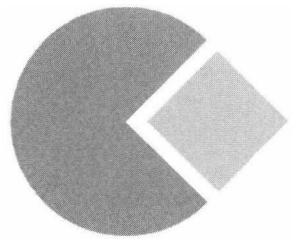


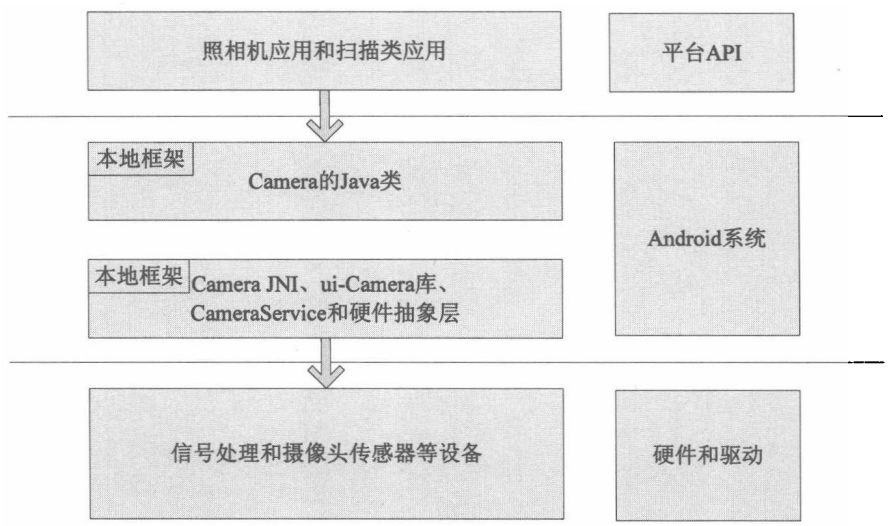
第 14 章　Camera 照相机驱动

无论是智能手机还是普通的手机，基本上都支持手机拍照和录制视频功能，虽然像素和清晰度各有不同。在 Android 系统中，照相机功能是通过 Camera 系统实现的。在本章将详细讲解 Android 平台中 Camera 系统的基本知识和移植方法，为读者步入本书后面知识的学习打下基础。

14.1 Camera 系统的结构

Camera 作为一个照相机系统，提供了取景器、视频录制和拍摄相片等功能，并且还提供了各种控制类的接口。Camera 系统分别提供了 Java 层的接口和本地接口，其中 Java 框架中的 Camera 类实现了 Java 层相机接口，为照相机和扫描类使用。而 Camera 的本地接口可以给本地程序调用，作为视频输入环节应用于摄像机和视频通话领域。

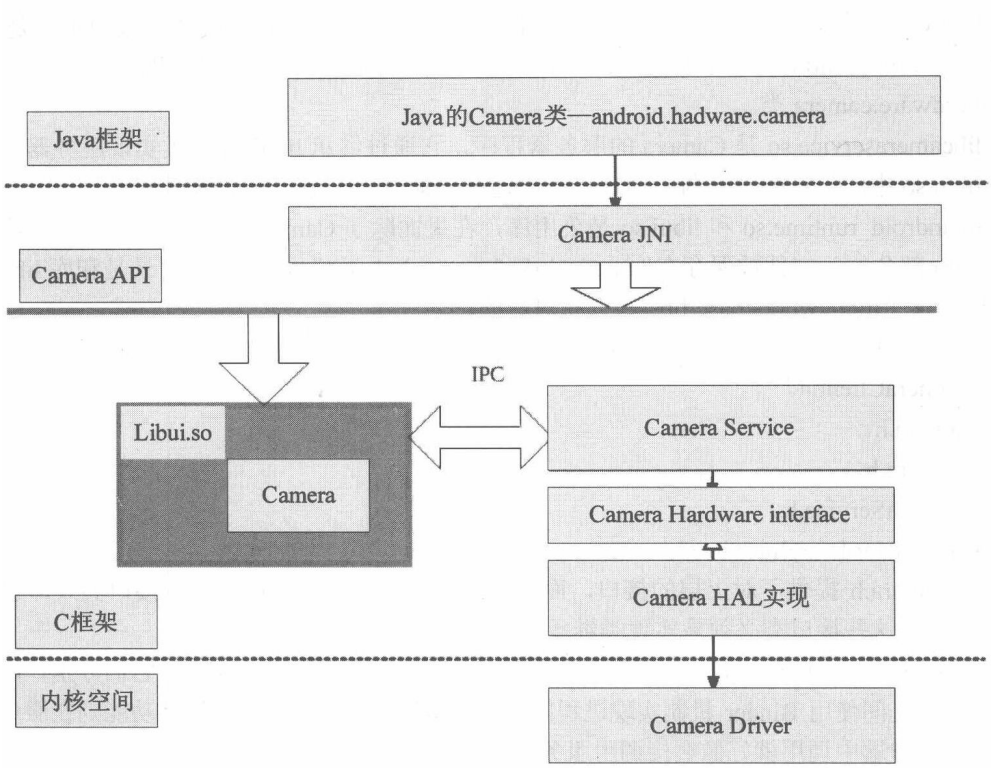
Android 照相机系统的基本层次结构如图 14-1 所示。



▲图 14-1　照相机系统的层次结构

Android 的 Camera 系统包括了 Camera 驱动程序层、Camera 硬件抽象层、AudioService、Camera 本地库、Camera 的 Java 框架类和 Java 应用层对 Camera 系统的调用。Camera 的系统结构如图 14-2

所示。



▲图 14-2　Camera 的系统结构

（1）Camera 的 Java 程序，代码目录是“frameworks/base/core/java/android/hardware/”。其中文件 Camera.java 是主要实现的文件，对应的 Java 层次的类是 android.hardware.Camera，这个类和 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），代码路径是“frameworks/base/core/jni/android_hardware_Camera.cpp”，这部分内容编译成为目标是 libandroid_runtime.so，主要的头文件保存在目录“frameworks/base/include/ui/”中。

（3）Camera 本地框架，其中头文件路径是“frameworks/base/include/ui/”或“frameworks/base/include/camera/”，源代码路径是“frameworks/base/libs/ui/”或“frameworks/base/libs/camera/”。这部分的内容被编译成库 libui.so 或 libcamera_client.so。

（4）Camera 服务部分，代码路径是“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。

（5）摄像头驱动程序，基于 Linux 的 Video for Linux 视频驱动框架。

（6）硬件抽象层，接口代码路径是“frameworks/base/include/ui/”或“frameworks/base/include/camera/”。其中的核心文件是 CameraHardwareInterface.h。

在 Camera 系统的各个库中，库 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。

库 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通过进程间通信（即 Binder 机制）的方式进行通信。

库 libandroid_runtime.so 和 libui.so 是公用库，在里面除了 Camera 外还有其他方面的功能。

Camera 部分的头文件被保存在“frameworks/base/include/ui/”目录中，此目录是和库 libmedia.so 的源文件目录“frameworks/base/libs/ui/”相对应的。

在 Camera 中主要包含下面的头文件。

- ICameraClient.h。
- Camera.h。
- ICamera.h。
- ICameraService.h。
- CameraHardwareInterface.h。

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。

当整个 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 机制实现进程间通信。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，进程间通信的部分对上层程序不可见。

从框架结构上来看，文件 ICameraService.h、ICameraClient.h 和 ICamera.h 定义了 Camera 的接口和架构，ICameraService.cpp 和 Camera.cpp 两个文件用于实现 Camera 架构，Camera 的具体功能在下层调用硬件相关的接口来实现。

由于 Android 系统是架构在 Linux 内核上的开源操作系统，所以驱动层的实现就是在 Linux 内核的基础上加上 Android 特有的一些机制，对于 Camera 系统，驱动层就是采用 Linux 系统上通用的 V4L2 接口实现数据采集、格式转换、大小缩放、数据传输的功能。

V4L 全称为 Video4Linux，是 Linux 系统上关于视频设备的通用驱动接口，现在的 V4L2 是在 V4L 上开发的第二代视频设备驱动，Android 就是基于这套标准的驱动架构来实现 Camera 功能的，当 Video 设备加载成功后，会在“/dev/”目录下生成设备节点，如图 14-3 所示。

```
# ls -l /dev |grep video
ls -l /dev |grep video
crw-rw-rw- root     root      81,   0 1970-01-01 00:00 video0
crw------- root     root      81,   1 1970-01-01 00:00 video1
crw------- root     root      81,  14 1970-01-01 00:00 video14
```

▲图 14-3　生成的设备节点

由此可见，一共有 3 个 Video 设备，上层应用可以通过调用 Video 暴露的接口来实现 Camera 功能。

14.2 移植的内容

因为 Camera 系统的标准化部分是硬件抽象层接口，所以在某平台移植 Camera 系统时，主要工作是移植 Camera 驱动程序和 Camera 硬件抽象层。

在 Linux 系统中，Camera 驱动程序使用了 Linux 标准的 Video for Linux 2（V4L2）驱动程序。无论是内核空间还是用户空间，都使用 V4L2 驱动程序框架来定义数据类和控制类。所以在移植 Android 中的 Camera 系统时，也是用标准的 V4L2 驱动程序作为 Camera 的驱动程序。

Camera 的硬件抽象层是 V4L2 和 CameraService 之间的接口，是一个 C++接口类，我们需要具体的实现者来继承这个类，并且实现里面的虚函数。Camera 的硬件抽象层需要具备取景器、视频录制、相片拍摄等功能。在 Camera 系统中，具体任务分配如下所示。

- V4L2 驱动程序：任务是获得 Video 数据。
- Camera 的硬件抽象层：任务是将纯视频流和取景器、实现预览、向上层发送数据等功能组织起来。
- 其他算法库和硬件：任务是实现自动对焦和成像增强等功能。

14.2.1 fimc 驱动模块的加载

S3c_fimc_core.c 是 Camera 驱动的核心框架程序，负责管理驱动函数的注册。代码如下：

```
"drivers/media/video/Samsung/fimc/S3c_fimc_core.c"
```

文件中以模块的方式加载驱动，会向系统中注册一个 fimc 驱动 s3c_fimc_driver，结构体定义如下：

```
static struct platform_driver s3c_fimc_driver = {
     .probe          = s3c_fimc_probe,
     .remove         = s3c_fimc_remove,
     .suspend   = s3c_fimc_suspend,
     .resume         = s3c_fimc_resume,
     .driver         = {
          .name= "s3c-fimc",
          .owner     = THIS_MODULE,
     },
};
```

当系统加载时会运行 s3c_fimc_probe()方法来注册 fimc 的控制器，其中控制器中的主要接口就是由 V4L2 驱动实现的，同时还会向系统注册 Video 设备，代码如下：

```
static int s3c_fimc_probe(struct platform_device *pdev)
{
     struct s3c_platform_fimc *pdata;
     struct s3c_fimc_control *ctrl;
     struct clk *srclk;
     int ret;
```

```
    //注册 fimc 的控制器
      ctrl = s3c_fimc_register_controller(pdev);
      if (!ctrl) {
           err("cannot register fimc controller\n");
           goto err_fimc;
      }
      pdata = to_fimc_plat(&pdev->dev);
      if (pdata->cfg_gpio)
           pdata->cfg_gpio(pdev);
      //获取时钟
      srclk = clk_get(&pdev->dev, pdata->srclk_name);
      if (IS_ERR(srclk)) {
           err("failed to get source clock of fimc\n");
           goto err_clk_io;
      }
    //获得 fimc 的时钟
      ctrl->clock = clk_get(&pdev->dev, pdata->clk_name);
      if (IS_ERR(ctrl->clock)) {
           err("failed to get fimc clock source\n");
           goto err_clk_io;
      }
      if (ctrl->clock->set_parent)
           ctrl->clock->set_parent(ctrl->clock, srclk);
      if (ctrl->clock->set_rate)
           ctrl->clock->set_rate(ctrl->clock, pdata->clockrate);
      clk_enable(ctrl->clock);
      //全局变量的初始化
      if (ctrl->id == 0) {
           ret = s3c_fimc_init_global(pdev);
           if (ret)
                goto err_global;
      }
    //注册 Video 设备
      ret = video_register_device(ctrl->vd, VFL_TYPE_GRABBER, ctrl->id);
      if (ret) {
           err("cannot register video driver\n");
           goto err_video;
      }
      return 0;
err_video:
      clk_put(s3c_fimc.cam_clock);
err_global:
      clk_disable(ctrl->clock);
      clk_put(ctrl->clock);
err_clk_io:
      s3c_fimc_unregister_controller(pdev);
err_fimc:
      return -EINVAL;
      }
```

其中 s3c_fimc_register_controller()这个函数将 V4L2 的接口注册进 fimc 驱动中，从而给应用层

提供了接口，例如 open()、close()等函数的实现都在这个文件中。V4l2 的驱动定义在 S3c_fimc_v4l2.c 中，文件路径是“drivers/media/video/samsung/fimc/S3c_fimc_v4l2.c”。

S3c_fimc_v4l2.c 中主要定义了 Video 设备的 ioctl 操作接口，并实现了这些接口，代码如下：

```
const struct v4l2_ioctl_ops s3c_fimc_v4l2_ops = {
     .vidioc_querycap          = s3c_fimc_v4l2_querycap,
     .vidioc_g_fbuf            = s3c_fimc_v4l2_g_fbuf,
     .vidioc_s_fbuf            = s3c_fimc_v4l2_s_fbuf,
     .vidioc_enum_fmt_vid_cap  = s3c_fimc_v4l2_enum_fmt_vid_cap,
     .vidioc_g_fmt_vid_cap          = s3c_fimc_v4l2_g_fmt_vid_cap,
     .vidioc_s_fmt_vid_cap          = s3c_fimc_v4l2_s_fmt_vid_cap,
     .vidioc_try_fmt_vid_cap        = s3c_fimc_v4l2_try_fmt_vid_cap,
     .vidioc_try_fmt_vid_overlay    = s3c_fimc_v4l2_try_fmt_overlay,
     .vidioc_overlay           = s3c_fimc_v4l2_overlay,
     .vidioc_g_ctrl            = s3c_fimc_v4l2_g_ctrl,
     .vidioc_s_ctrl            = s3c_fimc_v4l2_s_ctrl,
     .vidioc_streamon          = s3c_fimc_v4l2_streamon,
     .vidioc_streamoff         = s3c_fimc_v4l2_streamoff,
     .vidioc_g_input           = s3c_fimc_v4l2_g_input,
     .vidioc_s_input           = s3c_fimc_v4l2_s_input,
     .vidioc_g_output          = s3c_fimc_v4l2_g_output,
     .vidioc_s_output          = s3c_fimc_v4l2_s_output,
     .vidioc_enum_input        = s3c_fimc_v4l2_enum_input,
     .vidioc_enum_output       = s3c_fimc_v4l2_enum_output,
     .vidioc_reqbufs           = s3c_fimc_v4l2_reqbufs,
     .vidioc_querybuf          = s3c_fimc_v4l2_querybuf,
     .vidioc_qbuf              = s3c_fimc_v4l2_qbuf,
     .vidioc_dqbuf             = s3c_fimc_v4l2_dqbuf,
     .vidioc_cropcap           = s3c_fimc_v4l2_cropcap,
     .vidioc_g_crop            = s3c_fimc_v4l2_g_crop,
     .vidioc_s_crop            = s3c_fimc_v4l2_s_crop,
     .vidioc_s_parm            = s3c_fimc_v4l2_s_parm,
};
```

这些接口分别对应了上层应用程序的 ioctl 的命令，来实现不同的功能。例如设置数据格式的命令 VIDIOC_S_FMT 的实现为

```
static int s3c_fimc_v4l2_s_fmt_vid_cap(struct file *filp, void *fh,
                        struct v4l2_format *f){
     struct s3c_fimc_control *ctrl = (struct s3c_fimc_control *) fh;
     struct s3c_fimc_out_frame *frame = &ctrl->out_frame;
//判断数据流的类型，是输入还是输出
     if (f->type != V4L2_BUF_TYPE_VIDEO_CAPTURE)
          return -EINVAL;
     ctrl->v4l2.frmbuf.fmt = f->fmt.pix;
     if (frame->hq && f->fmt.pix.pixelformat == V4L2_PIX_FMT_JPEG) {
          frame->jpeg.enabled = 1;
          frame->jpeg.thumb = 1;
     }
```

```
        if (frame->hq && f->fmt.pix.pixelformat == V4L2_PIX_FMT_MJPEG)
                frame->jpeg.enabled = 1;
        if (ctrl->in_type != PATH_IN_DMA)
                f->fmt.pix.priv = V4L2_FMT_OUT;
        if (f->fmt.pix.priv == V4L2_FMT_IN)
                s3c_fimc_set_input_frame(ctrl, &f->fmt.pix);
        else
                s3c_fimc_set_output_frame(ctrl, &f->fmt.pix);
        return 0;
}
```

这是个拍照模式的设置数据格式的实现，代码中首先判断数据流的类型是否为V4L2_BUF_TYPE_VIDEO_CAPTURE，如果数据格式为 V4L2_PIX_FMT_JPEG，则最后会以 jpeg 的格式保存照片，然后对数据的输入输出进行判断，设定数据流的路由。具体的 V4L2 的标准接口和数据结构可以查看文件“include/linux/videodev2.h”。

ioctl 命令可以描述相关的数据结构，例如下面是描述 buffer 类型的 enum：

```
enum v4l2_buf_type {
        V4L2_BUF_TYPE_VIDEO_CAPTURE        = 1,
        V4L2_BUF_TYPE_VIDEO_OUTPUT         = 2,
        V4L2_BUF_TYPE_VIDEO_OVERLAY        = 3,
        V4L2_BUF_TYPE_VBI_CAPTURE          = 4,
        V4L2_BUF_TYPE_VBI_OUTPUT           = 5,
        V4L2_BUF_TYPE_SLICED_VBI_CAPTURE   = 6,
        V4L2_BUF_TYPE_SLICED_VBI_OUTPUT    = 7,
#if 1
        /* Experimental */
        V4L2_BUF_TYPE_VIDEO_OUTPUT_OVERLAY = 8,
#endif
        V4L2_BUF_TYPE_PRIVATE              = 0x80,
};
```

下面是申请帧缓冲的结构体：

```
struct v4l2_requestbuffers {
        __u32           count;//申请的个数
        enum v4l2_buf_type      type;
        enum v4l2_memory        memory;
        __u32           reserved[2];
};
```

下面是描述 Video 设备能力的结构体：

```
struct v4l2_capability {
        __u8 driver[16];
        __u8 card[32];
        __u8 bus_info[32];
        __u32   version;
        __u32 capabilities;
        __u32 reserved[4];
```

```
};
```

下面是描述视频格式的结构体：

```
struct v4l2_format {
        enum v4l2_buf_type type;
        union {
                struct v4l2_pix_format          pix;     //像素格式
                struct v4l2_window        win;
                struct v4l2_vbi_format          vbi;
                struct v4l2_sliced_vbi_format   sliced;
                __u8 raw_data[200];
        } fmt;
};
```

下面是描述视频中的一帧：

```
struct v4l2_buffer {
        __u32           index;
        enum v4l2_buf_type      type;//帧的类型
        __u32           bytesused;
        __u32           flags;
        enum v4l2_field      field;
        struct timeval       timestamp;
        struct v4l2_timecode timecode;
        __u32           sequence;
        enum v4l2_memory       memory;//内存地址
        union {
                __u32          offset;
                unsigned long   userptr;
        } m;
        __u32           length;
        __u32           input;
        __u32           reserved;
};
```

14.2.2　V4l2 驱动的用法

V4L2 驱动对 Video 设备的操作有一套标准的流程，具体说明如下。

（1）打开设备文件。

```
int fd=open("/dev/video0",O_RDWR);
```

（2）取得设备的 capability，看看设备具有什么功能，比如是否具有视频输入，或者音频输入输出等。　　n.性能;容量;才能，能力;

```
VIDIOC_QUERYCAP,struct v4l2_capability
```

（3）选择视频输入，一个视频设备可以有多个视频输入。

```
VIDIOC_S_INPUT,struct v4l2_input
```

（4）设置视频的制式和帧格式，制式包括 PAL、NTSC，帧的格式包括宽度和高度等。

```
VIDIOC_S_STD,VIDIOC_S_FMT,struct v4l2_std_id,struct v4l2_format
```

（5）向驱动申请帧缓冲，一般不超过 5 个。

```
struct v4l2_requestbuffers
```

（6）将申请到的帧缓冲映射到用户空间，这样就可以直接操作采集到的帧了，而不必去复制。

```
mmap
```

（7）将申请到的帧缓冲全部入队列，以便存放采集到的数据。

```
VIDIOC_QBUF,struct v4l2_buffer
```

（8）开始视频的采集。

```
VIDIOC_STREAMON
```

（9）出队列以取得已采集数据的帧缓冲，取得原始采集数据。

```
VIDIOC_DQBUF
```

（10）将缓冲重新入队列尾，这样可以循环采集。

```
VIDIOC_QBUF
```

（11）停止视频的采集。

```
VIDIOC_STREAMOFF
```

（12）关闭视频设备。

```
close(fd);
```

14.3 移植和调试

经过本章前面内容的讲解，已经了解了 Camera 系统的基本结构和我们需要移植的任务。在本节将详细讲解在 Android 平台中移植和调试 Camera 系统的方法，为读者步入本书后面知识的学习打下基础。

14.3.1 V4L2 驱动程序

在 Linux 系统中，Camera 驱动程序使用了 Linux 标准的 Video for Linux 2（V4L2）驱动程序。无论是内核空间还是用户空间，都使用 V4L2 驱动程序框架来定义数据类和控制类。所以在移植 Android 中的 Camera 系统时，也是用标准的 V4L2 驱动程序作为 Camera 的驱动程序。在 Camera 系统中，V4L2 驱动程序的任务是获得 Video 数据。

1. V4L2 API

V4L2 是 V4L 的升级版本，为 Linux 下视频设备程序提供了一套接口规范。包括一套数据结构和底层 V4L2 驱动接口。V4L2 驱动程序向用户空间提供字符设备，主设备号是 81。对于视频设备来说，次设备号是 0-63。如果次设备号在 64～127 之间的是 Radio 设备，次设备号在 192～223 之间的是 Teletext 设备，次设备号在 224～255 之间的是 VBI 设备。

V4L2 中常用的结构体在内核文件“include/linux/videodev2.h”中定义。

```
struct v4l2_requestbuffers      //申请帧缓冲，对应命令 VIDIOC_REQBUFS
struct v4l2_capability          //视频设备的功能，对应命令 VIDIOC_QUERYCAP
struct v4l2_input               //视频输入信息，对应命令 VIDIOC_ENUMINPUT
struct v4l2_standard            //视频的制式，比如 PAL，NTSC，对应命令 VIDIOC_ENUMSTD
struct v4l2_format              //帧的格式，对应命令 VIDIOC_G_FMT、VIDIOC_S_FMT 等
struct v4l2_buffer              //驱动中的一帧图像缓存，对应命令 VIDIOC_QUERYBUF
struct v4l2_crop                //视频信号矩形边框
v4l2_std_id                     //视频制式
```

常用的 ioctl 接口命令也在文件“include/linux/videodev2.h”中定义。

```
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              //设置当前驱动的频捕获格式
      VIDIOC_G_FMT              //读取当前驱动的频捕获格式
      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             //设置视频信号的矩形边框
      VIDIOC_G_CROP             //读取视频信号的矩形边框
      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。
```

2. 操作 V4L2 的流程

在 V4L2 中提供了很多访问接口，虽然可以根据具体需要选择操作方法，但是很少有驱动完全能够实现所有的接口功能。所以建议在使用时参考驱动源码，或仔细阅读驱动提供者的使用说明。下面简单列举一种 V4L2 的操作流程供读者朋友们参考。

（1）打开设备文件。如果需要使用非阻塞模式调用视频设备，当没有可用的视频数据时不会阻塞，而会立刻返回。

```
int fd = open(Devicename,mode);
Devicename：/dev/video0、/dev/video1 ……
Mode：O_RDWR [| O_NONBLOCK]
```

（2）获取设备的 capability。在此需要查看设备具有什么功能，比如是否具有视频输入特性。

```
struct v4l2_capability capability;
int ret = ioctl(fd, VIDIOC_QUERYCAP, &capability);
```

（3）选择视频输入。每一个视频设备可以有多个视频输入，如果只有一路输入，则可以没有这个功能。

```
struct v4l2_input input;
//……开始初始化 input
int ret = ioctl(fd, VIDIOC_QUERYCAP, &input);
```

（4）检测视频支持的制式。

```
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:
            //.....
              }
```

（5）设置视频捕获格式。

```
struct v4l2_format fmt;
fmt.type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_UYVY;
fmt.fmt.pix.height = height;
fmt.fmt.pix.width = width;
fmt.fmt.pix.field = V4L2_FIELD_INTERLACED;
ret = ioctl(fd, VIDIOC_S_FMT, &fmt);
if(ret) {
       perror("VIDIOC_S_FMT\n");
       close(fd);
       return -1;
}
```

（6）向驱动申请帧缓存。在结构 v4l2_requestbuffers 中定义了缓存的数量，驱动会根据这个申请对应数量的视频缓存。通过多个缓存可以建立 FIFO，这样可以提高视频采集的效率。

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

（7）获取每个缓存的信息，并 mmap 到用户空间。

```
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++) {//映射所有的缓存
        memset( &buf, 0, sizeof(buf) );
        buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        buf.memory = V4L2_MEMORY_MMAP;
        buf.index = numBufs;
        if (ioctl(fd, VIDIOC_QUERYBUF, &buf) == -1) {//获取到对应 index 的缓存信息，此处主要利
用 length 信息及 offset 信息来完成后面的 mmap 操作
                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;
        }
```

（8）开始采集视频。

```
int buf_type= V4L2_BUF_TYPE_VIDEO_CAPTURE;
int ret = ioctl(fd, VIDIOC_STREAMON, &buf_type);
```

（9）取出 FIFO 缓存中已经采样的帧缓存。可以根据返回的 buf.index 找到对应的 mmap 映射好的缓存，实现取出视频数据的功能。

```
struct v4l2_buffer buf;
memset(&buf,0,sizeof(buf));
buf.type=V4L2_BUF_TYPE_VIDEO_CAPTURE;
buf.memory=V4L2_MEMORY_MMAP;
buf.index=0;//此值由下面的 ioctl 返回
if (ioctl(fd, VIDIOC_DQBUF, &buf) == -1)
{
        return -1;
}
```

（10）将刚刚处理完的缓冲重新入队列尾，这样可以循环采集。

```
if (ioctl(fd, VIDIOC_QBUF, &buf) == -1) {
        return -1;
}
```

（11）停止视频的采集。

```
int ret = ioctl(fd, VIDIOC_STREAMOFF, &buf_type);
```

（12）关闭视频设备。

```
close(fd);
```

3. V4L2 驱动框架

在上述使用 V4L2 的流程中，各个操作都需要有底层 V4L2 驱动的支持。在内核中有一些非常完善的例子。例如在 Linux-2.6.26 内核目录“/drivers/media/video//zc301/”中，文件 zc301_core.c 实现了 ZC301 视频驱动代码。

（1）V4L2 驱动注册、注销函数。

在 Video 核心层文件“drivers/media/video/videodev.c”中提供了注册函数。

```
int video_register_device(struct video_device *vfd, int type, int nr)
```

- video_device：要构建的核心数据结构。
- Type：表示设备类型，此设备号的基地址受此变量的影响。
- Nr：如果 end-base>nr>0，次设备号=base（基准值，受 type 影响）+nr，否则将系统自动分配合适的次设备号。

我们具体需要的驱动只需构建 video_device 结构，然后调用注册函数即可。例如在文件 zc301_core.c 中的如下实现代码。

```
err = video_register_device(cam->v4ldev, VFL_TYPE_GRABBER,video_nr[dev_nr]);
```

在 Video 核心层文件“drivers/media/video/videodev.c”中提供了如下注销函数。

```
void video_unregister_device(struct video_device *vfd)
```

（2）构建 struct video_device。

在结构 video_device 中包含了视频设备的属性和操作方法，具体可以参考文件 zc301_core.c。

```
strcpy(cam->v4ldev->name, "ZC0301[P] PC Camera");
 cam->v4ldev->owner = THIS_MODULE;
 cam->v4ldev->type = VID_TYPE_CAPTURE | VID_TYPE_SCALES;
 cam->v4ldev->fops = &zc0301_fops;
 cam->v4ldev->minor = video_nr[dev_nr];
 cam->v4ldev->release = video_device_release;
 video_set_drvdata(cam->v4ldev, cam);
```

在上述 zc301 的驱动中，并没有实现 struct video_device 中的很多操作函数，例如 vidioc_querycap、vidioc_g_fmt_cap，这是因为在 struct file_operations zc0301_fops 中的 zc0301_ioctl 实现了前面的所有 ioctl 操作，所以无须在 struct video_device 再次实现 struct video_device 中的操作。

除此之外，也可以使用下面的代码来构建 struct video_device。

```
static struct video_device camif_dev =
  {
    .name = "s3c2440 camif",
    .type = VID_TYPE_CAPTURE|VID_TYPE_SCALES|VID_TYPE_SUBCAPTURE,
```

```
    .fops = &camif_fops,
    .minor = -1,
    .release = camif_dev_release,
    .vidioc_querycap = vidioc_querycap,
    .vidioc_enum_fmt_cap = vidioc_enum_fmt_cap,
    .vidioc_g_fmt_cap = vidioc_g_fmt_cap,
    .vidioc_s_fmt_cap = vidioc_s_fmt_cap,
    .vidioc_queryctrl = vidioc_queryctrl,
    .vidioc_g_ctrl = vidioc_g_ctrl,
    .vidioc_s_ctrl = vidioc_s_ctrl,
};
static struct file_operations camif_fops =
{
    .owner = THIS_MODULE,
    .open = camif_open,
    .release = camif_release,
    .read = camif_read,
    .poll = camif_poll,
    .ioctl = video_ioctl2, /* V4L2 ioctl handler */
    .mmap = camif_mmap,
    .llseek = no_llseek,
};
```

结构 video_ioctl2 是在文件 videodev.c 中实现的，video_ioctl2 中会根据 ioctl 不同的 cmd 来调用 video_device 中的操作方法。

4. 实现 Video 核心层

具体实现代码请参考内核文件“/drivers/media/videodev.c”，具体实现流程如下所示。

（1）注册 256 个视频设备。注册了 256 个视频设备和 video_class 类，video_fops 是这 256 个设备共同的操作方法。

```
static int __init videodev_init(void)
        {
                int ret;
                if (register_chrdev(VIDEO_MAJOR, VIDEO_NAME, &video_fops)) {
                        return -EIO;
                }
                ret = class_register(&video_class);
                ……
        }
```

（2）实现 V4L2 驱动的注册函数。注册 V4L2 驱动的过程只是创建了设备节点，例如“/dev/video0”。并且保存了 video_device 结构指针。

```
int video_register_device(struct video_device *vfd, int type, int nr)
        {
                int i=0;
                int base;
```

```
        int end;
        int ret;
        char *name_base;
    switch(type) //根据不同的 type 确定设备名称、次设备号
        {
                case VFL_TYPE_GRABBER:
                        base=MINOR_VFL_TYPE_GRABBER_MIN;
                        end=MINOR_VFL_TYPE_GRABBER_MAX+1;
                        name_base = "video";
                        break;
                case VFL_TYPE_VTX:
                        base=MINOR_VFL_TYPE_VTX_MIN;
                        end=MINOR_VFL_TYPE_VTX_MAX+1;
                        name_base = "vtx";
                        break;
                case VFL_TYPE_VBI:
                        base=MINOR_VFL_TYPE_VBI_MIN;
                        end=MINOR_VFL_TYPE_VBI_MAX+1;
                        name_base = "vbi";
                        break;
                case VFL_TYPE_RADIO:
                        base=MINOR_VFL_TYPE_RADIO_MIN;
                        end=MINOR_VFL_TYPE_RADIO_MAX+1;
                        name_base = "radio";
                        break;
                default:
                        printk(KERN_ERR "%s called with unknown type: %d\n",
                                __func__, type);
                        return -1;
        }
    /* 计算出次设备号 */
        mutex_lock(&videodev_lock);
        if (nr >= 0 && nr < end-base) {
                /* use the one the driver asked for */
                i = base+nr;
                if (NULL != video_device[i]) {
                        mutex_unlock(&videodev_lock);
                        return -ENFILE;
                }
        } else {
                /* use first free */
                for(i=base;i<end;i++)
                        if (NULL == video_device[i])
                                break;
                if (i == end) {
                        mutex_unlock(&videodev_lock);
                        return -ENFILE;
                }
        }
      video_device[i]=vfd; //保存 video_device 结构指针到系统的结构数组中，最终的次设备号
```

```
和 i 相关
            vfd->minor=i;
            mutex_unlock(&videodev_lock);
            mutex_init(&vfd->lock);
       /* sysfs class */
              memset(&vfd->class_dev, 0x00, sizeof(vfd->class_dev));
              if (vfd->dev)
                     vfd->class_dev.parent = vfd->dev;
            vfd->class_dev.class = &video_class;
            vfd->class_dev.devt = MKDEV(VIDEO_MAJOR, vfd->minor);
            sprintf(vfd->class_dev.bus_id, "%s%d", name_base, i - base);//最后在/dev 目录
下的名称
    ret = device_register(&vfd->class_dev);//结合 udev 或 mdev 可以实现自动在/dev 下创建设备节点
              ……
       }
```

（3）打开视频驱动。

使用下面的代码在用户空间调用 open()函数打开对应的视频文件。

```
int fd = open(/dev/video0, O_RDWR);
```

对应“/dev/video0”目录的文件操作结构是在文件“/drivers/media/videodev.c”中定义的 video_fops。代码如下所示。

```
static const struct file_operations video_fops=
       {
              .owner = THIS_MODULE,
              .llseek = no_llseek,
              .open = video_open,
       };
```

上述代码只是实现了 open 操作，后面的其他操作需要使用 video_open()来实现。

```
static int video_open(struct inode *inode, struct file *file)
       {
              unsigned int minor = iminor(inode);
              int err = 0;
              struct video_device *vfl;
              const struct file_operations *old_fops;
       if(minor>=VIDEO_NUM_DEVICES)
                     return -ENODEV;
              mutex_lock(&videodev_lock);
              vfl=video_device[minor];
              if(vfl==NULL) {
                     mutex_unlock(&videodev_lock);
                     request_module("char-major-%d-%d", VIDEO_MAJOR, minor);
                     mutex_lock(&videodev_lock);
                     vfl=video_device[minor]; //根据次设备号取出 video_device 结构
                     if (vfl==NULL) {
                            mutex_unlock(&videodev_lock);
```

```
                return -ENODEV;
            }
        }
        old_fops = file->f_op;
        file->f_op = fops_get(vfl->fops);//替换此打开文件的 file_operation 结构，后面
的其他针对此文件的操作都由新的结构来负责了，也就是由每个具体的 video_device 的 fops 负责
        if(file->f_op->open)
            err = file->f_op->open(inode,file);
        if (err) {
            fops_put(file->f_op);
            file->f_op = fops_get(old_fops);
        }
        ......
    }
```

14.3.2 硬件抽象层

在 Android 2.1 及其以前的版本中，Camera 系统的硬件抽象层的头文件保存在“frameworks/base/include/ui/”目录。在 Android 2.2 及其以后的版本中，Camera 系统的硬件抽象层的头文件保存在“frameworks/base/include/camera/”目录。在上述目录中主要包含了下面的头文件。

- CameraHardwareInterface.h：在里面定义了 C++接口类，此类需要根据系统的情况来实现继承。
- CameraParameters.h：在里面定义了 Camera 系统的参数，可以在本地系统的各个层次中使用这些参数。
- Camera.h：在里面提供了 Camera 系统本地对上层的接口。

1. Android 2.1 及其以前的版本

在 Android 2.1 及其以前的版本中，在文件 CameraHardwareInterface.h 中首先定义了硬件抽象层接口的回调函数类型，对应代码如下所示。

```
/** startPreview()使用的回调函数*/
typedef void (*preview_callback)(const sp<IMemory>& mem, void* user);

/** startRecord()使用的回调函数*/
typedef void (*recording_callback)(const sp<IMemory>& mem, void* user);

/** takePicture()使用的回调函数*/
typedef void (*shutter_callback)(void* user);

/** takePicture()使用的回调函数*/
typedef void (*raw_callback)(const sp<IMemory>& mem, void* user);

/** takePicture()使用的回调函数*/
typedef void (*jpeg_callback)(const sp<IMemory>& mem, void* user);

/** autoFocus()使用的回调函数*/
typedef void (*autofocus_callback)(bool focused, void* user);
```

然后定义类 CameraHardwareInterface，在类中定义了各个接口函数。代码如下所示。

```
class CameraHardwareInterface : public virtual RefBase {
public:
    virtual ~CameraHardwareInterface() { }
    virtual sp<IMemoryHeap>         getPreviewHeap() const = 0;
    virtual sp<IMemoryHeap>         getRawHeap() const = 0;
    virtual status_t    startPreview(preview_callback cb, void* user) = 0;
    virtual bool useOverlay() {return false;}
    virtual status_t setOverlay(const sp<Overlay> &overlay) {return BAD_VALUE;}
    virtual void        stopPreview() = 0;
    virtual bool        previewEnabled() = 0;
    virtual status_t    startRecording(recording_callback cb, void* user) = 0;
    virtual void        stopRecording() = 0;
    virtual bool        recordingEnabled() = 0;
    virtual void        releaseRecordingFrame(const sp<IMemory>& mem) = 0;
    virtual status_t    autoFocus(autofocus_callback,
                                  void* user) = 0;
    virtual status_t    takePicture(shutter_callback,
                                    raw_callback,
                                    jpeg_callback,
                                    void* user) = 0;
    virtual status_t    cancelPicture(bool cancel_shutter,
                                      bool cancel_raw,
                                      bool cancel_jpeg) = 0;
    /**返回 camera 系统的参数 */
    virtual CameraParameters  getParameters() const = 0;
    virtual void release() = 0;
    virtual status_t dump(int fd, const Vector<String16>& args) const = 0;
};
extern "C" sp<CameraHardwareInterface> openCameraHardware();
};
```

可以将上述代码中的接口分为如下几类。

- 取景预览：startPreview、stopPreview、useOverlay 和 setOverlay。
- 录制视频：startRecording、stopRecording、recordingEnabled 和 releaseRecordingFrame。
- 拍摄照片：takePicture 和 cancelPicture。
- 辅助功能：autoFocus（自动对焦）、setParameters 和 getParameters。

2. Android 2.2 及其以后的版本

在 Android 2.2 及其以前的版本中，在文件 Camera.h 中首先定义了通知信息的枚举值，对应代码如下所示。

```
enum {
    CAMERA_MSG_ERROR            = 0x001,    //错误信息
    CAMERA_MSG_SHUTTER          = 0x002,    //快门信息
    CAMERA_MSG_FOCUS            = 0x004,    //聚焦信息
    CAMERA_MSG_ZOOM             = 0x008,    //缩放信息
```

```
    CAMERA_MSG_PREVIEW_FRAME    = 0x010,    //帧预览信息
    CAMERA_MSG_VIDEO_FRAME      = 0x020,    //视频帧信息
    CAMERA_MSG_POSTVIEW_FRAME   = 0x040,    //拍照后停止帧信息
    CAMERA_MSG_RAW_IMAGE        = 0x080,    //原始数据格式照片信息
    CAMERA_MSG_COMPRESSED_IMAGE = 0x100,    //压缩格式照片信息
    CAMERA_MSG_ALL_MSGS         = 0x1FF     //所有信息
};
```

然后在文件 CameraHardwareInterface.h 中定义如下三个回调函数。

```
//通知回调
typedef void (*notify_callback)(int32_t msgType,
                                int32_t ext1,
                                int32_t ext2,
                                void* user);
//数据回调
typedef void (*data_callback)(int32_t msgType,
                              const sp<IMemory>& dataPtr,
                              void* user);
//带有时间戳的数据回调
typedef void (*data_callback_timestamp)(nsecs_t timestamp,
                                        int32_t msgType,
                                        const sp<IMemory>& dataPtr,
                                        void* user);
```

然后定义类 CameraHardwareInterface，在类中的各个函数和其他 Android 版本的相同。区别是回调函数不再由各个函数分别设置，所以在 startPreview 和 startRecording 缺少了回调函数的指针和 void*类型的附加参数。主要代码如下所示。

```
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,
    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 status_t    getBufferInfo(sp<IMemory>& Frame, size_t *alignedSize) = 0;
    virtual bool        useOverlay() {return false;}
    virtual status_t    setOverlay(const sp<Overlay> &overlay) {return BAD_VALUE;}
    virtual void        stopPreview() = 0;
    virtual bool        previewEnabled() = 0;
    virtual status_t    startRecording() = 0;
    virtual void        stopRecording() = 0;
    virtual bool        recordingEnabled() = 0;
    virtual void        releaseRecordingFrame(const sp<IMemory>& mem) = 0;
```

```
    virtual status_t    autoFocus() = 0;
    virtual status_t    cancelAutoFocus() = 0;
    virtual status_t    takePicture() = 0;
    virtual status_t    cancelPicture() = 0;
    virtual 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 d
}
```

因为在新版本的 Camera 系统中增加了 sendCommand()，所以需要在文件 Camera.h 中增加新命令和返回值。具体代码如下所示。

```
// 函数 sendCommand()使用的命令类型
enum {
    CAMERA_CMD_START_SMOOTH_ZOOM     = 1,
    CAMERA_CMD_STOP_SMOOTH_ZOOM      = 2,
    CAMERA_CMD_SET_DISPLAY_ORIENTATION = 3,
};

// 错误类型
enum {
    CAMERA_ERROR_UKNOWN  = 1,
    CAMERA_ERROR_SERVER_DIED = 100
};
```

3. 实现 Camera 硬件抽象层

在 startPreview()实现中保存预览回调函数并建立预览线程，在预览线程的循环中等待视频数据的到达。当视频帧到达后调用预览回调函数送出视频帧。

取景器实现预览的过程如下所示。

（1）在初始化的过程中，建立预览数据的内存队列（多种方式）。

（2）在 startPreview()中建立预览线程。

（3）在预览线程的循环中，等待视频数据到达。

（4）视频到达后使用预览回调机制将视频向上传送。

在此过程不需要使用预览回调函数，可以直接将视频数据输入到 Overlay 上。如果使用 Overlay 实现取景器，则需要有以下两个变化。

- 在 setOverlay()函数中，从 ISurface 接口中取得 Overlay 类。
- 在预览线程的循环中，不是用预览回调函数直接将数据输入到 Overlay 上。

录制视频的主要过程如下所示。

（1）在 startRecording()的实现（或者在 setCallbacks）中保存录制视频回调函数。

（2）录制视频可以使用自己的线程，也可以使用预览线程。

（3）通过录制回调函数将视频帧送出。

当调用 releaseRecordingFrame()后，表示上层通知 Camera 硬件抽象层，这一帧的内存已经用

完，可以进行下一次的处理。如果在 V4L2 驱动程序中使用原始数据（RAW），则视频录制的数据和取景器预览的数据为同一数据。当调用 releaseRecordingFrame()时，通常表示编码器已经完成了对当前视频帧的编码，对这块内存进行释放。在这个函数的实现中，可以设置标志位，标记帧内存可以再次使用。

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

（1）如果使用映射内核内存的方式（V4L2_MEMORY_MMAP），构建预览的内存 MemoryHeapBase 需要从 V4L2 驱动程序中得到内存指针。

（2）如果使用用户空间内存的方式（V4L2_MEMORY_USERPTR），MemoryHeapBase 中开辟的内存是在用户空间建立的。

（3）在预览的线程中，使用 VIDIOC_DQBUF 调用阻塞等待视频帧的到来，处理完成后使用 VIDIOC_QBUF 调用将帧内存再次压入队列，然后等待下一帧的到来。

14.4 实现 Camera 系统的硬件抽象层

在 Android 系统中已经实现了一个 Camera 硬件抽象层的“桩”，这样可以根据“宏”来配置。此“桩”使用假的方式实现取景器预览和照片拍摄功能。在 Camera 系统的“桩”实现中使用黑白格子来代替来自硬件的视频流，这样可以在不接触硬件的情况下让 Camera 系统不用硬件也可以运行。因为没有视频输出设备，所以不会使用 Overlay 来实现 Camera 硬件抽象层的“桩”。

14.4.1 Java 程序部分

在文件“packages/apps/Camera/src/com/android/camera/Camera.java”中，已经包含了对 Camera 的调用。在文件 Camera.java 中，对包的引用代码如下所示。

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

然后定义类 Camera，此类继承了活动 Activity 类，在它的内部包含了一个 android.hardware.Camera。对应代码如下所示。

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

调用 Camera 功能的代码如下所示。

```
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 {
}
```

在类 Camera 中，大部分代码使用 JNI 调用下层得到，例如下面的代码。

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

还有下面的代码。

```
public final void setPreviewDisplay(SurfaceHolder holder) {
      setPreviewDisplay(holder.getSurface());
}
private native final void setPreviewDisplay(Surface surface);
```

在上面的两段代码中，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。

14.4.2　Java 本地调用部分

Camera 的 Java 本地调用（JNI）部分在如下文件中实现。

```
frameworks/base/core/jni/android_hardware_Camera.cpp
```

在文件 android_hardware_Camera.cpp 中定义了一个 JNINativeMethod（Java 本地调用方法）类型的数组 gMethods，具体代码如下所示。

```
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_setHasPreview
Callback },
   {"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 Java 本地调用方法的名称，此名称是在 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。

通过函数 register_android_hardware_Camera()将 gMethods 注册为类“android/media/Camera”，其主要的实现如下所示。

```
int register_android_hardware_Camera(JNIEnv *env)
{
// 定义本地寄存器
return AndroidRuntime::registerNativeMethods(env, "android/hardware/Camera",camMethods,
NELEM(camMethods));
}
```

其中类“android/hardware/Camera”和 Java 类 android.hardware.Camera 相对应。

14.4.3 本地库 libui.so

文件“frameworks/base/libs/ui/Camera.cpp”的功能是实现文件 Camera.h 提供的接口，主要代码片段如下所示。

```
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;
}
```

上述代码通过如下调用代码得到一个名称为“media.camera”的服务，此调用返回值的类型是 IBinder，根据实现将其转换成类型 ICameraService 使用。

```
binder = sm->getService(String16("media.camera"));
```

函数 connect()的实现代码如下所示。

```
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;
}
```

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

函数 startPreview()的实现代码如下所示：

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

其他函数的实现过程与函数 setDataSource()的类似，在此不再详细介绍。

14.4.4　Camera 服务 libcameraservice.so

目录“frameworks/base/camera/libcameraservice/”中的文件实现了一个 Camera 服务，此服务是继承 ICameraService 的具体实现。在此目录中有如下和硬件抽象层中“桩”实现相关的文件。

- CameraHardwareStub.cpp：Camera 硬件抽象层“桩”实现。
- CameraHardwareStub.h：Camera 硬件抽象层“桩”实现的接口。
- CannedJpeg.h：包含一块 JPEG 数据，在拍照片时作为 JPEG 数据。
- FakeCamera.h 和 FakeCamera.cpp：实现假的 Camera 黑白格取景器效果。

在文件 Android.mk 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。文件 Android.mk 的主要代码如下所示。

```
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)
```

文件 CameraService.cpp 继承了 BnCameraService 的实现，在此类内部又定义了类 Client，CameraService::Client 继承了 BnCamera。在运作的过程中，函数 CameraService::connect()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。因为 CameraService::Client 本身继承了 BnCamera 类，而 BnCamera 类继承了 ICamera，所以可以将此类当成 ICamera 来使用。

类 CameraService 和 CameraService::Client 的继承代码如下所示。

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

在 CameraService 中，静态函数 instantiate()的功能是初始化一个 Camera 服务。

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

其实函数 CameraService::instantiate()注册了一个名称为“media.camera”的服务，此服务和文件 Camera.cpp 中调用的名称相对应。

Camera 整个运作机制是：在文件 Camera.cpp 中调用 ICameraService 的接口，此时实际上调用的是 BpCameraService。而 BpCameraService 通过 Binder 机制和 BnCameraService 实现两个进程的通信。因为 BpCameraService 的实现就是此处的 CameraService，所以 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，从函数 connect()中可以得到一个 ICamera 类型的指针，整个指针的实现实际上是 CameraService::Client。

上述 Camera 功能是 CameraService::Client 所实现的，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;
}
```

在构造函数中，通过调用 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。

其实真正的 Camera 功能已经通过实现 CameraHardwareInterface 类来完成。在这个库中，文件 CameraHardwareStub.h 和 CameraHardwareStub.cpp 定义了一个“桩”模块的接口，可以在没有 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文件 CameraHardwareStub.cpp 和它依赖的文件 FakeCamera.cpp。

类 CameraHardwareStub 的结构如下所示。

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

在类 CameraHardwareStub 中包含了线程类 PreviewThread，此线程可以处理 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。

在文件 FakeCamera.h 和 FakeCamera.cpp 中实现了类 FakeCamera，用于实现一个假的摄像头输入数据的内存。定义代码如下所示。

```
class FakeCamera {
public:
    FakeCamera(int width, int height);
    ~FakeCamera();

    void setSize(int width, int height);
    void getNextFrameAsRgb565(uint16_t *buffer);//获取 RGB565 格式的预览帧
    void getNextFrameAsYuv422(uint8_t *buffer);//获取 Yuv422 格式的预览帧
    status_t dump(int fd, const Vector<String16>& args);

private:
    void drawSquare(uint16_t *buffer, int x, int y, int size, int color, int shadow);
    void drawCheckerboard(uint16_t *buffer, int size);

    static const int kRed = 0xf800;
    static const int kGreen = 0x07c0;
    static const int kBlue = 0x003e;

    int        mWidth, mHeight;
    int        mCounter;
    int        mCheckX, mCheckY;
    uint16_t    *mTmpRgb16Buffer;
};
```

当在 CameraHardwareStub 中设置参数后会调用函数 initHeapLocked()，此函数的实现代码如下所示。

```
void CameraHardwareStub::initHeapLocked()
{
    int picture_width, picture_height;
   mParameters.getPictureSize(&picture_width, &picture_height);
   //建立内存堆栈，创建两块内存
   mRawHeap = new MemoryHeapBase(picture_width * 2 * picture_height);

   int preview_width, preview_height;
   mParameters.getPreviewSize(&preview_width, &preview_height);
   LOGD("initHeapLocked: preview size=%dx%d", preview_width, preview_height);

   // 从参数中获取信息
   int how_big = preview_width * preview_height * 2;

   if (how_big == mPreviewFrameSize)
      return;

   mPreviewFrameSize = how_big;

   mPreviewHeap = new MemoryHeapBase(mPreviewFrameSize * kBufferCount);
   // 建立内存队列
   for (int i = 0; i < kBufferCount; i++) {
      mBuffers[i] = new MemoryBase(mPreviewHeap, i * mPreviewFrameSize, mPreviewFrameSize);
   }

     delete mFakeCamera;
   mFakeCamera = new FakeCamera(preview_width, preview_height);
}
```

定义函数 startPrevie()来创建一个线程，此函数的实现代码如下所示。

```
status_t CameraHardwareStub::startPreview(preview_callback cb, void* user)
{
   Mutex::Autolock lock(mLock);
   if (mPreviewThread != 0) {
      // 已经启动
      return INVALID_OPERATION;
   }
   mPreviewCallback = cb;
   mPreviewCallbackCookie = user;
   mPreviewThread = new PreviewThread(this);//建立视频预览线程
   return NO_ERROR;
}
```

通过上面建立的线程可以调用预览回调机制，将预览的数据传递给上层的 CameraService。

创建预览线程函数 previewThread()，建立一个循环以得到假的摄像头输入数据的来源，并通过预览回调函数将输出传送到上层中去。函数 previewThread()的主要代码如下所示。

```
int CameraHardwareStub::previewThread()
{
    mLock.lock();
       int previewFrameRate = mParameters.getPreviewFrameRate();
        //获取当前预览缓冲区域的垂距
        ssize_t offset = mCurrentPreviewFrame * mPreviewFrameSize;
        sp<MemoryHeapBase> heap = mPreviewHeap;
            // 假设假照相机内部状态没有变化
        // (or is thread safe)
        FakeCamera* fakeCamera = mFakeCamera;
        sp<MemoryBase> buffer = mBuffers[mCurrentPreviewFrame];
            mLock.unlock();
    if (buffer != 0) {
        //计算在预览框架等待多久
        int delay = (int)(1000000.0f / float(previewFrameRate));
            //这总是合法的，即使内存消亡仍然在我们的过程中被映射
          void *base = heap->base();
          //用假照相机填装当前框架
        uint8_t *frame = ((uint8_t *)base) + offset;
        fakeCamera->getNextFrameAsYuv422(frame);

               // Notify the client of a new frame.
        mPreviewCallback(buffer, mPreviewCallbackCookie);
          //推进缓冲区
        mCurrentPreviewFrame = (mCurrentPreviewFrame + 1) % kBufferCount;
        //等待它...
        usleep(delay);
    }
    return NO_ERROR;
}
```

在上述文件中还定义了其他的函数，函数的功能一看便知，在此为节省篇幅将不再一一进行详细讲解，请读者参考开源的代码文件。

14.5 实现 Camera 系统

在本节将分别讲解在 MSM 平台实现 Camera 系统和在 OMAP 平台实现 Camera 系统的过程，为读者步入本书后面知识的学习打下基础。

14.5.1 在 MSM 平台实现 Camera 系统

在 MSM 平台中，和 Camera 系统相关的文件如下所示。

- drivers/media/video/msm/msm_v4l2.c：是 V4L2 驱动程序的入口文件。
- drivers/media/video/msm/ msm_camera.c：是公用库函数。
- drivers/media/video/msm/s5k3e2fx.c：摄像头传感器驱动文件，使用 i2c 接口控制。

文件 msm_camera.h 是和摄像头相关的头文件，在里面定义了各种额外的 ioctl 命令。

```
#define MSM_CAM_IOCTL_MAGIC 'm'
#define MSM_CAM_IOCTL_GET_SENSOR_INFO  _IOR(MSM_CAM_IOCTL_MAGIC, 1, struct msm_
camsensor_info *)
#define MSM_CAM_IOCTL_REGISTER_PMEM  _IOW(MSM_CAM_IOCTL_MAGIC, 2, struct msm_pmem_info
*)
#define MSM_CAM_IOCTL_UNREGISTER_PMEM  _IOW(MSM_CAM_IOCTL_MAGIC, 3, unsigned)
#define MSM_CAM_IOCTL_CTRL_COMMAND  _IOW(MSM_CAM_IOCTL_MAGIC, 4, struct msm_ctrl_cmd *)
#define MSM_CAM_IOCTL_CONFIG_VFE  _IOW(MSM_CAM_IOCTL_MAGIC, 5, struct msm_camera
_vfe_cfg_cmd *)
#define MSM_CAM_IOCTL_GET_STATS  _IOR(MSM_CAM_IOCTL_MAGIC, 6, struct msm_camera_stats
_event_ctrl *)
#define MSM_CAM_IOCTL_GETFRAME  _IOR(MSM_CAM_IOCTL_MAGIC, 7, struct msm_camera_get_
frame *)
#define MSM_CAM_IOCTL_ENABLE_VFE  _IOW(MSM_CAM_IOCTL_MAGIC, 8, struct camera_enable_cmd
*)
#define MSM_CAM_IOCTL_CTRL_CMD_DONE  _IOW(MSM_CAM_IOCTL_MAGIC, 9, struct camera_cmd *)
#define MSM_CAM_IOCTL_CONFIG_CMD  _IOW(MSM_CAM_IOCTL_MAGIC, 10, struct camera_cmd *)
#define MSM_CAM_IOCTL_DISABLE_VFE  _IOW(MSM_CAM_IOCTL_MAGIC, 11, struct camera_enable
_cmd *)
#define MSM_CAM_IOCTL_PAD_REG_RESET2  _IOW(MSM_CAM_IOCTL_MAGIC, 12, struct camera_
enable_cmd *)
#define MSM_CAM_IOCTL_VFE_APPS_RESET  _IOW(MSM_CAM_IOCTL_MAGIC, 13, struct camera_
enable_cmd *)
#define MSM_CAM_IOCTL_RELEASE_FRAME_BUFFER  _IOW(MSM_CAM_IOCTL_MAGIC, 14, struct camera
_enable_cmd *)
#define MSM_CAM_IOCTL_RELEASE_STATS_BUFFER  _IOW(MSM_CAM_IOCTL_MAGIC, 15, struct msm_
stats_buf *)
#define MSM_CAM_IOCTL_AXI_CONFIG  _IOW(MSM_CAM_IOCTL_MAGIC, 16, struct msm_camera_vfe
_cfg_cmd *)
#define MSM_CAM_IOCTL_GET_PICTURE  _IOW(MSM_CAM_IOCTL_MAGIC, 17, struct msm_camera_
ctrl_cmd *)
#define MSM_CAM_IOCTL_SET_CROP  _IOW(MSM_CAM_IOCTL_MAGIC, 18, struct crop_info *)
#define MSM_CAM_IOCTL_PICT_PP  _IOW(MSM_CAM_IOCTL_MAGIC, 19, uint8_t *)
#define MSM_CAM_IOCTL_PICT_PP_DONE  _IOW(MSM_CAM_IOCTL_MAGIC, 20, struct msm_snapshot_
pp_status *)
#define MSM_CAM_IOCTL_SENSOR_IO_CFG  _IOW(MSM_CAM_IOCTL_MAGIC, 21, struct sensor_cfg_
data *)
#define MSM_CAMERA_LED_OFF 0
#define MSM_CAMERA_LED_LOW 1
#define MSM_CAMERA_LED_HIGH 2
#define MSM_CAM_IOCTL_FLASH_LED_CFG  _IOW(MSM_CAM_IOCTL_MAGIC, 22, unsigned *)
#define MSM_CAM_IOCTL_UNBLOCK_POLL_FRAME  _IO(MSM_CAM_IOCTL_MAGIC, 23)
#define MSM_CAM_IOCTL_CTRL_COMMAND_2  _IOW(MSM_CAM_IOCTL_MAGIC, 24, struct msm_ctrl_cmd
*)
```

文件 msm_camera.c 辅助实现 Camera 系统的功能，在里面包含了供内核调用的文件，也提供了给用户空间的接口。其中在用户空间的设备节点就是“dev/msm_camera/”中的三个设备：配置设备 config0、控制设备 control0 和帧数据设备 frame0，上面的 ioctl 命令都是为这些设备节点使用的。

在文件 msm_camera.c 中为内核空间提供了接口。

```
int msm_v4l2_register(struct msm_v4l2_driver *drv)//注册 msm_v4l2_driver 驱动
{
       if (list_empty(&msm_sensors))
              return -ENODEV;
       drv->sync = list_first_entry(&msm_sensors, struct msm_sync, list);
       drv->open     = __msm_open;
       drv->release  = __msm_release;
       drv->ctrl     = __msm_v4l2_control;
       drv->reg_pmem = __msm_register_pmem;
       drv->get_frame = __msm_get_frame;
       drv->put_frame = __msm_put_frame_buf;
       drv->get_pict  = __msm_get_pic;
       drv->drv_poll  = __msm_poll_frame;
       return 0;
}
EXPORT_SYMBOL(msm_v4l2_register);                    //注销 msm_v4l2_driver 驱动
int msm_v4l2_unregister(struct msm_v4l2_driver *drv)
{
       drv->sync = NULL;
       return 0;
}
static int msm_device_init(struct msm_cam_device *pmsm,//开始注册 Camera 驱动
              struct msm_sync *sync,
              int node)
```

MSM 平台中的 Camera 硬件抽象层已经包含在 Android 代码中，此部分的内容保存在如下文件中。

- 文件 hardware/msm7k/libcamera/camera_ifc.h：定义 Camera 接口中的常量。
- 文件 hardware/msm7k/libcamera/QualcommCameraHardware.h：是硬件抽象层的头文件。
- 文件 hardware/msm7k/libcamera/QualcommCameraHardware.cpp：是硬件抽象层的实现。

在文件 QualcommCameraHardware.h 中定义了类 MemPool，此类表示一个内存。类 AshmemPool 和 PmemPool 是 MemPool 的继承者，PreviewPmemPool 和 RawPmemPool 是 MemPool 的继承者。实现代码如下所示。

```
struct MemPool : public RefBase {
   MemPool(int buffer_size, int num_buffers,
          int frame_size,
          int frame_offset,
          const char *name);
   virtual ~MemPool() = 0;
   void completeInitialization();
   bool initialized() const {
      return mHeap != NULL && mHeap->base() != MAP_FAILED;
   }
   virtual status_t dump(int fd, const Vector<String16>& args) const;
   int mBufferSize;
   int mNumBuffers;
```

```
    int mFrameSize;
    int mFrameOffset;
    sp<MemoryHeapBase> mHeap;
    sp<MemoryBase> *mBuffers;
    const char *mName;
};
struct AshmemPool : public MemPool {
    AshmemPool(int buffer_size, int num_buffers,
               int frame_size,
               int frame_offset,
               const char *name);
};
struct PmemPool : public MemPool {
    PmemPool(const char *pmem_pool,
             int buffer_size, int num_buffers,
             int frame_size,
             int frame_offset,
             const char *name);
    virtual ~PmemPool() { }
    int mFd;
    uint32_t mAlignedSize;
    struct pmem_region mSize;
};
struct PreviewPmemPool : public PmemPool {
    virtual ~PreviewPmemPool();
    PreviewPmemPool(int buffer_size, int num_buffers,
                    int frame_size,
                    int frame_offset,
                    const char *name);
};
struct RawPmemPool : public PmemPool {
    virtual ~RawPmemPool();
    RawPmemPool(const char *pmem_pool,
                int buffer_size, int num_buffers,
                int frame_size,
                int frame_offset,
                const char *name);
};
```

14.5.2 OMAP 平台实现 Camera 系统

在 OMAP 平台中，可以使用高级的 ISP（图像信号处理）模块通过外接（i2c 方式连接）的 Camera Sensor 驱动来获取视频帧的数据。

OMAP 平台中 Camera 系统相关的文件保存在“drivers/media/video/”目录中。此目录主要由如下三部分组成。

- Vedio for Linux 2 设备：实现文件是 omap34xxcam.h 和 omap34xxcam.c。
- ISP：实现文件是“isp”目录中的 isp.c、isph3a.c、isppreview.c、ispresizer.c，提供了通过 ISP 进行的 3A、预览、改变尺寸等功能。

- Camera Sensor 驱动：lv8093.c 或 imx046.c，使用 v4l2-int-device 结构来注册。

在文件 omap34xxcam.c 中通过 v4l2_int_master 定义了 v4l2_int 主设备，对应的代码如下所示。

```
static struct v4l2_int_master omap34xxcam_master = {
        .attach = omap34xxcam_device_register,          //注册设备
        .detach = omap34xxcam_device_unregister,        //注销设备
};
```

还需要定义 omap34xxcam_fops 来注册 video 中的 v4l2_file_operations 结构，定义代码如下所示。

```
static struct v4l2_file_operations omap34xxcam_fops = {
        .owner = THIS_MODULE,
        .unlocked_ioctl = video_ioctl2,
        .poll = omap34xxcam_poll,
        .mmap = omap34xxcam_mmap,
        .open = omap34xxcam_open,
        .release = omap34xxcam_release,
};
```

另外还需要通过文件 lv8093.c 或 imx046.c 实现 Camera 系统的传感器功能，并连接在系统的 i2c 总线上。通过结构 v4l2-int-device 从设备进行注册，在运行时被文件 omap34xxcam.c 直接调用。

OMAP 平台的 Camera 硬件抽象层是基于 OMAP 的 V4L2 驱动程序实现的，并调用 Overlay 系统作为视频输出，所以 Camera 硬件抽象层的 useOverlay()的返回值是 true。为了提高性能，需要直接映射 Overlay 中的内存以作为 Camera 输出的内存。当在 OMAP 的 Camera 硬件抽象层中调用 V4L2 驱动程序的时候，需要使用 V4L2_MEMORY_USERPTR 标识来表示来自用户空间的内存。

在 OMAP 平台的 Camera 硬件抽象层中可以使用自动对焦 AutoFocus、自动增强 AutoEnhance 和自动平衡 AutoWhiteBalance 等增强型功能。上述增强型功能是通过 OMAP SOC 内部的 ISP 模块提供的基本机制实现的，算法部分功能是由用户空间库所支持的。

14.6 借助 Sensor 驱动使用照相机系统

Sensor 是 Camera 的感光器件，是加载在 I2C 上的一个从设备，不同的 Sensor 有不同的像素值、曝光能力等，这里用的是 Ov3640 这款感光器，驱动文件为 Ov3640.c，其路径为“kernel/drivers/media/video/samsung/fimc/Ov3640.c”。其中在模块加载的时候会将 Ov3640 的驱动加载到 I2C 总线上，代码如下：

```
static __init int ov3640_init(void)
{
     //向 I2c 总线上添加 Ov3640 的驱动
     return i2c_add_driver(&ov3640_i2c_driver);
}
```

Ov3640 的驱动结构 ov3640_i2c_driver 如下。

```
static struct i2c_driver ov3640_i2c_driver = {
```

```
    .driver = {
        .name = "ov3640",
    },
    .id = I2C_DRIVERID_OV3640,
   //I2C 探测器函数
    .attach_adapter = ov3640_attach_adapter,
    .detach_client = ov3640_detach,
    .command = ov3640_command,
};
```

在驱动加载的时候会调用，下面是实现代码。

```
static int ov3640_attach_adapter(struct i2c_adapter *adap)
{
    int ret = 0;
  //向 fimc 中注册 ov3640 数据
    s3c_fimc_register_camera(&ov3640_data);
    ret = i2c_probe(adap, &addr_data, ov3640_attach);
    if (ret) {
        err("failed to attach ov3640 driver\n");
        ret = -ENODEV;
    }
    return ret;
}
```

当探测到 Ov3640 设备的时候，会将其作为一个 I2C 从设备挂到 I2C 总线上，代码如下：

```
static int ov3640_attach(struct i2c_adapter *adap, int addr, int kind)
{

    //定义一个 i2c_client
    struct i2c_client *c;
    c = kmalloc(sizeof(*c), GFP_KERNEL);
    if (!c)
        return -ENOMEM;
    memset(c, 0, sizeof(struct i2c_client));
    strcpy(c->name, "ov3640");
    c->addr = addr;
    c->adapter = adap;
    //将 ov3640_i2c_driver 作为这个 I2C 客户端的驱动
    c->driver = &ov3640_i2c_driver;
    ov3640_data.client = c;
    return i2c_attach_client(c);
}
```

当 Ov3640 挂载到 I2C 总线上，就会受到 I2C 总线的调度了，会有一个响应命令的函数。

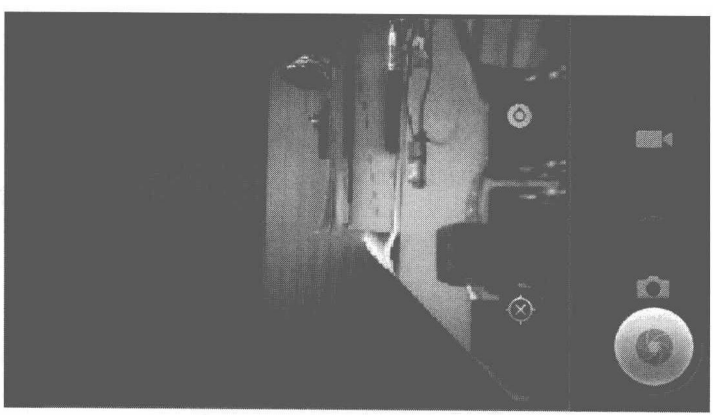
```
static int ov3640_command(struct i2c_client *client, u32 cmd, void *arg)
{
    switch (cmd) {
  //开始采集数据
```

```
        case I2C_CAM_INIT:
             ov3640_start(client);
             printk("external camera initialized\n");
             break;
       //改变分辨率
        case I2C_CAM_RESOLUTION:
             ov3640_change_resolution(client, (int) arg);
             break;
        default:
             err("unexpect command\n");
             break;
        }
        return 0;
}
```

I2C 总线上发送的命令会在这里进行响应，如开始采集数据，改变分辨率等。

```
static void ov3640_start(struct i2c_client *client)
{
     int i;
     for (i = 0; i < sizeof(ov3640_setting_15fps_VGA_640_480) / sizeof(ov3640_setting
_15fps_VGA_640_480[0]); i++) {
          s3c_fimc_i2c_write(client,
ov3640_setting_15fps_VGA_640_480[i],sizeof(ov3640_setting_15fps_VGA_640_480[i]));
     }
}
```

开始采集数据的时候，ov3640 会加载一个初始化的寄存器配置，以这个配置来采集数据，寄存器的配置对数据的影响非常大，包括采集的数据的大小、格式等属性是直接由寄存器值来决定的，这个一般是由摄像头厂家给出的配置方案，不过开发者也可以在手册的指导下进行局部微调，比如白平衡、曝光时间等，Camera 使用效果如图 14-4 所示。



▲图 14-4　开发板上的摄像机系统效果