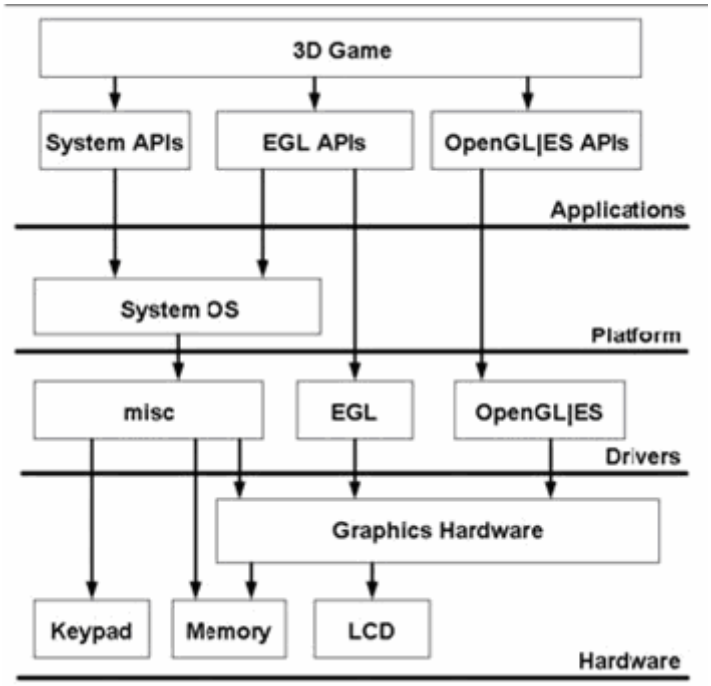
EGL 接口介绍

EGL 是 OpenGL ES 和底层 Native 平台视窗系统之间的接口。本章主要讲述 OpenGL ES 的 EGL API ，以及如何用它创建 Context 和绘制 Surface 等，并对用于 OpenGL 的其他视窗 API 做了比较分析，比如 WGL 和 GLX。本章中将涵盖如下几个方面：

1 EGL介绍

EGL 是为 OpenGL ES 提供平台独立性而设计。在本章中，你将详细地学习每个 EGL API ，并了解使用 EGL 时候需要注意的平台特性和限制。OpenGL ES 为附加功能和可能的平台特性开发提供了扩展机制，但仍然需要一个可以让 OpenGL ES 和本地视窗系统交互且平台无关的层。 OpenGL ES 本质上是一个图形渲染管线的状态机，而 EGL 则是用于监控这些状态以及维护 Frame buffer 和其他渲染 Surface 的外部层。

图 2-1 是一个典型的 EGL 系统布局图。



EGL 视窗设计是基于人们熟悉的用于 Microsoft Windows（WGL）和 UNIX（GLX）上的 OpenGL 的 Native 接口，与后者比较接近。 OpenGL ES 图形管线的状态被存储于 EGL 管理的一个 Context 中。Frame Buffers 和其他绘制 Surfaces 通过 EGL API 创建、管理和销毁。EGL 同时也控制和提供了对设备显示和可能的设备渲染配置的访问。

2 EGL 数据类型

EGL 包含了自己的一组数据类型，同时也提供了对一组平台相关的本地数据类型的支持。这些 Native 数据类型定义在 EGL 系统的头文件中。一旦你了解这些数据类型之间的不同，使用它们将变得很简单。多数情况下，为保证可移植性，开发人员将尽可能使用抽象数据类型而避免直接使 用系统数据类型。通过使用定义在 EGL 中 Native 类型，可以让你写的 EGL 代码运行在任意的 EGL 的实现上。 Native EGL 类型说明如下：

- NativeDisplayType 平台显示数据类型，标识你所开发设备的物理屏幕;
- NativeWindowType 平台窗口数据类型，标识系统窗口;
- NativePixmapType 可以作为 Framebuffer 的系统图像（内存）数据类型，该类型只用于离屏渲染.

下面的代码是一个 NativeWindowType 定义的例子。这只是一个例子，不同平台之间的实现千差万别。使用 native 类型的关键作用在于为开发者抽象化这些细节。QUALCOMM 使用 IDIB 结构定义 native 类型，如下：

```
struct IDIB {
    AEEVTBL(IBitmap) *pvt; // virtual table pointer
    IQueryInterface * pPaletteMap; // cache for computed palette mapping info
    byte * pBmp; // pointer to top row
    uint32 * pRGB; // palette
    NativeColor ncTransparent; // 32-bit native color value
    uint16 cx; // number of pixels in width
    uint16 cy; // number of pixels in height
    int16 nPitch; // offset from one row to the next
    uint16 cntRGB; // number of palette entries
    uint8 nDepth; // size of pixel in bits
    uint8 nColorScheme; // IDIB_COLORSCHEME_...(ie. 5-6-5)
    uint8 reserved[6];
};
```

接下来的小节中，我们将深入更多 EGL 数据类型细节。标准 EGL 数据类型如表 2.1 所示。

表 2.1 EGL 数据类型

数据类型	值
EGLBoolean	EGL_TRUE =1, EGL_FALSE=0
EGLint	int 数据类型
EGLDisplay	系统显示 ID 或句柄
EGLConfig	Surface 的 EGL 配置
EGLSurface	系统窗口或 frame buffer 句柄
EGLContext	OpenGL ES 图形上下文
NativeDisplayType	Native 系统显示类型
NativeWindowType	Native 系统窗口缓存类型
NativePixmapType	Native 系统 frame buffer

3 EGL Displays

EGLDisplay 是一个关联系统物理屏幕的通用数据类型。对于 PC 来说，Display 就是显示器的句柄。不管是嵌入式系统或 PC，都可能有多个物理显示设备。为了使用系统的显示设备，EGL 提供了 EGLDisplay 数据类型，以及一组操作设备显示的 API 。

下面的函数原型用于获取 Native Display ：

```
EGLDisplay eglGetDisplay (NativeDisplayType display);
```

其中 display 参数是 native 系统的窗口显示 ID 值。如果你只是想得到一个系统默认的 Display，你可以使用 EGL_DEFAULT_DISPLAY 参数。如果系统中没有一个可用的 native display ID 与给定的 display 参数匹配，函数将返回 EGL_NO_DISPLAY，而没有任何 Error 状态被设置。

由于设置无效的 display 值不会有任何错误状态，在你继续操作前请检测返回值。下面是一个使用 EGL API 获取系统 Display 的例子：

```
m_eglDisplay = eglGetDisplay( system.display);
if (m_eglDisplay == EGL_NO_DISPLAY || eglGetError() != EGL_SUCCESS))
throw error_egl_display;
```

4 Initialization初始化

和很多视窗 API 类似，EGL 在使用前需要初始化，因此每个 EGLDisplay 在使用前都需要初始化。初始化 EGLDisplay 的同时，你可以得到系统中 EGL 的实现版本号。了解当前的版本号在向后兼容性方面是非常有价值的。嵌入式和移动设备通常是持续的投放到市场上，所以你需要考虑到你的代码将被运行在形色色的实现上。通过动态查询 EGL 版本号，你可以为新旧版本的 EGL 附加额外的特性或运行环境。基于平台配置，软件开发可用清楚知道哪些 API 可用访问，这将会为你的代码提供最大限度的可移植性。

下面是初始化 EGL 的函数原型：

```
EGLBoolean eglInitialize (EGLDisplay dpy, EGLint *major, EGLint *minor);
```

其中 dpy 应该是一个有效的 EGLDisplay。函数返回时，major 和 minor 将被赋予当前 EGL 版本号。比如 EGL1.0，major 返回 1 ，minor 则返回 0 。给 major 和 minor 传 NULL 是有效的，如果你不关心版本号。

eglQueryString()函数是另外一个获取版本信息和其他信息的途径。通过 eglQueryString()获取版本信息需要解析版本字符串，所以通过传递一个指针给 eglInitializ()函数比较容易获得这个信息。注意在调用 eglQueryString()必须先使用 eglInitialize()初始化 EGLDisplay，否则将得到 EGL_NOT_INITIALIZED 错误信息。

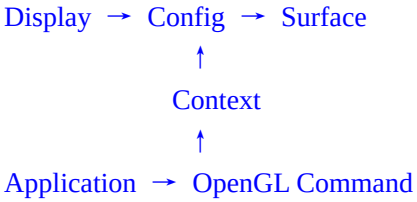
下面是获取 EGL 版本字符串信息的函数原型：

```
const char * eglQueryString (EGLDisplay dpy, EGLint name);
```

参数 name 可以是 EGL_VENDOR, EGL_VERSION, 或者 EGL_EXTENSIONS 。这个函数最常用来查询有哪些 EGL 扩展被实现。所有 EGL 扩展都是可选的，如果你想使用某个扩展特性，请检查该扩展是否被实现了，而不要想当然假定已经实现了。如果没有扩展被实现，将返回一个 Null 字符串，如果给定的 name 参数无效，则会得到 EGL_BAD_PARAMETER. 错误信息。

5 初始化EGL

OpenGL ES 是一个平台中立的图形库，在它能够工作之前，需要与一个实际的窗口系统关联起来，这与 OpenGL 是一样的。但不一样的是，这部份工作有标准，这个标准就是 EGL。而 OpenGL 时代在不同平台上有不同的机制以关联窗口系统，在 Windows 上是 wgl，在 X-Window 上是 xgl，在 Apple OS 上是 agl 等。EGL 的工作方式和部份术语都接近于 xgl。OpenGL ES 的初始化过程如下图所示意：



5.1 获取Display

Display 代表显示器，在有些系统上可以有多个显示器，也就会有多 Display。获得 Display 要调 EGLboolean eglGetDisplay(NativeDisplay dpy)，参数一般为 EGL_DEFAULT_DISPLAY。该参数实际的意义是平台实现相关的，在 X-Window 下是 XDisplay ID，在 MS Windows 下是 Window DC。

5.2 初始化egl

调用 EGLboolean eglInitialize(EGLDisplay dpy, EGLint *major, EGLint *minor)，该函数会进行一些内部初始化工作，并传回 EGL 版本号(major.minor)。

5.3 选择Config

所为 Config 实际指的是 FrameBuffer 的参数，在 MS Windows 下对应于 PixelFormat，在 X-Window 下对应 Visual。一般用 EGLboolean eglChooseConfig(EGLDisplay dpy, const EGLint * attr_list, EGLConfig * config, EGLint config_size, EGLint *num_config)，其中 attr_list 是以 EGL_NONE 结束的参数数组，通常以 id,value 依次存放，对于个别标识性的属性可以只有 id，没有 value。另一个办法是用 EGLboolean eglGetConfigs(EGLDisplay dpy, EGLConfig * config, EGLint config_size, EGLint *num_config) 来获得所有 config。这两个函数都会返回不多于 config_size 个 Config，结果保存在 config[]中，系统的总 Config 个数保存在 num_config 中。可以利用 eglGetConfig()中间两个参数为 0 来查询系统支持的 Config 总个数。Config 有众多的 Attribute，这些 Attribute 决定 FrameBuffer 的格式和能力，通过 eglGetConfigAttrib ()来读取，但不能修改。

5.4 构造Surface

Surface 实际上就是一个 FrameBuffer，通过 EGLSurface eglCreateWindowSurface

(EGLDisplay dpy, EGLConfig confg, NativeWindow win, EGLint *cfg_attr) 来创建一个可实际显示的 Surface。系统通常还支持另外两种 Surface：PixmapSurface 和 PBufferSurface，这两种都不 是可显示的 Surface，PixmapSurface 是保存在系统内存中的位图，PBuffer 则是保存在显存中的帧。

Surface 也有一些 attribute，基本上都可以根据字面意思理解， EGL_HEIGHT EGL_WIDTH EGL_LARGEST_PBUFFER EGL_TEXTURE_FORMAT EGL_TEXTURE_TARGET EGL_MIPMAP_TEXTURE EGL_MIPMAP_LEVEL ， 通过 eglSurfaceAttrib()设置、eglQuerySurface()读取。

5.5 创建Context

OpenGL 的 pipeline 从程序的角度看就是一个状态机，有当前的颜色、纹理坐标、变换矩阵、绚染模式等一大堆状态，这些状态作用于程序提交的顶点坐标等图元从而形成帧缓冲内的像素。在 OpenGL 的编程接口中，Context 就代表这个状态机，程序的主要工作就是向 Context 提供图元、设置状态，偶尔也从 Context 里获取一些信息。

用 EGLContext eglCreateContext(EGLDisplay dpy, EGLSurface write, EGLSurface read, EGLContext * share_list)来创建一个 Context。

5.6 绘制

应用程序通过 OpenGL API 进行绘制，一帧完成之后，调用 eglSwapBuffers(EGLDisplay dpy, EGLContext ctx)来显示。

EGL Configurations

EGLConfig 是一个用来描述 EGL surface 配置信息的数据类型。要获取正确的渲染结果，Surface 的格式是非常重要的。根据平台的不同，surface 配置可能会有限制，比如某个设备只支持 16 位色深显示，或是不支持 stencil buffer ，还有其他的功能限制或精度的差异。

下面是获取系统可用的 EGL 配置信息的函数原型：

EGLBoolean eglGetConfigs (EGLDisplay dpy, EGLConfig *configs,EGLint config_size, EGLint *num_config);

参数 configs 将包含在你的平台上有效的所有 EGL framebuffer 配置列表。支持的配置总数将通过 num_config 返回。实际返回的 configs 的配置个数依赖于程序传入的 config_size 。如果 config_size < num_config ，则不是所有的配置信息都将被返回。如果想获取系统支持的所有配置信息，最好的办法就是先给 eglGetConfig 传一个 NULL 的 configs 参数， num_config 将得到系统所支持的配置总数，然后用它来给 configs 分配合适的内存大小，再用得到的 configs 来调用 eglGetConfig 。

下面是如果使用 eglGetConfig() 函数的例子：

```
EGLConfig *configs_list;
EGLint num_configs;
// Main Display
m_eglDisplay = eglGetDisplay(EGL_DEFAULT_DISPLAY);
if( m_eglDisplay == EGL_NO_DISPLAY || eglGetError() != EGL_SUCCESS )
return FALSE;
if( eglInitialize( m_eglDisplay, NULL, NULL ) == EGL_FALSE || eglGetI
return FALSE;
// find out how many configurations are supported
if ( eglGetConfigs( m_eglDisplay, NULL, 0, &num_configs)   == EGL_FAI
return FALSE;
configs_list = malloc(num_configs * sizeof(EGLConfig));
if (configs_list == (EGLConfig *)0)
return FALSE;
// Get Configurations
if( eglGetConfigs( m_eglDisplay, configs_list, num_configs, &num_con
return FALSE;
```

由于当前平台的限制，通常只有很少的配置可用。系统支持的配置通常是利用系统硬件提供最好的性能。当你移植游戏到多个平台，它们的 EGL 配置可能会有细微的差别，我们希望作为通用的移植问题来直接处理这些问题。

选择一个 EGL Configuration

基于 EGL 的属性，你可以定义一个希望从系统获得的配置，它将返回一个最接近你的需求的配置。选择一个你特有的配置是有点不合适的，因为只是在你的平台上使用有效。eglChooseConfig() 函数将适配一个你所期望的配置，并且尽可能接近一个有效的系统配置。

下面是选择一个 EGL 配置的函数原型：

EGLBoolean eglChooseConfig(EGLDisplay dpy, const EGLint *attrib_list,
EGLConfig *configs, EGLint config_size, EGLint * num_config);

参数 attrib_list 指定了选择配置时需要参照的属性。参数 configs 将返回一个按照 attrib_list 排序的平台有效的所有 EGL framebuffer 配置列表。参数 config_size 指定了可以返回到 configs 的总配置个数。参数 num_config 返回了实际匹配的配置总数。

下面是如果使用 eglChoosetConfig() 函数的例子：

```
EGLint attrs[3] = { EGL_DEPTH_SIZE, 16, EGL_NONE };
EGLint num_configs;
EGLConfigs *configs_list;
// Get the display device
if ((eglDisplay = eglGetDisplay(EGL_NO_DISPLAY)) == EGL_NO_DISPLAY)
{
return eglGetError();
}
// Initialize the display
if (eglInitialize(eglDisplay, NULL, NULL) == EGL_FALSE)
{
return eglGetError();
}
// Obtain the total number of configurations that match
if (eglChooseConfig(eglDisplay, attrs, NULL, 0, &num_configs) == EGL_FALSE)
{
return eglGetError();
}
configs_list = malloc(num_configs * sizeof(EGLConfig));
if (configs_list == (EGLConfig *)0)
return eglGetError();
// Obtain the first configuration with a depth buffer of 16 bits
if (!eglChooseConfig(eglDisplay, attrs, &configs_list, num_configs, &num_configs))
{
return eglGetError();
}
```

如果找到多个合适的配置，有一个简单的排序算法用来匹配最接近你所查询的配置。表 2-2 显示了基于属性值的用来选择和排序的顺序，也包括了 EGL 规范中所有 EGL 配置属性及其默认值。

表 2.1 EGL 配置属性默认值和匹配法则

属性	数据类型	默认值	排序优先级	选择顺序
EGL_BUFFER_SIZE	int	0	3	Smaller value
EGL_RED_SIZE	int	0	2	Larger value
EGL_GREEN_SIZE	int	0	2	Larger value
EGL_BLUE_SIZE	Int	0	2	Larger value
EGL_ALPHA_SIZE	Int	0	2	Larger value
EGL_CONFIG_CAVET	enum	EGL_DONT_CARE	1(first)	Exact value
EGL_CONFIG_ID	int	EGL_DONT_CARE	9	Exact value
EGL_DEPTH_SIZE	int	0	6	Smaller value
EGL_LEVEL	int	0	-	Equal value
EGL_NATIVE_RENDERABLE	Boolean	EGL_DONT_CARE	-	Exact value
EGL_NATIVE_VISUAL_TYPE	int	EGL_DONT_CARE	8	Exact value
EGL_SAMPLE_BUFFERS	int	0	4	Smaller value
EGL_SAMPLES	int	0	5	Smaller value
EGL_STENCIL_SIZE	int	0	7	Smaller value
EGL_SURFACE_TYPE	bitmask			
EGL_WINDOW_BIT	-	Mask value		
EGL_TRANSPARENT_TYPE	enum	Exact value		
EGL_NONE	-			
EGL_TRANSPARENT_RED_VALUE	int			
EGL_DONT_CARE	-	Exact value		
EGL_TRANSPARENT_GREEN_VALUE	int			
EGL_DONT_CARE	-	Exact value		
EGL_TRANSPARENT_BLUE_VALUE	int			
EGL_DONT_CARE	-	Exact value		

Android OpenGL ES与EGL

1 名词解释

OpenGL ES (OpenGL for Embedded Systems，以下简称OpenGL)：OpenGL 三维图形 API 的子集，针对手机、PDA和游戏主机等嵌入式设备而设计。该API由Khronos集团定义推广，Khronos是一个图形软硬件行业协会，该协会主要关注图形和多媒体方面的开放标准。
EGL：EGL™ 是介于诸如OpenGL 或OpenVG的Khronos渲染API与底层本地平台窗口系统的接口。它被用于处理图形管理、表面/缓冲捆绑、渲染同步及支援使用其他Khronos API进行的高效、加速、混合模式 2D和 3D渲染。http://www.khronos.cn/index.shtml

2 Android中的OpenGL 与EGL

Android 2.0 版本之后图形系统的底层渲染均由 OpenGL 负责，OpenGL 除了负责处理 3D API 调用，还需负责管理显示内存及处理 Android SurfaceFlinger 或上层应用对其发出的 2D API 调用请求。

- 本地代码：
  framework/base/opengl/libs/egl
  Android EGL 框架，负责加载 OpenGL 函数库和 EGL 本地实现。
  framework/base/opengl/libagl
  Android 提供的 OpenGL 软件库
- JNI 代码：
  framework/base/core/jni/com_google_android_gles_jni_EGLImpl.cpp
  EGL 本地代码的 JNI 调用接口
  framework/base/core/jni/com_google_android_gles_jni_GLImpl.cpp
  framework/base/core/jni/android_opengl_GLESXXX.cpp
  OpenGL 功能函数的 JNI 调用接口
- Java 代码：
  framework/base/opengl/java/javax/microedition/khronos/egl
  framework/base/opengl/java/javax/microedition/khronos/opengles
  framework/base/opengl/java/com/google/android/gles_jni/
  framework/base/opengl/android/opengl
  EGL 和 OpenGL 的 Java 层接口，提供给应用开发者，通过 JNI 方式调用底层函数。

3 Android EGL实现

3.1 EGL 的主要功能

EGL 是用来管理绘图表面（Drawing surfaces），并且提供了如下的机制

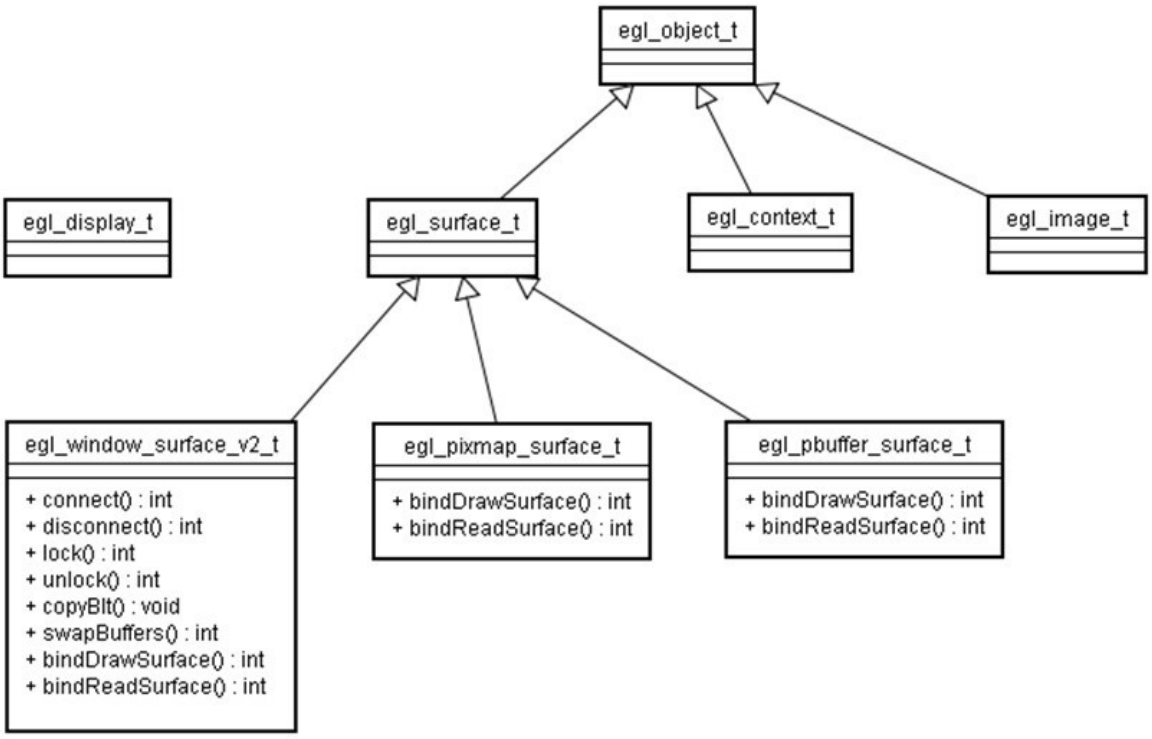
1. 与本地窗口系统进行通信
2. 查找绘图表面可用的类型和配置信息
3. 创建绘图表面
4. 同步 OpenGL ES 2.0 和其他的渲染 API（Open VG、本地窗口系统的绘图命令等）
5. 管理渲染资源，比如材质

3.2 EGL 数据结构

- egl_object_t 结构用来描述没一个 elg 对象。
- egl_display_t 结构用来存储 get_display 函数获取的物理显示设备。
- egl_surface_t 结构用来存储 surface 对象，系统可以同时拥有多个 surface 对象。
- egl_context_t 结构用用来存储 OpenGL 状态机信息。
- egl_image_t 结构用来存储 EGLImage 对象。

以下结构体在 libagl 中实现

- egl_window_surface_v2_t 继承 egl_surface_t，提供 egl_surface_t 功能的具体实现，属于可实际显示的 Surface。
- egl_pixmap_surface_t 存储保存在系统内存中的位图。
- egl_pbuffer_surface_t 存储保存在显存中的帧，以上两种位图属于不可显示的 Surface。



3.3 EGL 主要功能函数（省略了函数参数）

- EGLDisplay eglGetDisplay()：eglGetDisplay 调用 egl_display_t::get_display(dpy) 获取显示设备的句柄。
- EGLBoolean eglInitialize(EGLDisplay dpy, EGLint *major, EGLint *minor)：初

始化 EGL，获取 EGL 版本号。

- EGLBoolean eglTerminate()：结束一个 EGLDisplay，并不结束 EGL 本身。
- EGLBoolean eglGetConfigs()：获取 EGL 配置参数。
- EGLBoolean eglChooseConfig()：选择 EGL 配置参数，配置一个期望并尽可能接近一个有效的系统配置。
- EGLBoolean eglGetConfigAttrib()：获取 EGL 配置时需要参照的属性。
- EGLSurface eglCreateWindowSurface()：创建一个 EGL surface，surface 是可以实际显示在屏幕上类型。
- EGLSurface eglCreatePixmapSurface()，EGLSurface eglCreatePbufferSurface()：创建 EGL PixmapSurface 和 PbufferSurface 类型，这两种类型不可直接显示与屏幕上。
- EGLBoolean eglDestroySurface()：销毁一个 EGL surface 对象
- EGLBoolean eglQuerySurface()：查询 surface 的参数。
- EGLContext eglCreateContext()：创建一个 OpenGL 状态机。
- EGLBoolean eglDestroyContext()：销毁一个 OpenGL 状态机
- EGLBoolean eglMakeCurrent()：将 OpenGL context 与 surface 绑定。
- EGLBoolean eglQueryContext()：查询 context 的参数。
- EGLBoolean eglSwapBuffers()：绘制完图形后用于显示的函数。
- const char* eglQueryString()：查询 EGL 参数字串。

以上仅为 EGL 1.0 提供的基础功能，后续 EGL 版本陆续添加了更多的 API。

3.4 EGL 本地调用关系

framework/base/opengl/libs/egl 编译生成 libEGL.so。

libEGL.so 是 Android 系统的 EGL 框架，默认通过以下调用关系加载 OpenGL 库 libGLES_android.so：

eglGetDisplay()->egl_init_drivers()->egl_init_drivers_locked()-> loader.open()->load_driver()。

4 Android OpenGL 的实现

4.1 Android 提供的 OpenGL 库

framework/base/opengl/libagl 编译生成 libGLES_android.so。libGLES_android.so， 这是一个由系统提供的纯软件 3D 加速库，可以兼容各种环境。libGLES_android.so 中包含了一个 EGL 框架的实现和 OpenGL 各种 API 的实现。OpenGL 的 API 底层是通过 libpixelflinger.so 库实现的。

针对不同的硬件设计，GPU 厂商都会提供相应的硬件 3D 加速库，需要重写 libGLES_android.so 并实现相对应的 libpixelflinger.so，工程量较大，一般由 GPU 厂商的软件开发团队来完成。

4.2 EGL 加载 OpenGL API 的方法

libGLES_android.so 提供了两种 API，一种是 egl 实现 API，另一种是 OpenGL 标准 API。

4.2.1 加载EGL API

在函数 load_driver 中，通过 dlsym 从动态链接库中获取 egl 函数指针。其中 egl_names 包含 egl_entries.in 文件，egl_entries.in 文件则是 egl API 的声明。

```
void *Loader::load_driver(const char* driver_absolute_path,
        egl_connection_t* cnx, uint32_t mask)
{
... ...
        char const * const * api = egl_names;
        while (*api) {
            char const * name = *api;
            __eglMustCastToProperFunctionPointerType f =
                (__eglMustCastToProperFunctionPointerType)dlsym(dso, name);
            if (f == NULL) {
                // couldn't find the entry-point, use eglGetProcAddress()
                f = getProcAddress(name);
                if (f == NULL) {
                    f = (__eglMustCastToProperFunctionPointerType)0;
                }
            }
            *curr++ = f;
            api++;
        }
... ...
}
```

每次读取的函数地址赋值给了 cnx->egl 指向的 egl_t 结构体。

4.2.2 加载OpenGL API（这一段摘自师弟写的文章，要修）

Android 2.0 系统之前，加载 OpenGL API 和 EGL API 方式相同，均是通过 dlsym 加载。新版本的 Android 系统采用 TLS 技术进行动态加载。

TLS 让多线程程序设计更容易。TLS 是一个机制，通过它，程序可以拥有全局变量，但处于“每一线程各不相同”的状态。也就是说，进程中的所有线程都可以拥有全局变量，但这些变量只对某个线程 才有意义。一段代码在任何情况下都是相同的，而从 TLS 中取出的对每个线程却各不相同。

在 EGL 的 early_egl_init()中，对 TLS 机制进行初始化。将 TLS 里放入一个结构体指针，这个指针指向 gHooksNoContext，这个结构体里的每个函数指针被初始化为 gl_no_context。也就是现在如果通过 TLS 调用的 OpenGL ES API 都会调到 gl_no_context 这个函数中。

在 eglMakeCurrent 中，会将渲染上下文绑定到渲染面。在 EGL 中首先会处理完一系列和本地窗口系统的变量后，调用 libGLES_android.so 中的 eglMakeCurrent，调用成功的话会设置 TLS。将 TLS 指针指向前面已经初始化化好的 gl_hooks_t 结构体指针，这个结构体里的成员都已经指向了 libGLES_android.so 中的 OpenGL API 函数。

4.3 动态加载库的优化

Android 2.0 以上系统为 libagl 的 Android.mk 定义新的编译器参数-fvisibility=hidden。根据 GCC 编译器定义，使用该参数生成 的动态库将不会导出函数符号表，亦是动态库函数的 Vis 参数为 HIDDEN。这样可以防止导出多余函数，提高动态库的加载速度，对于需要导出的函数，可用 参数__attribute__((visibility("default")))手动指定其为“default”属性。
针对 libGLES_android.so 库，以 egl 和 gl 的 API 均由手动指定其属性为“default”。在 opengl/include/中，定义了 API 的额外参数 GL_API。

```
#define GL_API      KHRONOS_APICALL
#define KHRONOS_APICALL __attribute__((visibility("default")))
```

5 Android OpenGL 2D部分

libagl 中包含 egl 的实现和 OpenGL API，虽然 OpenGL API 属于软件实现，但是已为 2D 硬件加速预留了接口，主要位于 Texture 相关函数，调用 Android 的 Copybit 和 Gralloc 模块。

5.1 宏定义

libagl 的 Android.mk 中有如下定义，如果定义了 LIBAGL_USE_GRALLOC_COPYBITS 宏，编译时将加入 libagl 的 copybit.cpp 文件，并链接 libui 库。

```
# Set to 1 to use gralloc and copybits
LIBAGL_USE_GRALLOC_COPYBITS := 1
ifeq ($(LIBAGL_USE_GRALLOC_COPYBITS),1)
    LOCAL_CFLAGS += -DLIBAGL_USE_GRALLOC_COPYBITS
        LOCAL_SRC_FILES += copybit.cpp
        LOCAL_SHARED_LIBRARIES += libui
    Endif
```

5.2 2D 图形 API

void glDrawTexsvOES()v
void glDrawTexivOES()v
void glDrawTexsOES()v
void glDrawTexiOES()v
以上四个 API 调用函数：static void drawTexiOES(GLint x, GLint y, GLint z, GLint w, GLint h, ogles_context_t* c)
void glDrawTexfvOES()v
void glDrawTexxvOES()v
void glDrawTexfOES()v
void glDrawTexxOES()v
以上四个 API 调用函数：static void drawTexxOES(GLfixed x, GLfixed y, GLfixed z, GLfixed w, GLfixed h, ogles_context_t* c)

以上 8 个 API 的主要不同在于传入参数的类型。

drawTexiOES 函数用于处理整形数据，drawTexxOES 函数用于处理浮点转定点后的数据。以上两个函数如果定义了 LIBAGL_USE_GRALLOC_COPYBITS，则直接调用位于 copybit.cpp 中的 drawTexiOESWithCopybit(x, y, z, w, h, c)进行硬件加速，否则调用 drawTexxOESImp()纯软件实现。drawTexiOESWithCopybit 调用 copybit 函数处理 surface 以适应 copybit 模块。

5.3 libagl 中的 copybit 函数

copybit 函数共有 8 个参数，分别为 x、y 坐标，w、h 高宽，EGLTextureObject 对象，crop_rect 对象，transform 旋转方式，ogles_context_t OpenGL context 上下文。该函数主要执行以下操作：

判断源是否有 alpha 值；ν

判断是否需要进行 blending 操作；ν

选择纹理模式，并将纹理转换为 copybit 模块兼容模式；ν

确定 copybit 模块矩形框大小；ν

如果需要计算 alpha 通道:ν

调用 copybit 模块，将数据从目的地址考出至临时地址；υ

调用 copybit 模块，从源地址复制至目的地址；υ

调用 copybit 模块，从临时地址复制到目的地址，并带有 alpha 值。υ

如果不需要计算 alpha 值：ν

调用 copybit 模块，从源地址复制至目的地址；υ

针对需要计算 alpha 通道情况进行进一步解释：这种情况属于整个图形区域采用相同的 alpha 值。 需要表现的效果为背景透明效果，前景明显可见。由此得出计算公式为“前景 x(1-Alpha)+背景 x Alpha”，需要三个步骤，移出背景，移入前景，带 Alpha 参数移入背景。

Android 图形系统之 libui

1.libui 简介

libui 是 Android 图形库的本地框架，负责提供图形界面（Surface）的框架。libui 不仅仅负责图形界面框架，还提供处理事件输入、摄像头输出、Overlay 显示等框架，是整个图形用户交互（GUI）系统的中枢。

- 头文件位置 /frameworks/base/include/ui
- 源文件文章 /frameworks/base/libs/ui
- 编译生成动态链接库 libui.so

2 libui 库包含内容

1. Camera 相关框架及底层接口
2. Event / Key event 事件处理
3. Overlay 相关框架和底层接口
4. 定义一些图形显示相关数据结构（Rect、Region 和 PixelFormat）
5. Framebuffer 管理和显存分配
6. Surface 图形界面框架

本文将主要分析显示控制机制及图形界面框架。

3 Framebuffer管理和显存分配

3.1 设备初始化

libui 定义了 FramebufferNativeWindow 类，在构造函数中加载 Gralloc 硬件模块，并通过该模块提供的接口打开 Linux 的 Framebuffer 设备，同时将打开 Gralloc 模块。设备打开后，将初始化 FramebufferNativeWindow 的参数，将显示缓冲区设置为 2，同时向系统申请两块 NativeBuffer 结构用于存储显示设备的双缓冲，再调用 Gralloc 模块的 alloc 接口分配显存空间。以下为部分初始化代码：

```
if (hw_get_module(GRALLOC_HARDWARE_MODULE_ID, &module) == 0) {
        err = framebuffer_open(module, &fbDev);
        err = gralloc_open(module, &grDev);
        // initialize the buffer FIFO
        mNumBuffers = 2;
        mNumFreeBuffers = 2;
        mBufferHead = mNumBuffers-1;
        buffers[0] = new NativeBuffer(
                fbDev->width, fbDev->height, fbDev->format, GRALLOC_USAGE_HW_FB);
```

```
        buffers[1] = new NativeBuffer(
                fbDev->width, fbDev->height, fbDev->format, GRALLOC_USAGE_HW_FB);
        err = grDev->alloc(grDev, fbDev->width, fbDev->height, fbDev->format,
                GRALLOC_USAGE_HW_FB, &buffers[0]->handle, &buffers[0]->stride);
        err = grDev->alloc(grDev, fbDev->width, fbDev->height, fbDev->format,
                GRALLOC_USAGE_HW_FB, &buffers[1]->handle, &buffers[1]->stride);
... ...
```

3.2 共享式分配显存

相对于 Gralloc 模块采用独立内存空间作为显示缓冲区，libui 提供了一套通过 ashmem 机制，和系统主存共享分配显存的方案。这种方案适用于物理内存较少且对显示要求低的设备。该策略通过宏定义 GRALLOC_USAGE_HW_MASK 进行判断，如果标志位没有被置位，则初始化 sw_gralloc_handle_t 结构体。 sw_gralloc_handle_t::alloc 负责申请共享内存，根据图形格式计算需要申请的图形尺寸，计算公式为 size = bpp * w * h，另外需要保证 size 为 PAGE_SIZE 对齐。调用 ashmem_create_region 函数申请空间，调用 ashmem_set_prot_region 设置参数，最后用 mmap 获取申请空间的内存地址。

```
int fd = ashmem_create_region("sw-gralloc-buffer", size);
int prot = PROT_READ;
if (usage & GRALLOC_USAGE_SW_WRITE_MASK)
   prot |= PROT_WRITE;
ashmem_set_prot_region(fd, prot) ;
void* base = mmap(0, size, prot, MAP_SHARED, fd, 0);
```

sw_gralloc_handle_t::registerBuffer 会判断当前进程是否与 sw_gralloc_handle_t 相同，若不相同需要重新 mmap，获取新的内存地址。

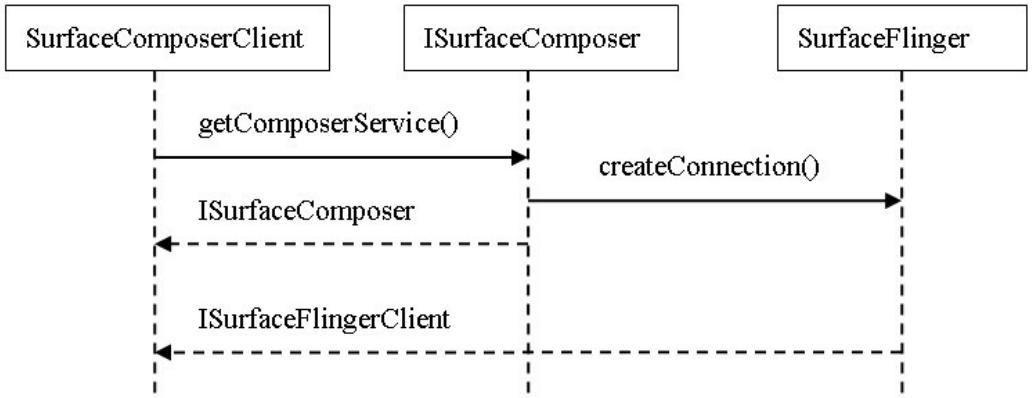
```
if (hnd->pid != getpid()) {
   void* base = mmap(0, hnd->size, hnd->prot, MAP_SHARED, hnd->fd, 0);
   hnd->base = intptr_t(base);
}
```

4 libui中的Surface图形界面框架

libui 库提供了 Surface 图形界面框架，包括上层应用的调用接口和于 SurfaceFlinger 库的通信接口，图形界面的具体实现由 SurfaceFlinger 库完成。Android 上层应用的调用接口主要包含 SurfaceComposerClient、 SurfaceControl 和 Surface 三个主要数据结构。

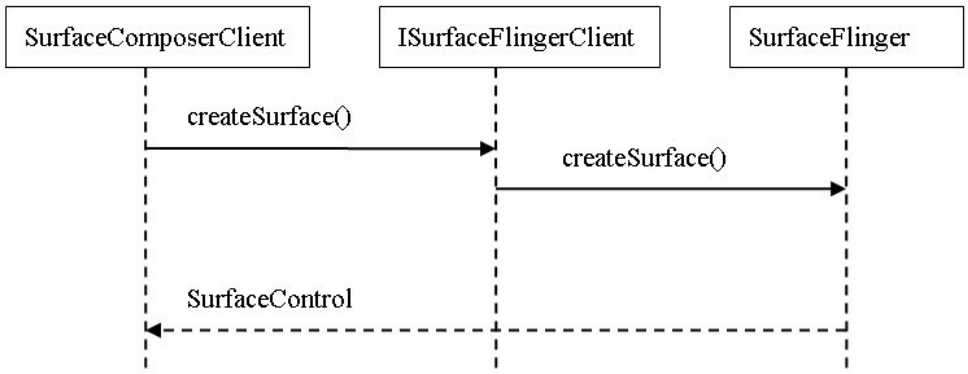
4.1 创建SurfaceComposerClient客户端

上层启动一个新的图形会话，首先要建立一个 SurfaceComposerClient 对象，用来创建图形会话客户端。SurfaceComposerClient 在其构造函数中调用 getComposerService()，返回 ISurfaceComposer 接口，在调用 ISurfaceComposer:createConnection()，通过 binder 和 SurfaceFlinger 库通讯，最终 SurfaceFlinger 库返回一个 ISurfaceFlingerClient 接口。



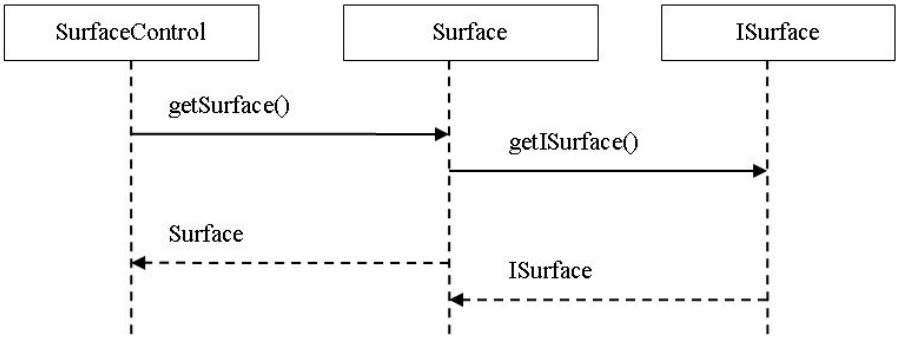
4.2 创建一个Surface

创建 SurfaceComposerClient 对象后，可以调用 createSurface()创建一个新的 Surface，实际上是调用 ISurfaceFlingerClient 接口的 createSurface()，通过 binder 进入 SurfaceFlinger 创建真正的显示 Layer。创建成功后将返回一个 SurfaceControl 类型的对象。



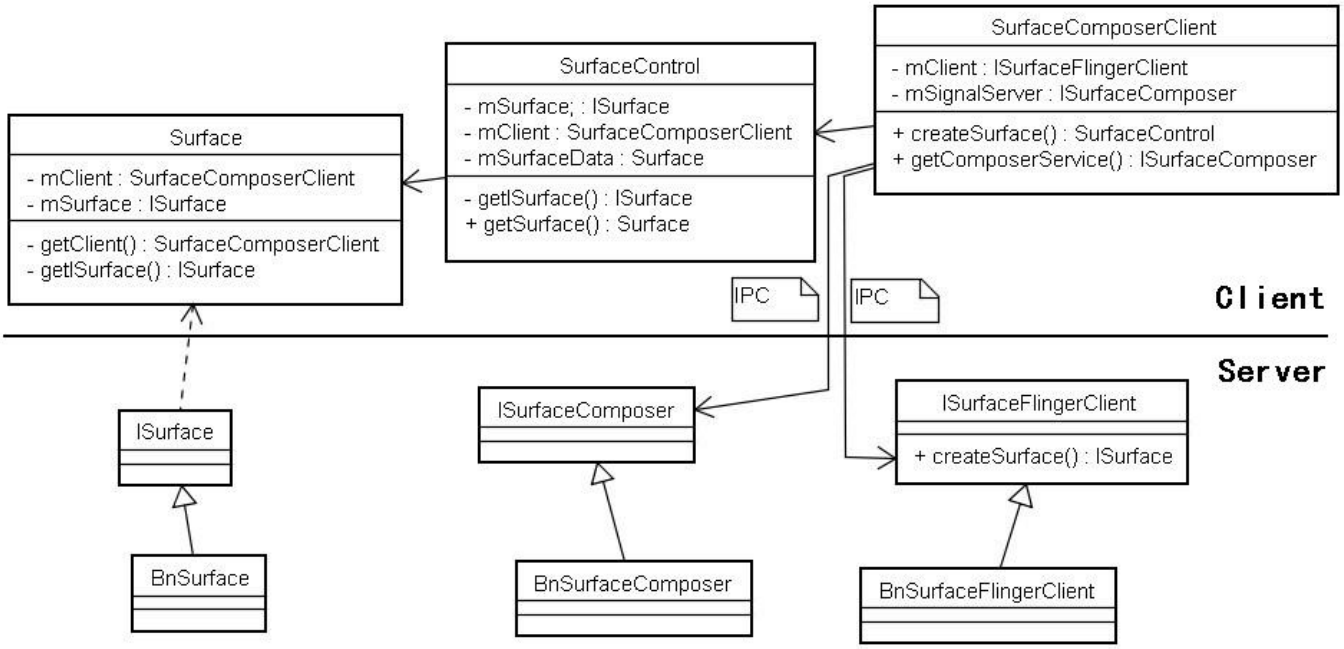
4.3 获取Surface和ISurface

SurfaceControl 对象可使用 getSurface()获取 Surface 对象。这两个数据结构分工不同，SurfaceControl 主要负责设置图形界面的参数，Surface 则提供了控制图形界面的接口。Surface 可以通过 getISurface()获取服务器端 ISurface 的接口，利用 ISurface 的接口，通过 binder 进程间通讯机制和服务器端传输 Surface 数据。



4.4 客户端和服务端模型

图中 Server 下层即 SurfaceFlinger 实现，这里没有列出。解释一下我理解的客户和服务端模型，上层图形应用程序利用 SurfaceComposerClient、SurfaceControl 及 Surface 对象向服务端申请 Surface，获取控制 Surface 的接口 ISurfaceXXX，利用该接口提供的 Binder 进程通讯机制和服务端进行通讯和数据交换。服务端指 libui 提供的 Surface 框架和 libsurfaceflinger 库，系统启动时将 SurfaceFlinger 注册为系统服务，负责处理所有客户端拥有的 Surface，决定当前显示内容及数据更新情况。



引用文章：

http://blog.csdn.net/DroidPhone/archive/2010/10/28/5972568.aspx

http://blog.csdn.net/yili_xie/archive/2009/11/12/4803527.aspx

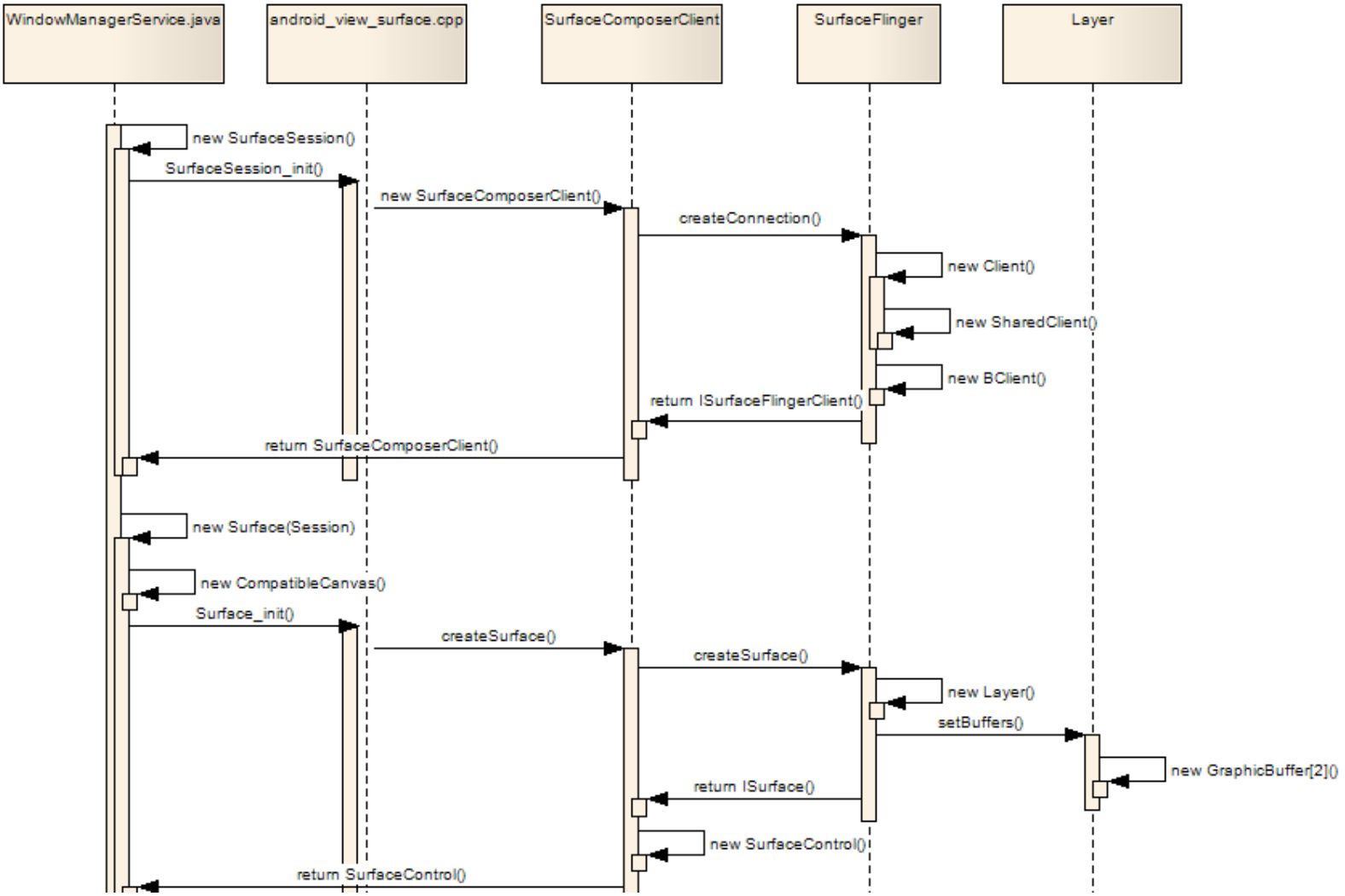
SurfaceFlinger 中的 SharedClient

SharedClient 用以客户端（Surface）和服务端（Layer）之间的显示缓冲区管理

SurfaceFlinger 在系统启动阶段作为系统服务被加载。应用程序中的每个窗口，对应本地代码中的 Surface，而 Surface 又对应于 SurfaceFlinger 中的各个 Layer，SurfaceFlinger 的主要作用是为这些 Layer 申请内存，根据应用程序的请求管理这些 Layer 显示、隐藏、重画等操作，最终由 SurfaceFlinger 把所有的 Layer 组合到一起，显示到显示器上。当一个应用程序需要在一个 Surface 上进行画图操作时，首先要拿到这个 Surface 在内存中的起始地址，而这块内存是在 SurfaceFlinger 中分配的，因为 SurfaceFlinger 和应用程序并不是运行在同一个进程中，如何在应用客户端（Surface）和服务端（SurfaceFlinger - Layer）之间传递和同步显示缓冲区？这正是本文要讨论的内容。

1 Surface 的创建过程

我们先看看 Android 如何创建一个 Surface，下面的序列图展示了整个创建过程。



创建 Surface 的过程基本上分为两步：

1. 建立 SurfaceSession

第一步通常只执行一次，目的是创建一个 SurfaceComposerClient 的实例，JAVA 层通过 JNI 调用本地代码，本地代码创建一个 SurfaceComposerClient 的实例，SurfaceComposerClient 通过 ISurfaceComposer 接口调用 SurfaceFlinger 的 createConnection，SurfaceFlinger 返回一个 ISurfaceFlingerClient 接口给 SurfaceComposerClient，在 createConnection 的过程中，SurfaceFlinger 创建了用于管理缓冲区切换 的 SharedClient，关于 SharedClient 我们下面再介绍，最后，本地层把 SurfaceComposerClient 的实例返回给 JAVA 层，完成 SurfaceSession 的建立。

2. 利用 SurfaceSession 创建 Surface

JAVA 层通过 JNI 调用本地代码 Surface_Init()，本地代码首先取得第一步创建的 SurfaceComposerClient 实例， 通过 SurfaceComposerClient，调用 ISurfaceFlingerClient 接口的 createSurface 方法，进入 SurfaceFlinger，SurfaceFlinger 根据参数，创建不同类型的 Layer，然后调用 Layer 的 setBuffers()方法， 为该 Layer 创建了两个缓冲区，然后返回该 Layer 的 ISurface 接口，SurfaceComposerClient 使用这个 ISurface 接口创建一个 SurfaceControl 实例，并把这个 SurfaceControl 返回给 JAVA 层。

由此得到以下结果：

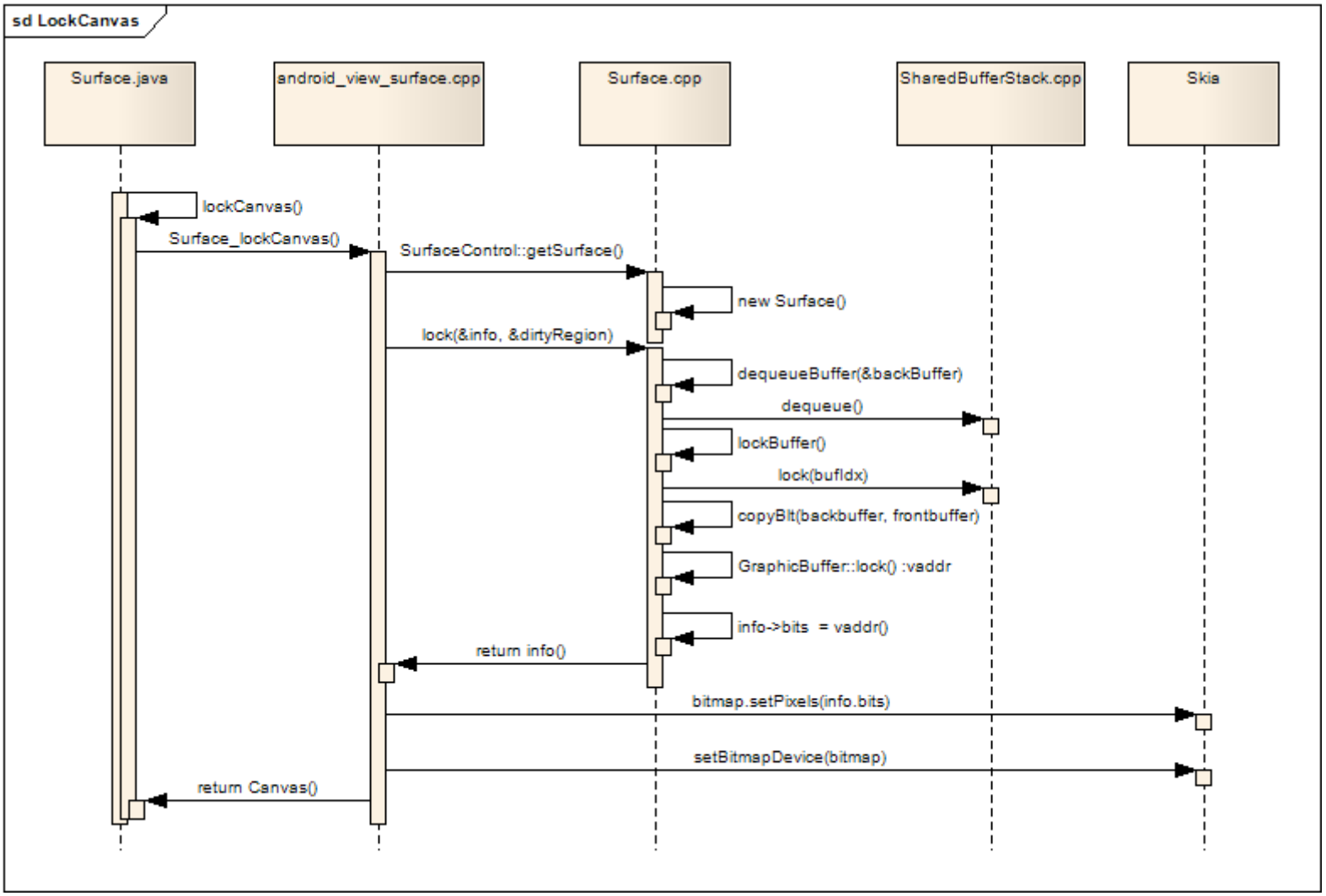
- JAVA 层的 Surface 实际上对应于本地层的 SurfaceControl 对象，以后本地代码可以使用 JAVA 传入的 SurfaceControl 对象，通过 SurfaceControl 的 getSurface 方法，获得本地 Surface 对象；
- Android 为每个 Surface 分配了两个图形缓冲区，以便实现 Page-Flip 的动作；
- 建立 SurfaceSession 时，SurfaceFlinger 创建了用于管理两个图形缓冲区切换的 SharedClient 对象，SurfaceComposerClient 可以通过 ISurfaceFlingerClient 接口的 getControlBlock()方法获得 这个 SharedClient 对象，查看 SurfaceComposerClient 的成员函数_init:

```
void SurfaceComposerClient::_init(
        const sp<ISurfaceComposer>& sm, const sp<ISurfaceFlingerClient>& conn)
{
    ......
    mClient = conn;
    if (mClient == 0) {
        mStatus = NO_INIT;
        return;
    }

    mControlMemory = mClient->getControlBlock();
    mSignalServer = sm;
    mControl = static_cast<SharedClient *>(mControlMemory->getBase());
}
```

2 获得 Surface 对应的显示缓冲区

虽然在 SurfaceFlinger 在创建 Layer 时已经为每个 Layer 申请了两个缓冲区，但是此时在 JAVA 层并看不到这两个缓冲 区，JAVA 层要想在 Surface 上进行画图操作，必须要先把其中的一个缓冲区绑定到 Canvas 中，然后所有对该 Canvas 的画图操作最后都会画到 该缓冲区内。下图展现了绑定缓冲区的过程：



图二　绑定缓冲区的过程

开始在 Surface 画图前，Surface.java 会先调用 lockCanvas()来得到要进行画图操作的 Canvas，lockCanvas 会进一步调用本地层的 Surface_lockCanvas，本地代码利用 JAVA 层传入的 SurfaceControl 对象，通过 getSurface() 取得本地层的 Surface 对象，接着调用该 Surface 对象的 lock()方法，lock()返回了改 Surface 的信息，其中包括了可用缓冲区的 首地址 vaddr，该 vaddr 在 Android 的 2D 图形库 Skia 中，创建了一个 bitmap，然后通过 Skia 库中 Canvas 的 API：Canvas.setBitmapDevice(bitmap)，把该 bitmap 绑定到 Canvas 中，最后把这个 Canvas 返回给 JAVA 层，这样 JAVA 层就可以在该 Canvas 上进行画图操作，而这些画图操作最终都会画在以 vaddr 为首地址的缓冲区中。

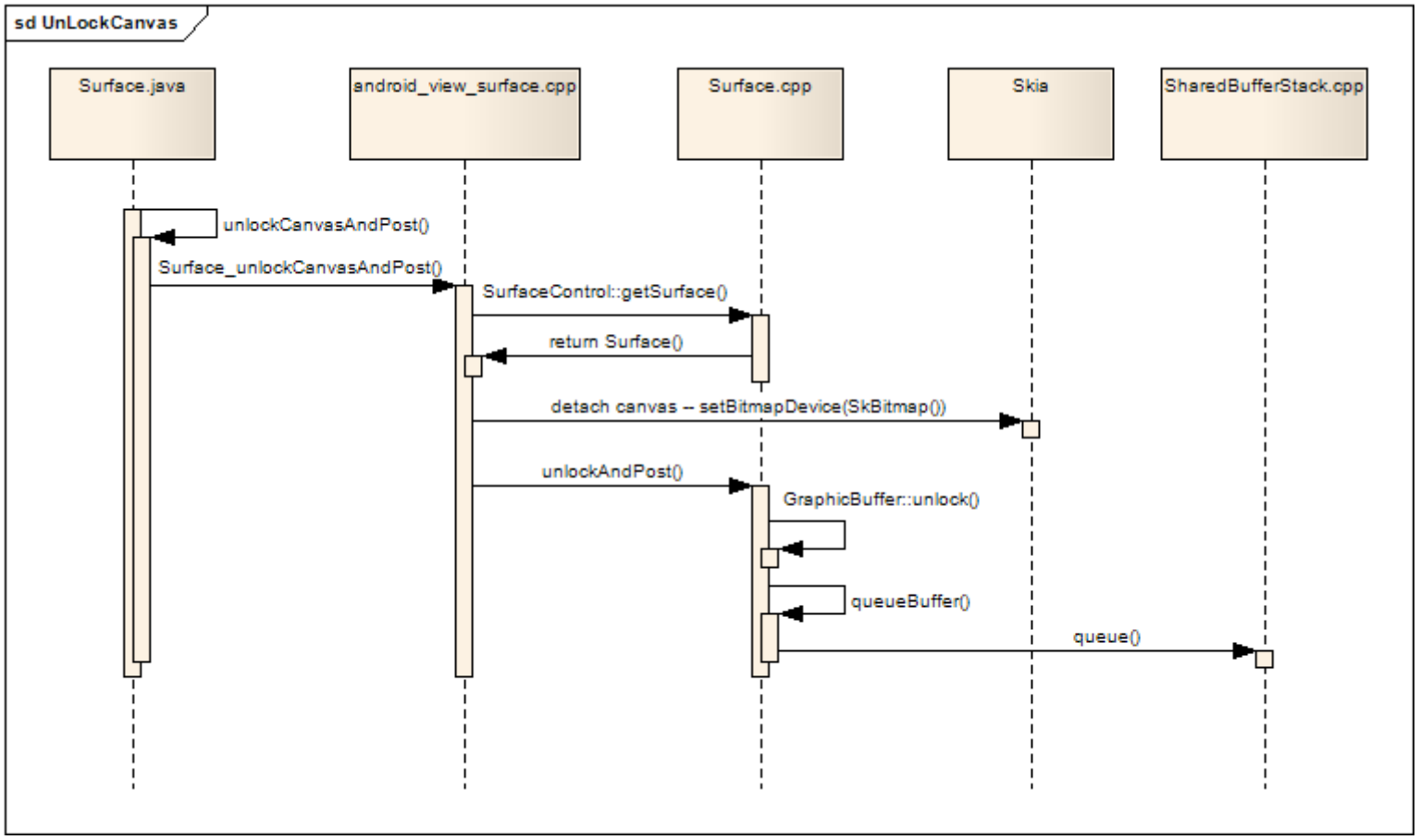
再看看在 Surface 的 lock()方法中做了什么：

- dequeueBuffer(&backBuffer)获取 backBuffer；
  - SharedBufferClient->dequeue()获得当前空闲缓冲区的编号；
  - 通过缓冲区编号获得真正的 GraphicBuffer:backBuffer；
  - 如果还没有对 Layer 中的 buffer 进行映射（Mapper），getBufferLocked 通过 ISurface 接口重新映射。
- 获取 frontBuffer；
- 根据两个 Buffer 的更新区域，把 frontBuffer 的内容拷贝到 backBuffer 中，这样保证了两个 Buffer 中显示内容的同步；

- backBuffer->lock() 获得 backBuffer 缓冲区的首地址 vaddr；
- 通过 info 参数返回 vaddr。

3 释放 Surface 对应的显示缓冲区

画图完成后，要想把 Surface 的内容显示到屏幕上，需要把 Canvas 中绑定的缓冲区释放，并且把该缓冲区从变成可投递（因为默认只有两个 buffer，所以实际上就是变成了 frontBuffer），SurfaceFlinger 的工作线程会在适当的刷新时刻，把系统中所有的 frontBuffer 混合在一起，然后通过 OpenGL 刷新到屏幕上。下图展现了解除绑定缓冲区的过程：



图三　解除绑定缓冲区的过程

- JAVA 层调用 unlockCanvasAndPost
- 进入本地代码：Surface_unlockCanvasAndPost
- 本地代码利用 JAVA 层传入的 SurfaceControl 对象，通过 getSurface()取得本地层的 Surface 对象
- 绑定一个空的 bitmap 到 Canvas 中
- 调用 Surface 的 unlockAndPost 方法
  - 调用 GraphicBuffer 的 unlock()，解锁缓冲区
  - 在 queueBuffer()调用了 SharedBufferClient 的 queue()，把该缓冲区更新为可投递状态

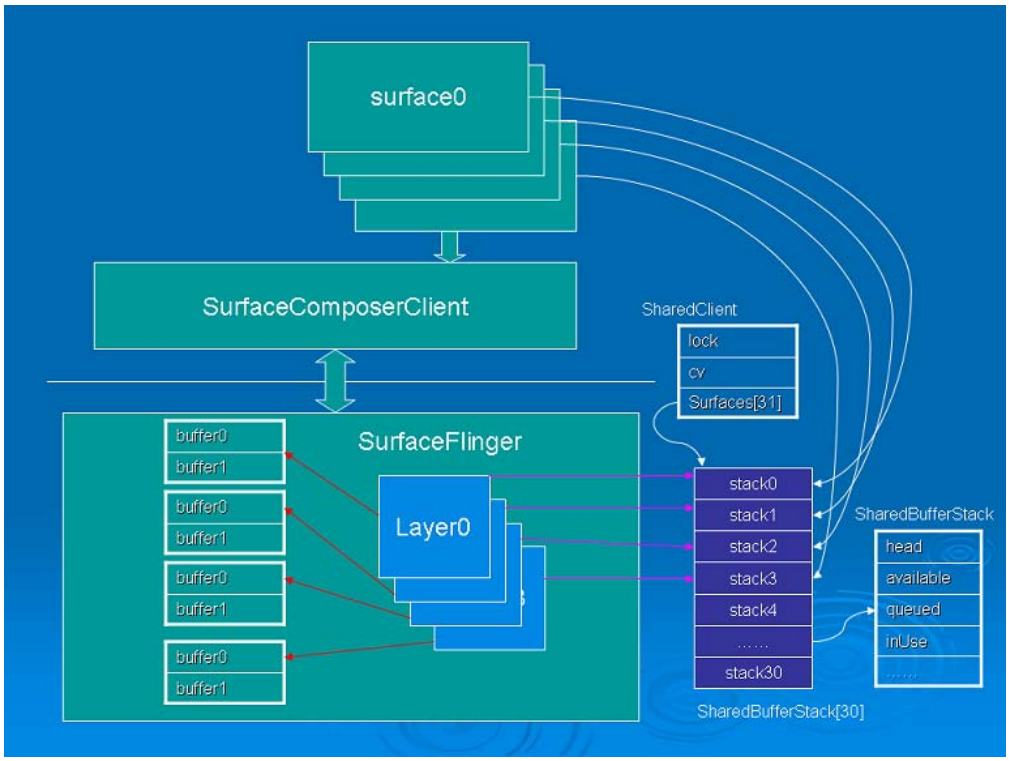
4 SharedClient 和 SharedBufferStack

从前面的讨论可以看到，Canvas 绑定缓冲区时，要通过 SharedBufferClient 的 dequeue 方法取得空闲的缓冲区，而解除绑定并提交缓冲区投递时，最后也要调用 SharedBufferClient 的 queue 方法通知 SurfaceFlinger 的工作线程。实际上，在 SurfaceFlinger 里，每个 Layer 也会关联一个 SharedBufferServer，SurfaceFlinger 的工作线程通过 SharedBufferServer 管理着 Layer 的缓冲区，在 SurfaceComposerClient 建立连接的阶段，SurfaceFlinger 就已经为该连接创建了一个 SharedClient 对象，SharedClient 对象中包含了一个 SharedBufferStack 数组，数组的大小是 31，每当创建一个 Surface，就会占用数组中的一个 SharedBufferStack，然后 SurfaceComposerClient 端的 Surface 会创建一个 SharedBufferClient 和该 SharedBufferStack 关联，而 SurfaceFlinger 端的 Layer 也会创建 SharedBufferServer 和 SharedBufferStack 关联，实际上每对 SharedBufferClient/SharedBufferServer 是控制着同一个 SharedBufferStack 对象，通过 SharedBufferStack，保证了负责对 Surface 的画图操作的应用端和负责刷新屏幕的服务端（SurfaceFlinger）可以使用不同的缓冲区，并且让他们之间知道对方何时锁定/释放缓冲区。

SharedClient 和 SharedBufferStack 的代码和头文件分别位于：

\frameworks\base\libs\surfaceflinger_client\SharedBufferStack.cpp

\frameworks\base\include\private\surfaceflinger\SharedBufferStack.h



图四　客户端和服务端缓冲区管理

5 SharedClient 、 SharedBufferStack 、 SharedBufferClient、SharedBufferServer

5.1 SharedClient

- 在 createConnection 阶段，SurfaceFlinger 创建 Client 对象：

```
sp<ISurfaceFlingerClient> SurfaceFlinger::createConnection()
{
    Mutex::Autolock _l(mStateLock);
    uint32_t token = mTokens.acquire();

    sp<Client> client = new Client(token, this);
    if (client->ctrlblk == 0) {
        mTokens.release(token);
        return 0;
    }
    status_t err = mClientsMap.add(token, client);
    if (err < 0) {
        mTokens.release(token);
        return 0;
    }
    sp<BClient> bclient =
        new BClient(this, token, client->getControlBlockMemory());
    return bclient;
```

- 再进入 Client 的构造函数中，它分配了 4K 大小的共享内存，并在这块内存上构建了 SharedClient 对象：

```
Client::Client(ClientID clientID, const sp<SurfaceFlinger>& flinger)
    : ctrlblk(0), cid(clientID), mPid(0), mBitmap(0), mFlinger(flinger)
{
    const int pgsize = getpagesize();
    const int cblksize = ((sizeof(SharedClient)+(pgsize-1))&~(pgsize-1));

    mCblkHeap = new MemoryHeapBase(cblksize, 0,
            "SurfaceFlinger Client control-block");

    ctrlblk = static_cast<SharedClient *>(mCblkHeap->getBase());
    if (ctrlblk) { // construct the shared structure in-place.
        new(ctrlblk) SharedClient;
    }
}
```

- 回到 createConnection 中，通过 Client 的 getControlBlockMemory()方法获得共享内存块的 IMemoryHeap 接口，接着创建 ISurfaceFlingerClient 的子类 BClient，BClient 的成员变量 mCblk 保存了 IMemoryHeap 接口指针；
- 把 BClient 返回给 SurfaceComposerClient，SurfaceComposerClient 通过 ISurfaceFlingerClient 接口的 getControlBlock()方法获得 IMemoryHeap 接口指针，同时保存在 SurfaceComposerClient 的成员变量 mControlMemory 中；
- 继续通过 IMemoryHeap 接口的 getBase ()方法获取共享内存的首地址，转换为 SharedClient 指针后保存在 SurfaceComposerClient 的成员变量 mControl 中；
- 至此，SurfaceComposerClient 的成员变量 mControl 和 SurfaceFlinger::Client.ctrlblk 指向了同一个内存块，该内存块上就是 SharedClient 对象。

5.2 SharedBufferStack 、 SharedBufferServer 、 SharedBufferClient

- SharedClient 对象中有一个 SharedBufferStack 数组：
  SharedBufferStack surfaces[NUM_LAYERS_MAX];
  NUM_LAYERS_MAX 被定义为 31，这样保证了 SharedClient 对象的大小正好满足 4KB 的要求。创建一个新的 Surface 时，进入 SurfaceFlinger 的 createSurface 函数后，先取在 createConnection 阶段创建的 Client 对象，通过 Client 在 0--NUM_LAYERS_MAX 之间取得一个尚未被使用的编号，这个编号实际上就是 SharedBufferStack 数组的索引：

```
int32_t id = client->generateId(pid);
```

- 然后以 Client 对象和索引值以及其他参数，创建不同类型的 Layer 对象，普通的 Layer 对象为例：

```
layer = createNormalSurfaceLocked(client, d, id,
                        w, h, flags, format);
```

- 在 createNormalSurfaceLocked 中创建 Layer 对象：

```
sp<Layer> layer = new Layer(this, display, client, id);
```

- 构造 Layer 时会先构造父类 LayerBaseClient，LayerBaseClient 中创建了 SharedBufferServer 对象，SharedBufferStack 数组的索引值和 SharedClient 被传入 SharedBufferServer 对象中。

```
LayerBaseClient::LayerBaseClient(SurfaceFlinger* flinger, DisplayID display,
        const sp<Client>& client, int32_t i)
    : LayerBase(flinger, display), lcblk(NULL), client(client), mIndex(i),
      mIdentity(uint32_t(android_atomic_inc(&sIdentity)))
{
    lcblk = new SharedBufferServer(
            client->ctrlblk, i, NUM_BUFFERS,
            mIdentity);
}
```

自此，Layer 通过 lcblk 成员变量(SharedBufferServer)和 SharedClient 共享内存区建立了关联，并且每个 Layer 对应于 SharedBufferStack 数组中的一项。

- 回到 SurfaceFlinger 的客户端 Surface.cpp 中，Surface 的构造函数如下：

```
Surface::Surface(const sp<SurfaceControl>& surface)
    : mClient(surface->mClient), mSurface(surface->mSurface),
      mToken(surface->mToken), mIdentity(surface->mIdentity),
      mFormat(surface->mFormat), mFlags(surface->mFlags),
      mBufferMapper(GraphicBufferMapper::get()), mSharedBufferClient(NULL),
      mWidth(surface->mWidth), mHeight(surface->mHeight)
{
    mSharedBufferClient = new SharedBufferClient(
            mClient->mControl, mToken, 2, mIdentity);

    init();
}
```

SharedBufferClient 构造参数 mClient->mControl 就是共享内存块中的 SharedClient 对象，mToken 就是 SharedBufferStack 数组索引值。

Surface 中的 mSharedBufferClient 成员和 Layer 中的 lcblk 成员(SharedBufferServer)，通过 SharedClient 中 的同一个 SharedBufferStack，共同管理着 Surface（Layer）中的两个缓冲区。