

# Android 的 Video 输入输出系统

# Android 的 Video 输入输出系统

- 第一部分 Video 输入输出系统的综述
- 第二部分 Overlay 系统
- 第三部分 Overlay 的硬件抽象层
- 第四部分 Camera 系统与上层接口
- 第五部分 Camera 的硬件抽象层

# 第一部分 Video 输入输出系统的综述

在 **Android** 系统中，视频的输入、输出具有特定的架构。

视频输入输出的两个部分是：

- ❑ 视频输入： **Camera** 系统

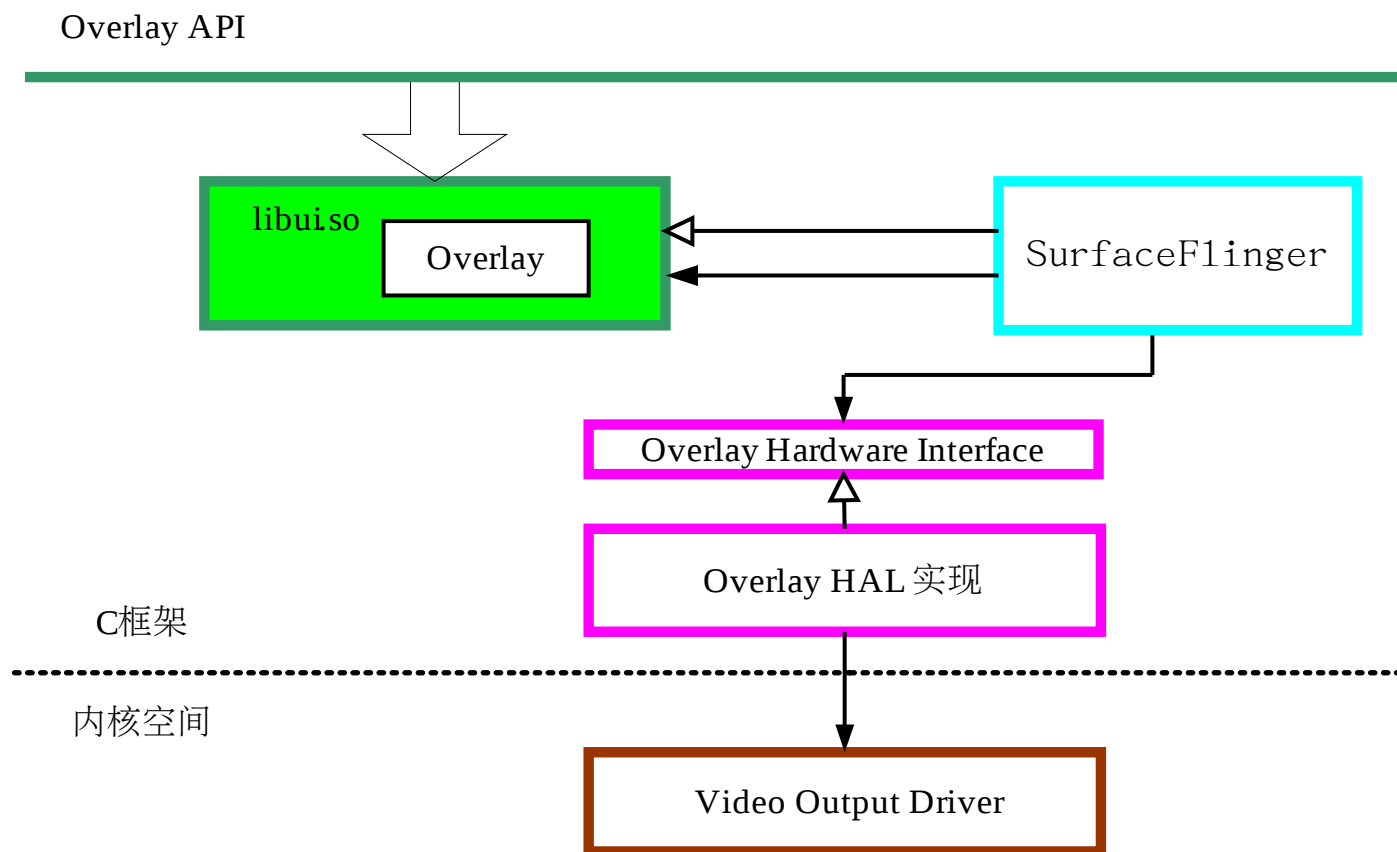
既作为视频输入的接口，也作为照相机应用的下层实现。

- ❑ 视频输出： **Overlay** 系统

一般作为视频输出的单独层次，在硬件支持中实现。

# 第一部分 Video 输入输出系统的综述

## Android 的 Overlay 系统结构



# 第一部分 Video 输入输出系统的综述

## Overlay 相关的代码路径：

Overlay 框架部分的头文件和源文件：

[frameworks/base/include/ui/](#)

[frameworks/base/libs/ui/](#)

主要为类是 IOverlay 和 Overlay ， 源代码被编译成库 libui.so 。

与 Overlay 相关的 SurfaceFlinger :

[framework/base/libs/surfaceflinger/](#)

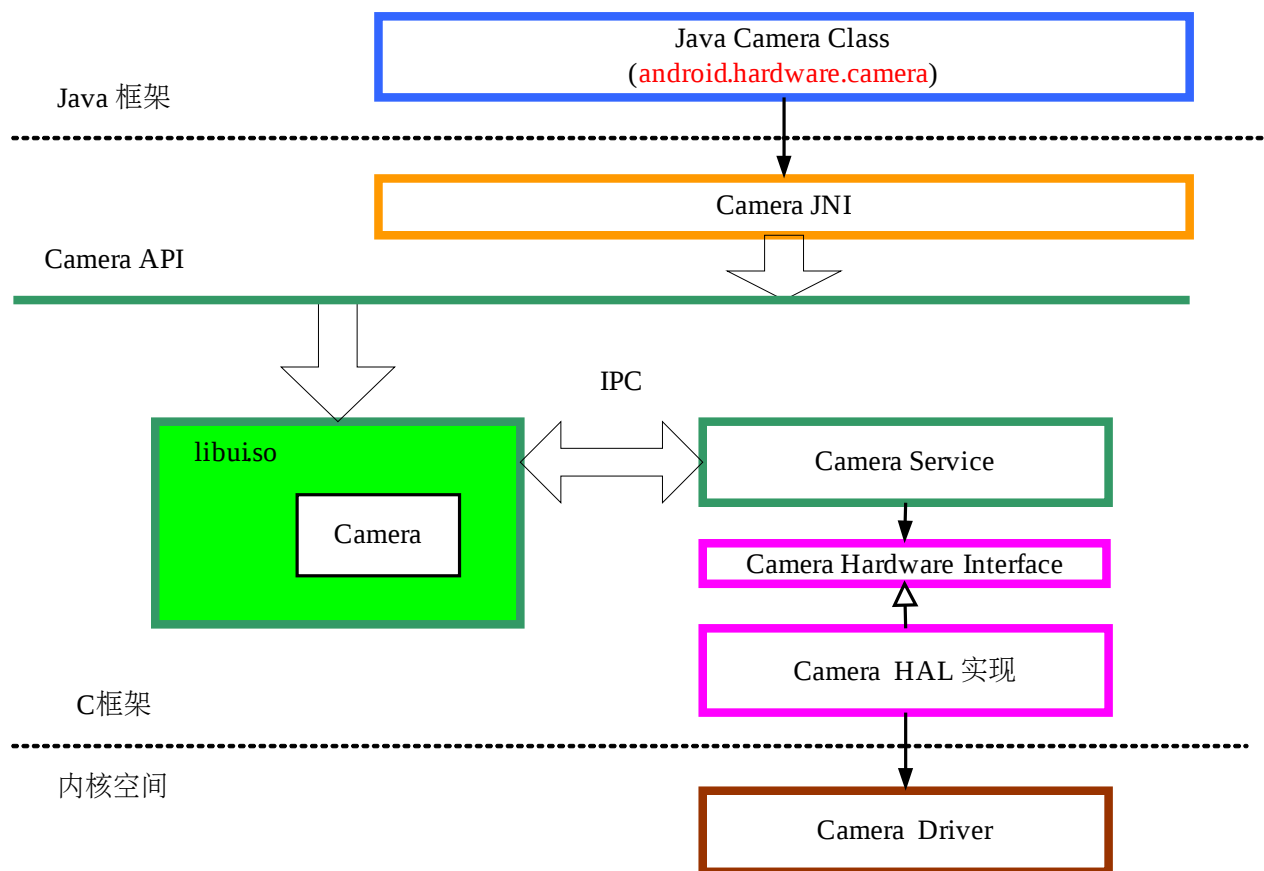
主要的类是 LayerBuffer 。

Overlay 的硬件抽象层的接口：

[hardware/libhardware/include/hardware/overlay.h](#)

# 第一部分 Video 输入输出系统的综述

## Android 的 Camera 系统结构



# 第一部分 Video 输入输出系统的综述

## Camera 相关的代码路径:

Camera 框架部分的头文件和源文件:

[frameworks/base/include/ui/](#)

[frameworks/base/libs/ui/](#)

这部分的内容被编译成库 libui.so 。

Camera 服务部分:

[frameworks/base/camera/libcameraservice/](#)

这部分内容被编译成库 libcameraservice.so 。

# 第一部分 Video 输入输出系统的综述

Camera 的 JAVA 本地调用部分（JNI）：

[frameworks/base/core/jni/android\\_hardware\\_Camera.cpp](#)

Camera 的 JAVA 类：

[frameworks/base/core/java/android/hardware/Camera.java](#)

Camera 的硬件抽象层的定义：

[frameworks/base/include/ui/](#)

目录之中的 CameraHardwareInterface.h

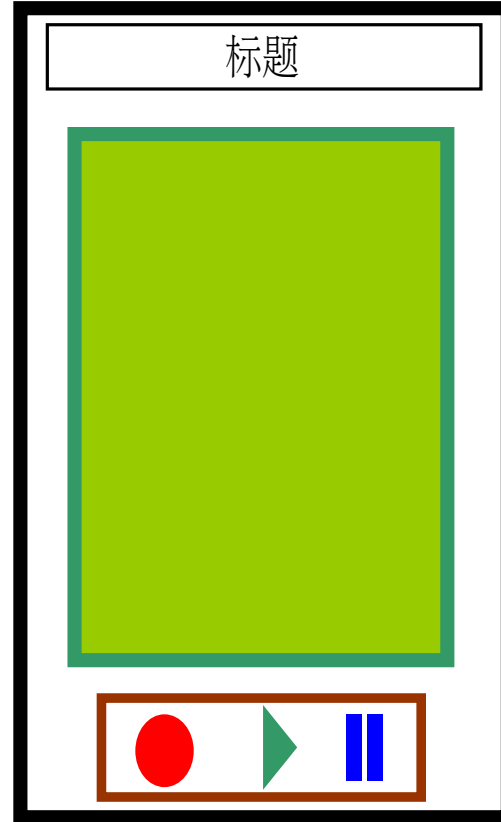


## 第二部分 Overlay 系统

在 Android 中，Overlay 系统提供 **overlay** 接口，这个接口的含义是叠加在主的显示层上的另外一个显示层，这个叠加的显示层通常作为视频的输出生或者照相机取景器的预览界面来使用。

Overlay 通过 **ISurface** 接口来使用，这个 Overlay 的使用与 **ISurface** 中的 **registerBuffers**，**postBuffer**，**unregisterBuffers** 几个接口是并立的，使用 **Overlay** 接口将和 **SurfaceFlinger** 中的显示等功能无关。

## 第二部分 Overlay 系统



## 第二部分 Overlay 系统

ISurface 接口的定义:

```
class ISurface : public IInterface
{
public:
    DECLARE_META_INTERFACE(Surface);
    /* ... */
    virtual status_t registerBuffers(const BufferHeap& buffers) = 0;
    virtual void postBuffer(ssize_t offset) = 0; // one-way
    virtual void unregisterBuffers() = 0;
    virtual sp<OverlayRef> createOverlay(
        uint32_t w, uint32_t h, int32_t format) = 0;
};
```

Overlay 接口虽然通过 SurfaceFlinger 的 **LayerBuffer** 来实现，但是 Overlay 是一个独立接口和 SurfaceFlinger 的其他部分没有依赖关系。

## 第二部分 Overlay 系统

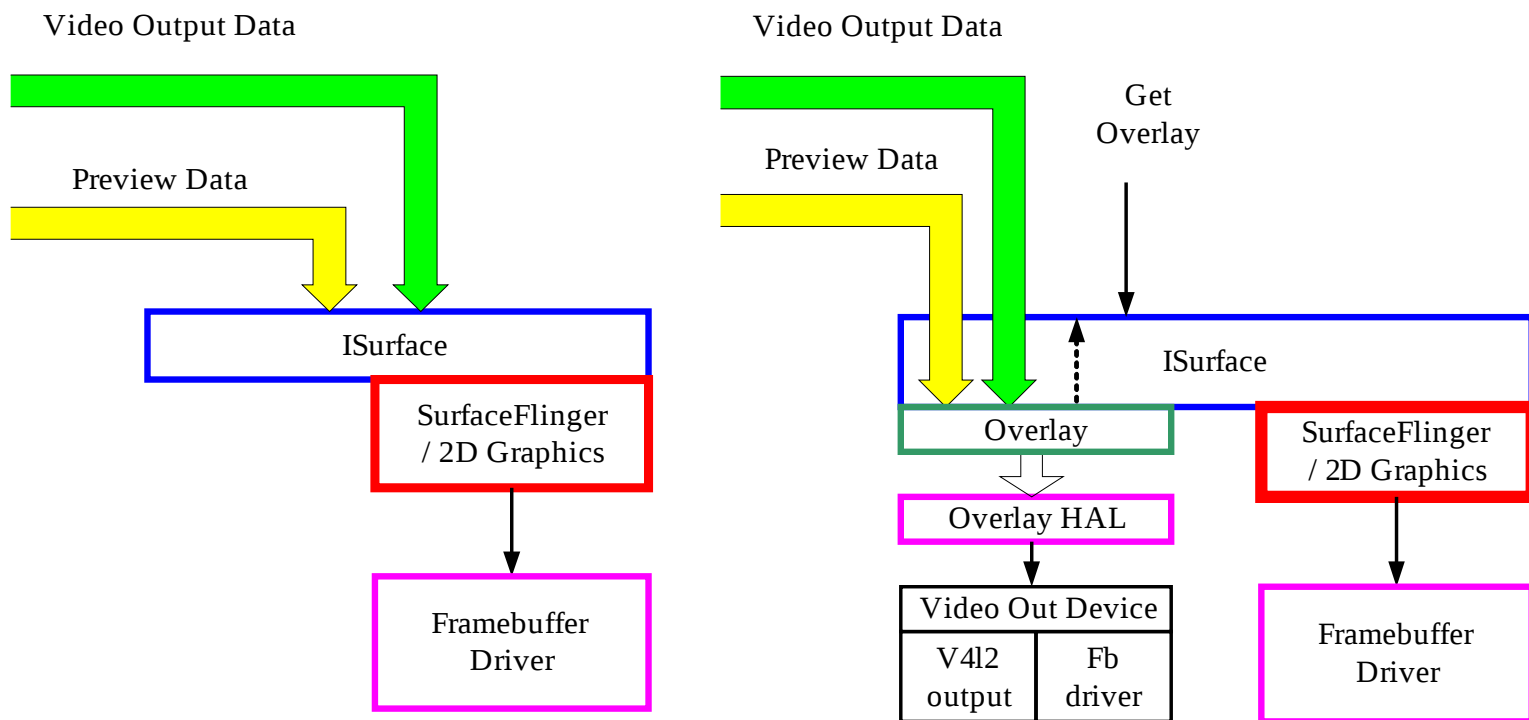
在文件 `Overlay.h` 中，定义了两个类 `OverlayRef` 和 `Overlay`。

```
class Overlay : public virtual RefBase
{
public:
    Overlay(const sp<OverlayRef>& overlayRef);
    void destroy();
    status_t dequeueBuffer(overlay_buffer_t*
buffer);
    status_t queueBuffer(overlay_buffer_t buffer);
    void* getBufferAddress(overlay_buffer_t buffer);
    /* ... */
};
```

类 `Overlay` 中的几个接口用于视频数据的输出，可以用队列，也可以直接使用地址。

## 第二部分 Overlay 系统

不使用 Overlay 和使用 Overlay 的对比:



## 第二部分 Overlay 系统

Overlay 与其他系统不同，它没有主动被 Android 系统所使用，因此如果移植了 Overlay 系统的硬件抽象层。还需要增加使用 Overlay 的部分。

Overlay 的使用场景主要有两个：

- 视频播放器的输出（PVPlayer）
- Preview 的输出（CameraHal）

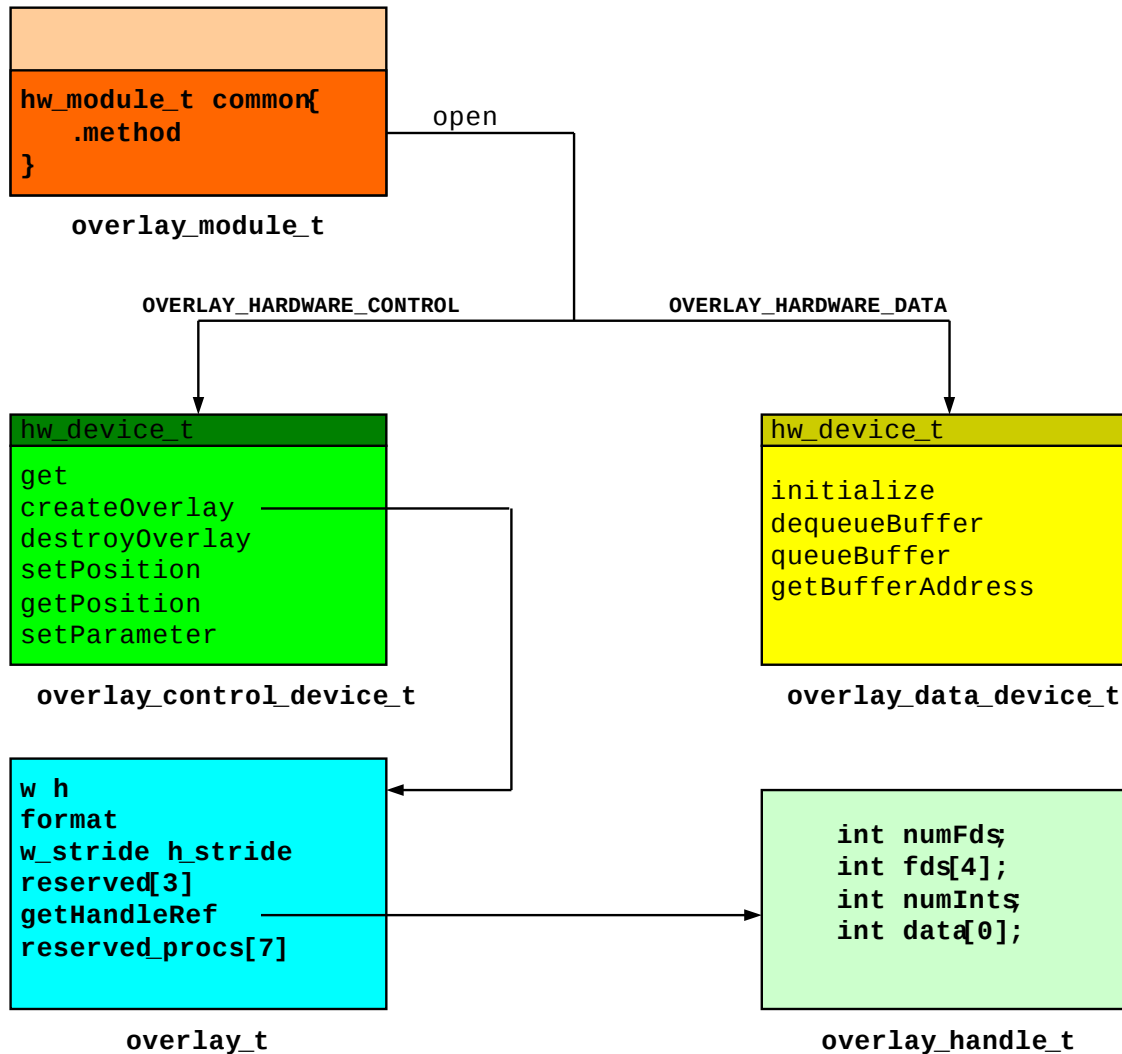
# 第三部分 Overlay 的硬件抽象层

Overlay 的硬件抽象层的接口在以下头文件中定义：  
<hardware/libhardware/include/hardware/overlay.h>

在这个头文件中，主要定义了两个类：  
`overlay_control_device_t` 和 `overlay_data_device_t`，  
它们分别继承了 `hw_device_t common`，通过这两个类  
实现 Overlay 的硬件抽象层。

实现一个 Overlay 的硬件抽象层使用的是  
Android 硬件模块的标准方法，通过类  
`overlay_module_t` 来完成。

# 第三部分 Overlay 的硬件抽象层





# 第三部分 Overlay 的硬件抽象层

Overlay 硬件抽象层的一个实现示例在以下中实现：  
[hardware/libhardware/modules/overlay/overlay.cpp](https://source.android.com/docs/core/hardware/libhardware/modules/overlay/overlay.cpp)

```
static struct hw_module_methods_t overlay_module_methods = {
    open: overlay_device_open
};
const struct overlay_module_t HAL_MODULE_INFO_SYM = {
    common: {
        tag: HARDWARE_MODULE_TAG,
        version_major: 1,
        version_minor: 0,
        id: OVERLAY_HARDWARE_MODULE_ID,
        name: "Sample Overlay module",
        author: "The Android Open Source Project",
        methods: &overlay_module_methods,
    }
};
```

# 第三部分 Overlay 的硬件抽象层

Overlay 的使用过程:

overlay\_control\_open

- overlay\_module\_t(hw\_module\_t common) :: open
- overlay\_control\_device\_t (hw\_device\_t common)
- overlay\_module\_t(hw\_module\_t common) :: open
- overlay\_data\_device\_t (hw\_device\_t common)

overlay\_control\_device\_t :: createOverlay

- overlay\_t :: getHandleRef
- overlay\_handle\_t

overlay\_data\_device\_t :: initialize(overlay\_handle\_t)

overlay\_data\_device\_t :: dequeueBuffer

overlay\_data\_device\_t :: queueBuffer

overlay\_data\_device\_t :: getBufferAddress

# 第三部分 Overlay 的硬件抽象层

**Overlay** 硬件抽象层需要基于一个视频显示的驱动来实现。

**Overlay** 的硬件抽象层通常基于两个驱动：

- **framerbuffer** 驱动程序
- **Video for Linux 2** 中的视频输出驱动。

基于 **framerbuffer** 驱动程序的实现，通常实现获得内存地址的接口即可。基于 **v4l2** 的实现可以提供流方式的接口，获得更好的性能，其中又分成使用内核内存和使用用户空间内存两种方式。

## 第四部分 Camera 系统与上层接口

4.1 Camera 框架和 CameraService

4.2 Camera 的 JNI 和 JAVA

## 4.1 Camera 框架和 CameraService

在 Android 系统中，Android 的 Camera 包含取景器（`viewfinder`）、视频数据获取（`Recording`）和拍摄照片的功能。

Camera 部分的主要头的框架部分包含在 `ui` 库的中，而 Camera 中间层的实现是 `CameraService`，`CameraService` 通过调用下层的 Camera 硬件抽象层来实现功能。

## 4.1 Camera 框架和 CameraService

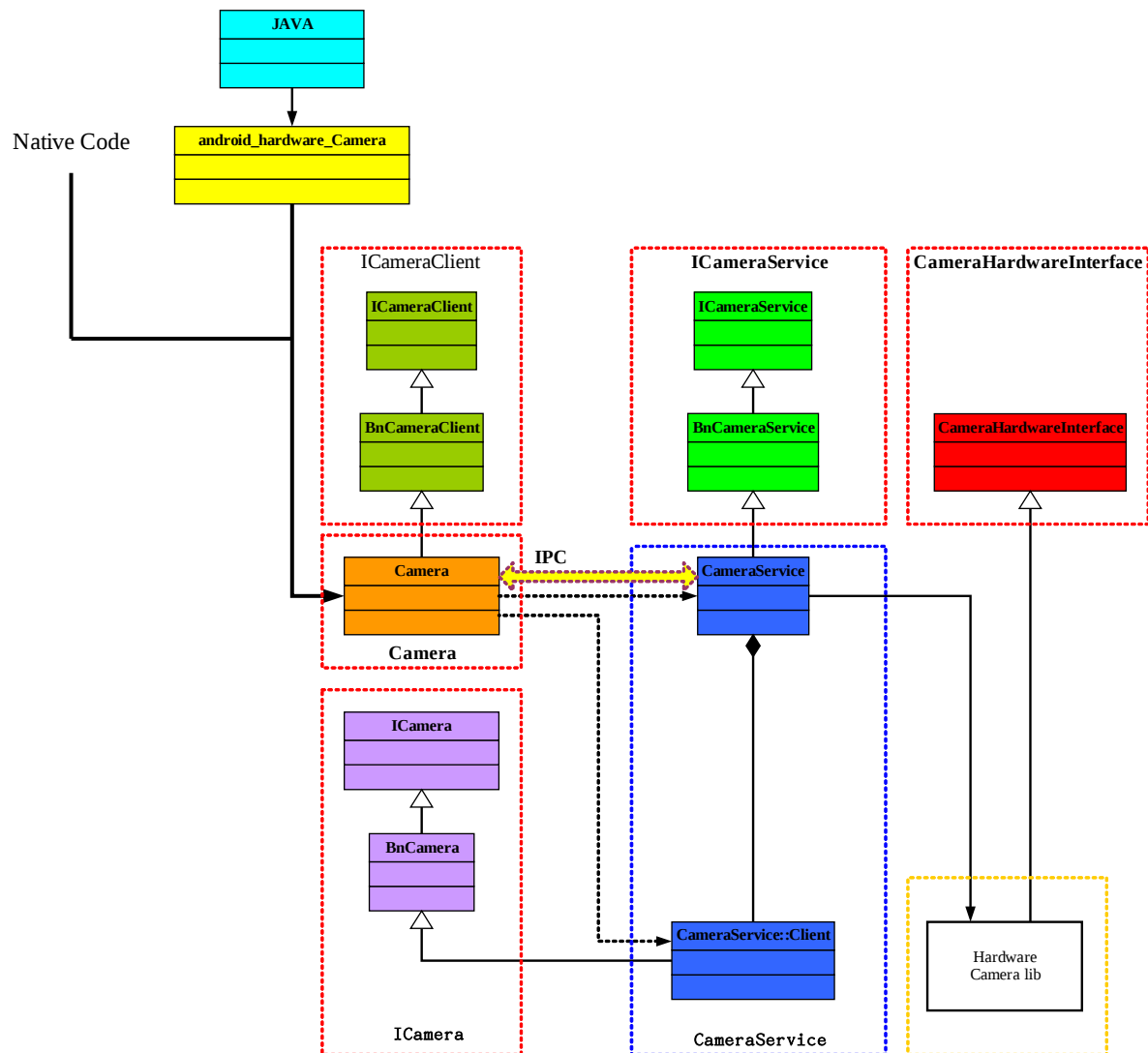
Camera 主要的头文件有以下几个：

- ❑ ICameraClient.h
- ❑ Camera.h
- ❑ ICamera.h
- ❑ ICameraService.h

ICameraService.h 、 ICameraClient.h 和 ICamera.h 三个类定义了 Camera 的接口和架构， ICameraService.cpp 和 Camera.cpp 两个文件用于 Camera 架构的实现， Camera 的具体功能在下层调用硬件相关的接口来实现。

Camera.h 是 Camera 系统对上层的接口。

## 4.1 Camera 框架和 CameraService



## 4.1 Camera 框架和 CameraService

Camera.h 是 Camera 系统对上层的接口；

ICameraService.h、ICameraClient.h 和 ICamera.h 三个类定义了 Camera 中间层实现的框架。它们接口的形式不同，但是具有 Camera 系统共同的几个方面：

- ❑ 预览功能（Preview）
- ❑ 视频获取功能（Recording）
- ❑ 拍照照片（takePicture）
- ❑ 参数设置



# 4.1 Camera 框架和 CameraService

Camera.h 中定义 Camera 对上层的接口。

```
// Typical use cases
#define FRAME_CALLBACK_FLAG_NOOP                0x00
#define FRAME_CALLBACK_FLAG_CAMCORDER          0x01
#define FRAME_CALLBACK_FLAG_CAMERA              0x05
#define FRAME_CALLBACK_FLAG_BARCODE_SCANNER    0x07
// msgType in notifyCallback and dataCallback functions
enum {
    CAMERA_MSG_ERROR            = 0x001,
    CAMERA_MSG_SHUTTER          = 0x002,
    CAMERA_MSG_FOCUS            = 0x004,
    CAMERA_MSG_ZOOM             = 0x008,
    CAMERA_MSG_PREVIEW_FRAME    = 0x010,
    CAMERA_MSG_VIDEO_FRAME      = 0x020,
    CAMERA_MSG_POSTVIEW_FRAME   = 0x040,
    CAMERA_MSG_RAW_IMAGE        = 0x080,
    CAMERA_MSG_COMPRESSED_IMAGE = 0x100,
    CAMERA_MSG_ALL_MSGS         = 0x1FF
};
// ref-counted object for callbacks
class CameraListener: virtual public RefBase
{
public:
    virtual void notify(int32_t msgType, int32_t ext1, int32_t ext2) = 0;
    virtual void postData(int32_t msgType, const sp<IMemory>& dataPtr) = 0;
    virtual void postDataTimestamp(nsecs_t timestamp, int32_t msgType, const sp<IMemory>& dataPtr) = 0;
};
```

## 4.1 Camera 框架和 CameraService

**CameraService** 是继承 **BnCameraService** 的实现，在这个类的内部又定义了类 **Client**，**CameraService::Client** 继承了 **BnCamera**。

在运作的过程中 **CameraService::connect()** 函数用于得到一个 **CameraService::Client**，在使用过程中，主要是通过调用这个类的接口来实现完成 **Camera** 的功能，由于 **CameraService::Client** 本身继承了 **BnCamera** 类，而 **BnCamera** 类是继承了 **ICamera**，因此这个类是可以被当成 **ICamera** 来使用的。

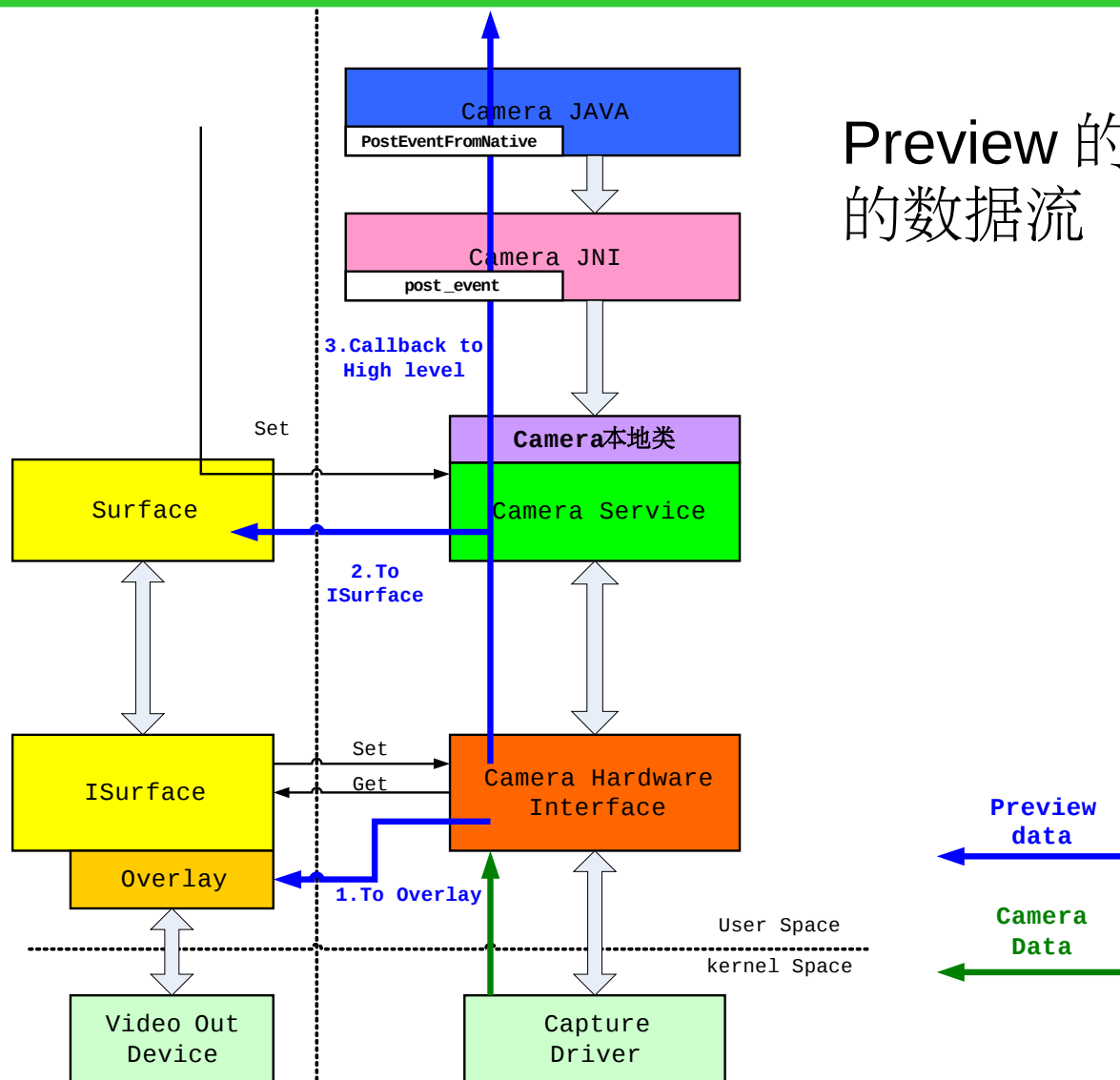
## 4.1 Camera 框架和 CameraService

照相机有三种实现 **ViewFinder**（**Preview**）的方式：

1. 在 **CameraHAL** 中，直接送给 **Overlay**
2. 在 **CameraService** 中，调用 **ISurface** 的 **postBuffer** 接口，送出数据。
3. **Camera** 类通过 **Callback** 送给上层，由上层处理

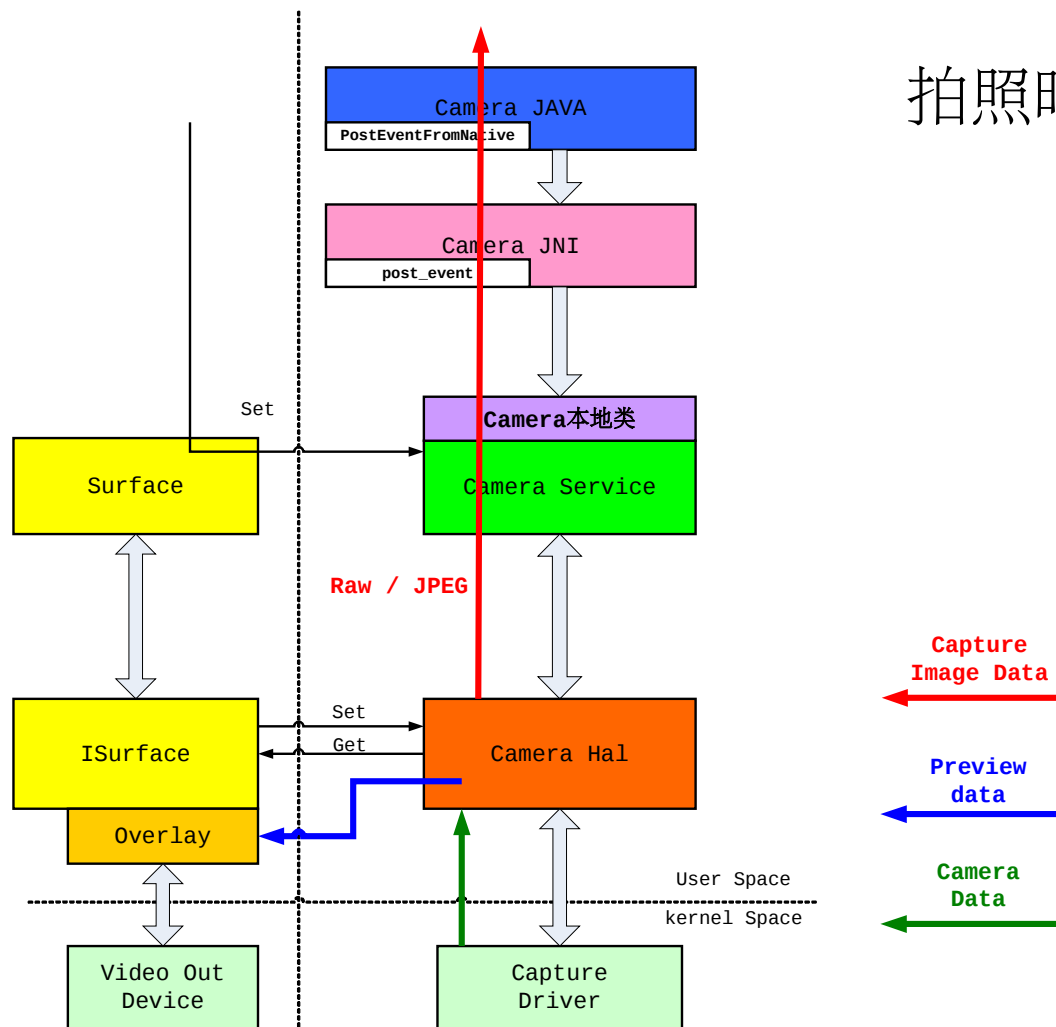
在 **Android** 照相机 / 摄像机应用程序中，使用的是 2 和 3 这两种方法，具体是 2 还是 3，由 **CameraService** 读取 **CameraHAL** 的 **useOverlay** 接口来实现。

## 4.1 Camera 框架和 CameraService



Preview 的三种方式  
的数据流

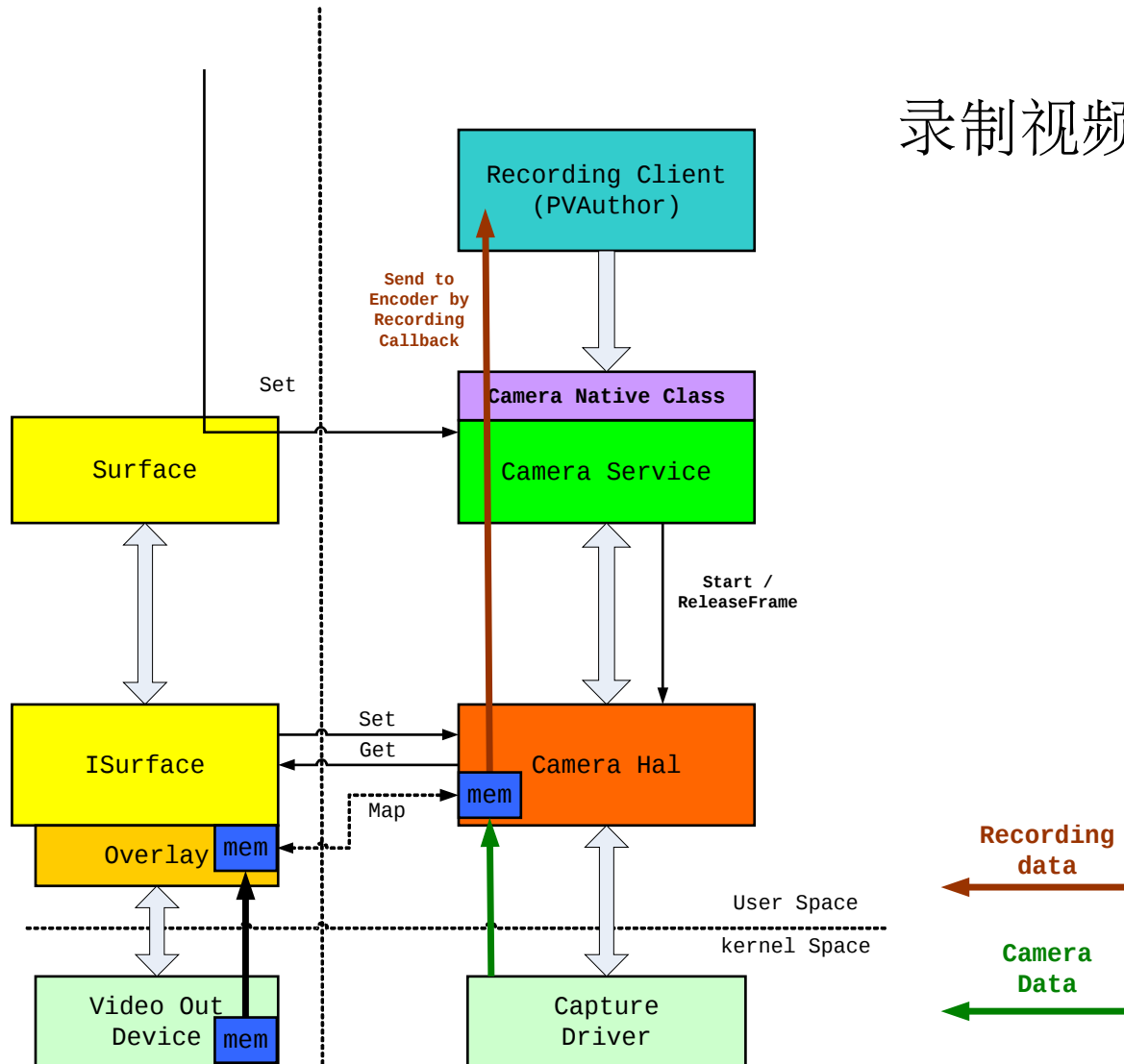
## 4.1 Camera 框架和 CameraService



拍照时的数据流

## 4.1 Camera 框架和 CameraService

录制视频时的数据流



## 4.2 Camera 的 JNI 和 JAVA

Android 的 Camera 使用 JNI 为向上层 JAVA 提供了接口。Camera 在 JAVA 中的类是：  
`android.hardware.Camera`。

Camera 的 JAVA 本地调用部分（JNI）：  
[frameworks/base/core/jni/android\\_hardware\\_Camera.cpp](#)

Camera 的 JAVA 类：  
[frameworks/base/core/java/android/hardware/Camera.java](#)

## 第五部分 Camera 的硬件抽象层

Camera 的硬件抽象层的在 UI 库的头文件 **CameraHardwareInterface.h** 文件定义。

在这个接口中，包含了控制通道和数据通道，控制通道用于处理预览和视频获取的开始 / 停止、拍摄照片、自动对焦等功能，数据通道通过回调函数来获得预览、视频录制、自动对焦等数据。

Camera 的硬件抽象层中还可以使用 **Overlay** 来实现预览功能。



## 第五部分 Camera 的硬件抽象层

CameraHardwareInterface.h 文件的定义:

```
/** Callback for startPreview() */
typedef void (*preview_callback)(const sp<IMemory>& mem, void* user);
/** Callback for startRecord() */
typedef void (*recording_callback)(const sp<IMemory>& mem, void* user);
/** Callback for takePicture() */
typedef void (*shutter_callback)(void* user);
/** Callback for takePicture() */
typedef void (*raw_callback)(const sp<IMemory>& mem, void* user);
/** Callback for takePicture() */
typedef void (*jpeg_callback)(const sp<IMemory>& mem, void* user);
/** Callback for autoFocus() */
typedef void (*autofocus_callback)(bool focused, void* user);
```

# 第五部分 Camera 的硬件抽象层

```
class CameraHardwareInterface : public virtual RefBase {
public:
    virtual ~CameraHardwareInterface() { }
    virtual sp<IMemoryHeap>      getPreviewHeap() const = 0;
    virtual sp<IMemoryHeap>      getRawHeap() const = 0;
    virtual status_t      startPreview(preview_callback cb, void* user) = 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(recording_callback cb, void* user) = 0;
    virtual void      stopRecording() = 0;
    virtual bool      recordingEnabled() = 0;
    virtual void      releaseRecordingFrame(const sp<IMemory>& mem) = 0;
    virtual status_t      autoFocus(autofocus_callback, void* user) = 0;
    virtual status_t      takePicture(shutter_callback, raw_callback,
                                      jpeg_callback, void* user) = 0;
    virtual status_t      cancelPicture(bool cancel_shutter, bool cancel_raw,
                                      bool cancel_jpeg) = 0;
    virtual status_t      setParameters(const CameraParameters& params) = 0;
    virtual CameraParameters getParameters() const = 0;
    virtual void release() = 0;
    virtual status_t dump(int fd, const Vector<String16>& args) const = 0;
};
```

# 第五部分 Camera 的硬件抽象层

Android 的 Éclair 版本中：  
CameraHardwareInterface.h 文件的定义：

```
typedef void (*notify_callback)(int32_t msgType,  
                                int32_t ext1,  
                                int32_t ext2,  
                                void* user);  
  
typedef void (*data_callback)(int32_t msgType,  
                               const sp<IMemory>& dataPtr,  
                               void* user);  
  
typedef void (*data_callback_timestamp)(nsecs_t timestamp,  
                                         int32_t msgType,  
                                         const sp<IMemory>& dataPtr,  
                                         void* user);
```

# 第五部分 Camera 的硬件抽象层

```
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 void stopRecording() = 0;
    virtual bool recordingEnabled() = 0;
    virtual void releaseRecordingFrame(const sp<IMemory>& mem) = 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;
};
```

## 第五部分 Camera 的硬件抽象层

在 CameraService 中，实现了一个 CameraHardwareInterface 的“桩”，它们在文件 CameraHardwareStub.cpp 和 FakeCamera.cpp 中实现。

当编译宏 **USE\_CAMERA\_STUB** 打开的时候，CameraService 将使用这个桩，这样整个 Camera 模块可以在没有硬件的情况下编译通过并可以假装运行。

## 第五部分 Camera 的硬件抽象层

取景器预览的主要步骤如下所示：

- ❑ 在初始化的过程中，建立预览数据的内存队列（多种方式）；
- ❑ 在 `startPreview()` 的实现中，保存预览回调函数，建立预览线程；
- ❑ 在预览线程的循环中，等待视频数据的到达；
- ❑ 视频帧到达后调用预览回调函数，将视频帧送出。

如果使用 **Overlay** 实现取景器，则需要有以下两个变化：

- ❑ 在 `setOverlay()` 函数中，从 **ISurface** 接口中取得 **Overlay** 类；
- ❑ 在预览线程的循环中，不需要使用预览回调函数，直接将视频数据输入到 **Overlay** 上。

## 第五部分 Camera 的硬件抽象层

对于 Linux 系统而言，摄像头驱动部分大多使用 Video for Linux 2 （ V4L2 ） 驱动程序，在此处主要的处理流程可以如下所示：

- 如果使用映射内核内存的方式（ V4L2\_MEMORY\_MMAP ），构建预览的内存 MemoryHeapBase 需要从 V4L2 驱动程序中得到内存指针；
- 如果使用用户空间内存的方式（ V4L2\_MEMORY\_USERPTR ）， MemoryHeapBase 中开辟的内存是在用户空间建立的；
- 在预览的线程中，使用 VIDIOC\_DQBUF 调用阻塞等待视频帧的到来，处理完成后 使用 VIDIOC\_QBUF 调用将帧内存再次压入队列，等待下一帧的到来。

## 第五部分 Camera 的硬件抽象层

录制视频的主要步骤如下所示：

- ❑ 在 `startRecording()` 的实现（**或者在 `setCallbacks`**）中，保存录制视频回调函数；
- ❑ 录制视频可以使用自己的线程，也可以使用预览线程；
- ❑ 通过录制回调函数将视频帧送出；

`releaseRecordingFrame()` 被调用后，表示上层通知 Camera 硬件抽象层，这一帧的内存已经用完，可以进行下一次的处理。

如果在 V4L2 驱动程序中使用原始数据（**RAW**），则视频录制的数据和取景器预览的数据为同一数据。

`releaseRecordingFrame()` 被调用时，通常表示编码器已经完成了对当前视频帧的编码，对这块内存进行释放。在这个函数的实现中，可以设置标志位，标记帧内存可以再次使用。



谢谢！