

第十七期：Android 摄像头的应用

eoe 特刊

ANDROID

eoeAndroid
摄像头
的应用



Application of cameras

ANDROID
eoe 开发者门户



优 亿 市 场

目 录

【序 言】

【Android Camera 框架】

1.1	Android Camera 介绍	4
1.2	client 端	4
1.3	service 端	6
1.4	Camera HAL(硬件抽象层)	7
1.5	Preview 数据流程	8
1.6	模拟器中的虚拟 camera	9
1.7	框架图	9
1.8	Overlay 简单介绍	9

【不同硬件平台上移植 Android 的 Camera 系统】

2.1	硬件环境描述。	13
2.2	Camera 硬件系统分析	13
2.3	Sensor 驱动框架设计	14
2.4	Sensor 驱动移植	15
2.5	Camera 系统测试	15
2.6	参考	15

【分析 Android Camera】

3.1	Camera 概述	17
3.2	Camera 的接口与架构	17
3.3	Camera 的主要实现分析	24
3.4	Camera Architecture	29
3.5	Camera 工作流程概述	30
3.6	Camera 库文件分析	32

【Camera 应用程序框架】

4.1	Camera V4L2 应用程序框架	36
4.2	Camera Hardware Stub	40
4.3	Camera HAL 设计初步	42

【Camera 实例教程】

5.1	Android 实现摄像头拍照	46
5.2	摄像头采集视频	51
5.3	camera 应用层的应用	55

【Android camera 原文】

6.1	Android SDK Quick Tip: Launching the Camera	57
6.2	Photo Capture & Display	59

【其 他】

7.1	BUG 提交	61
7.2	关于 eoeAndroid	61
7.3	庆祝优亿市场新版本发布，征集反馈送大礼！	61

【序 言】



高铜，eoe Android 特刊邀请的技术达人为每期特刊进行校稿和审查，并撰写和推荐一些优秀的稿件。

为人正直、有耐心，腼腆的性格，让他对于喜欢的事业非常认真，并努力充实自己。逻辑性强，条理清晰，热心，善于解决问题。

在了解的过程中，会发现他并不是那种很善言谈的人，但工作中，每一个细节都会把控的很好。

“摄像头的应用，对于 Android 未来的发展，有着至关重要的作用，用户认证、条形码、SNS、内容分享等方面都会涉及到摄像头的应用。”高铜说，“作为一个开发者，我依旧看好 Android 这个行业，并坚定的为此而努力奋斗。”

他认为，eoeAndroid 特刊是将很多分散的内容整理的更为具体和详细，所以 eoeAndroid 特刊对于 android 开发者是非常好的学习资料。“我会继续为 eoeAndroid 特刊进行校稿和审查，帮助更多的开发者。”

——采访记者：莫言默语

eoeANDROID

【Android Camera 框架】

1.1 Android Camera 介绍 (eoe Android ID: guizhiwen)

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。

1.2 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_addCallbackBuffer [419]
☞ android_hardware_Camera_autoFocus [429]
☞ android_hardware_Camera_cancelAutoFocus [441]
☞ android_hardware_Camera_getParameters [484]
☞ android_hardware_Camera_lock [505]
☞ android_hardware_Camera_native_setup [292]
☞ android_hardware_Camera_previewEnabled [395]
☞ android_hardware_Camera_reconnect [493]
☞ android_hardware_Camera_release [328]
☞ android_hardware_Camera_setDisplayOrientation [554]
☞ android_hardware_Camera_setHasPreviewCallback [404]
☞ android_hardware_Camera_setParameters [466]
☞ android_hardware_Camera_setPreviewDisplay [359]
☞ android_hardware_Camera_startPreview [374]
☞ android_hardware_Camera_startSmoothZoom [527]
☞ android_hardware_Camera_stopPreview [386]
☞ android_hardware_Camera_stopSmoothZoom [543]
☞ android_hardware_Camera_takePicture [453]
☞ android_hardware_Camera_unlock [516]
```

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。

1.3 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();
}

```

1.4 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 ioctl 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。

1.5 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。

1.6 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

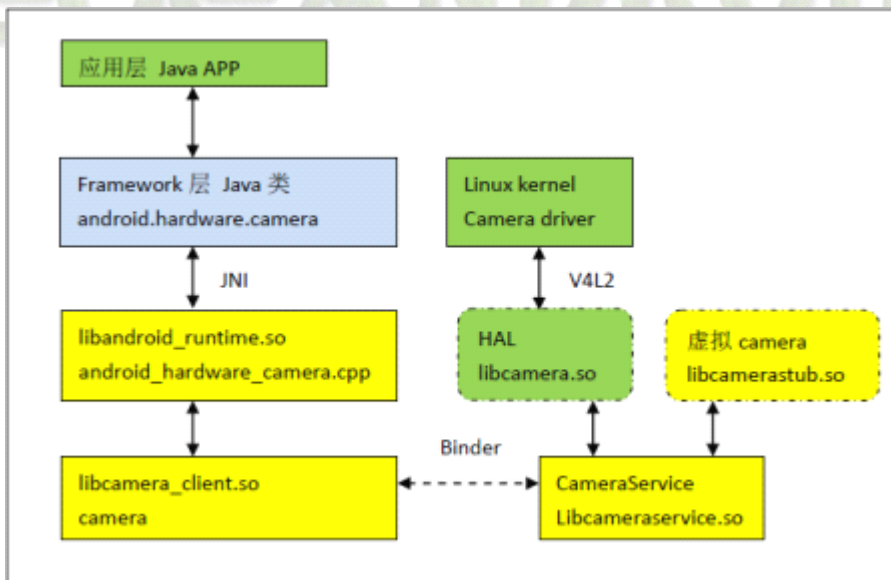
FakeCamera.cpp 文件提供虚拟的 preview 数据。CameraHardwareStub.cpp 文件中实现了 camera HAL(硬件抽象层)的功能。当宏 USE_CAMERA_STUB 为 true 时可以使用这个虚拟的 camera。

```

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

```

1.7 框架图



1.8 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。

1.8.1 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

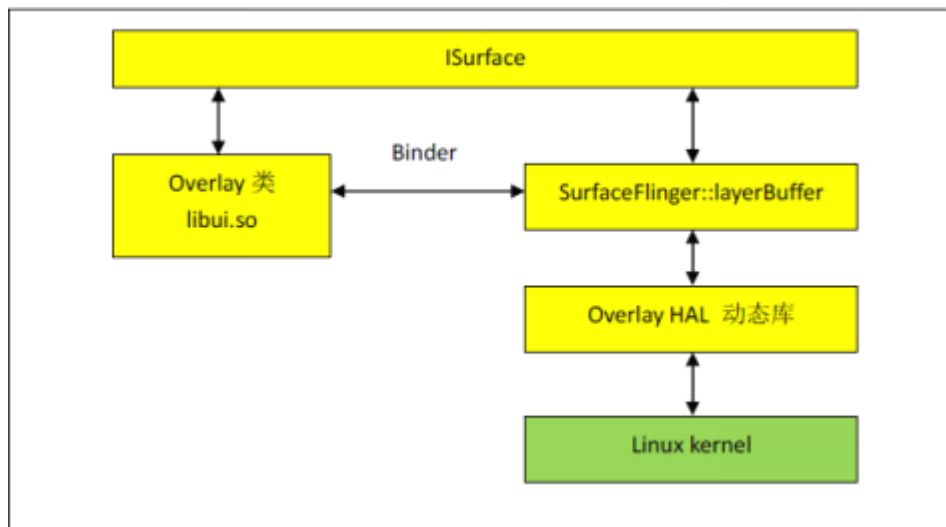
```
sp<Overlay> overlay = new Overlay(ref);
```

- 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。

1.8.2 Android overlay 框架



1.8.3 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。
- Overlay.cpp：定义提供 binder 所需的类。其中 LayerBuffer::OverlaySource::OverlayChannel 继承自 BnOverlay。

1.8.4 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。

1.8.5 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。

1.8.6 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 overlay 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。

【不同硬件平台上移植 Android 的 Camera 系统】

对于一个芯片公司，当一款芯片成型后，往往会根据不同客户的不同需求，设计不同的开发板卡，使用不同的外设，但它们的“芯”却是同一颗芯。这个芯片，我们假设代号为“AE300”，以这个为例，讲解遇到的六种情况。

2.1 硬件环境描述

当初使用芯片 AE300 做 Android 平台下 camera 移植时，可以自己设计开发板卡，使用了常用的 sensor。Sensor 的厂家很多，而且不同 sensor 都各有利弊。对于 sensor 的选择，一方面考虑芯片的支持能力，另一方面考虑客户的需求。这里，假设我们开发阶段使用的 sensor 型号分别为 SSA5526 和 SSA9076，SSA5526 做为后摄像头，200 万像素，SSA9076 做为前摄像头，30 万像素。目前，已经在这样的硬件环境下，将 Android 的 Camera 系统移植了上去。该项目叫做 Phone01。

假设有一家客户要使用我们的芯片 AE300，但是他们要用的 sensor 型号为：SSB0520 和 SSB0903。这两款 sensor 来自于另外一个厂家。它们的功能与我们开发阶段用的相似，像素值完全一样。但不同的是，每个厂家都有自己的寄存器地址选择和 bit 位意义，而且上电流程也未必一样。

从整个硬件平台上来看，由于所用芯片相同，只有 sensor 型号不同，所以可以说整个 Camera 系统的移植，实际上就是不同 sensor 驱动的移植。

2.2 Camera 硬件系统分析

从 sensor 本身的引脚来看，它们一般有如下一些需要配置的引脚：RESET，PWRDWN，VSYNC，HSYNC，PCLK，MCLK，SCA，SCL，AVDD，DVDD，IOVDD，还有就是数据引脚了。对于 30 万像素的 sensor 有 8 个数据引脚：D0——D7，对于 200 万像素的 sensor 有 10 个数据引脚：D0——D9。

这些引脚的意义大致如下：

RESET: 用来 reset sensor；RESET 一般是低有效，当脉冲为低时，reset sensor，而正常工作时，应该为高。SSA5526 中，其为低有效。

PWRDWN: power down 引脚，切断供电。PWRDWN 一般高有效，当脉冲为高时，进入省电模式，而正常工作时为低。但有些 sensor 却是低有效，比如 SSB0520。在使用不同 sensor 时，就需要注意这点。SSA5526 中，其为高有效。

HSYNC: 行同步，sensor 在抓取一行数据开始的时刻，通过 HSYNC 引脚向 Camera interface 发出信号，告知其。SSA5526 中，其为高有效。

VSYNC: 帧同步，这个与 HSYNC 对应，在 sensor 抓取一帧数据开始时，通过 VSYNC 引脚向 Camera interface 发出信号，告知其。SSA5526 中，其为低有效。

PCLK: 理解为 sensor 抓取一个像素的脉冲高低。SSA5526 中，其为低有效。

MCLK: sensor 工作的时钟频率。

SCA 和 SCL: 这是 I2C 的两条总线线路: SDA 为串行数据线, SCL 为串行时钟线。它们的工作原理, 可以参考注 1。

AVDD, DVDD, IOVDD 为电压引脚。AVDD 为模拟电源引脚, DVDD 和 IOVDD 为数字电源引脚。DVDD 和 IOVDD 的关系有待确认。AVDD 和 IOVDD 的关系参考注 2。

根据 sensor 自身的引脚, 我们就可以推想出芯片中 Camera 部分的引脚。如果芯片只支持一个 Camera, 那芯片只需要上述的引脚即可。但是当芯片支持双 Camera 时, 就需要额外的引脚。PWRDWN 需要两个引脚, 分别控制不同的 sensor, 这是因为不同 sensor 的 PWRDWN 有效脉冲不一样。另外, 数据传输引脚采用 CCIR656 标准, 统一设计了 8 个数据引脚。

2.3 Sensor 驱动框架设计

由于不同客户对 Sensor 的需求不同, 可能随时更换不同型号的 sensor, 所以应该设计一套统一的 sensor 驱动接口。当 sensor 型号发生变化时, 只需更换对应的文件即可。调用 sensor 驱动的程序不用变化。

鉴于此, 可以设计如下几个文件: sensor_driver.c/sensor_driver.h 用于定义并实现 sensor 的统一接口。Sensor_SSA5526.c/ Sensor_SSA9076.c/ Sensor_SSB0520.c/ Sensor_SSB0903.c 用于不同型号的 sensor 去实现支持 sensor 的统一接口。

在文件 sensor_driver.h 中, 定义一个可以描述 sensor 所有信息的结构体 struct sensor_info_t。它包含成员:

- 1、I2C 读写地址, SSA5526 中, I2C 写地址为 0x60, 读地址为 0x61;
- 2、RESET, PWRDWN, VSYNC, HSYNC, PCLK 的有效电平配置;
- 3、AVDD, DVDD, IOVDD 的电压值配置;
- 4、Sensor 型号识别信息;
- 5、Sensor 所支持的捕获图像的最大长和宽, 以及图像的格式等;
- 6、Sensor 支持不同长和宽捕获图像数据时的相应寄存器配置;
- 7、Sensor 支持的特效的函数接口, 比如白平衡调节, 亮度变化, 饱和度变化, 防闪烁等;

这些信息包含了整个 Sensor 的所有属性及功能, 通过获取该结构体, 对其操作, 可以控制该 Sensor。

定义两个这样的结构体:

```
struct sensor_info_t sensor_front_t;
struct sensor_info_t sensor_back_t;
```

文件 Sensor_SSA5526.c/ Sensor_SSB0520.c 去实现 sensor_back_t, 作为后摄像头; 文件 Sensor_SSA9076.c/ Sensor_SSB0903.c 去实现 sensor_front_t, 作为前摄像头。在文件 sensor_driver.c 使用 sensor_front_t 和 sensor_back_t 即可。

在文件 `sensor_driver.h` 中也定义了供 V4L2 驱动所调用的 `sensor` 统一接口, 包含如下接口:

- 1、`Int sensor_init(uint32_t sensor_id);`根据场景, 选择前或后 `sensor`, 进行初始化;
- 2、`Int sensor_deinit(void);`注销当前使用的 `sensor`;
- 3、`Int sensor_ioctl(uint32_t cmd, uint32_t arg);`通过 `IOCTRL`, 调用 `sensor` 的特效函数;
- 4、`Int sensor_set_size(uint32_t width, uint32_t height);`设置 `sensor` 所要抓取的图像的大小;

这几个接口在 `sensor_driver.c` 中实现。

项目不同, 使用的 `sensor` 也不同。在 `Makefile` 中, 可以根据项目名称来选择编译哪些型号的 `sensor` 文件。比如, 当前项目为 `PHONE02`, 使用的 `sensor` 为 `SSB0520` 和 `SSB0903`, 以前的项目为 `Phone01`, 使用的 `sensor` 为 `SSA5526` 和 `SSA9076`, 就可以这样写:

```
Obj-$(CONFIG_PHONE001) += SSA5526.o SSA9076.o
Obj-$(CONFIG_PHONE002) += SSB0520.o SSB0903.o
```

当编译不同项目时, 就会编译对应的 `sensor` 驱动文件。

2.4 Sensor 驱动移植

根据上面描述的 `Sensor` 驱动框架设计, 在移植 `sensor` 驱动时, 需要做的就是实现 `SSB0520.c/SSB0903.c`, 然后修改 `Makefile` 文件即可。对于文件 `SSB0520.c/SSB0903.c` 的实现, 应该参考 `SSA5526.c/SSA9076.c` 的实现, 使所有的 `sensor` 驱动文件都保持同样的风格, 有利于以后的驱动修改。

2.5 Camera 系统测试

`Sensor` 驱动移植完了, 别的文件都不用修改, 直接编译。等新的 `code` 下载到新的板卡上去后, 就可以开机进行 `Camera` 测试了。对于新的 `sensor`, 我们必须进行全面的测试, 以确保 `sensor` 驱动的正确性。

2.6 参考

2.6.1 [注 1] I2C 工作原理

ISDA 数据, SCL 时钟. 时钟是单片机工作的必要条件之一. 单片机的工作必要条件有三个, 电源, 时钟, 复位电路. SDA 是单片机工作时传输各种数据的接口。

I2C 总线是一种串行数据总线, 只有二根信号线, 一根是双向的数据线 SDA, 另一根是时钟线 SCL. 在 I2C 总线上传送的一个数据字节由八位组成. 总线对每次传送的字节数没有

限制，但每个字节后必须跟一位应答位。数据传送首先传送最高位 (MSB)。首先由主机发出启动信号“S” (SDA 在 SCL 高电平期间由高电平跳变为低电平)，然后由主机发送一个字节的数

据。启动信号后的第一个字节数据具有特殊含义：高七位是从机的地址，第八位是传送方向位，0 表示主机发送数据 (写)，1 表示主机接收数据 (读)。被寻址到的从机设备按传送方向位设置为对应工作方式。标准 I2C 总线的设备都有一个七位地址，所有连接在 I2C 总线上的设备都接收启动信号后的第一个字节，并将接收到的地址与自己的地址进行比较，如果地址相符则为主机要寻访的从机，应答在第九位时钟脉冲时向 SDA 线送出低电平作为应答。除了第一字节是通用呼叫地址之外第二字节开始即数据字节。数据传送完毕，由主机发出停止信号“P” (SDA 在 SCL 高电平期间由低电平跳变为高电平)。

2.6.2 【注 2】AVDD 和 IOVDD 的关系

AVDD 是模拟电源引脚，IOVDD 是数字电源引脚，它们应该分别供电，即最好不要采用一个电源供电，在不得不采用一个电源供电时，需要采取隔离、退耦措施，以避免数据电路电源上的噪声对模拟电路通过模拟电源产生干扰。在采用两个电源分别对模拟电路和数字电路分别供电时，可以采用不同的电源电压。如对模拟电路供电的电源电压为 3.0V 而对数字电路供电的电源电压为 3.3V；或者反过来。

eoeANDROID

【分析 Android Camera】

3.1 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端两个部分, 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 Android 的框架。

以开源的 Android 为例, Camera 的代码主要在以下的目录中:

1、Camera 的 JAVA 程序的路径:

A、[packages/apps/Camera/src/com/android/camera/](#)

在其中 Camera.java 是主要实现的文件

B、[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。

2、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](#)。

3、主要的头文件在以下的目录中:

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

4、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](#)。

5、Camera 服务部分:

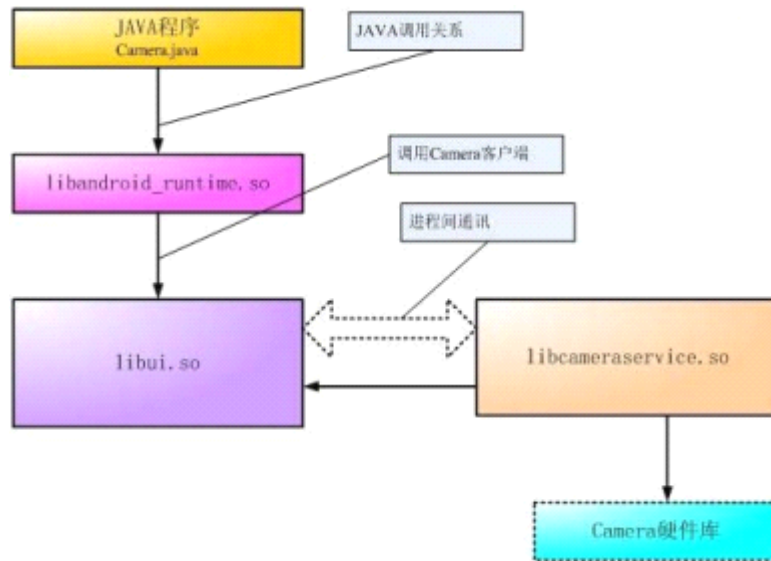
[frameworks/base/camera/libcameraservice/](#)

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 Camera 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。

3.2 Camera 的接口与架构

3.2.1 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。

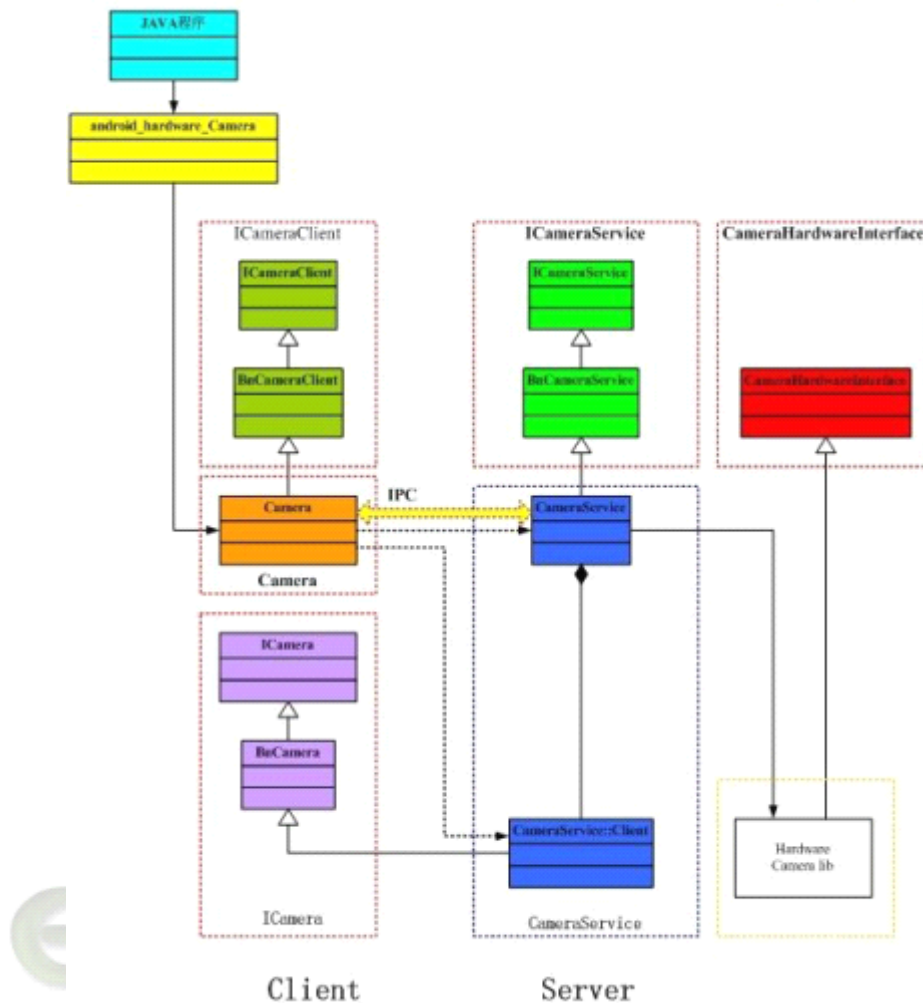
libandroid_runtime.so 和 libui.so 两个库是公用的，其中除了 Camera 还有其他方面的功能。

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 MediaPlayer 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 Binder 来实现进程间的通讯。

3.2.2 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。

3.2.3 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 类模板的定义 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。

3.2.4 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。

3.2.5 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。

3.2.6 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 定义 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。

3.3 Camera 的主要实现分析

3.3.1 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。

3.3.2 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",
                                                camMethods, NELEM(camMethods));
}

```

“android/hardware/Camera”对应 JAVA 的类 android.hardware.Camera。

3.3.3 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。

3.3.4 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。

3.4 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：

- 1、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。
- 2、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。
- 3、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 MediaPlayer 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 Binder 来实现进程间的通讯。

3.5 Camera 工作流程概述

3.5.1 Camera Service 的启动

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 },
}
```

```

        (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

3.5.1 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 icle) {
    super.onCreate(icle);
    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。

3.6 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 Camera 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：

```
LOCAL_PATH:= $(call my-dir)

#
# Set USE_CAMERA_STUB for non-emulator and non-simulator builds, if you want
# the camera service to use the fake camera. For emulator or simulator builds,
# we always use the fake camera.

ifeq ($(USE_CAMERA_STUB),)
USE_CAMERA_STUB:=false
ifeq ($(filter sooner generic sim,$(TARGET_DEVICE)),)
USE_CAMERA_STUB:=true
endif #libcamerastub
endif

ifeq ($(USE_CAMERA_STUB),true)
#
# libcamerastub
#

include $(CLEAR_VARS)

LOCAL_SRC_FILES:= \
```



```

    CameraHardwareStub.cpp      \
    FakeCamera.cpp

LOCAL_MODULE:= libcamerastub

LOCAL_SHARED_LIBRARIES:= libui

include $(BUILD_STATIC_LIBRARY)
endif # USE_CAMERA_STUB

#
# libcameraservice
#

include $(CLEAR_VARS)

LOCAL_SRC_FILES:=          \
    CameraService.cpp

LOCAL_SHARED_LIBRARIES:= \
    libui \
    libutils \
    libcutils \
    libmedia

LOCAL_MODULE:= libcameraservice

LOCAL_CFLAGS+=-DLOG_TAG=\"CameraService\"

ifeq ($(USE_CAMERA_STUB), true)
LOCAL_STATIC_LIBRARIES += libcamerastub
LOCAL_CFLAGS += -include CameraHardwareStub.h
else
LOCAL_SHARED_LIBRARIES += libcamera
endif

include $(BUILD_SHARED_LIBRARY)

```

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” ）；

```
class CameraHardwareInterface : public virtual RefBase {
public:
    virtual ~CameraHardwareInterface() { }
    virtual sp<IMemoryHeap>      getPreviewHeap() const;
    virtual sp<IMemoryHeap>      getRawHeap() const;
    virtual status_t      startPreview(preview_callback cb, void* user);
    virtual bool useOverlay() {return false;}
    virtual status_t setOverlay(const sp<Overlay> &overlay) {return BAD_VALUE;}
    virtual void      stopPreview();
    virtual bool      previewEnabled();
    virtual status_t      startRecording(recording_callback cb, void* user);
    virtual void      stopRecording();
    virtual bool      recordingEnabled();
    virtual void      releaseRecordingFrame(const sp<IMemory>& mem);
    virtual status_t      autoFocus(autofocus_callback, void* user);
    virtual status_t      takePicture(shutter_callback, raw_callback,
                                      jpeg_callback, void* user);
    virtual status_t      cancelPicture(bool cancel_shutter, bool cancel_raw,
                                      bool cancel_jpeg);
    virtual CameraParameters getParameters() const;
    virtual void release();
    virtual status_t dump(int fd, const Vector<String16>& args) const ;
};
```

- 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；例如：

```
extern "C" sp<CameraHardwareInterface> openCameraHardware()
{
    CameraHardwareInterface realCamera;
    return &realCamera;
}
```

```
}

```

- 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:

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE := libcamera

LOCAL_SHARED_LIBRARIES := \
    libutils \
    librpc \
    liblog

LOCAL_SRC_FILES += MyCameraHardware.cpp

LOCAL_CFLAGS +=

LOCAL_C_INCLUDES +=

LOCAL_STATIC_LIBRARIES += \
    libcamera-common \
    libclock-rpc \
    libcommondefs-rpc
include $(BUILD_SHARED_LIBRARY)
```

- 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)

libcamera.so 库直接操作 Camera 设备, 这样相对于 libcamera.so 库包含了 Camera 驱动, 而右图则将驱动从库中分离出来并形成一层 HAL 这样做的好处是: 移植不同型号或不同厂商的同类设备时只需修改 HAL 中很少代码即可。

【Camera 应用程序框架】

4.1 Camera V4L2 应用程序框架

V4L2 较 V4L 有较大的改动，并已成为 2.6 的标准接口，涵盖 video\dev\FM...，多数驱动都在向 V4L2 迁移。更好地了解 V4L2 先从应用入手，然后再深入到内核中结合物理设备 / 接口的规范实现相应的驱动。本文先就 V4L2 在视频捕捉或 camera 方面的应用框架。

V4L2 采用流水线的方式，操作更简单直观，基本遵循打开视频设备、设置格式、处理数据、关闭设备，更多的具体操作通过 ioctl 函数来实现。

4.1.1 打开视频设备

在 V4L2 中，视频设备被看做一个文件。使用 open 函数打开这个设备：

```
// 用非阻塞模式打开摄像头设备
int cameraFd;

cameraFd = open("/dev/video0", O_RDWR | O_NONBLOCK, 0);
// 如果用阻塞模式打开摄像头设备，上述代码变为：
// cameraFd = open("/dev/video0", O_RDWR, 0);
```

应用程序能够使用阻塞模式或非阻塞模式打开视频设备，如果使用非阻塞模式调用视频设备，即使尚未捕获到信息，驱动依旧会把缓存（DQBUF）里的东西返回给应用程序。

4.1.2 设定属性及采集方式

打开视频设备后，可以设置该视频设备的属性，例如裁剪、缩放等。这一步是可选的。在 Linux 编程中，一般使用 ioctl 函数来对设备的 I/O 通道进行管理：

```
int ioctl(int __fd, unsigned long int __request, ... /*args*/);
```

- 在进行 V4L2 开发中，常用的命令标志符如下 (some are optional)：
 - VIDIOC_REQBUFS：分配内存
 - VIDIOC_QUERYBUF：把 VIDIOC_REQBUFS 中分配的数据缓存转换成物理地址
 - VIDIOC_QUERYCAP：查询驱动功能
 - VIDIOC_ENUM_FMT：获取当前驱动支持的视频格式
 - VIDIOC_S_FMT：设置当前驱动的帧捕获格式
 - VIDIOC_G_FMT：读取当前驱动的帧捕获格式
 - VIDIOC_TRY_FMT：验证当前驱动的显示格式
 - VIDIOC_CROPCAP：查询驱动的修剪能力
 - VIDIOC_S_CROP：设置视频信号的边框
 - VIDIOC_G_CROP：读取视频信号的边框
 - VIDIOC_QBUF：把数据从缓存中读取出来
 - VIDIOC_DQBUF：把数据放回缓存队列
 - VIDIOC_STREAMON：开始视频显示函数
 - VIDIOC_STREAMOFF：结束视频显示函数

- VIDIOC_QUERYSTD: 检查当前视频设备支持的标准, 例如 PAL 或 NTSC。

1、检查当前视频设备支持的标准

在亚洲, 一般使用 PAL(720X576)制式的摄像头, 而欧洲一般使用 NTSC(720X480), 使用 VIDIOC_QUERYSTD 来检测:

```
v4l2_std_id std;
do{
    ret= ioctl(fd, VIDIOC_QUERYSTD, &std);
} while (ret== -1 && errno== EAGAIN);
switch(std) {
    case V4L2_STD_NTSC:
        //.....
    case V4L2_STD_PAL:
        //.....
}
```

2、设置视频捕获格式

当检测完视频设备支持的标准后, 还需要设定视频捕获格式, 结构如下:

```
struct v4l2_format fmt;
memset(&fmt, 0, sizeof(fmt));
fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
fmt.fmt.pix.width = 720;
fmt.fmt.pix.height = 576;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;
fmt.fmt.pix.field = V4L2_FIELD_INTERLACED;
if(ioctl(fd, VIDIOC_S_FMT, &fmt) == -1) {
    return -1;
}
```

v4l2_format 结构如下:

```
struct v4l2_format
{
    enum v4l2_buf_type type;    // 数据流类型, 必须永远是 V4L2_BUF_TYPE_VIDEO_CAPTURE
    union
    {
        struct v4l2_pix_format pix;
        struct v4l2_window win;
        struct v4l2_vbi_format vbi;
        __u8 raw_data[200];
    } fmt;
};
```

```

struct v4l2_pix_format
{
    __u32          width;          // 宽，必须是 16 的倍数
    __u32          height;         // 高，必须是 16 的倍数
    __u32          pixelformat;    // 视频数据存储类型，例如是 YUV4:2:2 还是 RGB
    enum v4l2_field field;
    __u32          bytesperline;
    __u32          sizeimage;
    enum v4l2_colorspace colorspace;
    __u32          priv;
};

```

3、分配内存

接下来可以为视频捕获分配内存：

```

struct v4l2_requestbuffers req;
if (ioctl(fd, VIDIOC_REQBUFS, &req) == -1) {
    return -1;
}

```

v4l2_requestbuffers 结构如下：

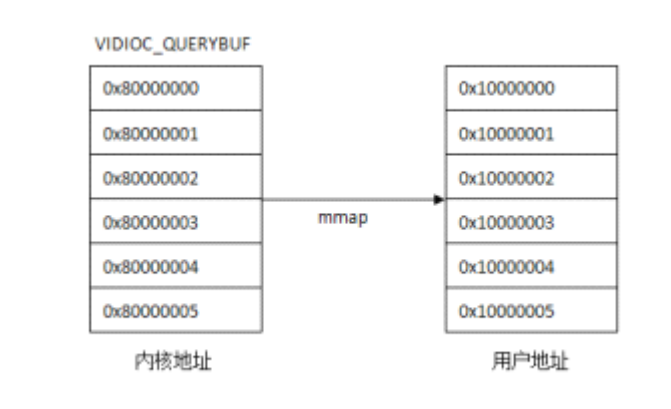
```

struct v4l2_requestbuffers
{
    __u32          count;          // 缓存数量，也就是说在缓存队列里保持多少张照片
    enum v4l2_buf_type type;       // 数据流类型，必须永远是 V4L2_BUF_TYPE_VIDEO_CAPTURE
    enum v4l2_memory memory;       // V4L2_MEMORY_MMAP 或 V4L2_MEMORY_USERPTR
    __u32          reserved[2];
};

```

4、获取并记录缓存的物理空间

使用 VIDIOC_REQBUFS，我们获取了 req.count 个缓存，下一步通过调用 VIDIOC_QUERYBUF 命令来获取这些缓存的地址，然后使用 mmap 函数转换成应用程序中的绝对地址，最后把这段缓存放入缓存队列：



```

typedef struct VideoBuffer{
    void *start;
    size_t length;
} VideoBuffer;

VideoBuffer*
buffers= calloc( req.count, sizeof(*buffers) );
struct v4l2_buffer buf;
for(numBufs= 0; numBufs< req.count; numBufs++) {
    memset( &buf, 0, sizeof(buf) );
    buf.type= V4L2_BUF_TYPE_VIDEO_CAPTURE;
    buf.memory= V4L2_MEMORY_MMAP;
    buf.index= numBufs;
    // 读取缓存
    if(ioctl(fd, VIDIOC_QUERYBUF, &buf) == -1) {
        return -1;
    }

    buffers[numBufs].length= buf.length;
    // 转换成相对地址
    buffers[numBufs].start= mmap(NULL, buf.length, PROT_READ|PROT_WRITE,
        MAP_SHARED, fd, buf.m.offset);

    if(buffers[numBufs].start== MAP_FAILED) {
        return -1;
    }

    // 放入缓存队列
    if(ioctl(fd, VIDIOC_QBUF, &buf) == -1) {
        return -1;
    }
}

```

5、视频采集方式

操作系统一般把系统使用的内存划分成用户空间和内核空间，分别由应用程序管理和操作系统管理。应用程序可以直接访问内存的地址，而内核空间存放的是供内核访问的代码和数据，用户不能直接访问。v4l2 捕获的数据，最初是存放在内核空间的，这意味着用户不能直接访问该段内存，必须通过某些手段来转换地址。

一共有三种视频采集方式：使用 read、write 方式；内存映射方式和用户指针模式。

read、write 方式：在用户空间和内核空间不断拷贝数据，占用了大量用户内存空

间，效率不高。

内存映射方式：把设备里的内存映射到应用程序中的内存控件，直接处理设备内存，这是一种有效的方式。上面的 mmap 函数就是使用这种方式。

用户指针模式：内存片段由应用程序自己分配。这点需要在 v4l2_requestbuffers 里将 memory 字段设置成 V4L2_MEMORY_USERPTR。

6、处理采集数据

V4L2 有一个数据缓存，存放 req.count 数量的缓存数据。数据缓存采用 FIFO 的方式，当应用程序调用缓存数据时，缓存队列将最先采集到的视频数据缓存送出，并重新采集一张视频数据。这个过程需要用到两个 ioctl 命令,VIDIOC_DQBUF 和 VIDIOC_QBUF:

```
struct v4l2_buffer buf;
memset(&buf, 0, sizeof(buf));
buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.memory = V4L2_MEMORY_MMAP;
buf.index = 0;

//读取缓存
if(ioctl(cameraFd, VIDIOC_DQBUF, &buf) == -1)
{
    return -1;
}
//.....视频处理算法
//重新放入缓存队列
if(ioctl(cameraFd, VIDIOC_QBUF, &buf) == -1)
{
    return -1;
}
```

4.1.3 关闭视频设备

使用 close 函数关闭一个视频设备

```
close(cameraFd)
```

如果使用 mmap, 最后还需要使用 munmap 方法。

4.2 Camera Hardware Stub

CameraHardwareStub 是 Android 提供的一个 fake camera 工具。其代码主要包含三个部分: init、preview 和 picture。

4.2.1 Init

CameraHardwareStub() () 构造函数:

```
initDefaultParameters()

setParameters(const CameraParameters& params)

initHeapLocked()
{
    mRawHeap = new MemoryHeapBase(picture_width * 2 * picture_height);
    mPreviewHeap = new MemoryHeapBase(mPreviewFrameSize * kBufferCount);
    mBuffers[i] = new MemoryBase(mPreviewHeap, i * mPreviewFrameSize, mPreviewFrameSize);
    mFakeCamera = new FakeCamera(preview_width, preview_height);
}
```

4.2.2 Preview

```
startPreview()

PreviewThread(this)

    previewThread()
    {
        mlock
        init...
        munlock
        fakeCamera->getNextFrameAsYuv422(frame);
    }
```

4.2.3 Picture

Picture 又可以分为 autofocus 和 picture 两个部分, autofocus 暂时不看, 无法实现。

```
takePicture()

beginPictureThread(void *cookie)

    pictureThread()
    {
        case CAMERA_MSG_RAW_IMAGE
            cam.getNextFrameAsYuv422((uint8_t *)mRawHeap->base());
        case CAMERA_MSG_COMPRESSED_IMAGE
```

```
memcpy(heap->base(), kCannedJpeg, kCannedJpegSize);
}
```

4.2.4 FakeCamera.cpp 中，除了本身的构造函数，只有 getNextFrameAsYuv422 经常被调用，所以如果从真实摄像头获取一帧数据，应该也可以。

4.3 Camera HAL 设计初步

使用 zc301 USB 摄像头，这个摄像头返回 JPEG 图形留，camera 的 preview 需要进行 jpeg 解码（没做），但是可以直接 take jpeg 照片。

4.3.1 修改你的 BoardConfig.mk

```
USE_CAMERA_STUB := false
```

将 stub 设置为 false，在编译时不会编译
android2.1/frameworks/base/camera/libcameraservice 中的
CameraHardwareStub.cpp
CameraHardwareStub.h
FakeCamera.cpp
FakeCamera.h
几个文件

4.3.2 hardware 下建立 Camera HAL 目录。

android2.1/hardware/your board/libcamera 复制以上几个文件。
CameraHardwareStub.cpp
CameraHardwareStub.h
FakeCamera.cpp
akeCamera.h

可以将其重命名
S3C6410CameraHardware.cpp
UsbCamera.cpp

文件中的 Fake 和 Stub 同样可以替换

4.3.3 编写 Android.mk 文件

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_SRC_FILES := S3C6410CameraHardware.cpp \
    UsbCamera.cpp
LOCAL_C_INCLUDES := \
    external/jpeg
LOCAL_SHARED_LIBRARIES := libutils libbinder libui liblog
```

```
LOCAL_STATIC_LIBRARIES:= \
    libjpeg
LOCAL_MODULE:= libcamera
include $(BUILD_SHARED_LIBRARY)
```

其中 jpeg 库是为了将来解码 jpeg 使用的，根据这个脚本可编译出 libcamera.so

4.3.4 修改 FakeCamera.cpp→UsbCamera.cpp

目前只实现了基本功能，该文件可以按照 V4L2 流程来写。网上也有现成的 patch，使用 mmap 方式，可惜是 1.6 的，没编译过去，待研究。

1、构造函数中进行初始化。

```
UsbCamera::UsbCamera(int width, int height)
    : mTmpRgb16Buffer(0)
{
    fd = open (DEFAULT_DEVICE, O_RDWR /* required */ | O_NONBLOCK, 0);
    LOGE("open /dev/video0 fd is %d", fd);
    ioctl (fd, VIDIOC_QUERYCAP, &cap);
    CLEAR (fmt);
    fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    fmt.fmt.pix.width = 640;
    fmt.fmt.pix.height = 480;
    fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;
    fmt.fmt.pix.field = V4L2_FIELD_INTERLACED;
    ioctl (fd, VIDIOC_S_FMT, &fmt);
    length = fmt.fmt.pix.bytesperline * fmt.fmt.pix.height;
}
```

2、获取一帧数据

```
void UsbCamera::getNextFrameAsYuv422(uint8_t *buffer)
{
    LOGE("read to get a pic from camera!");
    for (;;)
    {
        fd_set fds;
        struct timeval tv;
        int r;
        FD_ZERO (&fds);
        FD_SET (fd, &fds);
        /* Timeout. */
        tv.tv_sec = 3;
        tv.tv_usec = 0;
```

```

r = select (fd + 1, &fds, NULL, NULL, &tv);
if (-1 == r) {
    LOGE ("select"); }
if (0 == r) {
    LOGE ("select timeout\n"); }
if (read (fd, buffer, length));
break;
}
}

```

3、析构函数。

```

UsbCamera::~UsbCamera()
{
    delete[] mTmpRgb16Buffer;
    close (fd);
}

```

4.3.4 修改 CameraHardwareStub.cpp→S3C6410CameraHardware.cpp

1、改成 640x480。

```

void CameraHardware::initDefaultParameters()
{
    CameraParameters p;
    p.setPreviewSize(640,480);
    p.setPreviewFrameRate(1);
    p.setPreviewFormat("yuv422sp");//("yuv422sp");
    p.setPictureSize(640, 480);
    p.setPictureFormat("jpeg");//("jpeg");
    if (setParameters(p) != NO_ERROR) {
        LOGE("Failed to set default parameters?!");
    }
}

```

2、拍照部分要改，因为可以直接获取压缩的 jpeg 图片。

```

int CameraHardware::pictureThread()
{
    UsbCamera* usbCamera = mUsbCamera;

    if (mMsgEnabled & CAMERA_MSG_SHUTTER)
        mNotifyCb(CAMERA_MSG_SHUTTER, 0, 0, mCallbackCookie);
    if (mMsgEnabled & CAMERA_MSG_RAW_IMAGE) {
        //FIXME: use a canned YUV image!
        // In the meantime just make another fake camera picture.
        //int w, h;
    }
}

```

```

        //mParameters.getPictureSize(&w, &h);
        //sp<MemoryBase> mem = new MemoryBase(mRawHeap, 0, w * 2 * h);
        LOGE("CAMERA_MSG_RAW_IMAGE");
        //UsbCamera cam(w, h);
        //cam.getNextFrameAsYuv422((uint8_t *)mRawHeap->base());
        //mDataCb(CAMERA_MSG_RAW_IMAGE, mem, mCallbackCookie);
    }
    if (mMsgEnabled & CAMERA_MSG_COMPRESSED_IMAGE) {
        //sp<MemoryHeapBase> heap = new MemoryHeapBase(20000);
        //sp<MemoryBase> mem = new MemoryBase(heap, 0, 20000);
        //memcpy(heap->base(), kCannedJpeg, 20000);
        LOGE("CAMERA_MSG_COMPRESSED_IMAGE");
        int w, h;
        mParameters.getPictureSize(&w, &h);
        sp<MemoryBase> mem = new MemoryBase(mRawHeap, 0, w * 2 * h);
        //UsbCamera cam(w, h);
        usbCamera->getNextFrameAsYuv422((uint8_t *)mRawHeap->base());
        mDataCb(CAMERA_MSG_COMPRESSED_IMAGE, mem, mCallbackCookie);
    }
    return NO_ERROR;
}

```

3、status_t CameraHardware::setParameters(const CameraParameters& params)有个地方只让 take 320x240 的 pic，要注释掉。

```

/*  if (w != 320 && h != 240) {
        LOGE("Still picture size must be size of canned JPEG (%dx%d)",
            320, 240);
        return -1;
    }*/

```

至此 Camera HAL 已经可以拍照了。

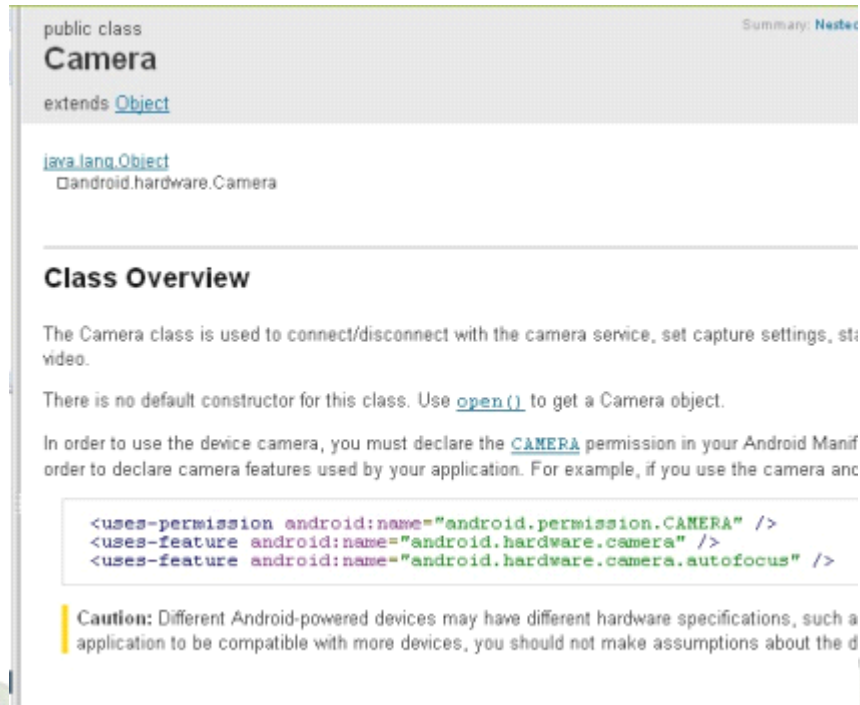
存在问题：

- 1) 需要做 jpeg->YUV 以实现 preview 功能，libjpeg 没用过，暂时不做。
- 2) UsbCamera.cpp 要改成 V4L2 标准流程，现在这种 read 模式太简单，效率低。

【Camera 实例教程】

5.1 Android 实现摄像头拍照 (eoeAndroid ID: liujunhui)

5.1.1 在 androidSDK 文档搜索 camera。内容已经描述了这个类的作用。结果如下：



There is no default constructor for this class. Use `open()` to get a `Camera` object.

翻译：这个类没有定义构造函数，用 `open()` 这个方法来得对象。这是一种设计模式，属于建造者模式类型。

In order to use the device camera, you must declare the `CAMERA` permission in your `AndroidManifest`. Also be sure to include the `<uses-feature>` manifest element in order to declare camera features used by your application. For example, if you use the camera and auto-focus feature, your `Manifest` should include the following:

简单来说，就是使用这个功能就要在 `Manifest` 中声明权限，就在 `Manifest` 文件中加入：

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-feature android:name="android.hardware.camera" />
<uses-feature android:name="android.hardware.camera.autofocus" />
```

到此，就明确了 `camera` 这个类照相一定要使用 `android.hardware.Camera` 这个类。要拍照，肯定要用到取景画面的 `SurfaceView` 控件，这个控件很多人都用过，所以这里简单的说下。

public class **SurfaceView**
extends [View](#)

[Summary](#) | [Inherited XML Attrs](#) | [Inherited Constants](#) | [Ctors](#) | [Method](#)

[java.lang.Object](#)
[android.view.View](#)
 android.view.SurfaceView

► Known Direct Subclasses
 GLSurfaceView, VideoView

Class Overview

Provides a dedicated drawing surface embedded inside of a view hierarchy. You can control the format of this surface and, care of placing the surface at the correct location on the screen

The surface is Z ordered so that it is behind the window holding its SurfaceView; the SurfaceView punches a hole in its win view hierarchy will take care of correctly compositing with the Surface any siblings of the SurfaceView that would normally place overlays such as buttons on top of the Surface, though note however that it can have an impact on performance since performed each time the Surface changes

SurfaceView 需要一个 surfaceHolder。它是系统提供的一个用来设置 surfaceView 的一个对象，通过 surfaceView.getHolder() 这个方法来获得。而 Camera 提供一个 setPreviewDisplay(SurfaceHolder) 的方法来连接 surfaceHolder，并通过它来控制 surfaceView。我们使用 android 的 Camera 类提供了 startPreview() 和 stopPreview() 来开启和关闭预览。

SurfaceHolder.Callback, 是 holder 用来显示 surfaceView 数据的接口, 它分别必须实现 3 个方法:

surfaceCreated()	当 surface 被创建后调用。
surfaceChanged()	当 surfaceView 发生改变后调用。
surfaceDestroyed()	当 surfaceView 销毁时进行调用。

surfaceHolder 通过 addCallback() 方法将响应的接口绑定到他身上。

5.1.2 要想照相，必须用到 SurfaceView 这个类。链接这两个类的中间桥梁是 SurfaceHolder。

我们现在来看代码：

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    Window window = getWindow();

    requestWindowFeature(Window.FEATURE_NO_TITLE); // 没有标题
    window.setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN, WindowManager.
LayoutParams.FLAG_FULLSCREEN); // 设置全屏

    window.addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON); // 高亮

    setContentView(R.layout.main);

    surfaceView = (SurfaceView) this.findViewById(R.id.surfaceView);
    surfaceView.getHolder().addCallback(new SurfaceListener());
}
```

```

/*下面设置 Surface 不维护自己的缓冲区，而是等待屏幕的渲染引擎将内容推送到用户面前*/
surfaceView.getHolder().setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
surfaceView.getHolder().setFixedSize(176, 144);    //设置分辨率
}

```

以上代码主要是对界面进行初始化和属性设置。

```

private final class SurfaceListener implements SurfaceHolder.Callback{
    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) {
    }
    @Override
    public void surfaceCreated(SurfaceHolder holder) {
        try {
            camera = Camera.open(); //打开摄像头
            Camera.Parameters parameters = camera.getParameters();
            WindowManager wm = (WindowManager) getSystemService(Context.WINDOW_SERVICE);
            Display display = wm.getDefaultDisplay();
            parameters.setPreviewSize(display.getWidth(), display.getHeight());
                                                                    //设置预览照片的大小

            parameters.setPreviewFrameRate(3); //每秒 3 帧
            parameters.setPictureFormat(PixelFormat.JPEG); //设置照片的输出格式
            parameters.set("jpeg-quality", 85); //照片质量
            parameters.setPictureSize(display.getWidth(), display.getHeight());
                                                                    //设置照片的大小

            camera.setParameters(parameters);
            camera.setPreviewDisplay(surfaceView.getHolder());
                                                                    //通过 SurfaceView 显示取景画面

            camera.startPreview();
            preview = true;
        } catch (Exception e) {
            Log.e(TAG, e.toString());
        }
    }

    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
        if(camera!=null){
            if(preview) camera.stopPreview();
            camera.release();
            camera = null;
        }
    }
}

```

```

    }

    @Override
    public boolean onKeyDown(int keyCode, KeyEvent event) {
        if(camera!=null && event.getRepeatCount()==0){
            switch (keyCode) {
                case KeyEvent.KEYCODE_SEARCH:
                    camera.autoFocus(null); //自动对焦
                    break;
                case KeyEvent.KEYCODE_DPAD_CENTER:
                case KeyEvent.KEYCODE_CAMERA: //拍照
                    camera.takePicture(null, null, new PictureCallbackListener());
                    break;
            }
        }
        return true;
    }
}


```

以上代码这个是内部类，主要是拍照时的操作。

```

@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if(camera!=null && event.getRepeatCount()==0){
        switch (keyCode) {
            case KeyEvent.KEYCODE_SEARCH:
                camera.autoFocus(null); //自动对焦
                break;
            case KeyEvent.KEYCODE_DPAD_CENTER:
            case KeyEvent.KEYCODE_CAMERA: //拍照
                camera.takePicture(null, null, new PictureCallbackListener());
                break;
        }
    }
    return true;
}

```

以上这个方法主要是绑定键盘事件，电话上的  按钮是就会拍照。

```

private final class PictureCallbackListener implements Camera.PictureCallback
{
    @Override
    public void onPictureTaken(byte[] data, Camera camera) {
        try {

```

```

        Bitmap bitmap = BitmapFactory.decodeByteArray(data, 0, data.length);

        File file = new File(Environment.getExternalStorageDirectory(), "ljh.jpg");
        FileOutputStream outStream = new FileOutputStream(file);
        bitmap.compress(CompressFormat.JPEG, 100, outStream);
        outStream.close();
        //重新浏览
        camera.stopPreview();
        camera.startPreview();

        preview = true;
    } catch (Exception e) {
        Log.e(TAG, e.toString());
    }
}
}

```

这个也是内部类，主要把我们拍照的相片存储。

最后我们不要忘记添加权限。上面有，就不重复了。看下效果，使用的是模拟器。



还有一种方法就是启动系统自带的系统相机。这个很简单。核心代码为：

```

Intent intent = new Intent(); //调用照相机
intent.setAction("android.media.action.STILL_IMAGE_CAMERA");

```

注意：要添加权限。

当 android 手机有这个相机软件的时候，如果把这个自带的软件卸载了，启动会出错。

5.2 摄像头采集视频 (eoeAndroid 笔名:kenan)

通过摄像头采集视频其实就是在图像预览中获取帧数保存起来。摄像头采集的格式有 JPEG YV12, NV16, NV21, RGB_565

5.2.1 main.xml 文件

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent" android:layout_height="fill_parent"
    android:orientation="vertical">
    <SurfaceView android:id="@+id/surface_camera"
        android:layout_width="176dp" android:layout_height="144dp">
    </SurfaceView>
</LinearLayout>
```

5.2.2 代码如下

```
package com.android.camarademo;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.List;

import com.aiyoulive.encoder.MetroEncoderJNI;
import android.app.Activity;
import android.content.Context;
import android.content.res.Configuration;
import android.graphics.Bitmap;
import android.graphics.PixelFormat;
import android.hardware.Camera;
import android.os.Bundle;
import android.os.Environment;
import android.util.Log;
import android.view.SurfaceHolder;
import android.view.SurfaceHolder.Callback;
import android.view.SurfaceView;
```

```
import android.view.Window;
import android.view.WindowManager;

public class CamaraDemo extends Activity implements Callback {
    /** Called when the activity is first created. */
    private static final String TAG = "CamaraDemo";
    private SurfaceView mSurfaceView = null;
    private SurfaceHolder mSurfaceHolder = null;
    private Camera mCamera = null;
    Bitmap mBitmap;
    FileOutputStream outStream;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getWindow().setFormat(PixelFormat.TRANSLUCENT);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                               WindowManager.LayoutParams.FLAG_FULLSCREEN);
        setContentView(R.layout.main);
        mSurfaceView = (SurfaceView) this.findViewById(R.id.surface_camera);
        mSurfaceHolder = mSurfaceView.getHolder();
        mSurfaceHolder.addCallback(this);
        mSurfaceHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);

        try {
            //File file=new File("/sdcard/abc.yuv");
            File file = new File(Environment.getExternalStorageDirectory(),
                                "abc.yuv");
            outStream = new FileOutputStream(file);
            //outStream = openFileOutput("abc.yuv", Activity.MODE_APPEND);
        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int width,
                              int height) {
        Log.d("data", "surfaceChanged");
        Camera.Parameters parameters = mCamera.getParameters();
        parameters.set("ORIENTATION", "PORTRAIT");
    }
}
```

```
parameters.set("ROTATION",-90);
parameters.setSceneMode("portrait");
parameters.setPreviewSize(width, height);    //获得支持的视频格式大小
List<Camera.Size> reviewSizes=parameters.getSupportedPreviewSizes();
for(int i=0;i<PreviewSizes.size();i++){
    Log.d(TAG, ""+PreviewSizes.get(i).height);
    Log.d(TAG, ""+PreviewSizes.get(i).width);
}

//  SharedPreferences mPreferences =
//      PreferenceManager.getDefaultSharedPreferences(context);
//  String sensor = mPreferences.getString("pref_camera_sensor_key",
//      "sub");
//  parameters.setSensorDev(sensor);
parameters.setPreviewFrameRate(4);
// parameters.setPreviewFormat(ImageFormat.YV12);
mCamera.setParameters(parameters);
mCamera.setPreviewCallback(new StreamIt(CamaraDemo.this));
mCamera.startPreview();
}

@Override
public void surfaceCreated(SurfaceHolder holder) {
    Log.d(TAG, "surfaceCreated");
    mCamera = Camera.open();
    try {
        mCamera.setPreviewDisplay(holder);
    } catch (IOException exception) {
        mCamera.release();
        mCamera = null;
        // TODO: add more exception handling logic here
    }
    int ret = MetroEncoderJNI.VideoInit(176, 144,2,4,64,4);
    Log.d(TAG, "MetroDecoderJNI.VideoInit()_return value =" + ret);
}

@Override
public void surfaceDestroyed(SurfaceHolder holder) {
    Log.d(TAG, "surfaceDestroyed");
    mCamera.setPreviewCallback(null);
    mCamera.stopPreview();
    mCamera.release();
    mCamera = null;
}
```



```

        try {
            outputStream.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    @Override
    public void onConfigurationChanged(Configuration newConfig) {
        // try {
        //     super.onConfigurationChanged(newConfig);
        //     if (this.getResources().getConfiguration().orientation ==
        //         Configuration.ORIENTATION_LANDSCAPE) {
        //     } else if (this.getResources().getConfiguration().orientation ==
        //         Configuration.ORIENTATION_PORTRAIT) {
        //     } catch (Exception ex) {
        //     }
        // }

        public void onOrientationChanged(int orientation) {
            // TODO Auto-generated method stub
            // if (OrientationEventListener.ORIENTATION_UNKNOWN != orientation) {
            //     if (0 == orientation)
            //         Log.e("MyNewLog", "Not change the Screen!!");
            //     else if (90 == orientation)
            //         Log.e("MyNewLog", "left side is at the top");
            //     else if (180 == orientation)
            //         Log.e("MyNewLog", "upside down");
            //     else if (270 == orientation)
            //         Log.e("MyNewLog", "right side is to the top");
            //     }
        }

        class StreamIt implements Camera.PreviewCallback {
            private Context context;

            public StreamIt(Context context){
                this.context=context;
            }

            @Override
            public void onPreviewFrame(byte[] data, Camera camera) {
                if (data != null) {

```

```

        byte[] pdata=new byte[data.length];
        int value = MetroEncoderJNI.VideoDecode(data, pdata);
        Log.d(TAG, "pdata="+value);
        Log.d(TAG, Integer.toString(data.length));

        try {
            //          File file=new File("/sdcard/abc.yuv");
            //          FileOutputStream outStream=new FileOutputStream(file);
            //          FileOutputStream
            outStream=context.openFileOutput("abc.yuv",Context.MODE_APPEND);
            outStream.write(pdata,0,value);
            //          outStream.write(data);
        } catch (FileNotFoundException e) {
            return;
        }

        catch (IOException e){
            return ;
        }

    } else {
        Log.d(TAG, "error");
    }
}
}

```

5.2.3 配置文件中加上权限

```

<uses-permission android:name="android.permission.CAMERA" />
<!-- 在 SDCard 中创建与删除文件权限 -->
<uses-permission android:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS"/>
<!-- 往 SDCard 写入数据权限 -->
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>

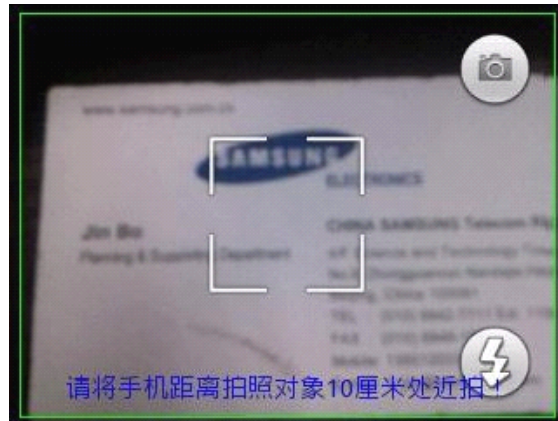
```

5.3 camera 应用层的应用 (eoe Android ID: helloyougirl)

5.3.1 Camera 的流程。

- 1、通过 Camera 的 open() 方法获取一个 Camera 对象实例，这里可能发生异常。

打开照相机之后，调用 startPreview() 方法就可以实现预览了，在使用照相机，在按下快门之前，你必须通过预览图片是否清晰或者焦距合理。



- 2、首先建立一个 SurfaceView, 获取它的 SurfaceHolder 对象, 通过这个 SurfaceHolder 对象用于图片的预览, 是通过 Camera 的 setPreviewDisplay(SurfaceHolder holder) 设置。
- 3、另外可以获取指定 Camera 的 Parameters, 设置相关属性, 假如想要这些 Parameters 的属性生效, 必须通过 Camera 的 setParameters(Parameters params)
- 4、另外要注意的 setPreviewSize() 和 setPictureSize() 两个方法可能会因为设置尺寸和实际尺寸不相同, 而抛出异常。
- 5、最关键的一点是如何获取我们所要的图片。

Camera 有一个方法 autoFocus(AutoFocusCallback back); 你可以自己实现 Camera.AutoFocusCallback 接口, 覆写 onAutoFocus(boolean focused, android.hardware.Camera camera)

在这个方法里面通过 Camera 类的 takePicture(ShutterCallback shutter, PictureCallback raw, PictureCallback jpeg) 方法获取图片, 这里你可以自己实现 PictureCallback 接口, 覆写 onPictureTaken(byte[] jpegData, android.hardware.Camera camera) 方法。

byte[] jpegData, 这就是我们所要想要的, 剩下的就是如何把这个字节数组变成一张图片, 可以通过 BitmapFactory 类的 decodeByteArray(byte[] data, int offset, int length)。

推荐: 在使用完 Camera 对象实例, 应立即调用 release() 方法, 断开连接, 释放 Camera 对象资源。

- 6、假如不想浏览所拍照出来的图片, 暂时退出拍照界面, 可以先通过 stopPreview() 关闭预览, 另外退出界面之前必须先关闭照相机, 通过调用 closeDriver() 完成。

5.3.2 具体流程就是这样，还有很多细节值得注意。

- 1、比如，想设置照相机的自动曝光，通过 Parameters 类的 setFlashMode(String mode)，通过情况下是没错的，但是为了使程序具有容忍性，应该先调用 getSupportedFlashModes()，这里一个 List<String>对象，然后使用它，我们可以判断手机是否支持自动曝光，只有检测到支持自动曝光，才设置自动曝光，这样看起来，程序更具包容性。
- 2、可以通过 set() 方法，设置图片的某些信息，比如，想设置图片的质量 set("jpeg-quality", 100)，这里的 key 是固定的。
- 3、可以通过 set("rotate", 0); 设置图片的旋转角度。当然也可以通过 remove(String key) 移除已经设定好的映射，这是不是让大家想起来了 java 的 Map 接口，对了，Parameters 就是这样的一个类。

eoeANDROID

【Android camera 原文】

6.1 Android SDK Quick Tip: Launching the Camera

This quick tip shows you how to launch the built-in Camera application and use the results for displaying the captured image. You will achieve this by creating an Intent within your application's Activity. You'll then learn how to get and process the Intent's results and use the resulting image within your application.

Step 1: Create an Android Application

Begin by creating an Android project. Implement your Android application as normal. Once you have a project set up and the application running, decide under what circumstances you want to launch the Camera and retrieve the resulting captured image. Will this occur when a button control is pressed?

What will you do with the resulting image? Presumably, you wish to display an appropriately-sized version of the captured image as part of your Activity's layout—for example, within an ImageView control. Implement the necessary button control, including any click handling. Design the appropriate layout with its ImageView control to contain the resulting photo. Once you have completed these tasks, you have a place to:

Drop in the code to launch the camera
Retrieve and display the resulting photo

Now you are ready to proceed with this quick tip.

Step 2: Creating the Intent

The Camera application can be launched to take a photo with the following Intent:

android.provider.MediaStore.ACTION_IMAGE_CAPTURE

Begin by creating an Intent of this type, as follows, within your button click handler:

```
Intent cameraIntent = new Intent(android.provider.MediaStore.ACTION_IMAGE_CAPTURE);
```

If you just start the Activity like this, the camera application will launch. However, you might want the resulting captured image as well. Luckily, you can retrieve a View-friendly version of the captured camera image as part of the Intent results data.

Step 3: Asking for Results

Intents are often started using the startActivity() method. However, in this

case, your application wants to wait for, and use, the results of the Camera application's image capture. Therefore, you want to send the Intent using the `startActivityForResult()` call instead. This way you can inspect the results and use the captured image. Therefore, the code should look something like this:

```
startActivityForResult(cameraIntent, CAMERA_PIC_REQUEST);
```

This will launch an Activity that provides a result, and then pass that result back to the Activity that launched it. If you're now wondering where `CAMERA_PIC_REQUEST` comes from, read on.

Step 4: Handling the Results

When the `startActivityForResult()` method is called, the Activity is launched. Once that Activity finishes, the calling Activity is presented with a result within its `onActivityResult()` handler. Therefore, you need to implement the `onActivityResult()` callback method within your application's Activity as follows:

```
protected void onActivityResult(int requestCode, int resultCode, Intent data) {  
    if (requestCode == CAMERA_PIC_REQUEST) {  
        // do something  
    }  
}
```

Yes, there it is again: `CAMERA_PIC_REQUEST`. This is a value that you need to define within your application as the request code returned by the Camera image capture Intent, like so:

```
private static final int CAMERA_PIC_REQUEST = 1337;
```

Beyond that, you can use this value to differentiate between different types of results. In this case, just verify that the `requestCode` is the one you set.

Step 5: Getting the Image

The image that is returned from this is appropriate for display on a small device screen. It comes in directly to the results as an Android Bitmap object:

```
Bitmap thumbnail = (Bitmap) data.getExtras().get("data");
```

What you do with the Bitmap object is up to you. Displaying it on screen is as easy as calling the `setImageBitmap()` method on an `ImageView` you have defined in your layout (here we've called it `photoResultView`).

6.2 Photo Capture & Display



- Capture of photo from built-in camera
- Captured in JPEG format at screen-size resolution (480x320)
- Compressed from 3MP format on capture
- Stored on local SD card (also accessible via USB mount or card reader)
- Upload to server via HTTP Post



reportphotoform.xml

```
<ImageView android:id="@+id/previewphoto" android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="top"/>
```

PhotoFormActivity

```
/** Called when the activity is first created. */
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    currentPhotoPath = this.getIntent().getStringExtra("photoFile");
    setContentView(R.layout.reportphotoform);

    ((Button)findViewById(R.id.btnTakePicture)).setOnClickListener(this);
    ((Button)findViewById(R.id.btnReportFormSubmit)).setOnClickListener(this);
    ((Button)findViewById(R.id.btnReportFormCancel)).setOnClickListener(this);

    if (currentPhotoPath != null)
    {
        Toast.makeText(getApplicationContext(), "Ready to send photo: " + currentPhotoPath,
            Toast.LENGTH_LONG).show();
    }

    ((ImageView)findViewById(R.id.previewphoto)).setImageURI(Uri.parse(currentPhotoPath));
}
```




camera.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="@drawable/bg">
    <SurfaceView android:id="@+id/surface"
        android:layout_width="fill_parent" android:layout_height="18dip"
        android:layout_weight="1">
    </SurfaceView>
</LinearLayout>
```

ImageCaptureActivity:

preview setup

```
public void surfaceChanged(SurfaceHolder holder, int
format, int w, int h)
{
    Log.e(getClass().getSimpleName(), "surfaceChanged");
    if (isPreviewRunning) {
        camera.stopPreview();
    }

    camera.setPreviewDisplay(holder);
    camera.startPreview();
    isPreviewRunning = true;
}
```

photo capture

```
public void onPreviewFrame(byte[] data, Camera c) {
    if (!sizeSet)
    {
        Log.i(getClass().getSimpleName(), "preview frame
        RAW: " + data);

        Camera.Parameters params = c.getParameters();
        params.setPictureFormat(PixelFormat.JPEG);
        params.setPictureSize(PHOTO_WIDTH, PHOTO_HEIGHT);
        c.setParameters(params);

        sizeSet = true;
    }
}
```



```
Camera.PictureCallback mPictureCallbackJpeg= new Camera.PictureCallback() {
    public void onPictureTaken(byte[] data, Camera c) {
        Log.e(getClass().getSimpleName(), "PICTURE CALLBACK JPEG: data.length = " + data.length);
        String filename = timeStampFormat.format(new Date());

        String baseDir = "/sdcard/";

        if (new File("/sdcard/dcm/Camera/").exists())
        {
            baseDir = "/sdcard/dcm/Camera/";
        }

        currentPhotoFile = baseDir + filename + ".jpg";

        try
        {
            FileOutputStream file = new FileOutputStream(new File(currentPhotoFile));
            file.write(data);

            sendPicture();
        }
        catch (Exception e){
            e.printStackTrace();
        }
    }
};

Camera.ShutterCallback mShutterCallback = new Camera.ShutterCallback() {
    public void onShutter() {
        Log.e(getClass().getSimpleName(), "SHUTTER CALLBACK");
    }
};

Camera.Parameters params = camera.getParameters();
params.setPictureFormat(PixelFormat.JPEG);
params.setPictureSize(PHOTO_WIDTH, PHOTO_HEIGHT);
camera.setParameters(params);
} };
```

【其他】

7.1 BUG 提交

如果你发现文档中有不妥的地方，请发邮件至 eoandroid@eoemobile.com 进行反馈，我们会定期更新、发布更新后的版本。

7.2 关于 eoeAndroid

eoeAndroid 是国内成立最早，规模最大的 Android 开发者社区，拥有海量的 Android 学习资料。分享、互助的氛围，让 Android 开发者迅速成长，逐步从懵懂到了解，从入门到开发出属于自己的应用，eoeAndroid 为广大 Android 开发者奠定坚实的技术基础。从初级到高级，从环境搭建到底层架构，eoeAndroid 社区为开发者精挑细选了丰富的学习资料和实例代码。让 Android 开发者在社区中迅速的成长，并在此基础上开发出更多优秀的 Android 应用。

7.3 庆祝优亿市场新版本发布，征集反馈送大礼！

经过长时间的努力和奋斗，优亿市场终于又迎来了新版本。做您最贴心的手机软件市场，是我们一直努力的目标。为了能让大家能够更好的交流和互动，特别开放优亿论坛！大家可以在论坛里分享自己，买手机、玩手机、使用各种应用的心得和体会。

为庆祝优亿市场新版本的发布，以及回报大家一直以来的支持与厚爱，特举办此次活动。

- 一等奖：1 名，微软（Microsoft）无线折叠鼠标 Arc Touch 黑色
- 二等奖：2 名，罗技（Logitech）MK260 无线光电键鼠套装
- 三等奖：3 名，摩天手（MOFII）G52 2.4G 无线光学蓝光鼠标（典雅黑）
- 安慰奖：5 名，eoe 特别制作的精美抱枕。

详情点击：<http://bbs.eoemarket.com/thread-20232-1-1.html>



北京易联致远无限技术有限公司

责任编辑: 莫言默语

美术支持: 阿彦

技术支持: gaotong86

中国最大的 Android 开发者社区 : www.eoeandroid.com

中国本土的 Android 软件下载平台 : www.eoemarket.com

eoeANDROID