第七章 Android Camera 框架

7.1 总体介绍

Android Camera 框架从整体上看是一个 client/service 的架构，有两个进程：一个是 client 进程，可以看成是 AP 端，主要包括 JAVA 代码与一些 native c/c++代码；另一个是 service 进程，属于服务端，是 native c/c++代码，主要负责和 linux kernel 中的 camera driver 交互，搜集 linux kernel 中 camera driver 传上来的数据，并交给显示系统(surface)显示。client 进程与 service 进程通过 Binder 机制通信，client 端通过调用 service 端的接口实现各个具体的功能。

需要注意的是真正的 preview 数据不会通过 Binder IPC 机制从 service 端复制到 client 端，但会通过回调函数与消息的机制将 preview 数据 buffer 的地址传到 client 端，最终可在 JAVA AP 中操作处理这个 preview 数据。

7.2 client 端

从 JAVA AP 的角度看 camera ap 就是调用 FrameWork 层的 android.hardware.camera 类来实现具体的功能。JAVA Ap 最终被打包成 APK。

FrameWork 层主要提供了 android.hardware.camera 类给应用层使用，这个类也是 JAVA 代码实现。Android.hardware.camera 类通过 JNI 调用 native 代码实现具体的功能。Android.hardware.camera 类中提供了如下的一个参数类给应用层使用：

```
public class Parameters {
       // Parameter keys to communicate with the camera driver。
       private static final String KEY_PREVIEW_SIZE = "preview-size";
       private static final String KEY_PREVIEW_FORMAT = "preview-format";
       ……
}
```

参数会以字典（map）的方式组织存储起来，关键字就是 Parameters 类中的这些静态字符串。这些参数最终会以形如“preview-size=640X480;preview-format=yuv422sp;……”格式的字符串传到 service 端。源代码位于：framework/base/core/java/android/hardware/camera.java

提供的接口示例：

- 获得一个 android.hardware.camera 类的实例

```
public static Camera open() {
       return new Camera();
}
```

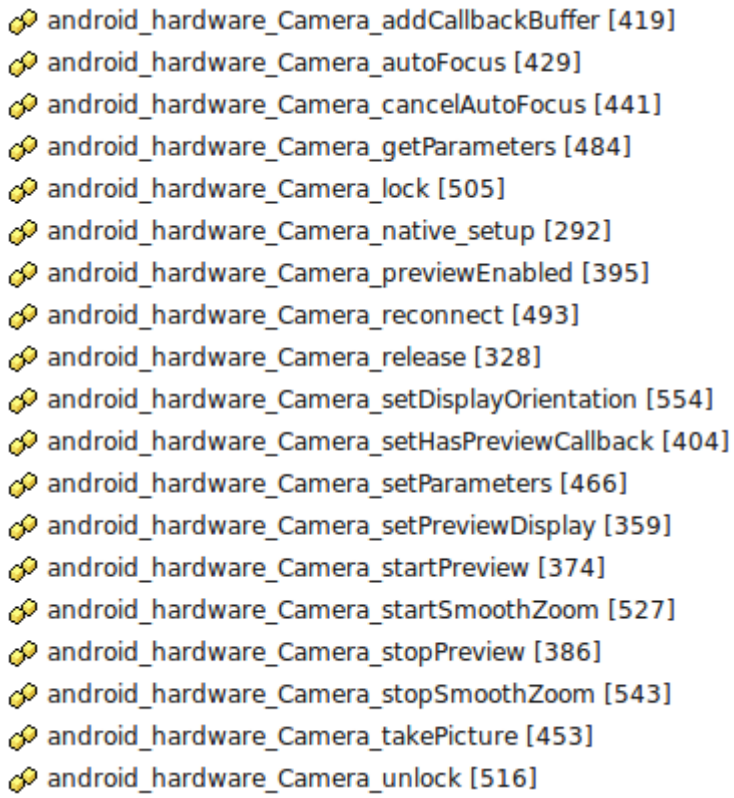
- 接口直接调用 native 代码（android_hardware_camera.cpp 中的代码）

```
public native final void startPreview();
```

```
public native final void stopPreview();
```

android.hardware.camera 类的 JNI 调用实现在 android_hardware_camera.cpp 文件中，源代码位置：framework/base/core/jni/android_hardware_camera.cpp（framework/base/core/jni/文件夹下的文件都被编译进 libandroid_runtime.so 公共库中。）

android_hardware_camera.cpp 文件中的 JNI 调用实现函数都如下图：



android_hardware_camera.cpp 文件中的 register_android_hardware_Camera(JNIEnv *env)函数会将上面的 native 函数注册到虚拟机中，以供 FrameWork 层的 JAVA 代码调用。这些 native 函数通过调用 libcamera_client.so 中的 Camera 类实现具体的功能。

核心的 libcamera_client.so 动态库源代码位于：frameworks/base/libs/camera/，实现了如下几个类：

- Camera---->Camera.cpp/Camera.h
- CameraParameters--->CameraParameters.cpp/CameraParameters.h
- Icamera--->ICamera.cpp/ICamera.h
- IcameraClient--->ICameraClient.cpp/ICameraClient.h
- IcameraService--->ICameraService.cpp/ICameraService.h

Icamera、IcameraClient、IcameraService 三个类是按照 Binder IPC 通信要求的框架实现的，用来与 service 端通信。

类 CameraParameters 接收 FrameWork 层的 android.hardware.camera::Parameters 类为参数，

解析与格式化所有的参数设置。

Camera 是一个很重要的类，它与 CameraService 端通过 Binder IPC 机制交互来实现具体功能。Camera 继承自 BnCameraClient，并最终继承自 ICameraClient。

Camera 类通过：

```
sp<IServiceManager> sm = defaultServiceManager();
sp<IBinder> binder = sm->getService(String16("media.camera"));
sp<ICameraService> mCameraService = interface_cast<ICameraService>(binder);
```

得到名字为“media.camera”的 CameraService。通过调用 CameraService 的接口 connect()返回得到 sp<ICamera> mCamera，并在 CameraService 端 new 一个 CameraService::Client 类 mClient。mClient 继承自 BnCamera，并最终继承自 ICamera。

之后 Camera 类通过这个 sp<ICamera> mCamera 对象调用函数就像直接调用 CameraService::Client 类 mClient 的函数。CameraService::Client 类实现具体的功能。

7.3 service 端

实现在动态库 libcameraservice.so 中，源代码位于：frameworks/base/camera/libcameraservice

Libcameraservice.so 中主要有下面两个类：

- Libcameraservice.so::CameraService 类，继承自 BnCameraService，并最终继承自 ICameraService
- Libcameraservice.so::CameraService::Client 类，继承自 BnCamera，并最终继承自 ICamera

CameraService::Client 类通过调用 Camera HAL 层来实现具体的功能。目前的 code 中只支持一个 CameraService::Client 实例。

Camera Service 在系统启动时 new 了一个实例，以“media.camera”为名字注册到 ServiceManager 中。在 init.rc 中有如下代码执行可执行文件/system/bin/mediaserver，启动多媒体服务进程。

```
service media /system/bin/mediaserver
```

Mediaserver 的 c 代码如下：

```
int main(int argc, char** argv)
{
    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm = defaultServiceManager();
    LOGI("ServiceManager: %p", sm。get());
    AudioFlinger::instantiate();
    MediaPlayerService::instantiate();
    CameraService::instantiate();
    AudioPolicyService::instantiate();
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
}
```

7.4 Camera HAL(硬件抽象层)

Libcameraservice.so::CameraService::Client 类调用 camera HAL 的代码实现具体功能，camera HAL 一般实现为一个动态库 libcamera.so（动态库名字可以改，只需要与 Android.mk 一致即可)。Android 只给了一个定义文件：/home/miracle/Work/android/android_src/froyo/frameworks/base/include/camera/CameraHardwareInterface.h

```
class CameraHardwareInterface :  public virtual RefBase {
public:
    virtual ~CameraHardwareInterface() { }
    virtual sp<IMemoryHeap> getPreviewHeap() const = 0;
    virtual sp<IMemoryHeap> getRawHeap() const = 0;
    virtual void setCallbacks(notify_callback notify_cb, data_callback data_cb,
                  data_callback_timestamp data_cb_timestamp, void* user) = 0;
    virtual void     enableMsgType(int32_t msgType) = 0;
    virtual void     disableMsgType(int32_t msgType) = 0;
    virtual bool     msgTypeEnabled(int32_t msgType) = 0;
    virtual status_t  startPreview() = 0;
    virtual bool      useOverlay() {return false;}
    virtual status_t setOverlay(const sp<Overlay> &overlay) {return BAD_VALUE;}
    virtual void       stopPreview() = 0;
    virtual bool       previewEnabled() = 0;
    virtual status_t    startRecording() = 0;
    virtual bool       recordingEnabled() = 0;
    virtual status_t    autoFocus() = 0;
    virtual status_t    cancelAutoFocus() = 0;
    virtual status_t    takePicture() = 0;
    virtual status_t    cancelPicture() = 0;
    virtual status_t    setParameters(const CameraParameters& params) = 0;
    virtual CameraParameters  getParameters() const = 0;
    virtual status_t sendCommand(int32_t cmd, int32_t arg1, int32_t arg2) = 0;
    virtual void release() = 0;
    virtual status_t dump(int fd, const Vector<String16>& args) const = 0;
};
extern "C" sp<CameraHardwareInterface> openCameraHardware();
};  // namespace android
```

可以看到在 JAVA Ap 中的功能调用最终会调用到 HAL 层这里，Camera HAL 层的实现是主要的工作，它一般通过 V4L2 command 从 linux kernel 中的 camera driver 得到 preview 数据。然后交给 surface(overlay)显示或者保存为文件。在 HAL 层需要打开对应的设备文件，并通过 ioctrl 访问 camera driver。Android 通过这个 HAL 层来保证底层硬件（驱动）改变，只需修改对应的 HAL 层代码，FrameWork 层与 JAVA Ap 的都不用改变。

7.5 Preview 数据流程

Android 框架中 preview 数据的显示过程如下：

1、 打开内核设备文件。CameraHardwareInterface.h 中定义的 openCameraHardware()打开 linux kernel 中的 camera driver 的设备文件（如/dev/video0），创建初始化一些相关的类的实例。
2、 设置摄像头的工作参数。CameraHardwareInterface.h 中定义的 setParameters()函数，在这一步可以通过参数告诉 camera HAL 使用哪一个硬件摄像头，以及它工作的参数（size，

format 等），并在 HAL 层分配存储 preview 数据的 buffers（如果 buffers 是在 linux kernel 中的 camera driver 中分配的，在这一步也会拿到这些 buffers mmap 后的地址指针）。

3、 设置显示目标。需在 JAVA APP 中创建一个 surface 然后传递到 CameraService 中。会调用到 libcameraservice.so 中的 setPreviewDisplay(const sp<ISurface>& surface)函数中。在这里分两种情况考虑：一种是不使用 overlay；一种是使用 overlay 显示。如果不使用 overlay 那设置显示目标最后就在 libcameraservice.so 中，不会进 Camera HAL 动态库。并将上一步拿到的 preview 数据 buffers 地址注册到 surface 中。 如果使用 overlay 那在 libcameraservice.so 中会通过传进来的 Isurface 创建 Overlay 类的实例，然后调用 CameraHardwareInterface.h 中定义的 setOverlay()设置到 Camera HAL 动态库中。

4、 开始 preview 工作。最终调用到 CameraHardwareInterface.h 中定义的 startPreview()函数。如果不使用 overlay，Camera HAL 得到 linux kernel 中的 preview 数据后回调通知到 libcameraservice.so 中。在 libcameraservice.so 中会使用上一步的 surface 进行显示。如果使用 overlay，Camera HAL 得到 linux kernel 中的 preview 数据后直接交给 Overlay 对象，然后有 Overlay HAL 去显示。

7.6 模拟器中的虚拟 camera

如果没有 camera 硬件，不实现真正的 Camera HAL 动态库，可以使用虚拟 camera。源代码位于：

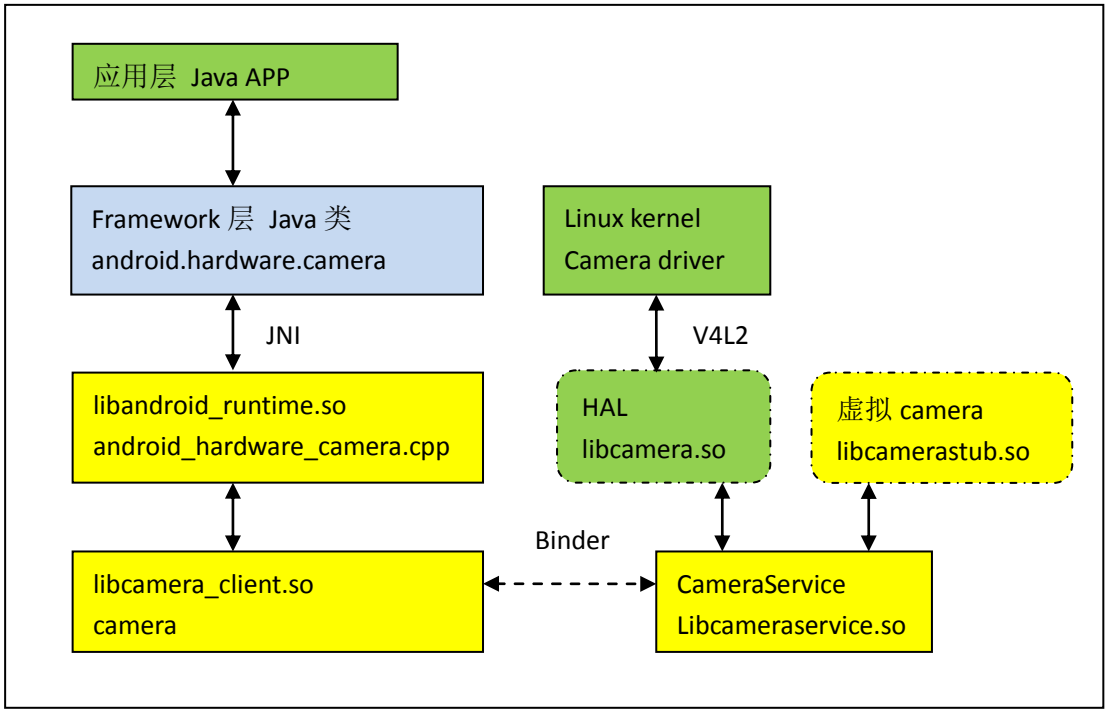
frameworks/base/camera/libcameraservice/FakeCamera.cpp

frameworks/base/camera/libcameraservice/CameraHardwareStub.cpp

FakeCamera.cpp 文件提供虚拟的 preview 数据。CameraHardwareStub.cpp 文件中实现了 camera HAL(硬件抽象层)的功能。当宏 USE_CAMERA_STUB 为 true 时可以使用这个虚拟的 camera。

```
ifeq ($(USE_CAMERA_STUB), true)
    LOCAL_STATIC_LIBRARIES += libcamerastub //虚拟的 camera
#if want show LOGV message, should use follow define。 add 0929
#LOCAL_CFLAGS += -DLOG_NDEBUG=0
    LOCAL_CFLAGS += -include CameraHardwareStub。h
else
    LOCAL_SHARED_LIBRARIES += libcamera //真正的 camera HAL 库
endif
```

7.7 框架图



7.8 Overlay 简单介绍

overlay 一般用在 camera preview，视频播放等需要高帧率的地方，还有可能 UI 界面设计的需求，如 map 地图查看软件需两层显示信息。overlay 需要硬件与驱动的支持。Overlay 没有 java 层的 code，也就没有 JNI 调用。一般都在 native 中使用。

Overlay 的使用方法

1. 头文件

overlay object 对外的接口

```
#include <ui/Overlay。h>
```

下面三个用于从 HAL 得到 overlay object

```
#include <surfaceflinger/Surface。h>
#include <surfaceflinger/ISurface。h>
#include <surfaceflinger/SurfaceComposerClient。h>
```

2. 相关动态库文件

```
libui.so
libsurfaceflinger_client.so
```

3. 调用步骤

- 创建 surfaceflinger 的客户端

```
sp<SurfaceComposerClient> client = new SurfaceComposerClient();
```

- 创建推模式 surface

```
sp<Surface> surface = client->createSurface(getpid(), 0, 320, 240,
PIXEL_FORMAT_UNKNOWN, IsurfaceComposer::ePushBuffers);
```

- 获得 surface 接口

```
sp<ISurface> isurface = surface->getISurface();
```

- 获得 overlay 设备

```
sp<OverlayRef> ref = isurface->createOverlay(320, 240, PIXEL_FORMAT_RGB_565);
```

这里会通过调用 overlay hal 层的 createoverlay()打开对应的设备文件。

- 创建 overlay 对象

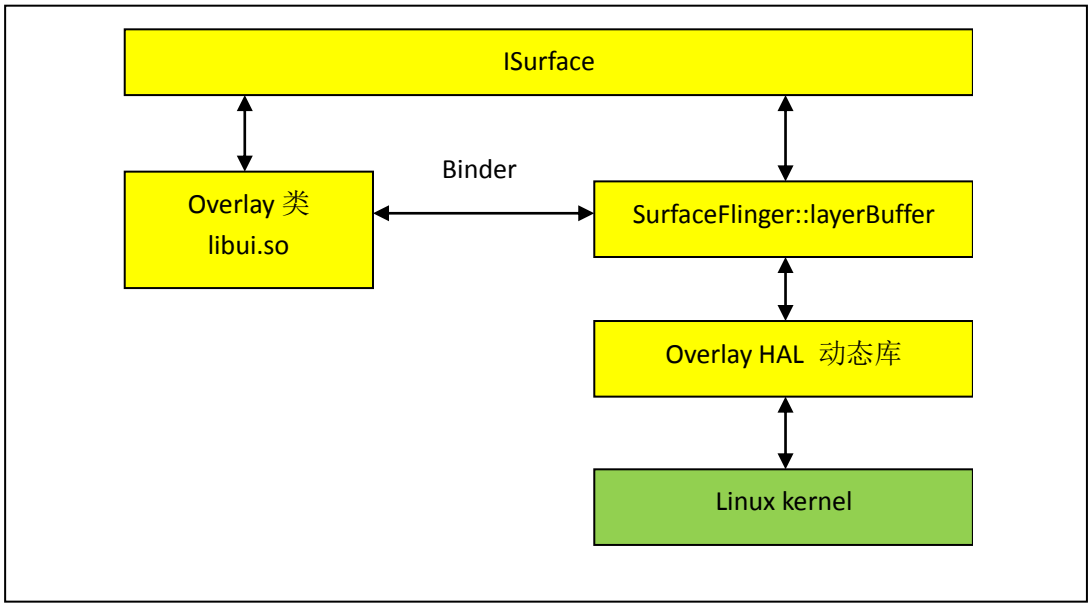
```
sp<Overlay> overlay = new Overlay(ref);
```

- 使用 overlay API

```
overlay_buffer_t buffer; //typedef void* overlay_buffer_t;
void* address = overlay->getBufferAddress(buffer);
```

address 指针就是 mmap 后的 overlay buffer 指针，只需将数据填充到这个 address 指针就可以看到画面了。

Android overlay 框架



overlay 本地框架代码

源代码位于：frameworks/base/libs/ui/ ，编译到 libui.so 中。

- Overlay.cpp：提供给外部程序调用的 Overlay object 接口与 API。定义在 frameworks/base/include/ui/Overlay.h 中。实现了两个类：OverlayRef 与 Overlay。外部程序通过这个 Overlay 对象来使用 overlay 的功能。Overlay.cpp 内部通过 binder 与 surfaceFlinger service 通信，最终调用到 Overlay HAL。

- IOVerlay.cpp：定义提供 binder 所需的类，其中 LayerBuffer::OverlaySource::OverlayChannel 继承自 BnOverlay。

overlay 的服务部分代码

源代码位于：frameworks/base/libs/surfaceflinger/

overlay 系统被包在 Surface 系统中，通过 surface 来控制 overlay 或者在不使用 overlay 的情况下统一的来管理。所以 overlay 的 service 部分也包含在 SurfaceFlinger service 中，主要的类 LayerBuffer。

android 启动的时候会启动 SurfaceFlinger service，SurfaceFlinger 启动时会实例化一个 DisplayHardware：

```
DisplayHardware* const hw = new DisplayHardware(this, dpy);
```

DisplayHardware 构造函数调用函数 init：

```
DisplayHardware::DisplayHardware(const sp<SurfaceFlinger>& flinger,
                                 uint32_t dpy)
:  DisplayHardwareBase(flinger, dpy)
{
  init(dpy);
}
```

Init 函数中：

```
if(hw_get_module(OVERLAY_HARDWARE_MODULE_ID, &module) == 0) {
    overlay_control_open(module, &mOverlayEngine);
}
```

获得 overlay 的 module 参数，调用 overlay_control_open 获取控制设备结构 mOverlayEngine。拥有了控制设备结构体就可以创建数据设备结构体 ，并具体控制使用 overlay 了。

overlay HAL 层

源代码位于：hardware/libhardware/include/hardware/overlay.h

android 只给出了接口的定义，需要我们自己实现具体的功能。 overlay hal 层生成的动态库在 SurfaceFlinger 中显式的加载。Overlay HAL 层具体功能如何实现取决于硬件与驱动程序。Android 提供了一个 Overlay Hal 层实现的框架代码，hardware/libhardware/modules/overlay/。因为 overlay hal 层生成的动态库是显式的动态打开(hw_get_module -> dlopen)，所以这个库

文件必须放在文件系统的 system/lib/hw/下。

多层 overlay

例如需要同时支持 overlay1 与 overlay2。

1. overlay hal 的 overlay_control_device_t 中要添加 overlay1 与 overlay2 的结构：

```
struct overlay_control_context_t {
   struct overlay_control_device_t device;
   /* our private state goes below here */
   struct overlay_t* overlay_video1;//overlay1
   struct overlay_t* overlay_video2;//overlay2
};
```

每个 overlay_t 代表一层 overlay，每层 ovelay 有自己的 handle。

在构造 OverlayRef 之前需指明使用哪一层 overlay：

```
sp<OverlayRef> ref = isurface->createOverlay(320, 240, PIXEL_FORMAT_RGB_565);
```

可以使用自定义参数调用 overlay_control_device_t::setParameter()来指定 Hal 层具体实现。

2，通过 Overlay object 来拿到 overlay1 与 overlay2 的 buffer 指针。