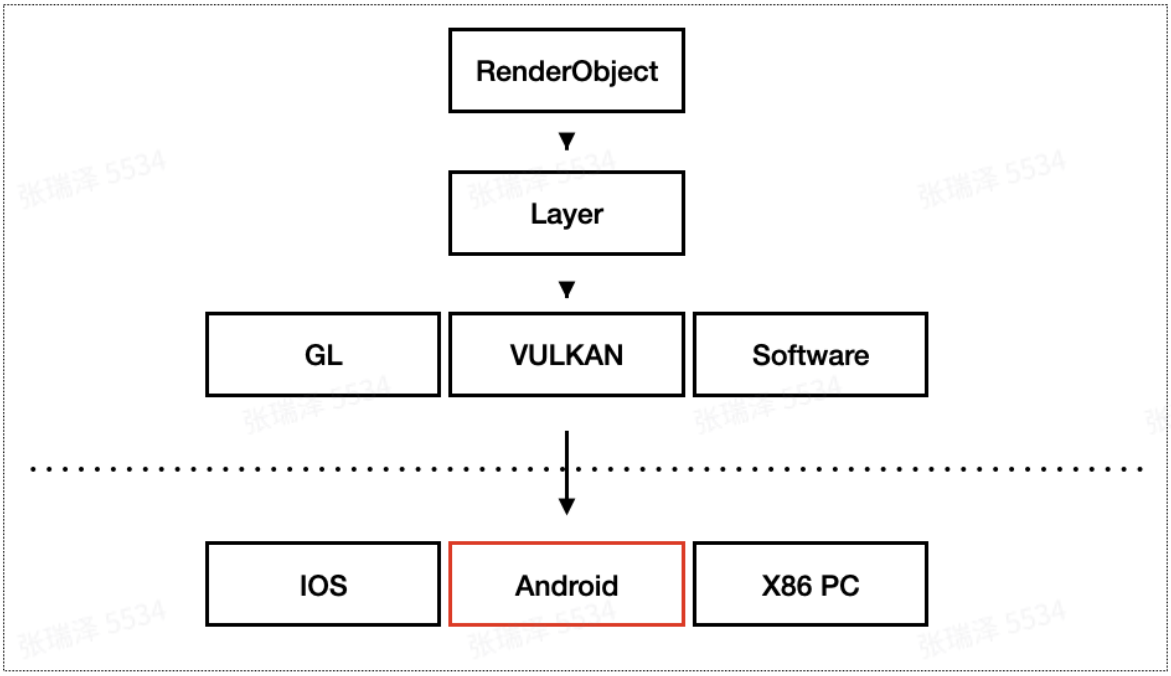
从Oryx Render到Android渲染框架

前言

Oryx Render从开始研发开始到现在已经大半年了。 Oryx-Renderer自渲染引擎概览 讨论了Oryx渲染引擎的基本原理。但是渲染链路到这一步其实并没有结束。



无论使用哪种具体的渲染方式，最终都要和特定的平台层对接。一些有心的同学可能会发现平台层代码里有类似FlutterSurfaceView/FlutterTextureView这样的文件，这些代码和框架接口会替我们完成最后显示在屏幕的工作。因此这次分享将关注oryx和android层的对接并延伸介绍android渲染框架，希望通过这种方式可以扩展我们对渲染引擎的理解。但是由于时间紧张和水平有限，我们将会从框架层面进行概要的介绍；如果有错误和疏漏，欢迎线下讨论。

Android Platform Rendering

重新看下FlutterView的构造方法，我们可以看到其和平台层交互的核心是**FlutterSurfaceView**(时间原因，这次不讨论FlutterTextureView)

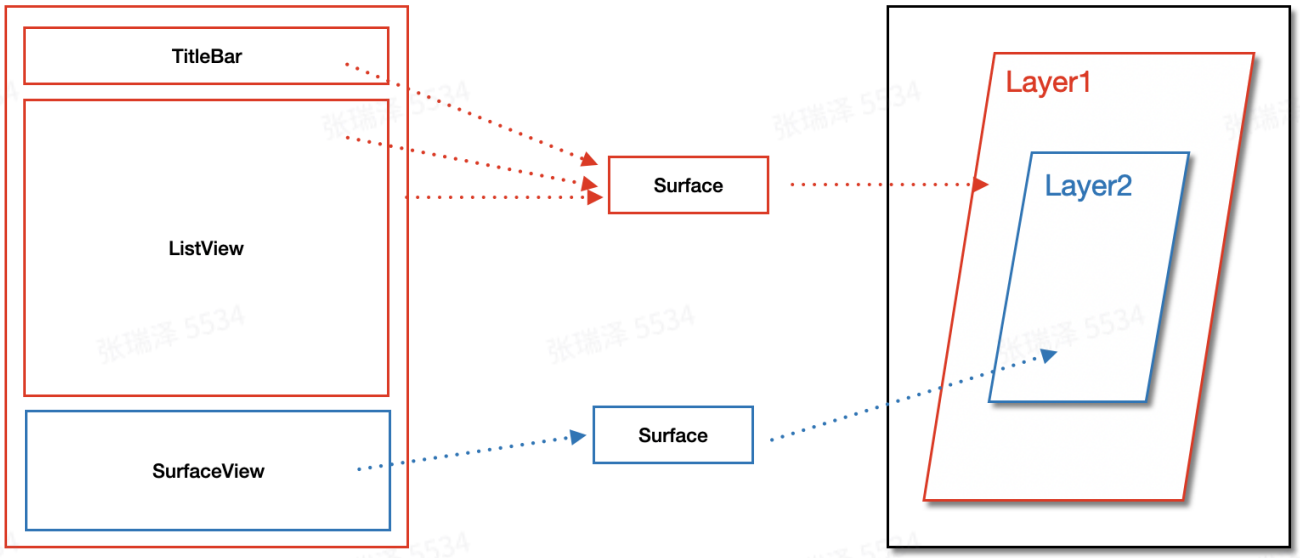
Java

```
FlutterSurfaceView flutterSurfaceView =
    new FlutterSurfaceView(
        host.getActivity(), host.getTransparencyMode() == TransparencyMode.transparent);
// Create the FlutterView that owns the FlutterSurfaceView.
flutterView = new FlutterView(host.getActivity(), flutterSurfaceView);
```

FlutterSurfaceView是SurfaceView的派生类。

SurfaceView

首先我们要看下Android APP UI 框架的基本结构



> 基于6.0版本绘制的图，有两点需要说明
>
> - 11的版本已经和图示有点不同但不影响核心的理解。在新版本中，SurfaceView和UI所在的Surface是同一个，Layer层面依然分开，但是具有父子关系。
> - Layer2其实位于Layer1之下，这是由SurfaceView的特性决定的。虽然Layer2在更下方，但是Layer1会提前绘制一片对应位置的透明空间，Layer2就可以把内容穿透显示出来。

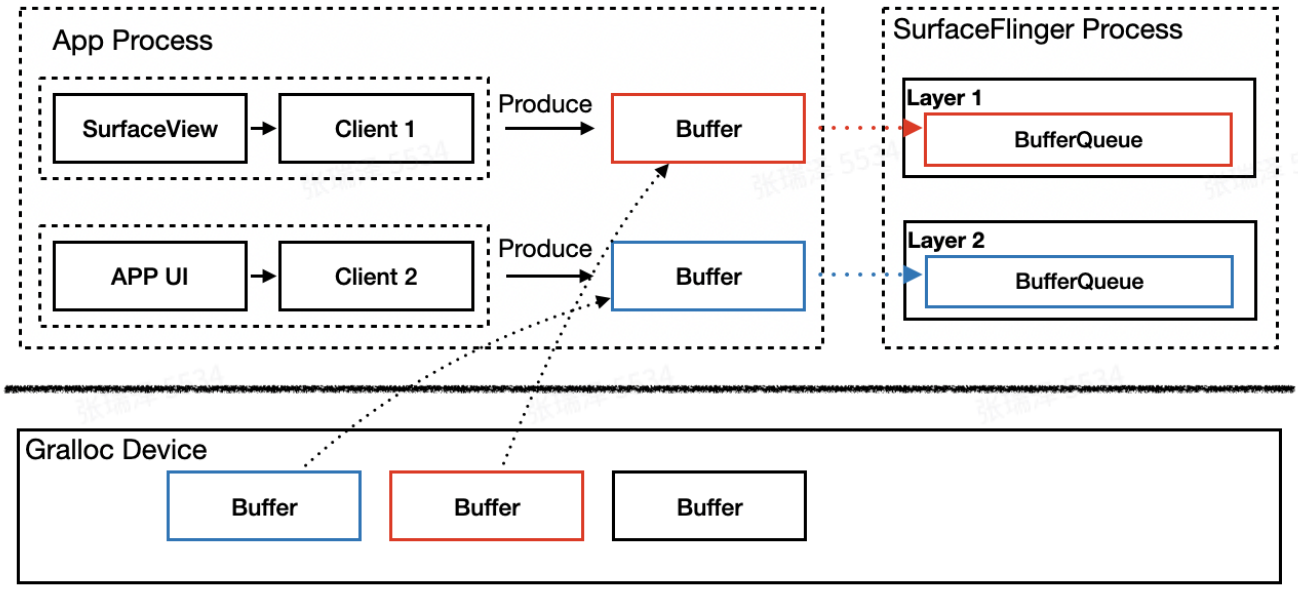
这里省略了测量和布局的流程，关注于渲染。总的来说，对于一般的UI组件，比如image/text/button等都是位于同一个Surface上，并最终绘制在同一个图层上。但是SurfaceView不同，其拥有自己独立的图层。也就是说FlutterSurfaceView在APP层面单独创建了自己的图层进行自身内容的渲染，这样的好处是不影响其他UI组件的渲染。那么下一个问题是，图层是如何被绘制的呢

渲染数据的生成

第一个需要关注的问题是渲染数据从何而来？一种通过Bitmap的方式进行传递，另一种则是通过记录Rendering operations的方式。

> 以Bitmap作为直接渲染数据载体的情况主要是软件渲染(CPU渲染)，目前的Androids手机使用这种方式的已经比较少了。
>
> 通过记录和传递Rendering Op的方式则出现在Hardware accelerate rendering, 是目前主流的渲染方式。数据以op集合的方式传递，并在回放(replay operations)阶段通过OPGNGL命令填充源数据。

然而无论数据的初始载体是什么，最终都会分配到对应的Buffer中，如图:



> Gralloc 可以简单理解为图形缓冲分配器。使用类似于匿名共享内存的方式在APP进程和系统进程之间进行跨进程共享。
>
> SurfaceFlinger则是系统服务进程，其职责包括VSYNC监听，管理所有可能要渲染的图层并消费渲染数据, 调度合成并上屏幕。

第一个需要注意到的是里面的生产者-消费者模型。生产帧(UI操作)和消费帧(VSYNC驱动)是Android渲染框架里的基础模型。

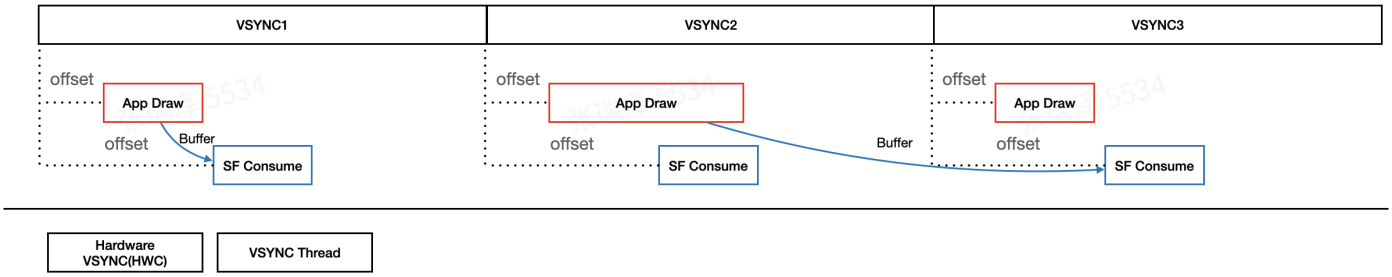
第二个是应用进程和实际进行渲染的进程是不同的。所以这里的Buffer都涉及到跨进程通信。不过本次不对通信方式过多展开。

> Android 底层使用"Binder"作为底层跨进程通信的核心方式。本质上通过C-S模型管理共享内存块的方式进行通信。

渲染数据的消费

刚刚说了渲染数据的生成。但是应该何时消费呢，这里涉及到对Vsync信号的理解。

Vsync(Vertical Synchronization,垂直同步)是一种在PC上很早就广泛使用的技术，可以理解为是一种定时中断。而在Android 4.1(JB)中已经开始引入VSync机制来同步渲染。对于早期最大支持60帧的手机而言，则Vsync的时间间隔是1000/60=16.7ms。



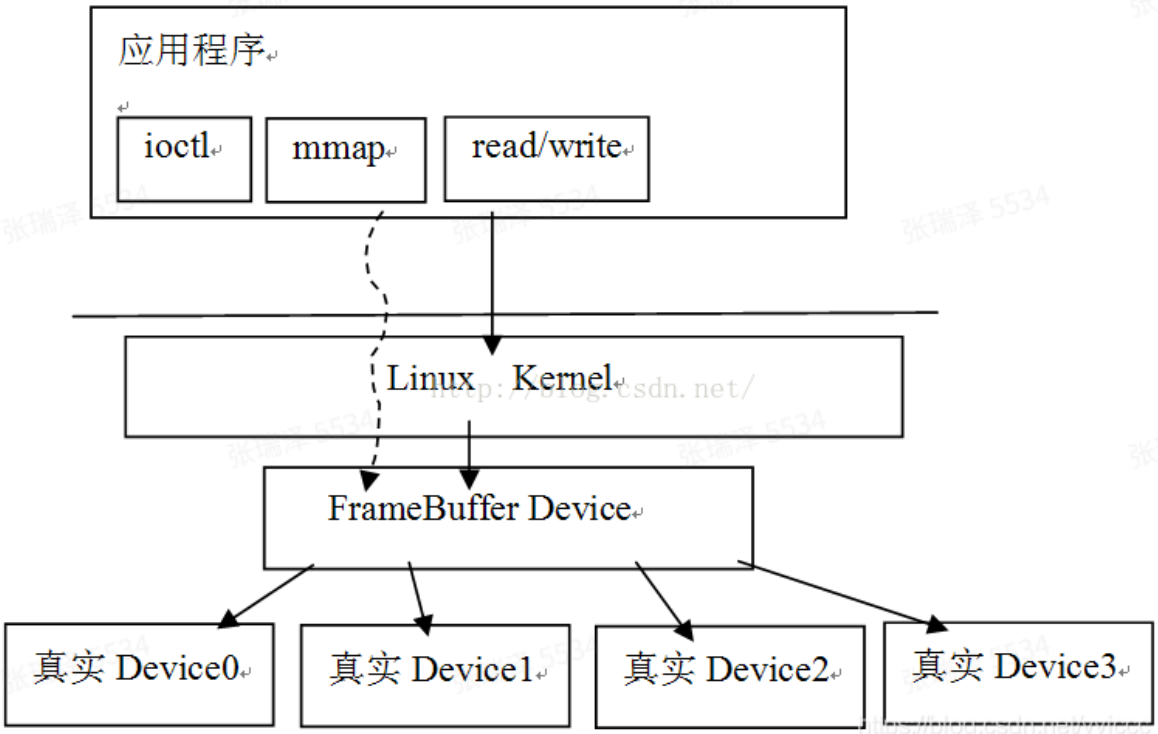
这里展示了Android Vsync模型的概要。红色边框代表生产者，蓝色为消费者，箭头代表生产出的数据流向。基本上有以下几点

- Android支持硬件或者软件模型生成Vsync信号。
- 支持"Offset"模型(具体做不做取决于厂商)。图示展示了"Offset"模型的作用。对于传统的Vsync模型，VSYNC1 生成的帧数据在Vsync2消费合成上屏。但是在Offset模型下，如果VSYNC1生成帧数据比较快，在Offset时间内完成，则可以在本帧内即刻消费上屏。

总结下，生成的数据会在本帧或者下一帧的开始阶段被SurfaceFlinger消费掉，也就是对Layer合成并上屏。

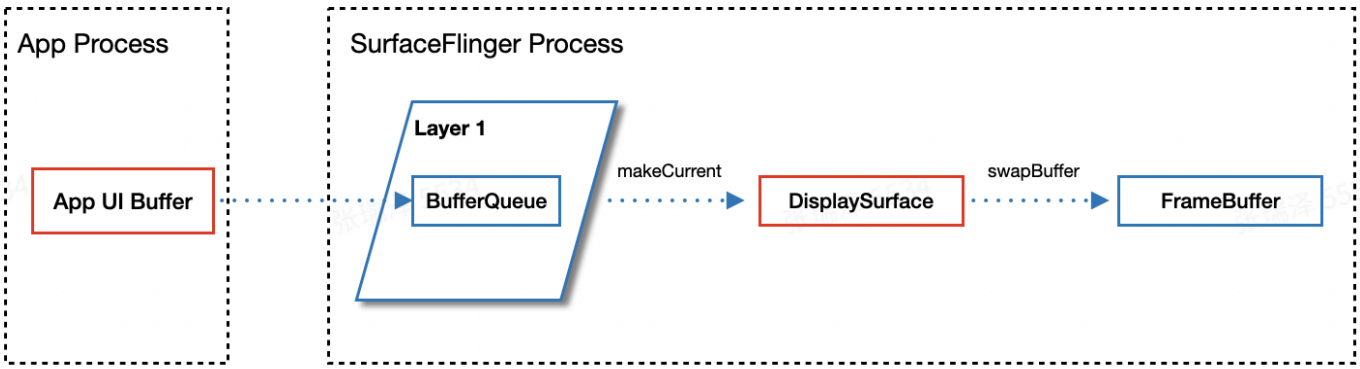
渲染数据的上屏

最后聊下渲染数据的上屏，首先需要提下FrameBuffer(帧缓冲)的概念。



FrameBuffer(简称FB)是指一个存储图形/图像帧数据的缓冲区。FB是Linux系统中的一个虚拟设备，设备文件对应为/dev/fb%d（比如/dev/fb0）。这个虚拟设备将不同硬件厂商实现的真实设备统一在一个框架。然后应用层就可以通过类似'ioctl" "mmap"这样的方式就可以操作显示设备。对这个缓冲进行写操作就相当于在屏幕上绘画。

了解了这一点，再去了解数据写到FB的基础模型。



红色生产者,蓝色消费者。渲染数据被SF消费的时候，SF会确认Layer对象所属的显示设备也就是"DisplaySurface"，然后通过一系列的GL命令进行数据填充，这里简单看下实现。

C++

```
EGLBoolean eglMakeCurrent(  EGLDisplay dpy, EGLSurface draw,
                            EGLSurface read, EGLContext ctx)
{
    // some code

    // dequeue a buffer
    int fenceFd = -1;
    if (nativeWindow->dequeueBuffer(nativeWindow, &buffer,
            &fenceFd) != NO_ERROR) {
        return setError(EGL_BAD_ALLOC, EGL_FALSE);
    }

   // some code
}
```

可以看到makeCurrent调用dequeueBuffer从BufferQueue中取出一个Free Buffer 用于后续渲染数据填充。

再看下swapBuffer:

PHP

```
void DisplayDevice::swapBuffers(HWComposer& hwc) const {
    if (hwc.hasClientComposition(mHwcDisplayId) || hwc.hasFlipClientTargetReques
t(mHwcDisplayId)) {
        mSurface->swapBuffers();
    }
    status_t result = mDisplaySurface->advanceFrame();
    // some code
}
```

看下内部的swapBuffers调用的最终实现:

JavaScript

```
EGLBoolean eglSwapBuffers(EGLDisplay dpy, EGLSurface draw)
{
    // some code
    nativeWindow->queueBuffer(nativeWindow, buffer, -1);
    // dequeue a new buffer
}
```

可以看到eglSwapBuffers会通过调用queueBuffer将刚刚填充的渲染数据加入到BufferQueue中。

然后advanceFrame将会消费这个Buffer:

C++

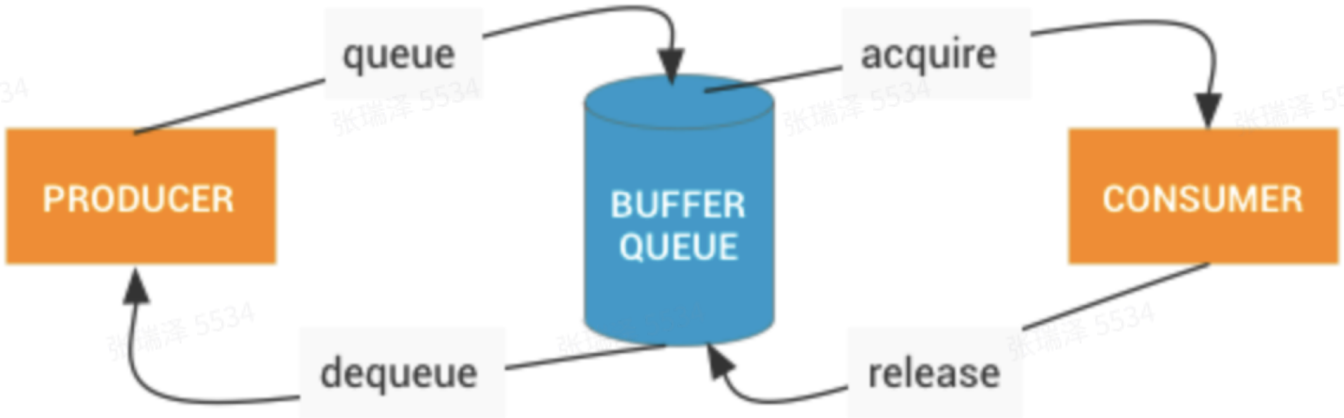
```
status_t FramebufferSurface::advanceFrame() {
    status_t result = nextBuffer(slot, buf, acquireFence, dataspace);
}

status_t FramebufferSurface::nextBuffer(uint32_t& outSlot,
        sp<GraphicBuffer>& outBuffer, sp<Fence>& outFence,
        Dataspace& outDataspace) {
    BufferItem item;
    status_t err = acquireBufferLocked(&item, 0);
    status_t result = mHwc.setClientTarget(mDisplayType, outSlot, outFence, outBuffer, outDataspace);
}
```

acquireBufferLocked会获取之前加入到队列中的Buffer 并通过setClientTarget把Buffer设置给HWC用于硬件合成并做好最终上屏的准备。

> 硬件混合渲染器 (HWC) HAL 用于合成从 SurfaceFlinger 接收的图层，从而减少 OpenGL ES (GLES) 和 GPU 执行的合成量。

最后我们再回顾下这个生产者消费者模型:



至此，就把Android设备渲染的基本原理介绍到这。然后再回去看SurfaceView的基本原理。

SurfaceView的绘制

这里先给个SurfaceView常规用法(软件渲染)的例子:

TypeScript

```
public void surfaceCreated(@NonNull SurfaceHolder holder) {
    // 1. Get canvas instance
    Canvas canvas = holder.lockCanvas();
    // 2. do some draw operations
    // 3. upload draw result
    holder.unlockCanvasAndPost();
}
```

可以看到基本上分三步，也是渲染的核心

获取canvas画板(lockCanvas)

C++

```
static jlong nativeLockCanvas(JNIEnv* env, jclass clazz,
        jlong nativeObject, jobject canvasObj, jobject dirtyRectObj) {
    ANativeWindow_Buffer outBuffer;
    status_t err = surface->lock(&outBuffer, dirtyRectPtr);

    SkBitmap bitmap;
    ssize_t bpr = outBuffer.stride * bytesPerPixel(outBuffer.format);
    bitmap.setInfo(info, bpr);
    if (outBuffer.width > 0 && outBuffer.height > 0) {
        bitmap.setPixels(outBuffer.bits);
    } else {
        // be safe with an empty bitmap.
        bitmap.setPixels(NULL);
    }

    Canvas* nativeCanvas = GraphicsJNI::getNativeCanvas(env, canvasObj);
    nativeCanvas->setBitmap(bitmap);
}
```

- Lock操作会在SF中创建对应的Layer(图层)和BufferQueue，并调用dequeueBuffer获取用于填充数据的缓冲区。
- SetPixels将缓冲区的起始地址设置给Bitmap对象。
- 最后可以看到，软件渲染的canvas就是在渲染这个Bitmap。 软件渲染一般用的是skia canvas，这个和flutter是一致的。

渲染

渲染层就是调用SK canvas的接口在Bitmap上进行绘制，没有特别要提的。

上传结果(unlockCanvasAndPost)

C++

```
static void nativeUnlockCanvasAndPost(JNIEnv* env, jclass clazz,
        jlong nativeObject, jobject canvasObj) {
    sp<Surface> surface(reinterpret_cast<Surface *>(nativeObject));
    // unlock surface
    status_t err = surface->unlockAndPost();
}

status_t Surface::unlockAndPost()
{
    int fd = -1;
    status_t err = mLockedBuffer->unlockAsync(&fd);
    err = queueBuffer(mLockedBuffer.get(), fd);
}
```

核心实现就是之前提过的queueBuffer。至此，SurfaceView的渲染数据已经加入到缓冲队列，用于后续SF消费。

现在我们清楚了SurfaceView的大致原理，可以进一步探索FlutterSurfaceView是如何运作的了。

FlutterSurfaceView

如果阅读过Flutter的平台层代码，就会知道FlutterSurfaceView并不是刚刚举例的标准用法。我们看下Flutter是怎么做的。

第一步是把平台层Surface传递到native层。

C++

```
static void SurfaceCreated(JNIEnv* env,
                           jobject jcaller,
                           jlong shell_holder,
                           jobject jsurface) {
  auto window = fml::MakeRefCounted<AndroidNativeWindow>(
      ANativeWindow_fromSurface(env, jsurface));
  ANDROID_SHELL_HOLDER->GetPlatformView()->NotifyCreated(std::move(window));
}

bool AndroidSurfaceSoftware::SetNativeWindow(
    fml::RefPtr<AndroidNativeWindow> window) {
  native_window_ = std::move(window);
  int32_t window_format = ANativeWindow_getFormat(native_window_->handle());
  return true;
}
```

Flutter用于渲染层对接的有多个类，除了AndroidSurfaceSoftware还有AndroidSurfaceGL和Vulkan。本质原理到最后都是一样的，比如说AndroidSurfaceGL的实现中通过eglCreateWindowSurface

与平台层Surface进行绑定，然后通过makeCurrent/SwapBuffer进行缓冲获取和上传操作。这里用AndroidSurfaceSoftware主要是原理展示更加清晰。

下面我们看下在flutter raster线程中是如何利用AndroidSurfaceSoftware进行渲染的最后一步的。

C++

```cpp
RasterStatus Rasterizer::DrawToSurface(flutter::LayerTree& layer_tree) {
    auto frame = surface_->AcquireFrame(layer_tree.frame_size());
    // draw layers
    RasterStatus raster_status = compositor_frame->Raster(layer_tree, false);
    frame->Submit();
}

SurfaceFrame::SubmitCallback on_submit =
    [self = weak_factory_.GetWeakPtr()](const SurfaceFrame& surface_frame,
                                        SkCanvas* canvas) -> bool {
  canvas->flush();
  return self->delegate_->PresentBackingStore(surface_frame.SkiaSurface());
};

bool AndroidSurfaceSoftware::PresentBackingStore(
    sk_sp<SkSurface> backing_store) {
  SkPixmap pixmap;
  if (!backing_store->peekPixels(&pixmap)) {
    return false;
  }

  ANativeWindow_Buffer native_buffer;
  if (ANativeWindow_lock(native_window_->handle(), &native_buffer, nullptr)) {
    return false;
  }

  std::unique_ptr<SkCanvas> canvas = SkCanvas::MakeRasterDirect(
        native_image_info, native_buffer.bits,
        native_buffer.stride * SkColorTypeBytesPerPixel(color_type));

  SkBitmap bitmap;
  if (bitmap.installPixels(pixmap)) {
     canvas->drawBitmapRect(
           bitmap, SkRect::MakeIWH(native_buffer.width, native_buffer.height),
           nullptr);
  }

  ANativeWindow_unlockAndPost(native_window_->handle());
  return true;
}

```

这里的流程相对有点长。我们忽略掉具体的渲染操作，关注最后submit一步。

Flutter在绘制完layer-tree以后将数据都填充到了SkSurface上。

1. 通过peekPixels将surface的像素转移到SkPixmap上。
2. ANativeWindow_lock类似于之前的lockCanvas方法，通过dequeue获取一个空的Buffer
3. 将Buffer的首地址设置给一个新建的SkCanvas.
4. 将第一步得到的SkPixmap填充到Bitmap上
5. 调用drawBitmapRect绘制在第三步获得的SkCanvas上，也就是填充Buffer。
6. ANativeWindow_unlockAndPost基本就是之前聊的unlockCanvasAndPost方法，将填充的Buffer加入队列。

至此虽然看起来要复杂很多，其实和前面聊的SurfaceView三个步骤本质是一样的。都是

1. Lock canvas(dequeue)
2. Draw (flush)
3. Unlock and post(enqueue)。

至此就和Android平台侧对接上了。

结尾

本次分享主要是尽可能简明的介绍了Android的渲染层框架和Flutter侧的对接。也出于方便理解原理的目的，介绍的链路并不完整。主要包括:

- 硬件加速渲染。软件渲染目前已经使用的比较少了。硬件加速渲染的核心特点是单独使用单独的Render Thread进行渲染而不是主线程，这样可以避免主线程过于阻塞
- TextureView/FlutterTextureView原理。TextureView不会新建独立的Layer而是以类似View的方式连接在Host Layer上。

但是其最终的核心原理是共同的。这些点有机会后续可以进一步分享。