)，即在其全部区域中去除两部分区域：透明区域和被上层 layer 的不透明区域覆盖的区域。

这个过程只是两个 layer 的情况，多个 layer 的情况也类似，只是需要把所有上层 layer 的可见区域都叠加起来，作为一个新的 layer 和下层 layer 叠加，这实际上又是两个 layer 在叠加。

SurfaceFlinger 使用 computeVisibleRegions(...)函数计算可见区域，原理和上面所述的一致，此处就不赘述了。

3.2.3 叠加

计算完各个 layer 的可见区域后，就可以叠加各个 layer 的可见区域了。SurfaceFlinger 使用 handleRepaint()函数完成绘制，其中主要使用 SurfaceFlinger::composeSurfaces(...)叠加并

重绘界面。这是通过调用各个 layer 的 draw 方法完成的，而实际的绘图在 Layer::onDraw 方法中，其可以简化为：

```
void Layer::onDraw(const Region& clip) {
    glBindTexture(GL_TEXTURE_EXTERNAL_OES, mTextureName);
    drawWithOpenGL(clip);
}
```

其中，参数 clip 就是这个 layer 的可见区域与全局脏区相交的区域。这个函数会先使用 glBindTexture 绑定该 layer 的纹理对象到当前 OpenGL 环境中，这之后 OpenGL 操作的都是这个纹理，直到下一次调用 glBindTexture。值得注意的是，在 handlePageFlip 中，该 layer 对应的应用程序所绘制的新的界面信息已经和纹理对象绑定，所以，之后就能通过 opengl 的纹理贴图功能，把界面信息作为纹理贴到需要更新的区域，这是通过 drawWithOpenGL（位于 Layer 的父类 LayerBase）实现的，该函数可以简化为：

```
void LayerBase::drawWithOpenGL(const Region& clip) const{
    ...
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);
    glVertexPointer(2, GL_FLOAT, 0, mVertices);
    glTexCoordPointer(2, GL_FLOAT, 0, texCoords);
    glDrawArrays(GL_TRIANGLE_FAN, 0, mNumVertices);
    ...
}
```

其中，mVertices 是这个 Layer 的需要更新的区域的四个顶点坐标（更新区域为矩形）；mNumVertices 代表上述顶点的个数；texCoords 表示纹理对象的四个顶点坐标，他们和 mVertices 是对应的。glVertexPointer 和 glTexCoordPointer 两个函数分别指定了需要绘制的图形的顶点坐标和纹理的坐标。最后，glDrawArrays 就绘制了一个矩形，其顶点坐标为 mVertices，同时，OpenGL 会自动把纹理（其中包含新界面信息）填充到这个矩形中，这样，该 layer 对应的应用程序的界面就更新了。

至此，所有需要更新的 layer 的像素信息都被 OpenGL 绘制在其 backbuffer 中了，即 backbuffer 中保存的就是最新的界面信息。而 backbuffer 是由 gralloc 模块从 framebuffer 设备中分配的。注意，这一步会涉及到内存复制，即把保存在 ashm 中的纹理信息复制到 framebuffer。

3.2.4 结果输出

上一步应经把最新的界面信息保存在 backbuffer 中了，所以只需要交换 backbuffer 和 frontbuffer 就能把新界面显示出来了。

这一步是在 SurfaceFlinger::postFramebuffer()函数中完成的，这个函数最终会使用 eglSwapBuffers(...)函数把上一步绘制好的 backbuffer 和正在显示的 frontbuffer 交换，之后新的界面就显示在屏幕上了。

而 eglSwapBuffers 经过一系列步骤，最终会调用 gralloc 模块中 framebuffer_device_t 中的 post(…)方法交换前后 buffer，把最新的界面显示在屏幕上。

SurfaceFlinger 叠加各个 Surface 的过程可以总结如下：首先，把保存有新的界面信息的缓冲区作为每一个 Layer 的纹理对象；然后，根据每一个 Layer 的信息和 Layer 之间的重叠关系，每一个 Layer 的可见区域；之后，叠加需要更新的 Layer，并保存在 backbuffer 中；最后，交换 backbuffer 和 frontbuffer，新界面就会显示在屏幕上。

四、 四、示例程序

为了更清楚的说明 Surface 和 SurfaceFlinger 的交互关系和使用方法，下面的附件中附带了一个简单的示例程序，其中，会在屏幕上显示 0-9 这 10 个数字，然后退出程序。程序用 C++语言实现，其中会使用 Surface::lock(…)从 SurfaceFlinger 锁定并获取一块缓冲区，然后，使用 Skia 在这块缓冲区中绘制图像，最后使用 Surface:: unlockAndPost()提交缓冲区并请求 SurfaceFlinger 重绘界面。

只要把附件中的文件解压并放到 android 源代码的 framework/base/cmds 目录下，然后使用 mm 编译就可得到可执行文件 SurfaceFlingerTest，将其放到板子上，然后以 root 身份执行即可。

附件： surfaceflinger_test.zip

参考资料

1. Android 系统匿名共享内存 Ashmem（Anonymous Shared Memory）简要介绍和学习计划
2. Android HAL(硬件抽象层)介绍以及调用
3. Android 帧缓冲区（Frame Buffer）硬件抽象层（HAL）模块 Gralloc 的实现原理分析
4. Android 进程间通信（IPC）机制 Binder 简要介绍和学习计划
5. ANDROID 窗口系统机制之显示机制
6. 浅谈 Google Skia 图形处理引擎
7. OpenGL 超级宝典（第4版） [美] Richard S. Wright 等 张琪 付非译 人民邮电出版社