

Anbox图形渲染实现

1. Anbox图形渲染相关组件

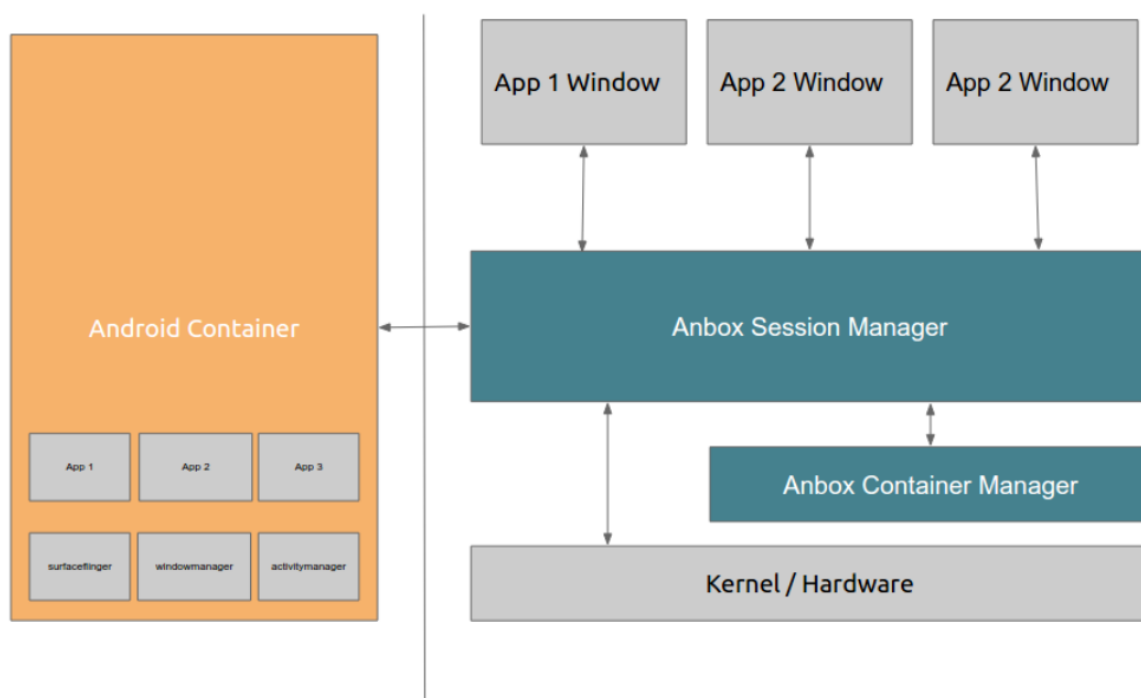


图1.1 Anbox基本架构

上图便是Anbox官方给出的架构图，从图中可以看到，Anbox主要有两个部分：Session Manager和Container Manager(Container Manager部分内容不在本文分析范围内)。其中Session Manager主要做了以下三部分工作：

1.1 虚拟Surfaceflinger

Surfaceflinger即Android的图像系统，可以将Android里的不同应用渲染的结构进行图层的合成，使得所有应用的渲染结果最终合成在一个窗口上。

1.2 虚拟Windowmanager

Windowmanager是指Android的窗口管理器，功能是给Android里的应用提供对应的窗口，即给每个应用提供自己的渲染界面。

1.3 虚拟Activitymanager

Activitymanager是指管理Android内部的进程管理器，主要功能是管理应用的启动、关闭等进程。

本文主要从Android端中的OpenGL ES初始化开始分析，到最终的指令经过Anbox端的翻译库进行翻译，最终由宿主机端的OpenGL进行图形渲染。下图为具体实现流程图：

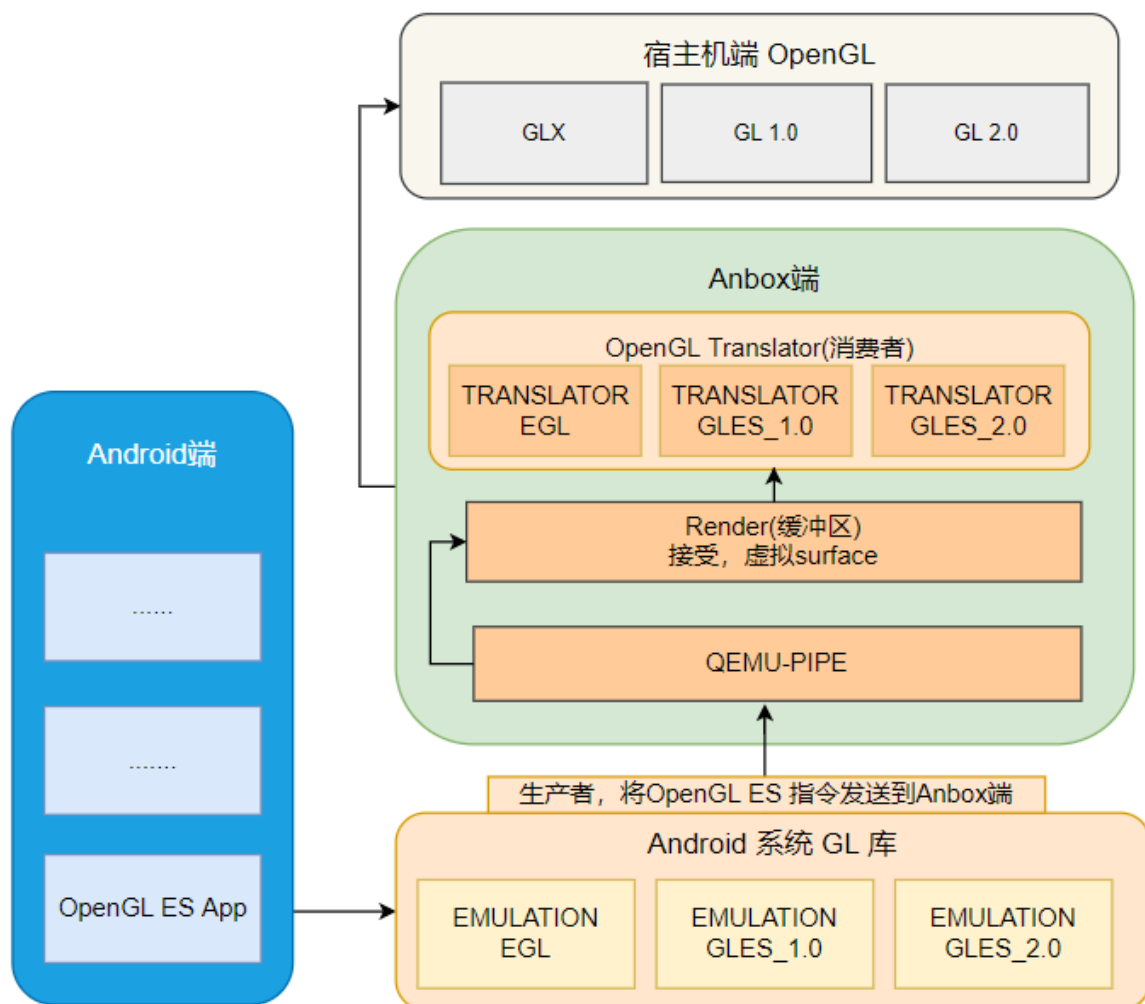


图1.2 OpenGL图形渲染流程图

2. OpenGL ES

Anbox的图形渲染部分实现与OpenGL ES息息相关，其实现了在Android模拟OpenGL相关库，接收相关指令并传输出去。下文将对OpenGL ES相关实现进行分析，在了解OpenGL ES之前，我们需要先了解什么是OpenGL。

2.1 什么是OpenGL

OpenGL是一套图像编程接口，对于开发者来说，其实就是一套C语言编写的API接口，通过这些接口，开发者可以调用显卡来进行计算机的图形开发。虽然OpenGL是一套API接口，但它并没有具体实现这些接口，接口的实现是由显卡的驱动程序来完成的。显卡驱动也是其他模块和显卡沟通的入口，开发者通过调用OpenGL的图像编程接口发出渲染命令，这些渲染命令被称为DrawCall，显卡驱动会将渲染命令翻译能被GPU理解的指令，然后通知GPU读取数据进行操作。

2.2 什么是OpenGL ES

OpenGL ES又是什么呢？它是为了更好地适应嵌入式等硬件较差的设备，推出的OpenGL的剪裁版，基本和OpenGL是一致的。Android从4.0开始默认开启硬件加速，也就是默认使用OpenGL ES来进行图形的生成和渲染工作。

2.3 如何使用OpenGL ES

了解Android如何使用OpenGL ES前，需要先对EGL有个简单了解，在本文中不涉及对EGL的深入分析。OpenGL虽然是跨平台的，但是在各个平台上也不能直接使用，因为每个平台的窗口都是不一样的，而EGL就是适配Android本地窗口系统和OpenGL ES桥接层。

OpenGL ES 定义了平台无关的 GL 绘图指令，EGL则定义了控制 displays，contexts 以及 surfaces 的统一的平台接口。

2.3.1 初始化与加载

Android7 加载OpenGL ES和EGL的源码在</frameworks/native/opengl/libs/EGL/>。

当 `eglGetDisplay` 被调用时，OpenGL 的库文件就被加载了，具体过程是：

(1) EGL初始化

```
EGLDisplay eglGetDisplay(EGLNativeDisplayType display)//获取设备屏幕
```

```
EGLBoolean eglInitialize(EGLDisplay display, // 指定EGL显示连接
                        EGLint *majorVersion, // 指定 EGL实现返回的 主版本号，可能为
NULL
                        EGLint *minorVersion); // 返回 EGL实现返回的 次版本号，可能为
NULL
```

在EGL初始化过程中调用了OpenGL初始化函数 `Loader::open`：

```
static EGLBoolean egl_init_drivers_locked() {
    if (sEarlyInitState) {
        // initialized by static ctor. should be set here.
        return EGL_FALSE;
    }

    // get our driver loader
    Loader& loader(Loader::getInstance());

    // dynamically load our EGL implementation
    egl_connection_t* cnx = &gEGLImpl;
    if (cnx->dso == 0) {
        cnx->hooks[egl_connection_t::GLESv1_INDEX] =
            &gHooks[egl_connection_t::GLESv1_INDEX];
        cnx->hooks[egl_connection_t::GLESv2_INDEX] =
            &gHooks[egl_connection_t::GLESv2_INDEX];
        //调用初始化OpenGL函数
        cnx->dso = loader.open(cnx);
    }

    return cnx->dso ? EGL_TRUE : EGL_FALSE;
}
```

(2) OpenGL ES初始化

`Loader::open(egl_connection_t* cnx)` 初始化图形驱动，主要是初始化这些函数表和指针。

`Loader::open(egl_connection_t* cnx)` 的定义如下。

```
static void* load_wrapper(const char* path) {
    void* so = dlopen(path, RTLD_NOW | RTLD_LOCAL);
    ALOGE_IF(!so, "dlopen(\"%s\") failed: %s", path, dlerror());
    return so;
}

#ifndef EGL_WRAPPER_DIR
#ifdef __LP64__
#define EGL_WRAPPER_DIR "/system/lib64"
#else
#define EGL_WRAPPER_DIR "/system/lib"
#endif
#endif

//设置模拟器属性（是否在模拟器中运行？在模拟中运行时是否有GPU支持？）
static void setEmulatorGlesvalue(void) {
    char prop[PROPERTY_VALUE_MAX];
    property_get("ro.kernel.qemu", prop, "0");
    if (atoi(prop) != 1) return;

    property_get("ro.kernel.qemu.gles", prop, "0");
    if (atoi(prop) == 1) {
        ALOGD("Emulator has host GPU support, qemu.gles is set to 1.");
        property_set("qemu.gles", "1");
        return;
    }

    // for now, checking the following
    // directory is good enough for emulator system images
    const char* vendor_lib_path =
        //NDK编译后的库不同手机上运行，可能加载的是 32 位 的 或者 64 的 库
#ifdef __LP64__
        "/vendor/lib64/egl";
#else
        "/vendor/lib/egl";
#endif
    //判断库文件是否存在
    const bool has_vendor_lib = (access(vendor_lib_path, R_OK) == 0);
    if (has_vendor_lib) {
        //存在则表示模拟器被GPU支持，通过客户的OpenGL ES 实现
        ALOGD("Emulator has vendor provided software renderer, qemu.gles is set to 2.");
        //设置qemu.gles值为2
        property_set("qemu.gles", "2");
    } else {
        //模拟器不被GPU所支持，设置qemu.gles值为0
        ALOGD("Emulator without GPU support detected. "
            "Fallback to legacy software renderer, qemu.gles is set to 0.");
        property_set("qemu.gles", "0");
    }
}
```

```

}

void* Loader::open(egl_connection_t* cnx)
{
    void* dso;
    driver_t* hnd = 0;
    // 设置模拟器属性（是否在模拟器中运行？在模拟中运行时是否有GPU支持？）
    setEmulatorGlesValue();
    //加载设备特有的图形驱动库，包括 EGL 库，OpenGL ES 1.0 和 2.0 的库。
    dso = load_driver("GLES", cnx, EGL | GLESv1_CM | GLESv2);
    if (dso) {
        hnd = new driver_t(dso);
    } else {
        // Always load EGL first
        dso = load_driver("EGL", cnx, EGL);
        if (dso) {
            hnd = new driver_t(dso);
            hnd->set( load_driver("GLESv1_CM", cnx, GLESv1_CM), GLESv1_CM );
            hnd->set( load_driver("GLESv2", cnx, GLESv2), GLESv2 );
        }
    }
}

LOG_ALWAYS_FATAL_IF(!hnd, "couldn't find an OpenGL ES implementation");
//加载图形驱动 wrapper，它们都位于 /system/lib64 或 /system/lib
cnx->libEgl = load_wrapper(EGL_WRAPPER_DIR "/libEGL.so");
cnx->libGles2 = load_wrapper(EGL_WRAPPER_DIR "/libGLESv2.so");
cnx->libGles1 = load_wrapper(EGL_WRAPPER_DIR "/libGLESv1_CM.so");

LOG_ALWAYS_FATAL_IF(!cnx->libEgl,
    "couldn't load system EGL wrapper libraries");

LOG_ALWAYS_FATAL_IF(!cnx->libGles2 || !cnx->libGles1,
    "couldn't load system OpenGL ES wrapper libraries");

return (void*)hnd;
}

```

这里的 `driver_t` 是Loader类的内部结构体：

```

struct driver_t {
    explicit driver_t(void* gles);
    ~driver_t();
    status_t set(void* hnd, int32_t api);
    void* dso[3];
};

```

`struct driver_t` 包含设备生产商提供的设备特有 EGL 和 OpenGL ES 实现库的句柄，如果 EGL 接口和 OpenGL 接口由单独的库实现，它包含一个库的句柄，即这个单独的库，如果 EGL 接口由不同的库实现，它则包含所有这些库的句柄。

在open函数中还调用了 `setEmulatorGlesValue` 函数，这个函数用于检查、设置一些模拟器属性（例如是否在模拟器中运行以及在模拟运行时是否有GPU支持）。

`Loader::load_driver()` 函数主要完成驱动库加载,以下是相关代码，加载驱动库可细分为三步进行：

```

/* This function is called to check whether we run inside the emulator,
 * and if this is the case whether GLES GPU emulation is supported.
 *
 * Returned values are:
 * -1  -> not running inside the emulator
 *  0  -> running inside the emulator, but GPU emulation not supported
 *  1  -> running inside the emulator, GPU emulation is supported
 *       through the "emulation" host-side OpenGL ES implementation.
 *  2  -> running inside the emulator, GPU emulation is supported
 *       through a guest-side vendor driver's OpenGL ES implementation.
 */
static int
checkGlesEmulationStatus(void)
{
    /* We're going to check for the following kernel parameters:
     *
     *      qemu=1                      -> tells us that we run inside the
emulator
     *      android.qemu.gles=<number> -> tells us the GLES GPU emulation status
     *
     * Note that we will return <number> if we find it. This let us support
     * more additionnal emulation modes in the future.
     */
    char prop[PROPERTY_VALUE_MAX];
    int result = -1;

    /* First, check for qemu=1 */
    property_get("ro.kernel.qemu",prop,"0");
    if (atoi(prop) != 1)
        return -1;

    /* We are in the emulator, get GPU status value */
    property_get("qemu.gles",prop,"0");
    return atoi(prop);
}

. . . . .
void Loader::init_api(void* dso,
    char const * const * api,
    __eglMustCastToProperFunctionPointerType* curr,
    getProcAddressType getProcAddress)
{
    const ssize_t SIZE = 256;
    char scrap[SIZE];
    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) {
            // Try without the OES postfix
            ssize_t index = ssize_t(strlen(name)) - 3;
            if ((index>0 && (index<SIZE-1)) && (!strcmp(name+index, "OES"))) {

```

```

        strncpy(scrap, name, index);
        scrap[index] = 0;
        f = (__eglMustCastToProperFunctionPointerType)dlsym(dso, scrap);
        //ALOGD_IF(f, "found <%s> instead", scrap);
    }
}
if (f == NULL) {
    // Try with the OES postfix
    ssize_t index = ssize_t(strlen(name)) - 3;
    if (index>0 && strcmp(name+index, "OES")) {
        snprintf(scrap, SIZE, "%sOES", name);
        f = (__eglMustCastToProperFunctionPointerType)dlsym(dso, scrap);
        //ALOGD_IF(f, "found <%s> instead", scrap);
    }
}
if (f == NULL) {
    //ALOGD("%s", name);
    f = (__eglMustCastToProperFunctionPointerType)gl_unimplemented;

    /*
     * GL_EXT_debug_label is special, we always report it as
     * supported, it's handled by GLES_trace. If GLES_trace is not
     * enabled, then these are no-ops.
     */
    if (!strcmp(name, "glInsertEventMarkerEXT")) {
        f = (__eglMustCastToProperFunctionPointerType)gl_noop;
    } else if (!strcmp(name, "glPushGroupMarkerEXT")) {
        f = (__eglMustCastToProperFunctionPointerType)gl_noop;
    } else if (!strcmp(name, "glPopGroupMarkerEXT")) {
        f = (__eglMustCastToProperFunctionPointerType)gl_noop;
    }
}
*curr++ = f;
api++;
}
}

void *Loader::load_driver(const char* kind,
    egl_connection_t* cnx, uint32_t mask)
{
    class MatchFile {
    public:
        static String8 find(const char* kind) {
            String8 result;
            int emulationStatus = checkGlesEmulationStatus();
            switch (emulationStatus) {
                case 0:
#ifdef __LP64__
                    result.setTo("/system/lib64/egl/libGLES_android.so");
#else
                    result.setTo("/system/lib/egl/libGLES_android.so");
#endif
                return result;
                case 1:
                    // Use host-side OpenGL through the "emulation" library

```

```

    #if defined(__LP64__)
        result.appendFormat("/system/lib64/egl/lib%s_emulation.so",
kind);
    #else
        result.appendFormat("/system/lib/egl/lib%s_emulation.so",
kind);
    #endif

    return result;
default:
    // Not in emulator, or use other guest-side implementation
    break;
}

String8 pattern;
pattern.appendFormat("lib%s", kind);
const char* const searchPaths[] = {
    #if defined(__LP64__)
        "/vendor/lib64/egl",
        "/system/lib64/egl"
    #else
        "/vendor/lib/egl",
        "/system/lib/egl"
    #endif
};

// first, we search for the exact name of the GLES userspace
// driver in both locations.
// i.e.:
//     libGLES.so, or:
//     libEGL.so, libGLESv1_CM.so, libGLESv2.so

for (size_t i=0 ; i<NELEM(searchPaths) ; i++) {
    if (find(result, pattern, searchPaths[i], true)) {
        return result;
    }
}

// for compatibility with the old "egl.cfg" naming convention
// we look for files that match:
//     libGLES_*.so, or:
//     libEGL_*.so, libGLESv1_CM_*.so, libGLESv2_*.so

pattern.append("_");
for (size_t i=0 ; i<NELEM(searchPaths) ; i++) {
    if (find(result, pattern, searchPaths[i], false)) {
        return result;
    }
}

// we didn't find the driver. gah.
result.clear();
return result;
}

private:

```



```

static bool find(String8& result,
    const String8& pattern, const char* const search, bool exact) {
    if (exact) {
        String8 absolutePath;
        absolutePath.appendFormat("%s/%s.so", search, pattern.string());
        if (!access(absolutePath.string(), R_OK)) {
            result = absolutePath;
            return true;
        }
        return false;
    }

    DIR* d = opendir(search);
    if (d != NULL) {
        struct dirent cur;
        struct dirent* e;
        while (readdir_r(d, &cur, &e) == 0 && e) {
            if (e->d_type == DT_DIR) {
                continue;
            }
            if (!strcmp(e->d_name, "libGLES_android.so")) {
                // always skip the software renderer
                continue;
            }
            if (strstr(e->d_name, pattern.string()) == e->d_name) {
                if (!strcmp(e->d_name + strlen(e->d_name) - 3, ".so")) {
                    result.clear();
                    result.appendFormat("%s/%s", search, e->d_name);
                    closedir(d);
                    return true;
                }
            }
        }
        closedir(d);
    }
    return false;
}
};

```

```

String8 absolutePath = MatchFile::find(kind);
if (absolutePath.isEmpty()) {
    // this happens often, we don't want to log an error
    return 0;
}
const char* const driver_absolute_path = absolutePath.string();

void* dso = dlopen(driver_absolute_path, RTLD_NOW | RTLD_LOCAL);
if (dso == 0) {
    const char* err = dlerror();
    ALOGE("load_driver(%s): %s", driver_absolute_path, err?err:"unknown");
    return 0;
}

if (mask & EGL) {

```

```

        ALOGD("EGL loaded %s", driver_absolute_path);
        getProcAddress = (getProcAddressType)dlsym(dso, "eglGetProcAddress");

        ALOGE_IF(!getProcAddress,
            "can't find eglGetProcAddress() in %s", driver_absolute_path);

        egl_t* egl = &cnx->egl;
        __eglMustCastToProperFunctionPointerType* curr =
            (__eglMustCastToProperFunctionPointerType*)egl;
        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++;
        }
    }

    if (mask & GLESv1_CM) {
        ALOGD("GLESv1_CM loaded %s", driver_absolute_path);
        init_api(dso, gl_names,
            (__eglMustCastToProperFunctionPointerType*)
                &cnx->hooks[egl_connection_t::GLESv1_INDEX]->gl,
            getProcAddress);
    }

    if (mask & GLESv2) {
        ALOGD("GLESv2 loaded %s", driver_absolute_path);
        init_api(dso, gl_names,
            (__eglMustCastToProperFunctionPointerType*)
                &cnx->hooks[egl_connection_t::GLESv2_INDEX]->gl,
            getProcAddress);
    }

    return dso;
}

```

第一步，找到驱动库文件的路径:

```

class MatchFile {
public:
    static String8 find(const char* kind) {
        String8 result;
        int emulationStatus = checkGlesEmulationStatus();
        switch (emulationStatus) {
            case 0:
                #if defined(__LP64__)

```

```

        result.setTo("/system/lib64/egl/libGLES_android.so");
#else
        result.setTo("/system/lib/egl/libGLES_android.so");
#endif

        return result;
    case 1:
        // Use host-side OpenGL through the "emulation" library
#ifdef __LP64__
        result.appendFormat("/system/lib64/egl/lib%s_emulation.so",
kind);
#else
        result.appendFormat("/system/lib/egl/lib%s_emulation.so",
kind);
#endif

        return result;
    default:
        // Not in emulator, or use other guest-side implementation
        break;
}

String8 pattern;
pattern.appendFormat("lib%s", kind);
const char* const searchPaths[] = {
#ifdef __LP64__
    "/vendor/lib64/egl",
    "/system/lib64/egl"
#else
    "/vendor/lib/egl",
    "/system/lib/egl"
#endif
};

// first, we search for the exact name of the GLES userspace
// driver in both locations.
// i.e.:
//     libGLES.so, or:
//     libEGL.so, libGLESv1_CM.so, libGLESv2.so

for (size_t i=0 ; i<NELEM(searchPaths) ; i++) {
    if (find(result, pattern, searchPaths[i], true)) {
        return result;
    }
}

// for compatibility with the old "egl.cfg" naming convention
// we look for files that match:
//     libGLES_*.so, or:
//     libEGL_*.so, libGLESv1_CM_*.so, libGLESv2_*.so

pattern.append("_");
for (size_t i=0 ; i<NELEM(searchPaths) ; i++) {
    if (find(result, pattern, searchPaths[i], false)) {
        return result;
    }
}
}

```

```

        // we didn't find the driver. gah.
        result.clear();
        return result;
    }

private:
    static bool find(String8& result,
        const String8& pattern, const char* const search, bool exact) {
        if (exact) {
            String8 absolutePath;
            absolutePath.appendFormat("%s/%s.so", search, pattern.string());
            if (!access(absolutePath.string(), R_OK)) {
                result = absolutePath;
                return true;
            }
            return false;
        }

        DIR* d = opendir(search);
        if (d != NULL) {
            struct dirent cur;
            struct dirent* e;
            while (readdir_r(d, &cur, &e) == 0 && e) {
                if (e->d_type == DT_DIR) {
                    continue;
                }
                if (!strcmp(e->d_name, "libGLES_android.so")) {
                    // always skip the software renderer
                    continue;
                }
                if (strstr(e->d_name, pattern.string()) == e->d_name) {
                    if (!strcmp(e->d_name + strlen(e->d_name) - 3, ".so")) {
                        result.clear();
                        result.appendFormat("%s/%s", search, e->d_name);
                        closedir(d);
                        return true;
                    }
                }
            }
            closedir(d);
        }
        return false;
    }
};

String8 absolutePath = MatchFile::find(kind);
if (absolutePath.isEmpty()) {
    // this happens often, we don't want to log an error
    return 0;
}

```

对于使用模拟器即使用 GLES 软件渲染模拟的情况，EGL 和 OpenGL ES 库对应为
/system/lib64/egl/libGLES_android.so 或 /system/lib/egl/libGLES_android.so。

如果使用物理设备，那么对于特定图形驱动库文件，EGL 库文件或 OpenGL ES 库文件查找则按照如下的顺序进行加载：

- ① `/vendor/lib64/egl` 或 `/vendor/lib/egl` 目录下文件名符合 `lib%s.so` 模式的库文件，例如 `/vendor/lib64/egl/libGLES.so`。
- ② `/system/lib64/egl` 或 `/system/lib/egl` 目录下文件名符合 `lib%s.so` 模式的库文件，例如 `/system/lib64/egl/libGLES.so`。
- ③ `/vendor/lib64/egl` 或 `/vendor/lib/egl` 目录下文件名符合 `lib%s_*.so` 模式的库文件，例如针对于 Pixel 设备的 `/vendor/lib64/egl/libEGL_adreno.so`。
- ④ `/system/lib64/egl` 或 `/system/lib/egl` 目录下文件名符合 `lib%s_*.so` 模式的库文件。

也就是说Android 会优先采用 `/vendor/` 下设备供应商提供的图形驱动库。

第二步，通过 `dlopen` 函数加载库文件：

```
const char* const driver_absolute_path = absolutePath.string();

void* dso = dlopen(driver_absolute_path, RTLD_NOW | RTLD_LOCAL);
if (dso == 0) {
    const char* err = dlerror();
    ALOGE("load_driver(%s): %s", driver_absolute_path, err?err:"unknown");
    return 0;
}
```

第三步，初始化函数表：

```
if (mask & EGL) {
    ALOGD("EGL loaded %s", driver_absolute_path);
    getProcAddress = (getProcAddressType)dlsym(dso, "eglGetProcAddress");

    ALOGE_IF(!getProcAddress,
        "can't find eglGetProcAddress() in %s", driver_absolute_path);

    egl_t* egl = &cnx->egl;
    __eglMustCastToProperFunctionPointerType* curr =
        (__eglMustCastToProperFunctionPointerType*)egl;
    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++;
    }
}

if (mask & GLESv1_CM) {
    ALOGD("GLESv1_CM loaded %s", driver_absolute_path);
```

```

        init_api(dso, gl_names,
                (__eglMustCastToProperFunctionPointerType*)
                &cnx->hooks[egl_connection_t::GLESv1_INDEX]->gl,
                getProcAddress);
    }

    if (mask & GLESv2) {
        ALOGD("GLESv2 loaded %s", driver_absolute_path);
        init_api(dso, gl_names,
                (__eglMustCastToProperFunctionPointerType*)
                &cnx->hooks[egl_connection_t::GLESv2_INDEX]->gl,
                getProcAddress);
    }
    return dso;
}

```

初始化函数表主要通过 `dlsym` 函数,根据函数名,一个个找到对应的地址,并赋值给函数指针来完成:

```

static void* load_wrapper(const char* path) {
    void* so = dlopen(path, RTLD_NOW | RTLD_LOCAL);
    ALOGE_IF(!so, "dlopen(\"%s\") failed: %s", path, dlerror());
    return so;
}

```

2.4 Android 加载 OpenGL ES总结

Android加载OpenGL ES流程图如下所示:

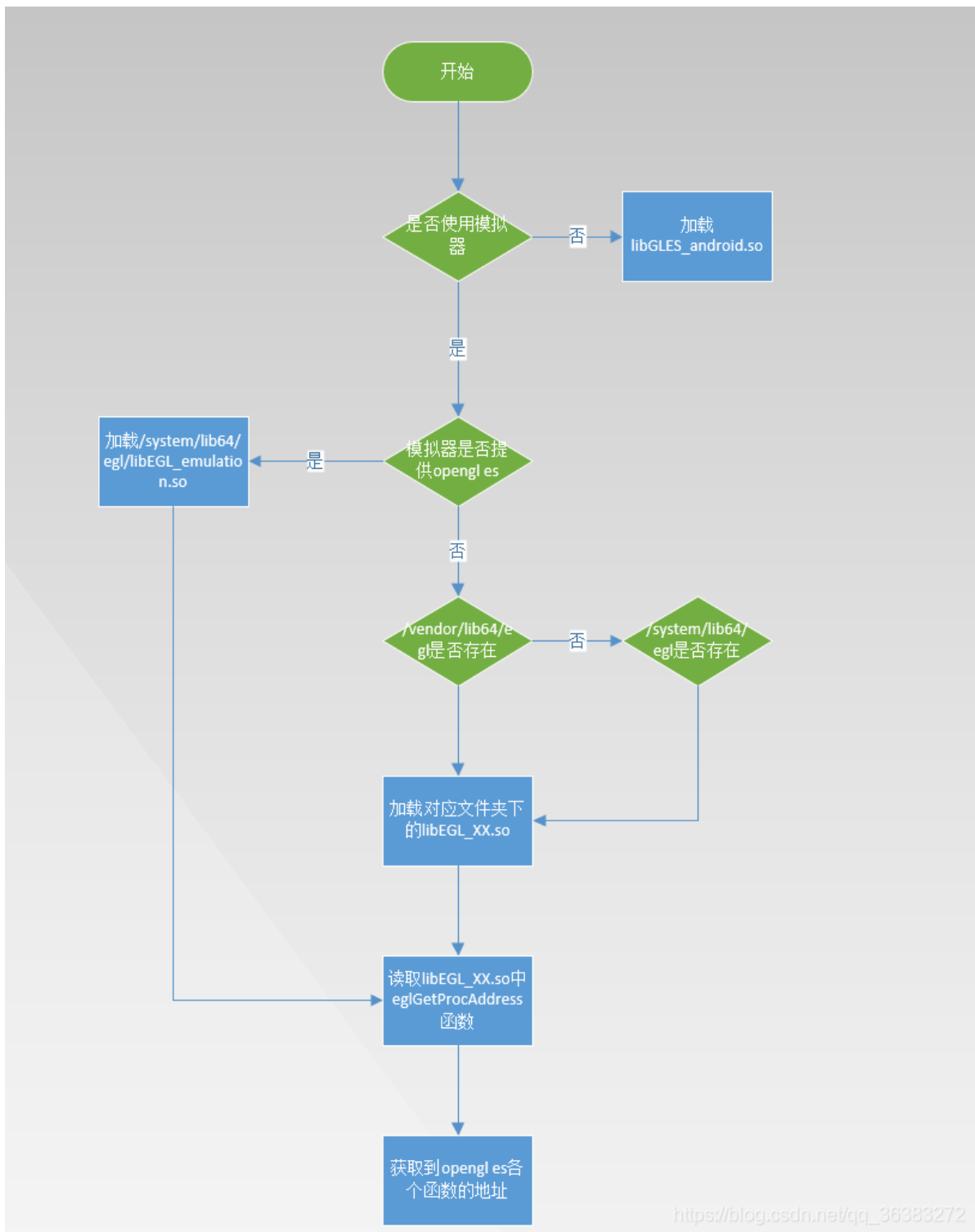


图2.1 加载OpenGL ES流程图

从Android加载OpenGL ES的流程可以看出，只要Anbox在Android的 `/system/lib64/egl` 下安装对应的OpenGL ES库，则可以使Android加载Anbox提供的渲染库了。

Anbox在 `anbox/android/opengl/system` 下有几个文件夹，查看 `egl/Android.mk` 文件。

可以看到 `$(call emugl-begin-shared-library,libEGL_emulation)`，这一行的意思是编译为动态库，库名为 `libEGL_emulation`。

Anbox中 `anbox/android` 文件夹是放在Android源码中，编译后存在Android镜像中，随着编译完成，Anbox在Android的 `/system/lib64/egl` 下就安装了三个动态库，分别为 `libEGL_emulation.so`、`libGLESv1_CM_emulation.so`和`libGLESv2_emulation.so`。也就是Anbox作为虚拟的硬件厂商给Android提供的OpenGL ES渲染库。

但Anbox实际渲染工作并不是上述三个库文件，这三个库文件作用为采集Android中APP渲染的OpenGL ES指令，并通过高速传输通道qemu-pipe传输将指令传至宿主机Anbox进程中，实际渲染工作由宿主机执行。

具体渲染的机制可以参考下图：

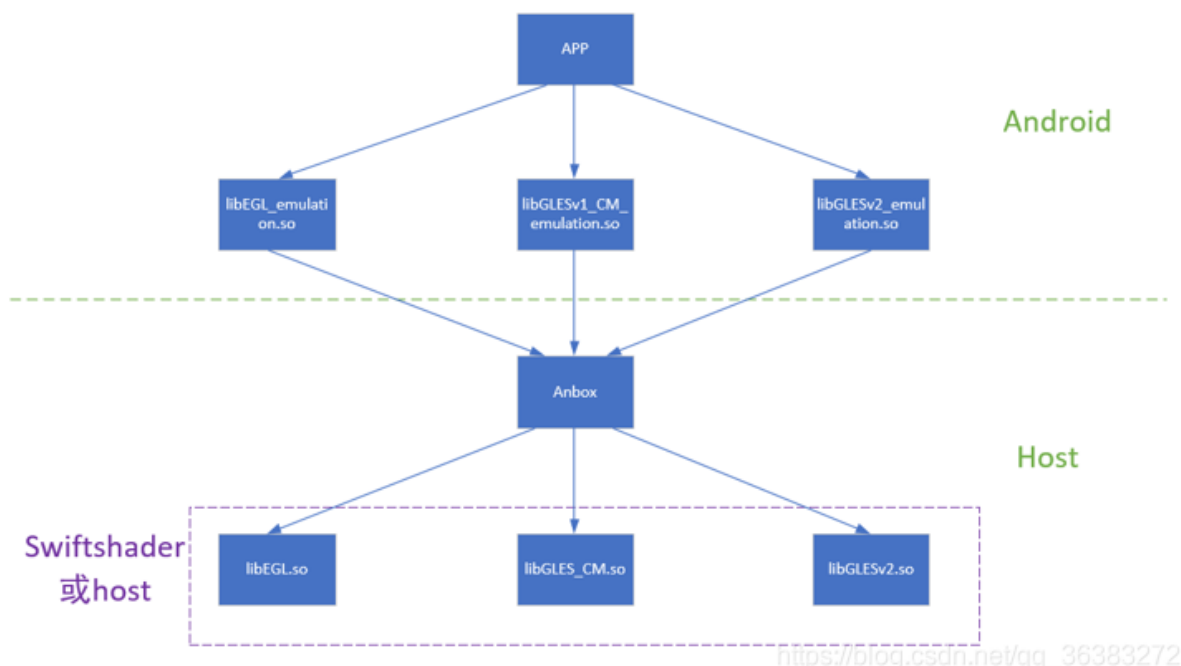


图2.2 Anbox渲染机制

其中： `swiftshader` 是谷歌提供的一种纯软件渲染的方式，其代码可在github进行搜索下载。

而 `host` 则是使用宿主机中默认的OpenGL ES库。即如果宿主机中显卡及显卡驱动支持OpenGL ES标准规范，则可以直接使用宿主机中的显卡进行渲染。但目前大部分显卡和驱动都不能直接支持OpenGL ES，因此Anbox会将Android的OpenGL ES命令翻译成Linux所能支持的OpenGL命令，再由Linux中的OpenGL进行渲染。

此部分代码在 `anbox/external/android-emugl` 和 `anbox/src/anbox/graphics` 中。

3. Anbox接收与渲染

到这里，Android的OpenGL ES命令已经准备完成，接下来只要将指令发送给宿主机进行转换和渲染即可。

首先从Anbox源码入手进行分析，这一部分涉及到了渲染环境的初始化。

3.1 初始化： `session manager`入口

`session manager` 的入口函数在 `anbox/src/anbox/cmds/session_manager.cpp`.首先是处理一系列的启动参数，对渲染来说，主要参数是 `software-rendering`

```
flag(cli::make_flag(cli::Name{"software-rendering"},
                      cli::Description{"Use software rendering instead of
hardware accelerated GL rendering"})
```

使用软件渲染或硬件渲染:

```
const auto should_force_software_rendering =
utils::get_env_value("ANBOX_FORCE_SOFTWARE_RENDERING", "false");
auto gl_driver = graphics::GLRenderersServer::Config::Driver::Host;
```



```

if (should_force_software_rendering == "true" || use_software_rendering_)
    gl_driver = graphics::GLRendererServer::Config::Driver::Software;

graphics::GLRendererServer::Config renderer_config {
    gl_driver,
    single_window_
};
auto gl_server = std::make_shared<graphics::GLRendererServer>
(renderer_config, window_manager);

platform->set_window_manager(window_manager);
platform->set_renderer(gl_server->renderer());
window_manager->setup();

```

根据启动参数，决定使用软件渲染或硬件渲染，然后生成了一个 `GLRendererServer` 的共享指针，`GLRendererServer` 函数信息位于 `anbox/src/anbox/graphics/gl_renderer_server.cpp` 中，源码为：

```

GLRendererServer::GLRendererServer(const Config &config, const
std::shared_ptr<wm::Manager> &wm)
    : renderer_(std::make_shared<::Renderer>()) {

    std::shared_ptr<LayerComposer::Strategy> composer_strategy;
    if (config.single_window)
        composer_strategy = std::make_shared<SingleWindowComposerStrategy>(wm);
    else
        composer_strategy = std::make_shared<MultiWindowComposerStrategy>(wm);

    composer_ = std::make_shared<LayerComposer>(renderer_, composer_strategy);

    auto gl_libs = emugl::default_gl_libraries();
    if (config.driver == Config::Driver::Software) {
        auto swiftshader_path = fs::path(utils::get_env_value("SWIFTSHADER_PATH"));
        const auto snap_path = utils::get_env_value("SNAP");
        if (!snap_path.empty())
            swiftshader_path = fs::path(snap_path) / "lib" / "anbox" / "swiftshader";
        if (!fs::exists(swiftshader_path))
            throw std::runtime_error("Software rendering is enabled, but SwiftShader
library directory is not found.");

        gl_libs = std::vector<emugl::GLLibrary>{
            {emugl::GLLibrary::Type::EGL, (swiftshader_path / "libEGL.so").string()},
            {emugl::GLLibrary::Type::GLSV1, (swiftshader_path /
"libGLES_CM.so").string()},
            {emugl::GLLibrary::Type::GLSV2, (swiftshader_path /
"libGLESv2.so").string()},
        };
    }
    emugl_logger_struct log_funcs;
    log_funcs.coarse = logger_write;
    log_funcs.fine = logger_write;

    if (!emugl::initialize(gl_libs, &log_funcs, nullptr))
        BOOST_THROW_EXCEPTION(std::runtime_error("Failed to initialize OpenGL
renderer"));
}

```

```

renderer_->initialize(0);

registerRenderer(renderer_);
registerLayerComposer(composer_);
}

```

根据源码可知，先根据使用的窗口策略生成对应的图层合成策略智能指针，LayerComposer这个模块里面就是把不同应用的图层合成到同一个画布上，即虚拟一个Android的Surfaceflinger。然后根据软件渲染还是硬件渲染读取对应的库函数，以软件渲染为例，Anbox先去读取swiftshader的环境变量，如果存在，则将swiftshader库的路径设置为环境变量读取到的路径；然后Anbox读取SNAP的环境变量，也就是说，如果Anbox是使用snap打包启动的，则直接去对应的路径下读取swiftshader的库。如果读取到了swiftshader库的路径，则在对应路径下读取libEGL.so、libGLES_CM.so和libGLSv2.so。即OpenGL ES对应的三个库文件。

接着初始化这三个库文件，进入 `anbox/src/anbox/graphics/emugl/RenderApi.cpp`：

```

bool initialize(const std::vector<GLLibrary> &libs, emugl_logger_struct
*log_funcs, logger_t crash_func) {
    set_emugl_crash_reporter(crash_func);
    if (log_funcs) {
        set_emugl_logger(log_funcs->coarse);
        set_emugl_cxt_logger(log_funcs->fine);
    }

    for (const auto &lib : libs) {
        const auto path = lib.path.c_str();
        switch (lib.type) {
            case GLLibrary::Type::EGL:
                if (!init_egl_dispatch(path))
                    return false;
                break;
            case GLLibrary::Type::GLSv1:
                if (!gles1_dispatch_init(path, &s_gles1))
                    return false;
                break;
            case GLLibrary::Type::GLSv2:
                if (!gles2_dispatch_init(path, &s_gles2))
                    return false;
                break;
            default:
                break;
        }
    }
}

```

打开对应路径，将egl需要用到的函数全部通过 `findsymbol` 或 `eglGetProcAddress` 的方式得到函数地址，方便后续直接调用。这样即可得到了 OpenGL ES 所有函数地址。

回到 `anbox/src/anbox/graphics/gl_renderer_server.cpp`，初始化OpenGL函数后，会去初始化渲染器，来到了 `anbox/src/anbox/graphics/emugl/Renderer.cpp` 的 `initialize` 函数，其实就是在display :0上初始化OpenGL ES相关的环境。执行完相关函数后 `gl_renderer_server` 的初始化就完成了，回到 `anbox/src/anbox/cmds/session_manager.cpp`，下一步 `window_manager->setup()` 创建本地窗口，其实就是Android窗口，所有Android的画面都会显示在这个窗口上。以单窗口为例，来到 `anbox/src/anbox/wm/single_window_manager.cpp`

```

void SinglewindowManager::setup() {
    if (auto p = platform_.lock()) {
        window_ = p->create_window(0, window_size_, "Anbox - Android in a Box");
        if (!window_->attach())
            WARNING("Failed to attach window to renderer");
    } else {
        throw std::runtime_error("Can't create window as we don't have a platform
abstraction");
    }
}

```

先是创建了一个 `window` 的对象，然后调用这个对象的 `attach` 函数，来到 `anbox/src/anbox/wm/window.cpp`:

```

bool window::attach() {
    if (!renderer_)
        return false;
    attached_ = renderer_->createNativeWindow(native_handle());
    return attached_;
}

```

可以看到，这里调用的 `renderer` 创建本地窗口。来到 `anbox/src/anbox/graphics/emugl/Renderer.cpp`

```

RendererWindow *Renderer::createNativeWindow(
    EGLNativeWindowType native_window) {
    m_lock.lock();

    auto window = new RendererWindow;
    window->native_window = native_window;
    window->surface = s_egl.eglCreateWindowSurface(
        m_eglDisplay, m_eglConfig, window->native_window, nullptr);
    if (window->surface == EGL_NO_SURFACE) {
        delete window;
        m_lock.unlock();
        return nullptr;
    }

    if (!bindwindow_locked(window)) {
        s_egl.eglDestroySurface(m_eglDisplay, window->surface);
        delete window;
        m_lock.unlock();
        return nullptr;
    }

    s_gles2.glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
        GL_STENCIL_BUFFER_BIT);
    s_egl.eglSwapBuffers(m_eglDisplay, window->surface);

    unbind_locked();

    m_nativewindows.insert({native_window, window});
}

```

```

m_lock.unlock();

return window;
}

```

3.2 指令传输与渲染

3.2.1 宿主机端

上文已经提到，Anbox的入口函数在 `session manage` 中。代码位置 `anbox/src/anbox/cmds/session_manager.cpp`：

```

auto qemu_pipe_connector =
    std::make_shared<network::PublishedSocketConnector>(
        utils::string_format("%s/qemu_pipe", socket_path), rt,
        std::make_shared<qemu::PipeConnectionCreator>(gl_server->renderer(),
rt));

```

Anbox建立了一个名字为`qemu_pipe`的unix socket服务端,并等待Android端连接。`PublishedSocketConnector` 函数在 `anbox/src/anbox/network/published_socket_connector.cpp` 中，构造函数直接调用 `start_accept` 函数。

```

void PublishedSocketConnector::start_accept() {
    auto socket = std::make_shared<boost::asio::local::stream_protocol::socket>
(runtime->service());

    acceptor_.async_accept(*socket,
                           [this, socket](boost::system::error_code const& err) {
    on_new_connection(socket, err);
                           });
}

```

当有客户端连接时则调用 `on_new_connection` 函数：

```

void
PublishedSocketConnector::on_new_connection
(std::shared_ptr<boost::asio::local::stream_protocol::socket> const& socket,
boost::system::error_code const& err) {
    if (!err)
        connection_creator_->create_connection_for(socket);

    if (err.value() == boost::asio::error::operation_aborted)
        return;
    start_accept();
}

```

这个函数的先是调用 `connection_creator` 中的 `create_connection_for` 函数，然后继续监听客户端连接。

这个 `connection_creator` 就是 `session manage` 中传进来的 `qemu::PipeConnectionCreator`。查看 `anbox/src/anbox/qemu/pipe_connection_creator.cpp` 的 `create_connection_for` 函数

```

void PipeConnectionCreator::create_connection_for(
    std::shared_ptr<boost::asio::local::stream_protocol::socket> const
        &socket) {
    auto const messenger = std::make_shared<network::LocalSocketMessenger>
(socket);
    const auto type = identify_client(messenger);
    auto const processor = create_processor(type, messenger);
    if (!processor)
        BOOST_THROW_EXCEPTION(std::runtime_error("Unhandled client type"));

    auto const &connection = std::make_shared<network::SocketConnection>(
        messenger, messenger, next_id(), connections_, processor);
    connection->set_name(client_type_to_string(type));
    connections_->add(connection);
    connection->read_next_message();
}

```

先是创建了一个 `LocalSocketMessenger`，用来与客户端通信。然后通过函数 `identify_client` 判断客户端的类型，对于渲染来说，类型就是 `pipe:opengles`。接着根据客户端类型创建对应的 `processor`，最后再创建对应的 `connection`。创建完成后读取客户端消息。在创建 `processor` 时，会创建处理 OpenGL ES 指令的线程，在

`anbox/src/anbox/graphics/opengles_message_processor.cpp`:

```

OpenGLMessageProcessor::OpenGLMessageProcessor(
    const std::shared_ptr<Renderer> &renderer,
    const std::shared_ptr<network::SocketMessenger> &messenger)
    : messenger_(messenger),
      stream_(std::make_shared<BufferedIOStream>(messenger_)) {
    // We have to read the client flags first before we can continue
    // processing the actual commands
    unsigned int client_flags = 0;
    auto err = messenger_->receive_msg(
        boost::asio::buffer(&client_flags, sizeof(unsigned int)));
    if (err) ERROR("%s", err.message());

    render_thread_.reset(RenderThread::create(renderer, stream_.get(),
std::ref(global_lock)));
    if (!render_thread_->start())
        BOOST_THROW_EXCEPTION(
            std::runtime_error("Failed to start renderer thread"));
}

```

这里会创建一个线程 `render_thread` 专门来处理 OpenGL ES 的指令:

```

intptr_t RenderThread::main() {
    RenderThreadInfo threadInfo;
    ChecksumCalculatorThreadInfo threadChecksumInfo;

    threadInfo.m_g1Dec.initGL(gles1_dispatch_get_proc_func, NULL);
    threadInfo.m_g12Dec.initGL(gles2_dispatch_get_proc_func, NULL);
    initRenderControlContext(&threadInfo.m_rcDec);

    ReadBuffer readBuf(STREAM_BUFFER_SIZE);
}

```

```

while (true) {
    int stat = readBuf.getData(m_stream);
    if (stat <= 0)
        break;

    bool progress;
    do {
        progress = false;

        std::unique_lock<std::mutex> l(m_lock);

        size_t last =
            threadInfo.m_glDec.decode(readBuf.buf(), readBuf.validData(),
m_stream);
        if (last > 0) {
            progress = true;
            readBuf.consume(last);
        }

        last =
            threadInfo.m_gl2Dec.decode(readBuf.buf(), readBuf.validData(),
m_stream);
        if (last > 0) {
            progress = true;
            readBuf.consume(last);
        }

        last = threadInfo.m_rcDec.decode(readBuf.buf(), readBuf.validData(),
m_stream);
        if (last > 0) {
            readBuf.consume(last);
            progress = true;
        }

    } while (progress);

}

threadInfo.m_gl2Dec.freeShader();
threadInfo.m_gl2Dec.freeProgram();

// Release references to the current thread's context/surfaces if any
renderer_>bindContext(0, 0, 0);
if (threadInfo.currContext || threadInfo.currDrawSurf ||
threadInfo.currReadSurf)
    ERROR("RenderThread exiting with current context/surfaces");

renderer_>drainWindowSurface();
renderer_>drainRenderContext();

return 0;
}

```

```

intptr_t RenderThread::main() {
    RenderThreadInfo threadInfo;
    ChecksumCalculatorThreadInfo threadChecksumInfo;

    threadInfo.m_g1Dec.initGL(gles1_dispatch_get_proc_func, NULL);
    threadInfo.m_g12Dec.initGL(gles2_dispatch_get_proc_func, NULL);
    initRenderControlContext(&threadInfo.m_rcDec);

    ReadBuffer readBuf(STREAM_BUFFER_SIZE);

    while (true) {
        int stat = readBuf.getData(m_stream);
        if (stat <= 0)
            break;

        bool progress;
        do {
            progress = false;

            std::unique_lock<std::mutex> l(m_lock);

            size_t last =
                threadInfo.m_g1Dec.decode(readBuf.buf(), readBuf.validData(),
m_stream);
            if (last > 0) {
                progress = true;
                readBuf.consume(last);
            }

            last =
                threadInfo.m_g12Dec.decode(readBuf.buf(), readBuf.validData(),
m_stream);
            if (last > 0) {
                progress = true;
                readBuf.consume(last);
            }

            last = threadInfo.m_rcDec.decode(readBuf.buf(), readBuf.validData(),
m_stream);
            if (last > 0) {
                readBuf.consume(last);
                progress = true;
            }

        } while (progress);

    }

    threadInfo.m_g12Dec.freeShader();
    threadInfo.m_g12Dec.freeProgram();

    // Release references to the current thread's context/surfaces if any
    renderer->bindContext(0, 0, 0);
    if (threadInfo.currContext || threadInfo.currDrawSurf ||
threadInfo.currReadSurf)

```

```

        ERROR("RenderThread exiting with current context/surfaces");

    renderer_>drainWindowSurface();
    renderer_>drainRenderContext();

    return 0;
}

```

这个线程里，显示初始化三个解码器，分别是 `GLESV1Decoder`、`GLESV2Decoder` 和 `renderControl_decoder_context_t`，然后根据收到的socket客户端的信息，来分别解析这三种指令。

解码的相关的代码在 `anbox/external/android-emugl/host/libs`，此部分内容是Anbox从 `Android Emulator` 中复制而来，在此不再进行讨论。

3.2.2 Android端

Android端采集OpenGL ES相关的指令，并通过三个对应的编码器将指令传输出来，Anbox就可以实现将Android里的所有OpenGL ES指令在宿主机上执行，从而进行相应的渲染。

那么Android是如何采集OpenGL ES指令并通过编码器传输出来？在

`anbox/android/opengl/system/egl/Android.mk` 可以得知这个文件夹下的源文件会编译为库文件 `libEGL_emulation.so`：

```
$(call emugl-begin-shared-library,libEGL_emulation)
```

在前文已经介绍过了Android加载OpenGL ES的流程，所以只要将 `libEGL_emulation.so` 放在 `/system/lib64/egl` 下，Android就会自动去加载该库，并将其作为OpenGL ES的默认库文件，这样Android里所有的OpenGL ES调用都会经过该库。也就是说Android里所有的OpenGL ES相关的指令都可以被收集。

那么，指令又是如何传输出来的呢？在前文中提到，通过 `egl` 来调用 `opengl` 时首先要调用

`eglInitialize` 函数进行初始化，在 `anbox/android/opengl/system/egl/egl.cpp` 的中 `eglInitialize` 函数可以看到：

```

EGLBoolean eglInitialize(EGLDisplay dpy, EGLint *major, EGLint *minor)
{
    VALIDATE_DISPLAY(dpy, EGL_FALSE);

    if (!s_display.initialize(&s_eglIface)) {
        return EGL_FALSE;
    }
    if (major!=NULL)
        *major = s_display.getVersionMajor();
    if (minor!=NULL)
        *minor = s_display.getVersionMinor();
    return EGL_TRUE;
}

```

这个函数调用了 `s_display.initialize(&s_eglIface)`，我们来到

`anbox/android/opengl/system/egl/eglDisplay.cpp`的`initialize` 函数。这个函数首先加载了 `libGLESV1_CM_emulation.so` 和 `libGLESV2_emulation.so`。也就是 `OpenGL ES 1.0` 和 `OpenGL ES2.0` 的库。然后在调用 `get` 函数：


```
HostConnection *hcon = HostConnection::get();
```

在 `anbox/android/opengl/system/OpenGLSystemCommon/HostConnection.cpp` 中可以看到:

```
HostConnection *HostConnection::get()
{
    /* TODO: Make this configurable with a system property */
    const int useQemuPipe = USE_QEMU_PIPE;

    // Get thread info
    EGLThreadInfo *tinfo = getEGLThreadInfo();
    if (!tinfo) {
        return NULL;
    }

    if (tinfo->hostConn == NULL) {
        HostConnection *con = new HostConnection();
        if (NULL == con) {
            return NULL;
        }

        if (useQemuPipe) {
            QemuPipeStream *stream = new QemuPipeStream(STREAM_BUFFER_SIZE);
            if (!stream) {
                ALOGE("Failed to create QemuPipeStream for host
connection!!!\n");
                delete con;
                return NULL;
            }
            if (stream->connect() < 0) {
                ALOGE("Failed to connect to host (QemuPipeStream)!!!\n");
                delete stream;
                delete con;
                return NULL;
            }
            con->m_stream = stream;
        }
        else /* !useQemuPipe */
        {
            TcpStream *stream = new TcpStream(STREAM_BUFFER_SIZE);
            if (!stream) {
                ALOGE("Failed to create TcpStream for host connection!!!\n");
                delete con;
                return NULL;
            }

            if (stream->connect("10.0.2.2", STREAM_PORT_NUM) < 0) {
                ALOGE("Failed to connect to host (TcpStream)!!!\n");
                delete stream;
                delete con;
                return NULL;
            }
            con->m_stream = stream;
        }
    }
}
```

```

        // send zero 'clientFlags' to the host.
        unsigned int *pClientFlags =
            (unsigned int *)con->m_stream->allocBuffer(sizeof(unsigned
int));
        *pClientFlags = 0;
        con->m_stream->commitBuffer(sizeof(unsigned int));

        ALOGD("HostConnection::get() New Host Connection established %p, tid %d\n",
con, gettid());
        tinfo->hostConn = con;
    }
    return tinfo->hostConn;
}

```

可知Anbox用的是 `qemuPipe`，先创建 `QemuPipeStream`，再进行 `stream->connect()`。
查看 `anbox/android/opengl/system/OpenGLSystemCommon/QemuPipeStream.cpp`：

```

int QemuPipeStream::connect(void)
{
    m_sock = qemu_pipe_open("opengles");
    if (!valid()) return -1;
    return 0;
}

```

在connect的时候会以"opengles"为标志连接 `qemu_pipe`，也就与前文所讲到Anbox创建的 `unix socket` 中的 `pipe:opengles` 对应上了。这样Android端与宿主机端就连接上了，然后只需要通过这个通道将Android里的OpenGL ES指令传输到宿主机端，就可以实现Android内所有APP的渲染了。

3.2.3 宿主机渲染器

上文提到当Anbox获得Android的OpenGL ES指令后，会将该指令转换为协议流发送给理解协议格式的特殊渲染库或进程，流程如图所示：

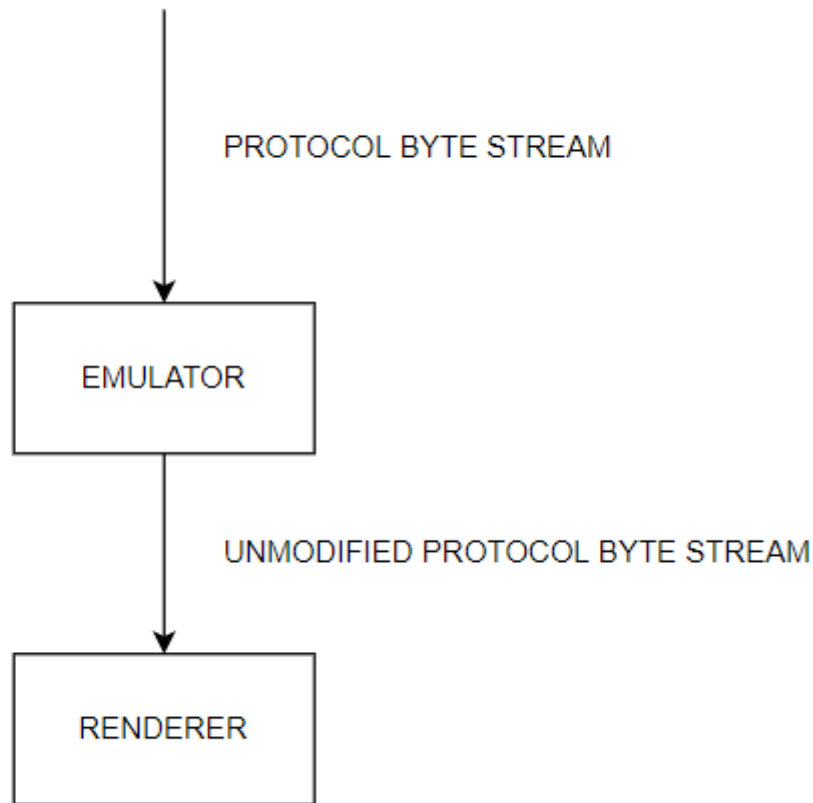


图3.1 渲染流程图

宿主渲染器库位于 `$ANDROID/external/qemu/android/android-emugl/host/libs/libOpenGlRender` 下，它提供了一个由 `$$ANDROID/external/qemu/android/android-emugl/host/libs/libOpenGlRender/render_api.h` 下（比如用于模拟器）的头文件描述的接口。

简而言之，渲染库负责以下内容：

- 提供一个虚拟的离屏视频 Surface 用于在运行时渲染所有的东西。它的维度必须通过在库初始化之后，紧接着调用 `initOpenGlRender()` 来固定。
- 提供一种方式在一个宿主应用程序的 UI 中显示虚拟的视频 Surface。这通过调用 `createOpenGlSubwindow()` 完成，它接收 window ID 或父 window 的句柄，一些显示维度和一个旋转角度作为参数。这允许 Surface 在显示时被放缩/旋转，甚至是在视频 surface 的维度没有改变时。
- 提供一种方式监听从客户系统进入的 EGL/GLES 命令。这通过给 `initOpenGlRender()` 提供一个所谓的“端口号”完成。

默认情况下，端口号对应一个渲染器将绑定和监听的本地 TCP 端口号。到该端口的每个新连接将对应创建一个新的客户系统与宿主系统间的连接，每个这样的连接对应客户系统中的一个不同的线程。

处于性能原因，监听 Unix sockets（在 Linux 或 OS X 上），或 Win32 命名管道（在 Windows 上）都是可能的。为了做到这一点，必须在库初始化（比如 `initLibrary()`）和构建（比如 `initOpenGlRender()`）之间调用 `setStreamType()`。

注意在这些模式中，端口号依然被用于区分多个模拟器实例。这些细节通常由模拟器代码处理。

注意更早的接口版本允许渲染器库的客户端提供它自己的 `IStream` 实现。然而，因为许多原因这不是很方便。如果它有意义，这也许可以再次做到，但现在的性能数字是相当好的。

4. OpenGL ES—翻译—OpenGL

当渲染器从协议流解码 EGL/GLES 命令后，会根据指令的类型派发给适当的翻译器库，流程如图所示：

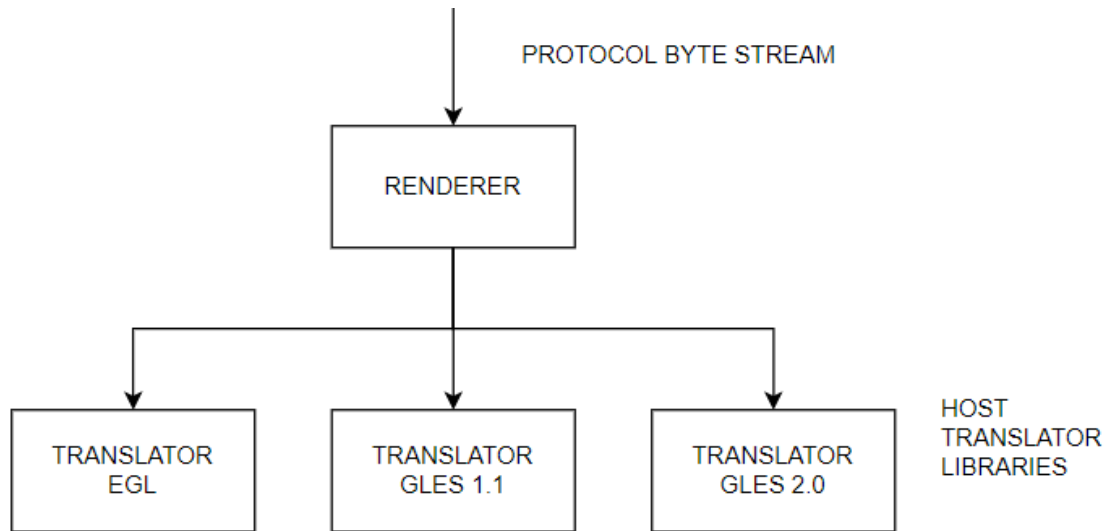


图4.1 指令翻译流

事实上，协议流是双向流动的，尽管大多数命令使得数据从客户系统传送到宿主机。但完整流程应该是：

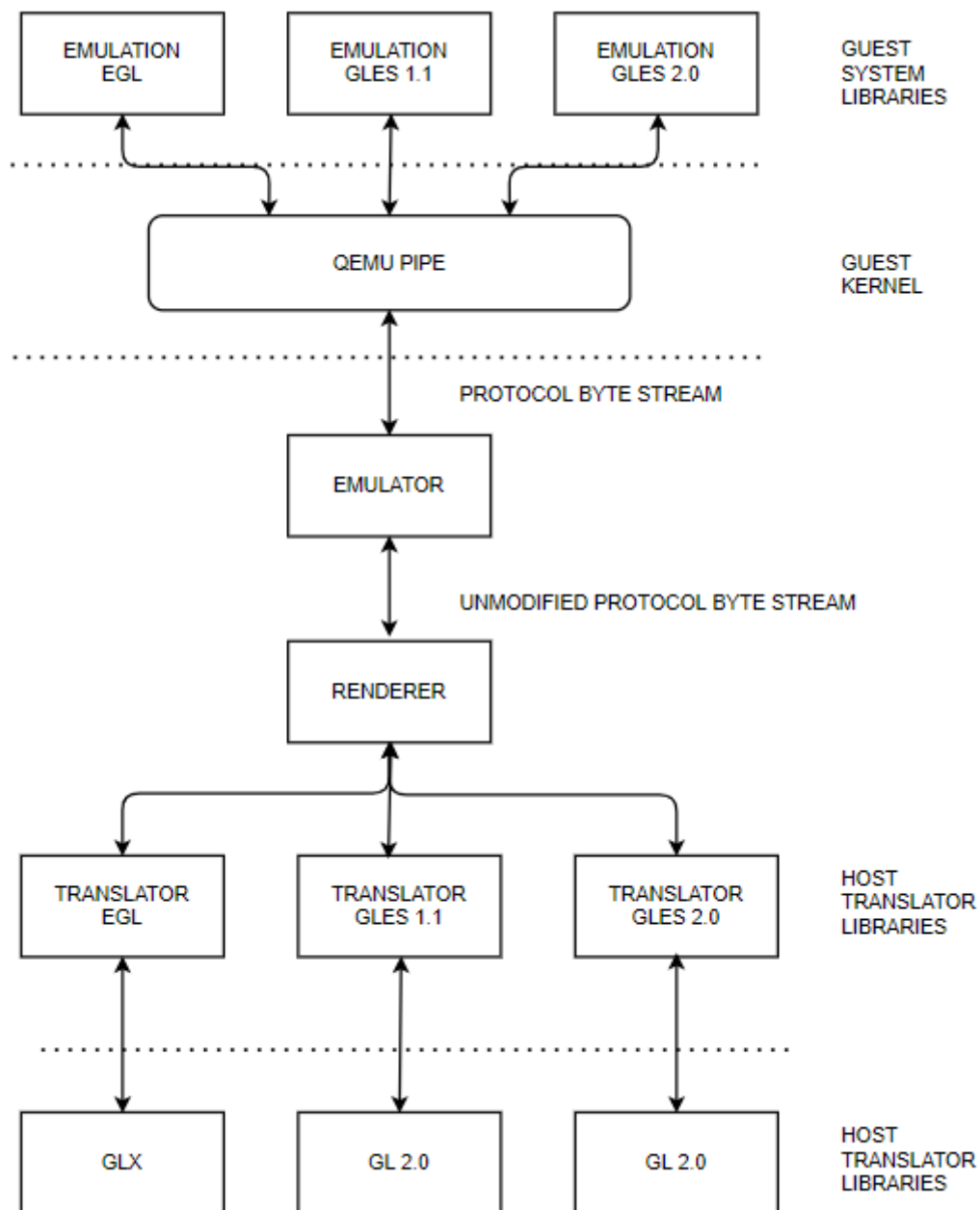


图4.2 翻译流完整流程

4.1 翻译器库

Anbox提供了三个宿主端的翻译器库：

```

libEGL_translator    -> EGL 1.2 翻译
libGLES_CM_translator -> GLES 1.1 翻译
libGLES_V2_translator -> GLES 2.0 翻译
  
```

这些库的源码位于 Android 源码树的下列路径下：

```

$ANDROID/external/qemu/android/android-emugl/host/libs/Translator/EGL
$ANDROID/external/qemu/android/android-emugl/host/libs/Translator/GLES_CM
$ANDROID/external/qemu/android/android-emugl/host/libs/Translator/GLES_V2
  
```

翻译器库也使用如下目录中定义的通用的程序：

当指令经过翻译程序翻译完成后，Anbox便可调用Linux下的OpenGL库进行图形渲染。

参考资料

1. android函数的调用过程,Android OpenGL库加载和调用过程 链接:(https://blog.csdn.net/weixin_39880895/article/details/117314150)
2. Android模拟器图形绘原理(二十二) 链接:(https://unbroken.blog.csdn.net/article/details/119487447?spm=1001.2101.3001.6661.1&utm_medium=distribute.pc_relevant_t0.none-task-blog-2%Edefault%ECTRLIST%Edefault-1-119487447-blog-107620008.pc_relevant_aa2&depth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-2%Edefault%ECTRLIST%Edefault-1-119487447-blog-107620008.pc_relevant_aa2&utm_relevant_index=1)
3. Android 硬件 OpenGL ES 模拟设计概述 链接:(<https://blog.csdn.net/tq08g2z/article/details/78004065>)
4. 掌握Android图像显示原理(中) 链接:(<https://blog.csdn.net/tyuiof/article/details/108675978>)
5. Android EGL 一、初始化配置 链接:(<https://blog.csdn.net/kongbaidepao/article/details/108550631>)
6. Android 图形驱动初始化 链接:(<https://blog.csdn.net/tq08g2z/article/details/77991008>)
7. Android SurfaceFlinger之OpenGL库加载过程 链接:(<https://www.cnblogs.com/ztguang/p/12644892.html>)
8. Anbox源码分析 (一) 链接: (https://blog.csdn.net/qg_36383272/article/details/105163579)
9. Anbox源码分析 (二) ——Anbox渲染原理 链接:(https://blog.csdn.net/qg_36383272/article/details/105680570)
10. Anbox源码分析 (三) ——Anbox渲染原理(源码分析) 链接:(https://blog.csdn.net/qg_36383272/article/details/105957455)
11. Anbox源码分析 (四) ——Anbox渲染原理(源码分析) 链接:(https://blog.csdn.net/qg_36383272/article/details/107919857)