Android 的 Camera 系统分析

一、Camera 构架分析

Android 的 Camera 包含取景（preview）和拍摄照片（take picture）的功能 。目前 Android 发布版的 Camera 程序 虽然功能比较简单，但是其程序的架构分成客户端和服务器两个部分，它们建立在 Android 的进程间通讯 Binder 的结构上。Android 中 Camera 模块同样遵循 Andorid 的框架，如下图所示

Camera Architecture

Camera 模块主要包含了 libandroid_runtime.so、libui.so 和 libcameraservice.so 等几个库文件 ，它们之间的调用关系如下所示：

在 Camera 模块的各个库中，libui.so 位于核心的位置，它对上层的提供的接口主要是 Camera 类。

libcameraservice.so 是 Camera 的 server 程序，它通过继承 libui.so 中的类实现 server 的功能，并且与 libui.so 中的另外一部分内容通过进程间通讯（即 Binder 机制）的方式进行通讯。

libandroid_runtime.so 和 libui.so 两个库是公用的,其中除了 Camera 还有其他方面的功能。整个 Camera 在运行 的时候，可以大致上分成 Client 和 Server 两个部分，它们分别在两个进程中运行，它们之间使用 Binder 机制实现进程间通讯。这样在 client 调用接口，功能则在 server 中实现，但是在 client 中调用就好像直接调用 server 中的功能，进程间通讯的部分对上层程序不可见。

从框架结构上来看，源码 中 ICameraService.h、ICameraClient.h 和 ICamera.h 三个类定义 了 MeidaPlayer 的接口和 架构，ICameraService.cpp 和 Camera.cpp 两个文件则用于 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。

从 Camera 的整体结构上，类 Camera 是整个系统 核心，ICamera 类提供了 Camera 主要功能的接口，在客户端方面调用；CameraService 是 Camera 服务，它通过调用实际的 Camera 硬件接口来实现功能。事实上，图中红色虚线框的部分都是 Camera 程序的框架部分，它主要利用了 Android 的系统的 Binder 机制来完成通讯。蓝色的部分通过调用 Camera 硬件相关的接口完成具体的 Camera 服务功能，其它的部分是为上层的 JAVA 程序提供 JNI 接口。在整体结构上，左边可以视为一个客户端，右边是一个可以视为服务器，二者通过 Android 的 Bimder 来实现进程间的通讯。

二、Camera 的工作流程概述：

①．App_main process: 进程通过 AndroidRuntime 调用 register_jni_procs 向 JNI 注册模块的 native 函数 供 JVM 调用。

```
AndroidRuntime::registerNativeMethods(env, "android/hardware/Camera",
camMethods, NELEM(camMethods));
```

其中 camMethods 定义如下:

```
static JNINativeMethod camMethods[] = {
  { "native_setup",
    "(Ljava/lang/Object;)V",
    (void*)android_hardware_Camera_native_setup },
  { "native_release",
    "()V",
    (void*)android_hardware_Camera_release },
```

```
    { "setPreviewDisplay",
      "(Landroid/view/Surface;)V",
      (void *)android_hardware_Camera_setPreviewDisplay },
    { "startPreview",
      "()V",
      (void *)android_hardware_Camera_startPreview },
    { "stopPreview",
      "()V",
      (void *)android_hardware_Camera_stopPreview },
    { "previewEnabled",
      "()Z",
      (void *)android_hardware_Camera_previewEnabled },
    { "setHasPreviewCallback",
      "(ZZ)V",
      (void *)android_hardware_Camera_setHasPreviewCallback },
    { "native_autoFocus",
      "()V",
      (void *)android_hardware_Camera_autoFocus },
    { "native_takePicture",
      "()V",
      (void *)android_hardware_Camera_takePicture },
    { "native_setParameters",
      "(Ljava/lang/String;)V",
      (void *)android_hardware_Camera_setParameters },
    { "native_getParameters",
      "()Ljava/lang/String;",
      (void *)android_hardware_Camera_getParameters },
    { "reconnect",
      "()V",
      (void*)android_hardware_Camera_reconnect },
    { "lock",
      "()I",
      (void*)android_hardware_Camera_lock },
    { "unlock",
      "()I",
      (void*)android_hardware_Camera_unlock },
};
```

JNINativeMethod 的第一个成员是一个字符 串，表示了 JAVA 本地调用方法的名称，这个名称是在 JAVA 程序中调用的名称；第二个成员也是一个字符串，表示 JAVA 本地调用方法的参数和返回值；第三个成员是 JAVA 本地调用方法对应的 C 语言函数。

②．Mediaserver proces：进程注册了以下几个 server: AudioFlinger、 MediaPlayerServer、CameraService.

```
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();
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
}
```

当向 ServiceManager 注册了 CameraService 服务后就可以响应 client 的请求了

2.client 端向 service 发送请求

①.在 java 应用 层调用 onCreate()函数得到一个上层的 Camera 对象

```
public void onCreate(Bundle icicle) {
super.onCreate(icicle);
Thread openCameraThread = new Thread(
new Runnable() {
public void run() {
                    mCameraDevice = android.hardware.Camera.open();
}
}
);
……
}
```

②．通过 Camera 对象的调用成员函数，而这些成员函数会调用已向 JNI 注册过的 native 函数来调用 ICamera 接口的成员函数向 Binder Kernel Driver 发送服务请求。

③. Binder Kernel Driver 接收到 client 的请求后，通过唤醒 service 的进程来处理 client 的请求，处理完后通过回调函数传回数据 并通知上层处理已完成。

三．Camera 库文件分析

上面已提到 Camera 模块主要包含 libandroid_runtime.so、libui.so、libcameraservice.so 和一个与 Camera 硬件相关的底层库。其中 libandroid_runtime.so、libui.so 是与 Android 系统构架相关的不需要对其进行修改， libcameraservice.so 和 Camera 硬件相关的底层库则是和硬件设备相关联的，而 Canera 硬件相关的底层库实际上就是设备的 Linux 驱动，所以 Camera 设备的系统集成主要通过移植 Camera Linux 驱动和修改 libcameraservice.so 库来完成。

libcameraservice.so 库通过以下规则来构建：

构建规则略

在上面的构建规则中可以看到使用了宏 USE_CAMERA_STUB 决定 是否使用真的 Camera，如果宏为真，则使用 CameraHardwareStub.cpp 和 FakeCamera.cpp 构造一个假的 Camera，如果为假则使用 libcamera 来构造一个实际上的 Camera 服务。

在 CameraHardwareStub.cpp 中定义了 CameraHardwareStub 类，它继承并实现了抽象类 CameraHardwareInterface 中定义的真正操作 Camera 设备的所有的纯虚函数。通过 openCameraHardware （） 将返回一个 CameraHardwareInterface 类的对象， 但由于

CameraHardwareInterface 类是抽象类所以它并不能创建对象，而它的派生类 CameraHardwareStub 完全实现了其父类的纯虚函数所以 openCameraHardware（）返回一个指向派生类对象的基类指针用于底层设备的操作。由于 CameraHardwareStub 类定义的函数是去操作一个假的 Camera，故通过 openCameraHardware 返回的指针主要用于仿真环境 对 Camera 的模拟 操作，要想通过 openCameraHardware 返回的指针操作真正的硬件设备则需完成以下步骤：

1. 将 CameraHardwareInterface 类中的所有纯虚函数的声明改为虚函数的声明（即去掉虚函数声明后的“= 0” ）;

2. 编写一个源文件去定义 CameraHardwareInterface 类中声明的所有虚函数，并实现 openCameraHardware()函数让该函数返回一个 CameraHardwareInterface 类对象的指针;

3. 仿照其他.mk 文件编写 Android.mk 文件用于生成一个包含步骤 2 编写的源文件和其他相关文件的 libcamera.so 文件;例如

4. 将宏 USE_CAMERA_STUB 改成 false，这样生成 libcameraservice.so 时就会包含 libcamera.so 库。（注：如果 CameraHardwareInterface 类的成员函数并没有直接操作硬件而是调用 Camera 的 linux 驱动来间接对硬件操作，那么包含这样的 CameraHardwareInterface 类的 libcamera.so 库就相当于一个 HAL）

Android 的 Camera 部分分析（二）

第一部分 Camera 概述

Android 的 Camera 包含取景器（viewfinder）和拍摄照片的功能。目前 Android 发布版的 Camera 程序虽然功能比较简单，但是其程序的架构分成客户端和服务器两个部分，它们建立在 Android 的进程间通讯 Binder 的结构上。

以开源的 Android 为例，Camera 的代码主要在以下的目录中：

Camera 的 JAVA 程序的路径：
packages/apps/Camera/src/com/android/camera/
在其中 Camera.java 是主要实现的文件

frameworks/base/core/java/android/hardware/Camera.java
这个类是和 JNI 中定义的类是一个，有些方法通过 JNI 的方式调用本地代码得到，有些方法自己实现。

Camera 的 JAVA 本地调用部分（JNI）：
frameworks/base/core/jni/android_hardware_Camera.cpp

这部分内容编译成为目标是 libandroid_runtime.so。

主要的头文件在以下的目录中：
frameworks/base/include/ui/

Camera 底层库在以下的目录中：
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。

为了实现一个具体功能的 Camera，在最底层还需要一个硬件相关的 Camer 库（例如通过调用 video for linux 驱动程序和 Jpeg 编码程序实现）。这个库将被 Camera 的服务库 libcameraservice.so 调用。

第二部分 Camera 的接口与架构

2.1 Camera 的整体框架图 Camera 的各个库之间的结构可以用下图的表示：

在 Camera 系统的各个库中，libui.so 位于核心的位置，它对上层的提供的接口主要是 Camera 类，类 libandroid_runtime.so 通过调用 Camera 类提供对 JAVA 的接口，并且实现了 android.hardware.camera 类。

libcameraservice.so 是 Camera 的服务器程序，它通过继承 libui.so 的类实现服务器的功能，并且与 libui.so 中的另外一部分内容则通过进程间通讯（即 Binder 机制）的方式进行通讯。
libandroid_runtime.so 和 libui.so 两个库是公用的,其中除了 Camera 还有其他方面的功能。

Camera 部分的头文件在 frameworks/base/include/ui/目录中，这个目录是和 libmedia.so 库源文件的目录 frameworks/base/libs/ui/相对应的。

Camera 主要的头文件有以下几个：

ICameraClient.h
Camera.h
ICamera.h
ICameraService.h
CameraHardwareInterface.h

在这些头文件 Camera.h 提供了对上层的接口，而其他的几个头文件都是提供一些接口类（即包含了纯虚函数的类），这些接口类必须被实现类继承才能够使用。

整个 Camera 在运行的时候，可以大致上分成 Client 和 Server 两个部分，它们分别在两个进程中运行，它们之间使用 Binder 机制实现进程间通讯。这样在客户端调用接口，功能则在服务器中实现，但是在客户端中调用就好像直接调用服务器中的功能，进程间通讯的部分对上层程序不可见。

从框架结构上来看，ICameraService.h、ICameraClient.h 和 ICamera.h 三个类定义了 MeidaPlayer 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。

从 Camera 的整体结构上，类 Camera 是整个系统核心，ICamera 类提供了 Camera 主要功能的接口，在客户端方面调用，CameraService 是 Camera 服务，它通过调用实际的 Camera 硬件接口来实现功能。事实上，图中红色虚线框的部分都是 Camera 程序的框架部分，它主要利用了 Android 的系统的 Binder 机制来完成通讯。蓝色的部分通过调用 Camera 硬件相关的接口完成具体的 Camera 服务功能，其它的部分是为上层的 JAVA 程序提供 JNI 接口。在整体结构上，左边可以视为一个客户端，右边是一个可以视为服务器，二者通过 Android 的 Bimder 来实现进程间的通讯。

2.2 头文件 ICameraClient.h

ICameraClient.h 用于描述一个 Camera 客户端的接口，定义如下所示：

```
class ICameraClient: public IInterface
{
public:
        DECLARE_META_INTERFACE(CameraClient);
        virtual void shutterCallback() = 0;
        virtual void rawCallback(const sp<IMemory>& picture) = 0;
        virtual void jpegCallback(const sp<IMemory>& picture) = 0;
        virtual void frameCallback(const sp<IMemory>& frame) = 0;
        virtual void errorCallback(status_t error) = 0;
        virtual void autoFocusCallback(bool focused) = 0;
};

class BnCameraClient: public BnInterface<ICameraClient>
{
public:
        virtual status_t onTransact( uint32_t code,
                                                                const Parcel& data,
                                                                Parcel* reply,
                                                                uint32_t flags = 0);
};
```

在定义中，ICameraClient 类继承 IInterface，并定义了一个 Camera 客户端的接口，BnCameraClient 继承了 BnInterface<ICameraClient>，这是为基于 Android 的基础类 Binder 机制实现在进程通讯而构建的。根据 BnInterface 类模版的定义 BnInterface<ICameraClient>类相当于双继承了 BnInterface 和 ICameraClient。

IcameraClient 这个类的主要接口是几个回调函数 shutterCallback、rawCallback 和 jpegCallback 等，它们在相应动作发生的时候被调用。作为 Camera 的“客户端”，需要自己实现几个回调函数，让服务器程序去“间接地”调用它们。

2.3 头文件 Camera.h

Camera.h 是 Camera 对外的接口头文件，它被实现 Camera JNI 的文件 android_hardware_Camera.cpp 所调用。Camera.h 最主要是定义了一个 Camera 类：

```
class Camera : public BnCameraClient, public IBinder:: DeathRecipient
{
public:
```

```
        static    sp<Camera>    connect();
                                            ~Camera();
                        void                disconnect();
                        status_t        getStatus() { return mStatus; }
                        status_t        setPreviewDisplay(const sp<Surface>& surface);
                        status_t        startPreview();
                        void                stopPreview();
                        status_t        autoFocus();
                        status_t        takePicture();
                        status_t        setParameters(const String8& params);
                        String8          getParameters() const;
                        void                setShutterCallback(shutter_callback cb, void *cookie);
                        void                setRawCallback(frame_callback cb, void *cookie);
                        void                setJpegCallback(frame_callback cb, void *cookie);
                        void                setFrameCallback(frame_callback cb, void *cookie);
                        void                setErrorCallback(error_callback cb, void *cookie);
            void                setAutoFocusCallback(autofocus_callback cb, void *cookie);
        // ICameraClient interface
        virtual void                shutterCallback();
        virtual void                rawCallback(const sp<IMemory>& picture);
        virtual void                jpegCallback(const sp<IMemory>& picture);
        virtual void                frameCallback(const sp<IMemory>& frame);
        virtual void                errorCallback(status_t error);
        virtual void                autoFocusCallback(bool focused);
//……
}
```

从接口中可以看出Camera类刚好实现了一个Camera的基本操作，例如播放（startPreview）、停止（stopPreview）、暂停（takePicture）等。在Camera类中connect()是一个静态函数，它用于得到一个Camera的实例。在这个类中，具有设置回调函数的几个函数：setShutterCallback、setRawCallback和setJpegCallback等，这几个函数是为了提供给上层使用，上层利用这几个设置回调函数，这些回调函数在相应的回调函数中调用，例如使用setShutterCallback设置的回调函数指针被shutterCallback所调用。

在定义中，ICameraClient 类双继承了IInterface和IBinder::DeathRecipient，并定义了一个Camera客户端的接口，BnCameraClient 继承了BnInterface<ICameraClient>，这是为基于Android的基础类Binder机制实现在进程通讯而构建的。事实上，根据BnInterface类模版的定义BnInterface<ICameraClient>类相当于双继承了BnInterface和ICameraClient。这是Android一种常用的定义方式。

继承了DeathNotifier类之后,这样当这个类作为IBinder使用的时候,当这个Binder即将Died的时候被调用其中的binderDied函数。继承这个类基本上实现了一个回调函数的功能。

2.4 头文件ICamera.h

ICamera.h描述的内容是一个实现Camera功能的接口，其定义如下所示：

```
class ICamera: public IInterface
{
public:
```

```
        DECLARE_META_INTERFACE(Camera);
        virtual void            disconnect() = 0;
        virtual status_t   setPreviewDisplay(const sp<ISurface>& surface) = 0;
        virtual void            setHasFrameCallback(bool installed) = 0;
        virtual status_t   startPreview() = 0;
        virtual void            stopPreview() = 0;
        virtual status_t   autoFocus() = 0;
        virtual status_t   takePicture() = 0;
        virtual status_t   setParameters(const String8& params) = 0;
        virtual String8     getParameters() const = 0;
};
class BnCamera: public BnInterface<ICamera>
{
public:
        virtual status_t       onTransact( uint32_t code,
                                                                        const Parcel& data,
                                                                        Parcel* reply,
                                                                        uint32_t flags = 0);
};
```

ICamera.h 描述的内容是一个实现 Camera 功能的接口，其定义如下所示：

在 camera 类中，主要定义 Camera 的功能接口，这个类必须被继承才能够使用。值得注意的是，这些接口和 Camera 类的接口有些类似，但是它们并没有直接的关系。事实上，在 Camera 类的各种实现中，一般都会通过调用 ICamera 类的实现类来完成。

2.5 头文件 ICameraService .h

ICameraService.h 用于描述一个 Camera 的服务，定义方式如下所示：

```
class ICameraService : public IInterface
{
public:
        DECLARE_META_INTERFACE(CameraService);
        virtual sp<ICamera>         connect(const sp<ICameraClient>& cameraClient) = 0;
};
class BnCameraService: public BnInterface<ICameraService>
{
public:
        virtual status_t       onTransact( uint32_t code,
                                                                        const Parcel& data,
                                                                        Parcel* reply,
                                                                        uint32_t flags = 0);
};
```

由于具有纯虚函数， ICameraService 以及 BnCameraService 必须被继承实现才能够使用，在 ICameraService 只定义了一个 connect()接口，它的返回值的类型是 sp<ICamera>，这个 ICamera 是提供实现功能的接口。注意，ICameraService 只有连接函数 connect()，没有断开函数，断开的功能由 ICamera 接口来提供。

2.6 头文件 CameraHardwareInterface.h

CameraHardwareInterface.h 定义的是一个 Camera 底层的接口，这个类的实现者是最终实现 Camera 的。

CameraHardwareInterface 定以 Camera 硬件的接口，如下所示：

```
class CameraHardwareInterface : public virtual RefBase {
public:
        virtual ~CameraHardwareInterface() { }
        virtual sp<IMemoryHeap>                 getPreviewHeap() const = 0;
        virtual status_t     startPreview(preview_callback cb, void* user) = 0;
        virtual void                stopPreview() = 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;
};
```

使用 C 语言的方式导出符号：

```
extern "C" sp<CameraHardwareInterface> openCameraHardware();
```

在程序的其他地方，使用 openCameraHardware()就可以得到一个 CameraHardwareInterface，然后调用 CameraHardwareInterface 的接口完成 Camera 的功能。

3.1 JAVA 程序部分

在 packages/apps/Camera/src/com/android/camera/ 目录的 Camera.java 文件中，包含了对 Camera 的调用

在 Camera.java 中包含对包的引用：

```
import android.hardware.Camera.PictureCallback;
import android.hardware.Camera.Size;
```

在这里定义的Camera类继承了活动Activity类，在它的内部，包含了一个 android.hardware.Camera

```
public class Camera extends Activity implements View.OnClickListener, SurfaceHolder.Callback
{                          android.hardware.Camera mCameraDevice;
}
```

对Camera功能的一些调用如下所示：

```
mCameraDevice.takePicture(mShutterCallback, mRawPictureCallback, mJpegPictureCallback);
mCameraDevice.startPreview();
mCameraDevice.stopPreview();
```

startPreview、stopPreview 和takePicture等接口就是通过JAVA本地调用（JNI）来实现的。

frameworks/base/core/java/android/hardware/目录中的Camera.java文件提供了一个JAVA类：Camera。

```
public class Camera {
}
```

在这个类当中，大部分代码使用JNI调用下层得到，例如：

```
public void setParameters(Parameters params) {
        Log.e(TAG, "setParameters()");
        //params.dump();
        native_setParameters(params.flatten());
}
```

再者，例如以下代码：

```
public final void setPreviewDisplay(SurfaceHolder holder) {
        setPreviewDisplay(holder.getSurface());
}

private native final void setPreviewDisplay(Surface surface);
```

两个setPreviewDisplay参数不同，后一个是本地方法，参数为Surface类型，前一个通过调用后一个实现，但自己的参数以SurfaceHolder为类型。

3.2 Camera的JAVA本地调用部分

Camera的JAVA本地调用（JNI）部分在frameworks/base/core/jni/目录的android_hardware_Camera.cpp中的文件中实现。

android_hardware_Camera.cpp之中定义了一个JNINativeMethod（JAVA本地调用方法）类型的数组gMethods，如下所示：

```
static JNINativeMethod camMethods[] = {
```

```
{"native_setup","(Ljava/lang/Object;)V",(void*)android_hardware_Camera_native_setup },
    {"native_release","()V",(void*)android_hardware_Camera_release },
    {"setPreviewDisplay","(Landroid/view/Surface;)V",(void
*)android_hardware_Camera_setPreviewDisplay },
    {"startPreview","()V",(void *)android_hardware_Camera_startPreview },
    {"stopPreview", "()V", (void *)android_hardware_Camera_stopPreview },
    {"setHasPreviewCallback","(Z)V",(void *)android_hardware_Camera_setHasPreviewCallback },
    {"native_autoFocus","()V",(void *)android_hardware_Camera_autoFocus },
    {"native_takePicture", "()V", (void *)android_hardware_Camera_takePicture },
    {"native_setParameters","(Ljava/lang/String;)V",(void *)android_hardware_Camera_setParameters },

    {"native_getParameters", "()Ljava/lang/String;",(void *)android_hardware_Camera_getParameters }

};
```

JNINativeMethod 的第一个成员是一个字符串，表示了 JAVA 本地调用方法的名称，这个名称是在 JAVA 程序中调用的名称；第二个成员也是一个字符串，表示 JAVA 本地调用方法的参数和返回值；第三个成员是 JAVA 本地调用方法对应的 C 语言函数。

register_android_hardware_Camera 函数将 gMethods 注册为的类"android/media/Camera"，其主要的实现如下所示。

```
int register_android_hardware_Camera(JNIEnv *env)
{
    // Register native functions
    return AndroidRuntime::registerNativeMethods(env, "android/hardware/Camera",
                                                                                camMet
hods, NELEM(camMethods));
}
```

"android/hardware/Camera"对应 JAVA 的类 android.hardware.Camera。

3.3 Camera 本地库 libui.so

frameworks/base/libs/ui/中的 Camera.cpp 文件用于实现 Camera.h 提供的接口，其中一个重要的片段如下所示：

```
const sp<ICameraService>& Camera::getCameraService()
{
    Mutex::Autolock _l(mLock);
    if (mCameraService.get() == 0) {
        sp<IServiceManager> sm = defaultServiceManager();
        sp<IBinder> binder;
        do {
            binder = sm->getService(String16("media.camera"));
            if (binder != 0)
                break;
            LOGW("CameraService not published, waiting...");
            usleep(500000); // 0.5 s
        } while(true);
```

```
            if (mDeathNotifier == NULL) {
                    mDeathNotifier = new DeathNotifier();
            }
            binder->linkToDeath(mDeathNotifier);
            mCameraService = interface_cast<ICameraService>(binder);
    }
    LOGE_IF(mCameraService==0, "no CameraService!?");
    return mCameraService;
}
```

其中最重要的一点是 binder = sm->getService(String16("media.camera"));;这个调用用来得到一个名称为"media.camera"的服务,这个调用返回值的类型为 IBinder,根据实现将其转换成类型 ICameraService 使用。

一个函数 connect 的实现 如下所示：

```
sp<Camera> Camera::connect()
{
    sp<Camera> c = new Camera();
    const sp<ICameraService>& cs = getCameraService();
    if (cs != 0) {
            c->mCamera = cs->connect(c);
    }
    if (c->mCamera != 0) {
            c->mCamera->asBinder()->linkToDeath(c);
            c->mStatus = NO_ERROR;
    }
    return c;
}
```

connect 通过调用 getCameraService 得到一个 ICameraService，再通过 ICameraService 的 cs->connect(c)得到一个 ICamera 类型的指针。 调用 connect 将得到一个 Camera 的指针,正常情况下 Camera 的成员 mCamera 已经初始化完成。

一个具体的函数 startPreview 如下所示：

```
status_t Camera::startPreview()
{
    return mCamera->startPreview();
}
```

这些操作可以直接对 mCamera 来进行，它是 ICamera 类型的指针。

其他一些函数的实现也与 setDataSource 类似。

libmedia.so 中的其他一些文件与头文件的名称相同，它们是：

frameworks/base/libs/ui/ICameraClient.cpp

frameworks/base/libs/ui/ICamera.cpp

frameworks/base/libs/ui/ICameraService.cpp

在此处，BnCameraClient 和 BnCameraService 类虽然实现了 onTransact()函数，但是由于还有纯虚函数没有实现，因此这个类都是不能实例化的。

ICameraClient.cpp 中的 BnCameraClient 在别的地方也没有实现；而 ICameraService.cpp 中的 BnCameraService 类在别的地方被继承并实现，继承者实现了 Camera 服务的具体功能。

3.4 Camera 服务 libcameraservice.so

frameworks/base/camera/libcameraservice/ 用于实现一个 Camera 的服务，这个服务是继承 ICameraService 的具体实现。

在这里的 Android.mk 文件中，使用宏 USE_CAMERA_STUB 决定是否使用真的 Camera，如果宏为真，则使用 CameraHardwareStub.cpp 和 FakeCamera.cpp 构造一个假的 Camera，如果为假则使用 CameraService.cpp 构造一个实际上的 Camera 服务。

CameraService.cpp 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。

CameraService 和 CameraService::Client 两个类的结果如下所示：

```
class CameraService : public BnCameraService
{
    class Client : public BnCamera {};
    wp<Client>                              mClient;
}
```

在 CameraService 中的一个静态函数 instantiate()用于初始化一个 Camera 服务，寒暑如下所示：

```
void CameraService::instantiate() {
    defaultServiceManager()->addService( String16("media.camera"), new CameraService());
}
```

事实上，CameraService::instantiate()这个函数注册了一个名称为"media.camera"的服务，这个服务和 Camera.cpp 中调用的名称相对应。

Camera 整个运作机制是：在 Camera.cpp 中可以调用 ICameraService 的接口，这时实际上调用的是 BpCameraService，而 BpCameraService 又通过 Binder 机制和 BnCameraService 实现两个进程的通讯。而 BpCameraService 的实现就是这里的 CameraService。因此，Camera.cpp 虽然是在另外一个进程中运行，但是调用 ICameraService 的接口就像直接调用一样，从 connect()中可以得到一个 ICamera 类型的指针，真个指针的实现实际上是 CameraService::Client。

而这些 Camera 功能的具体实现，就是 CameraService::Client 所实现的了，其构造函数如下所示：

```
CameraService::Client::Client(const sp<CameraService>& cameraService,
            const sp<ICameraClient>& cameraClient) :
    mCameraService(cameraService), mCameraClient(cameraClient), mHardware(0)
{
    mHardware = openCameraHardware();
    mHasFrameCallback = false;
}
```

构造函数中，调用 openCameraHardware()得到一个 CameraHardwareInterface 类型的指针，并作为其成员 mHardware。以后对实际的 Camera 的操作都通过对这个指针进行。这是一个简单的直接调用关系。

事实上，真正的 Camera 功能已通过实现 CameraHardwareInterface 类来完成。在这个库当中 CameraHardwareStub.h 和 CameraHardwareStub.cpp 两个文件定义了一个桩模块的接口，在没有 Camera 硬件的情况下使用，例如在仿真器的情况下使用的文件就是 CameraHardwareStub.cpp 和它依赖的文件 FakeCamera.cpp。

CameraHardwareStub 类的结构如下所示：

```
class CameraHardwareStub : public CameraHardwareInterface {
       class PreviewThread : public Thread {
       };
};
```

在类CameraHardwareStub当中，包含一个线程类PreviewThread，这个线程用于处理PreView，即负责刷新取景器的内容。实际的Camera硬件接口通常可以通过对v4l2 捕获驱动的调用来实现，同时还需要一个JPEG编码程序将从驱动中取出的数据编码成JPEG文件。