Android camera

本文档分为三部分介绍Android camera 开发与调试。第一部分主要介绍Android 的 原生开发好的camera 架构； 第二部分主要介绍夹在驱动和Android 原生架构之间的hardware 层开发与调试，这部分是介绍的重点；第三部分介绍我们的camera 驱动开发与调试以及驱动的板机配置。

首先介绍**Android camera** 应用层开发：

要想开发Android camera 应用，必须熟悉Android 为我们定义好的camera API， 以及这些API 的调用流程。

Android 为我们提供的camera API 不多，但是功能很强， 有关于拍照的，有关于录像的。关于拍照的有：

1, 打开：

Camera open(int cameraId);

Camera open();

说明： 我们在使用camera 之前首先打开， 目前Android 可以支持的camera 数量是两个，不过这个可以改动，后面会提到，所以带参数的open ,参数只能是0和1, 不带参数的open只打开cameraID 等于0的camera, 当然这个id得根据具体硬件平台来决定，不过要记住，id 是从0开始计数的。id==0 代表后置， id==1 代表前置。

2, 设置预览：

void setPreviewDisplay(SurfaceHolder holder);

说明： 如果你要预览camera, 必须给camera 设置一个画布， 就是这里的holder, 这个holder可以从SurfaceView 控件得到，只要你布局一个SurfaceView, 然后调用这个类的getHolder 方法就好了，设置这个holder, 其实就是从surfaceflinger 里面申请了一个graph buffer, hardware 层代码会得到这个buffer,将camera sensor 的数据拷贝到这个buffer 里面，注意的是 这个buffer 里面的数据是rgb 格式的。

void setPreviewTexture(SurfaceTexture surfaceTexture);

说明： 这个也是给预览用的，和上面的效果一样， 所以它和 setPreviewDisplay 只能选择一个，这个和上面的区别是这个接口效率和预览色彩都比上面的好， 所以要是给camera 设置预览画布，建议优先使用这个， 不过这个接口在Android 的老版本中好像是没有的。

注意了: setPreviewDisplay 和 setPreviewTexture 必须在调用 startPreview 之前调用，要不屏幕将是黑的，无法看到预览， 而且要注意了，一定要在画布创建完成后在使用，不如

setPreviewDisplay 一般在 surfaceCreate 的回调里面使用。设置早或者晚，都不能正常预览。

3, 开始预览：

void startPreview();

说明： 开始camera 的预览， 一般是在画布创建完成后调用，这个方法一般放在一个线程里面，防止阻塞UI线程。

小节：我们可以看到， 开发Android camera 应用是多么的简单，只要三步就可以预览了：
open ==> setPreviewDisplay/setPreviewTexture ==> startPreview

假如camera 现在可以预览了，你可能还不满足，想给它添加更多的功能，想做一个交互性更好的camera 应用，那么就必须继续学习Android camera API, Android 提供的camera API 能最大程度的满足你的各种复杂功能的开发需求。比如

1, 修改camera 的参数， 例如 预览大小

Parameters getParameters();

void setParameters(Parameters params);

说明： 这两个方法是修改参数使用的，你必须先得到一份参数，Android camera 为我们提供了一份默认的参数， 你需要的是修改这份默认参数， 都有什么参数， 参数的值修改方法，都在 Parameters这个类中，所以你要学习 Parameters 类的api， 我这里就不介绍了，这个类很简单。

参数修改好后，调用 setParameters 设置给camera 驱动就可以了，要注意的是，有些参数要在预览之前设置，有些可以边预览边设置， 这些需要自己摸索了。

2, 参数设定也许不能满足你的要求，Android 也为你提供了自主修改预览数据的接口.

void setPreviewCallback(PreviewCallback cb);

void setOneShotPreviewCallback(PreviewCallback cb);

void setPreviewCallbackWithBuffer(PreviewCallback cb);

说明： 这三个方法都是给Android camera 设置一个回调， 你只需要实现回调就好了，回调的参数是由驱动报给应用的，其中的data 参数是驱动上报的yuv数据，具体是yuv420sp 还是 yuv422 还是 yuv420P, 取决于你对camera 参数的设置，这里你可以对这个数据进行各种变换，显示在屏幕上，或者进行其他处理，不如二维码识别， 指纹识别，人脸识别，颜色转化等等。

这三个方法是有区别的：

setPreviewCallback ： 可以使你实现的回调在预览的时候，不停的被调用。

setOneShotPreviewCallback： 可以使你实现的回调只被调用一次， 如果你想回调在处理完以后仍然被调用，需要在最后在重复调用 setOneShotPreviewCallback， 使它成为一个循环，二维码识别调用的就是这个，数据识别完以后， 如果成功，就显示结果，如果失败， 继续重复调用 setOneShotPreviewCallback；

setPreviewCallbackWithBuffer：可以使用你自己分配的buffer, 你调用这个方法之前必

须先调用 addCallbackBuffer 方法， 添加一个你自己分配的buffer, 这样在回调里面data 对应的 buffer ,就是有你自己分配的，以上两个api, 是由系统分配的buffer. 注意了，这个方法也是不停的被回调。

3, 有的camera 有对焦功能，你可以从camera 参数里面得到这些属性， 如果你想在对焦完成后做点什么事情，比如拍照，你需要添加对焦的回调。

void autoFocus(AutoFocusCallback cb);

说明： 如果你想对焦了，比如点击一个按钮对焦，你需要调用这个方法，他会帮助你对焦，对焦完成后的参数，会在 AutoFocusCallback 回调里面提供，你这需要实现这个回答就行了。

void cancelAutoFocus();

说明： 取消对焦，这个方法一般没有用，因为一但开始了对焦，很大程度是有驱动完成的动作，上层无法在干预了，直到对焦成功或失败，都会在 AutoFocusCallback回调里面有所体现。

4, camera 拍照

void takePicture(ShutterCallback shutter, PictureCallback raw,
PictureCallback jpeg);

说明： 这个方法是在开始预览以后才能调用的，除非修改hardware 层代码，也可以在不预览的情况下直接拍照。这个方法有三个回调，需要用户来实现，他们被回调的时机有所不同。

ShutterCallback： 当触发拍照时调用， 也就是按下拍照按钮的那一刻被回调。

第二个 PictureCallback： 当驱动得到camera 的数据的时候调用，所以他的参数一般是yuv 格式的。

第三个 PictureCallback： 一般是在将camera 的预览yuv 数据压缩成jpeg 格式的时候调用，所以它的参数是 jpeg 格式的。你可以将这个数据直接写入一个文件， 就是一张jpeg 图像了。

5, 预览数据的放大缩小

void startSmoothZoom(int value);

void stopSmoothZoom();

void setZoomChangeListener(OnZoomChangeListener listener);

说明： 这两个方法实现了对预览数据的放大缩小， 有些camera 不支持， 你需要从参数里面判断。如果支持放大缩小，你可能想知道放大缩小的倍数， 就需要实现 OnZoomChangeListener回调了，在放大缩小完成以后，这个回调会被调用

6, 调整在屏幕上的显示角度

void setDisplayOrientation(int degrees);

说明： 如果你有重力感应，屏幕可以旋转， 那么你使用camera 的时候，当然希望根据你的旋转，预览跟着旋转了，你就需要根据重力感应传的角度，camera 本身的角度，来计算出正确的显示角度，通过个这方法传给驱动。

7, 当出现错误的时候，也有个回调

void setErrorCallback(ErrorCallback cb)；

说明： 如果出现硬件故障， ErrorCallback 这个回调会被调用，你需要实现对硬件故障的排除处理，需要实现这个方法。

8, 录像：

录像的图像数据来源很多， 不如屏幕录像，数据来自framebuffer, 电影录像，数据来自播放器，camera 录像，数据来自 camera,等等，所以Android 实现了一个MediaRecorder 的类，要想录像，你需要将camera 设置给这个类， 并调用里面的startRecording. MediaRecorder 类里面的方法很简单，基本上使用上面的方法设置参数， 再加上MediaRecorder里面的几个方法，就可以录像了， 这里就不做介绍了。

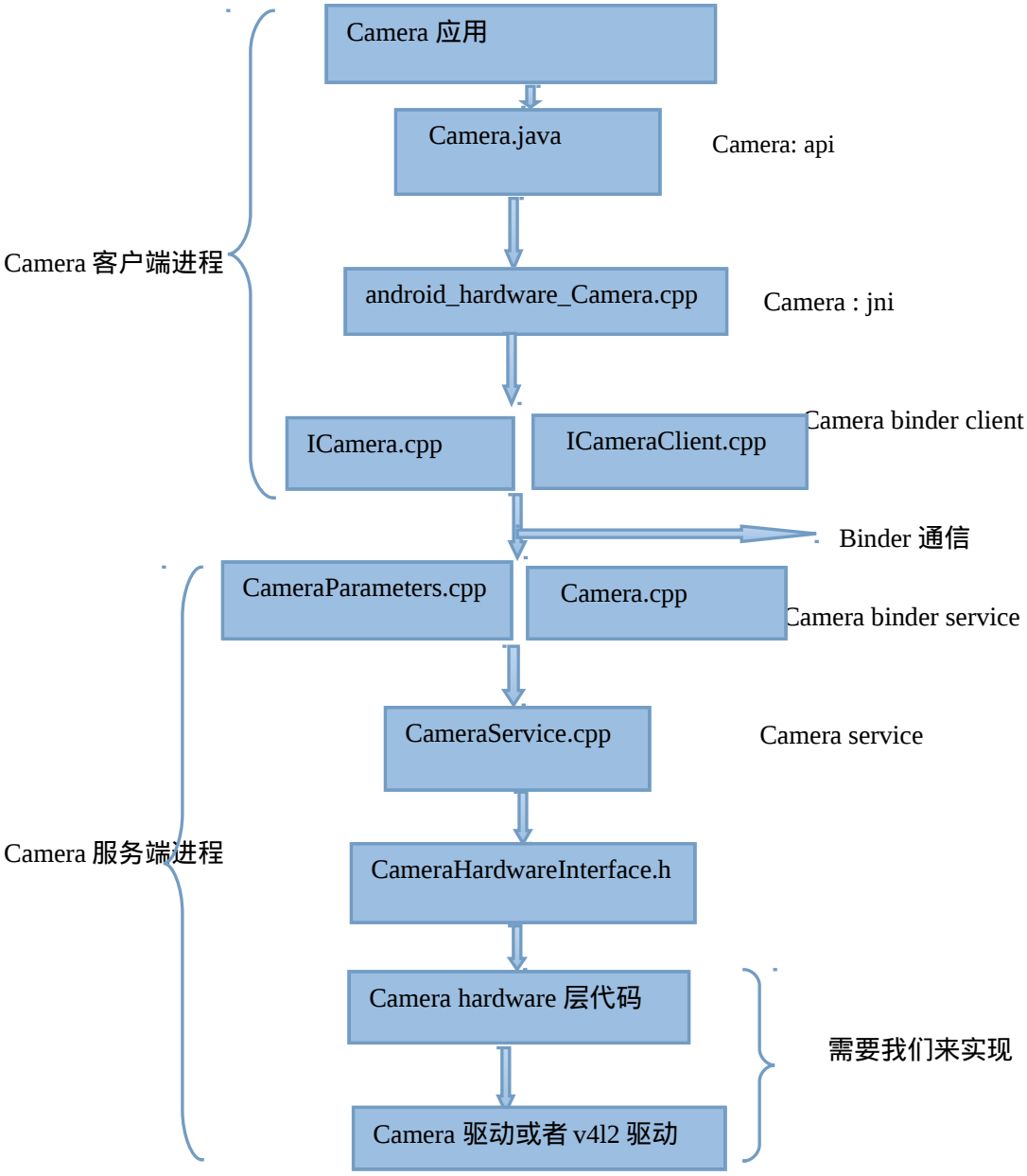
开发Android 应用的流程和API 基本介绍完毕， 看来开发一个不算复杂的Android camera 应用是很简单的事情， 但是要明白的是，之所以简单，是因为有强大的后端支持。Android 对这套API的支持在后端提供了一个复杂的camera framework , 这部分一共分为两部分，一部分是客户端，一部分是服务端， 它们分别在两个不同的进程中， 它们之间的通信，是通过Android 提供的高效的进程间通信工具binder 实现的。binder 是Android 实现的一种进程间通信驱动， 类似于共享内存，但是通信效率上比共享内存高很多，因为camera 涉及到传输预览数据，数据量有可能会很大，共享内存不太适合，所以使用binder 通信，要了解binder ，请自行在网上学习， 他主要是通过/dev/binder 设备来通信的，不过在Android 中不用之间打开这个设备，Android 为操作这个设备提供了方便的API, 你只需要实现客户端binder 接口， 服务端binder 接口， 就可以使用binder 通信了。

对不起，我们捎带说了一下binder， 因为如果要想弄清camera framework, 必须理解binder, 最起码知道用的是binder. 下面从两个进程说起。

1, camera 应用和camera API

camera 应用层代码 和 camera API是一个进程，它们在java 虚拟机上跑。它们要向 cameraservice 通过binder 提出请求， cameraservice是一个进程，它隶属于 mediaserver的一部分，所以可以这么说，mediaserver里面有camera 服务的功能。

这部分涉及的类不多，主要有8个类，当然还有一些辅助文件， 下面是他们的调用图：



通过上图，可以很清晰的看清整个 Android camera 的调用， 虽然不是很详细，但是框架基本是这样了， 只有理清楚了框架， 我们改造 Android camera 才会轻松上手，不会丢落。Android camera framework 层基本不用修改，一般是正确的，除非你有新的功能添加，或者其他需求，比如说 Android 默认只支持两个 camera , 而你想让他多支持几个，你就需要修改 CameraService.h 里面的 #define MAX_CAMERAS 2 定义，可以把 2 改为 4，让他支持四个。

下面我们讲一下需要我们实现的代码：

1， Camera hardwear 层代码，

要想实现这层代码需要上下兼顾，你需要知道你必须实现哪些接口， 你也要清楚这些接口的调用流程，你还要知道 Android hardware 层代码实现框架，因为 Android 为我们也定义好了 hardware 层的代码框架， 我们必须严格按照这个框架来实现，不然上层就

会调用失败， 你还必须知道驱动怎么调用，所以要想实现这层代码需要知道的东西很多，不过按部就班的来，也很快，我们可以分块来介绍。

第一部分: Android hardware 层框架:

说起 Android hardware 层框架，其实就是一个 Android 定义好的一个结构体，这个结构体是各种硬件的抽象，无论你是写 camera hardwear 还是写其他的 hardwear 比如 sensor 的 hardware, 你都需要实现这个结构体，这个结构体在 hardware/libhardware/include/hardware/hardware.h 中定义， 你打开这个文件就可以看到: hw_module_t 这个结构体， 你需要做的就是实现这个结构体，你定义这个结构体的时候，他的名字必须是“HMI”。

比如说我要实现一个 hardwear 层代码，那么我首先要做的就是声明 hw_module_t 这个结构体变量，并且名字是 HMI,必须是 HMI:

(1): hw_module_t HMI;

(2) 给这个结构体变量成员赋值:

HMI.tag = HARDWARE_MODULE_TAG; //必须是这个名字

//你要实现的 hardwear 层代码的版本，方便你以后升级，比如说你实现了 1.0 的 API, 又实现了 2.0 的 API, 那么你在 framework 层添加的代码既要兼容 1.0 的，又要兼容 2.0 的，那个这个成员变量就管用了。

HMI.module_api_version = HARDWARE_MODULE_API_VERSION(1,0);

HMI.hal_api_version = HARDWARE_HAL_API_VERSION;// 这个是 hardware 框架的版本，目前 hardwear 框架没有改动过，所以一直是 1.0 的版本。

HMI.id = “camera”; //这个是用来识别到底是什么硬件的 hardwear 层，你可以随便命名， 只要和 framework 层用一样的就行，这个是识别模块的关键。

HMI.name = “camera module”; //这个也是随便命名的，方便代码阅读的时候，清楚是什么模块.

HMI.author = “XXXX”; // 这个同上，方便追查是谁写的代码.

HMI.methods = &youMethods; //这个是 hardware 层的入口，用来打开硬件，你可以这样定义:

hw_module_methods_t youMethods;

HMI.dso = NULL;

HMI.reserved = {0}; 后面两个是留给 hardware 层框架代码使用的，跟你没有关系.

(3) 完成赋值后，只要实现上面定义的 **youMethods** ，就 **ok** 了， 你的 **hardwear** 层代码就大功告成了。**hw_module_methods_t** 这个结构体类型中只有一个 **open** 方法变量:

int (*open)(const struct hw_module_t* module, const char* id,

```
struct hw_device_t** device);
```

说一说这个方法吧，别看他简单，就三个参数，前两个是输入参数，最后一个是输出参数。要是实现这个方法其实也是很容易的：

指针：module 是第一步变量的地址，就是 &HMI 了, 是由 framework 层先从 hardware 层得到，在传给 hardwear 层的， 明白了吧.

Id: 这个 id 有时候有意义，有时候没有意义，完全取决于你的 framework 层的实现了， 比如拿 camera 来说吧: id==0, 代表后置摄像头，id==1 代表前置摄像头.

device: 这个指针的指针是你要实现的关键，你需要在 open 方法里面申请一块静态空间或者堆空间， 让后将这个空间的地址赋值给 *device, 例如我在 open 方法里面这些写：

```
static struct hw_device_t   camera_device;
camera_device.module = module;
*device = &camera_device;
```

(4) 上面提到的 **open** 方法里面的 **struct hw_device_t** 结构体类型是很灵活的，不局限于 **hardwear.h** 里面的定义，你可以这样定义：

```
typedef  struct  sensor_device {
    struct   hw_device_t    hw;
    void  (*set_sensor) (struct  sensor_device* dev);
    void  (*get_sensor_parameters)(struct  sensor_device* dev);
    ..... // 添加你自定义的方法或者变量
    } sensor_device_t;
    static  sensor_device_t  mySensor;
    *device  = (struct hw_device_t*) &mySensor;
```

你实现的 framework 层怎么使用，怎么设计，你就在 结构体 sensor_device 里面添加什么，就好了，这部分就是自定义用的。

注意了: struct hw_device_t hw; 这个必须定义在第一个位置上，不然 mySensor 就不能强制转换成 struct hw_device_t 变量了。

总结： 经过以上四步， 你就实现了一个硬件的 **hardwear** 层代码， 是不是和科学，也很简单， 是不是感慨 **Android** 真强大， 呵呵，祝你很快学会 **hardwear** 层代码框架。

第二部分: camera hardware 层接口:

你可以在 hardware/libhardware/include/hardware/ 这个目录找到两个或者三个关于 camera 的头文件，例如：

```
camera.h,  camea2.h,   camera3.h
```

这三个文件分别定义了 camera 三个版本的 api, camera 的 framework 层是兼容这三个版本的，也就是说你乐意实现那个都行，或者闲着没事，三

个版本都实现也没有关系，都最后到底使用那个版本的 api, 可以在设备的 BoardConfig.mk 里面取舍，不过这都是后话了，不过这里先提一下吧：

你在 BoardConfig.mk 里面定义一个宏：

CAMERA_VISION := [1/2/3];

在 camera hardwear 层的 Android.mk 里面根据这个宏的值 编译不同版本的 api 代码.

言归正传， 我只说 camera 版本 1 的 api 实现，关于其他两个版本我目前还不知道怎么实现，这需要研究 camera framework 层的代码调用流程。

Camera hal 版本 1 的接口在 camera.h 的 camera_device_t 结构体里面定义：

```
typedef struct camera_device {
  /**
   * camera_device.common.version must be in the range
   * HARDWARE_DEVICE_API_VERSION(0,0)-(1,FF).
CAMERA_DEVICE_API_VERSION_1_0 is
   * recommended.
   */
  hw_device_t common; // 必须是第一个定义, 上面有原因
  camera_device_ops_t *ops; // 自定义的调用接口，需要我们来实现
  void *priv; //私有指针， 随你怎么使用
} camera_device_t;
```

下面我们重点分析一下 camera_device_ops_t 结构体:

```
typedef struct camera_device_ops {
  int (*set_preview_window)(struct camera_device *,
        struct preview_stream_ops *window);
  void (*set_callbacks)(struct camera_device *,
        camera_notify_callback notify_cb,
        camera_data_callback data_cb,
        camera_data_timestamp_callback data_cb_timestamp,
        camera_request_memory get_memory,
        void *user);
  void (*enable_msg_type)(struct camera_device *, int32_t msg_type);
  void (*disable_msg_type)(struct camera_device *, int32_t msg_type);
  int (*msg_type_enabled)(struct camera_device *, int32_t msg_type);
  int (*start_preview)(struct camera_device *);
  void (*stop_preview)(struct camera_device *);
  int (*preview_enabled)(struct camera_device *);
  int (*store_meta_data_in_buffers)(struct camera_device *, int enable);
  int (*start_recording)(struct camera_device *);
  void (*stop_recording)(struct camera_device *);
  int (*recording_enabled)(struct camera_device *);
  void (*release_recording_frame)(struct camera_device *,
```

```
            const void *opaque);
    int (*auto_focus)(struct camera_device *);
    int (*cancel_auto_focus)(struct camera_device *);
    int (*take_picture)(struct camera_device *);
    int (*cancel_picture)(struct camera_device *);
    int (*set_parameters)(struct camera_device *, const char *parms);
    char *(*get_parameters)(struct camera_device *);
    void (*put_parameters)(struct camera_device *, char *);
    int (*send_command)(struct camera_device *,
            int32_t cmd, int32_t arg1, int32_t arg2);
    void (*release)(struct camera_device *);
    int (*dump)(struct camera_device *, int fd);
} camera_device_ops_t;
```

这里面一共自定义了 23 个方法，这 23 方法实现起来都不难，难就难在他们的调用顺序，如果把他们的调用顺序理清楚了，也就知道怎么来实现了。

流程：

（1）取得 camera number, 以及每一个 camera Information, 这两个 api 在结构体：HMI 里面定义, 例如我们的 camera_module_t ：

```
typedef struct camera_module {
    hw_module_t common;  // 这个必须在第一个定义，因为 camera_module_t 需要和 hw_module_ 兼容
    int (*get_number_of_cameras)(void);
    int (*get_camera_info)(int camera_id, struct camera_info *info);
} camera_module_t;
```

camera_module_t HMI; // 看上面的 hardwear 层框架分析， 名字必须是 HMI。这个结构体如果初始化，你可以参考上面的 hardwear 层框架分析，这个结构体就是 camera hardware 层代码的总入口了，所以你必须定义成全局变量，它里面多了两个自定义方法：

```
int  get_number_of_camera (void) {
        return 2;  //你可以根据你的平台的具体 camera 数量在这里写死，也可以通过扫描 /dev/ 下面的设备名，计算得出，随你便。
    }
int  get_camera_info (int camera_id,  struct camera_info *info);
```

这个方法就是让你填充 info 这个指针用的，这个指针指向的空间由 framework 层配，不需你来考虑。

```
int get_camera_info (int camera_id ,  struct camera_info *info) {
    if  (camera_id == 0) {
        info-> facing = 0;  //前置摄像头
        info -> orientation = 0; //  根据 sensor 在平台上的安装方向来定义, 可
```

```
        以取值为： 0, 90, 180, 270
        } else if (camera_id == 1) {
            info -> facing = 1; // 后置摄像头
            info -> orientation = 90;
        } else {
            return -1;
        }
        return 0;
    }
```

（2）通过 open 方法得到 camera_device_t 的结构体指针， 通过这个结构体指针得到这个结构体里面的 camera_device_ops_t 结构体指针， 得到这个 camera 操作指针后就可以通过里面定义的接口来操作具体的 camera sensor 了。如果不知道 open 方法在什么地方，请看上面的 hardwear 框架分析.

（3）拿到 camera 操作方法指针后，就按部就班的调用，完成 camera 的正确使用流程。

Camera 拍照流程：

set_callbacks ==> enable_msg_type ==> set_preview_window ==> get_parameters ==> set_parameters ==> start_preview ==> auto_focus ==> cancel_auto_focus ==> stop_preview ==> take_picture ==> start_preview ==> stop_preview ==> release

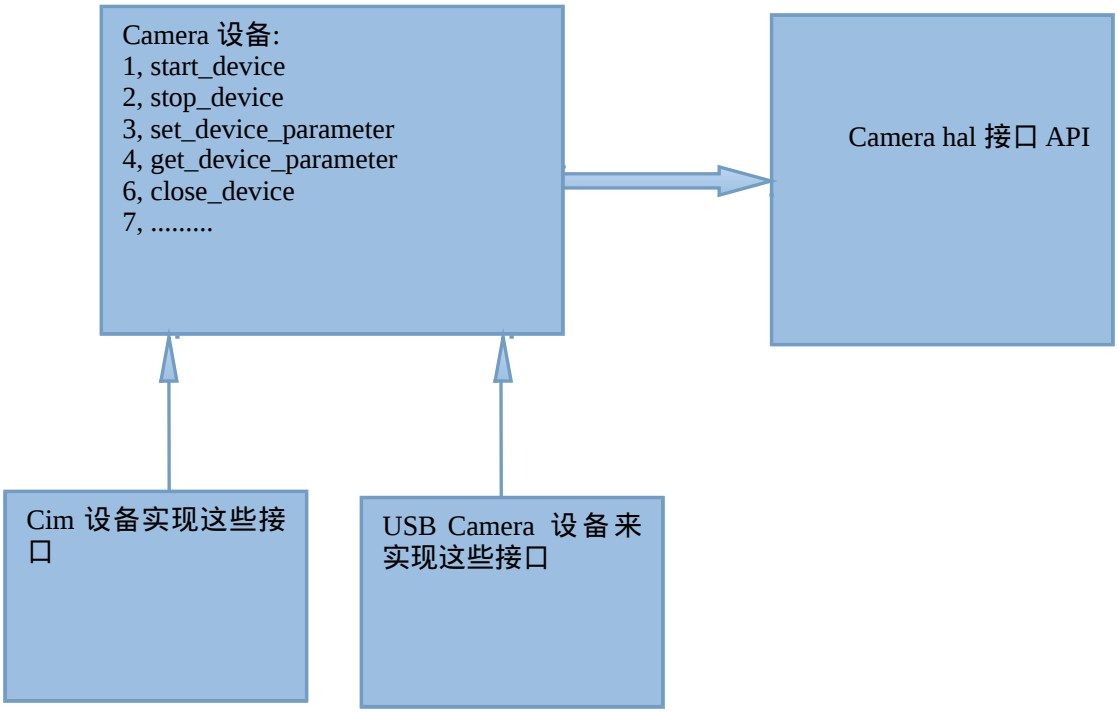
Camera 录像流程：

set_callbacks ==> enable_msg_type ==> set_preview_window ==> get_parameters ==> set_parameters ==> start_preview ==> start_recording ==> recording_enabled ==> release_recording_frame ==> stop_recording ==> stop_preview ==> release

小结： 上面简单说了一下 **camera hardware** 的实现流程，并没有讲到具体的实现，我感觉，每个人和每个人的实现方法不一样，主要是知道实现流程，具体逻辑，怎么设计，因人而异了。

第三部分: camera hardware 层多设备兼容：

为了实现多设备兼容，比如说 cim 驱动和 v4l2 驱动共存，你的平台上在 cim 控制器上接入了一个 camera sensor, 在 usb host 上接入了一个 camera sensor, 拿么怎么让他们可以相互切换着工作呢。想想就明白了，这是两类设备，但是他们实现了同样的功能，我们可以抽象出一个共同的功能列表，然后由这两类设备分别实现。



通过 camera device 这层抽象，就把 cim 和 usb camera 当成了一种 camera 设备了， 我们的 hardware 层就是这样实现的。

第四部分: camera cim 驱动:

首先在 kernel/arch/mips/xburst/soc-47XX/board/xxx/misc.c

里面添加:

```
if defined(CONFIG_JZ_CIM0) || defined(CONFIG_JZ_CIM1)))
struct cam_sensor_plat_data {
        int facing;
        int orientation;
        int mirror;   //camera mirror
        //u16   gpio_vcc;       /* vcc enable gpio */   remove the gpio_vcc   , DO NOT use this pin for sensor power up ,cim will controls this
        uint16_t        gpio_rst;       /* resert  gpio */
        uint16_t        gpio_en;        /* camera enable gpio */
        int cap_wait_frame;    /* filter n frames when capture image */
```

```
};

#ifdef CONFIG_OV3640  // 具体的 sensor, 例如: CONFIG_GC0308
static struct cam_sensor_plat_data ov3640_pdata = {
	.facing = 1,
	.orientation = 0,
	.mirror = 0,
	.gpio_en = GPIO_OV3640_EN,  // 看原理图 找到具体在使用那个 gpio, 这个变量在 borad.h 里面定义
	.gpio_rst = GPIO_OV3640_RST,
	.cap_wait_frame = 6,
};

#if  defined(CONFIG_I2C_GPIO)
static struct i2c_board_info camera//_i2c1_devs[] __initdata = {
#ifdef CONFIG_OV3640   // 具体的 sensor, 例如: CONFIG_GC0308
	{
		I2C_BOARD_INFO("ov3640", 0x3c),
		.platform_data  = &ov3640_pdata, //上面定义的结构体
	},
#endif
};
#endif  /*I2C1*/
```

在 board_init 里面添加:

```
#if defined(CONFIG_I2C_GPIO)
	i2c_register_board_info(1, camera_i2c1_devs,
ARRAY_SIZE(mensa_i2c1_devs));
#endif
```

板级代码就这样添加好了，下面就是选择了:

在 kernel 目录里面:

Make menuconfig

Device drvicers ==> misc dev ices ==> ingenci camera interface module ==> ov5640(是具体而定)

第五部分: camera usb 驱动:

这部分不用添加板级代码了，直接在驱动里面选择官方自带的驱动就 ok 了。

Device drvicers ==> Multimedia support ==> Video For Linux

==> video capture adapter ==>

V4L USB device ==> USB video class

==> UVC input events
选择好后编译重新编译 kernel 生成 boot.img, 烧录进板子， 如果重启后插入 usb host camera ，如果 /dev/videoX 设备出现的话，说明驱动选择正确。
注意： 我们的 usb otg camera 不支持，可能 otg 驱动有问题，感兴趣的可以调试一下 usb otg camera.

第六部分: camera 板级配置

如果你的板子上有 camera, 你还需要配置
build/target/board/XXX/BoardConfig.mk 和
build/target/board/XXX/MediaProfile.xml
BoardConfig.mk: 添加:
BOARD_HAS_CAMERA := true
CAMERA_SUPPORT_VIDEOSNAPSHORT := true // 是否支持边录像边拍照
COVERT_WITH_SOFT := true // 是否支持 ipu
CAMERA_VERSION := 1 // camera 的版本信息
这些宏直接影响 hardware 层的 camera 代码的编译。

media_profiles.xml:

这个文件主要定义了关于录像的支持力度， 大小啊，录像出来的格式啊 等等，我经常看到 板子上明明只有一个 camera sensor, 这个文件里面却定义了两个，所以提醒还是要稍微关注一下这个文件。
Camera 大小:
VGA: 640*480
QVGA: 320*240
480P: 640*480 或者 720*480
720P: 1280*720

第七部分： 关于实现 USB camera 在 Android 上的热插拔工程

有的客户需要 usb camera 热插拔功能，其实这个并不难，只需要添加两个文件，和修改一个配置就行了。
原理：其实 linux kernel 本身是支持热插拔的，所以驱动就不用改动了，再插入和拔出 usb camera 的时候 kernel 都会上报 uevent 事件，我们只需要在 hardwear 层添加对 usb camera uevent 的事件的监听，在监听回调里面实现 camera 的动态关闭和打开就行了。

(1), 要满足第一个条件，需要给 **mediaserver** 添加 **root** 权限，因为 **cameraserver** 是 **mediaserver** 里面的一部分，没有 **root** 权限进程接受不到 **uevent** 事件，所以你只需要修改 **init.rc,**

```
service media /system/bin/mediaserver
 class main
 user media
 group audio media camera inet net_bt net_bt_admin net_bw_acct drmrpc graphics
    ioprio rt 4
```

改为：

```
service media /system/bin/mediaserver
 class main
 user root
 group audio media camera inet net_bt net_bt_admin net_bw_acct drmrpc graphics
    ioprio rt 4
```

(2), 在 **camera hardwear** 层添加一个类 **NetlinkEventListener** 具体这个类怎么写请参考 **/system/vold** 里面的代码，关键是当收到事件后怎么处理。

(3)处理流程：

1，关闭拔出的 usb camera ，或者 添加刚插入的 usb camera 设备节点

2, 释放拔出的 usb camera 分配的内存 或者 对刚插入的 usb camera 进行一般初始化，等待应用打开

小结： 热插拔 **usb camera** 就是这么简单，关键是驱动已经支持了，就啥都好办了.

总结： 关于 Android 的 camera 使用和配置以及开发就先说这么多， 虽然不够详细，但是够全面， 如果你对那部分感兴趣，请自行研究，后者直接联系我，我愿意和你一块探讨及合作。

我的联系方式：

邮箱： wwzhao@ingenic.cn

分机： 8037

姓名： 赵伟伟