《ANDROID 多媒体编程从初学到精通》

作者：华清远见

第 5 章　图像框架

惟有道者能备患于未形也。 -- 《管子•牧民》

Android 目前支持的图像格式有 JPEG、GIF、PNG、BMP 等，目前仅对 JPEG 提供了编码支持。

彩色图像的元数据多为 RGB 或 YCbCr 格式。与灰度图像只有一个分量表示图像的灰度不同，彩色图像通常有 3 个分量，RGB 格式的元数据采用的是红、绿、蓝 3 个分量；YCbCr 格式的元数据采用的是亮度、色调、饱和度 3 个分量，目前 YCbCr 格式的元数据占据主流地位。

图像记录的功能多是基于 Camera 来实现的，根据摄像头采用的传感器不同，支持的图像分辨率有所差别。目前基于 Android 平台已经有 500M 像素的智能终端面世。

在智能机中，图像处理多通过专有 DSP 进行相关的编解码。在本章中，将基于 Camera 进行图像记录的详细描述，简要地介绍图像解码的过程。

在 Android 2.3 中，增强了对多摄像头的支持。

5.1 Camera 拍照框架

Android 的图像编码，涉及的主要是 Camera 服务，其架构主要涉及驱动和 HAL 模块、Camera 原生服务，以及上层对服务的封装和应用等 3 个层次的内容。图 5-1 显示的是 Camera 的软件架构。

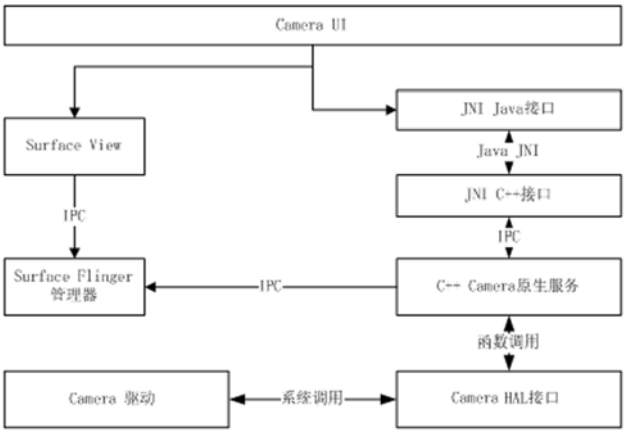

图 5-1　Camera 软件架构

1）Camera 驱动和 HAL 模块

这一层次内容主要包括运行在内核空间的 Camera 驱动和运行在用户空间的 Camera 的 HAL 实现及接口等。开发中涉及 I2C、GPIO、PMIC、VFE aDSP 等。其元数据一般为 YUV 数据或者 Bayer 数据。

2）Camera 原生服务

Camera 的原生服务主要是提供 Camera 服务的原生服务，以及提供给上层 Java 部分的原生接口等。

3）上层应用和接口

这一层次的内容主要包括调用原生服务的 Java JNI 的 Java 接口和与用户直接交互的 UI 部分。

在本章中将会着重介绍驱动和 HAL 模块、Camera 原生服务等框架性的内容，而上层应用和接口部分将会放置在 11.1 节视频记录中介绍。

5.1.1　Camera 原生服务

Camera 原生服务代码位于 frameworks\base\camera\libcameraservice 和 frameworks\base\ include\ui 目录下。在 Android 中，Camera 的原生服务是基于 C/S 架构的。

Camera 原生服务主要的文件包括：Camera.h、ICamera.h、ICameraClient.h、ICameraService.h、CameraService.cpp、CameraHardwareStub.cpp、FakeCamera.cpp 等。其中 FakeCamera.cpp 提供了在模拟器上的 Camera 仿真功能。图 5-2 显示了 Camera 原生服务框架的类图。

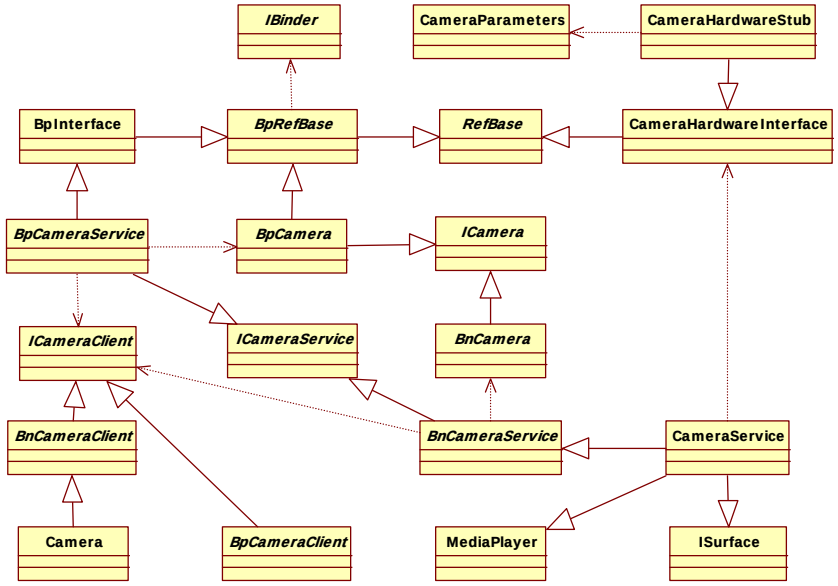

图 5-2　Camera 原生服务框架类图

其中在 ICamera 接口内部有两个内部类：BpCamera 和 BnCamera，BnCamera 为本地 ICamera 对象，BpCamera 为远程 ICamera 对象在本地进程的代理。ICameraClient 为 Camera 原生服务的客户端，ICameraClient 定义了若干个回调函数，接收服务器端传来的拍照音、预览、拍照、录像的回调数据。ICameraService 为 Camera 原生服务的服务器端接口。

在 C++层，Camera 原生服务和其他服务一样，客户端和服务器端的通信都是基于 IBinder 进行的。

为了使上层应用能够利用 Camera 原生服务，首先需要将 Camera 原生服务在服务管理器中注册。下面是 Camera 原生服务的注册过程：

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

为了使用 Camera 原生服务，首先需要从服务管理器中获得 Camera 原生服务的句柄。具体如下：

代码 5-1　获得 Camera 原生服务的句柄

```
const sp<ICameraService>& Camera::getCameraService()
{
    Mutex::Autolock _l(mLock);
    if (mCameraService.get()==0) {
        sp<IServiceManager> sm=defaultServiceManager(); //服务管理器
        sp<IBinder> binder;
        do {
            binder=sm->getService(String16("media.camera")); //Camera 原生服务
            if (binder != 0)
                break;
            LOGW("CameraService not published, waiting...");
            usleep(500000); // 0.5s
        } while(true);
        if (mDeathNotifier==NULL) {
            mDeathNotifier=new DeathNotifier();
        }
        binder->linkToDeath(mDeathNotifier);
        mCameraService=interface_cast<ICameraService>(binder);
    }
    LOGE_IF(mCameraService==0, "no CameraService!?");
    return mCameraService;
}
```

另外，Camera 的原生服务和 Java 层的原生接口实现位于 android_hardware_Camera.cpp 文件中。在 Camera 中，原生接口包括 native_setup、native_release、setPreviewDisplay、startPreview、stopPreview、previewEnabled 、 setHasPreviewCallback 、 native_autoFocus 、 native_takePicture 、 native_setParameters 、 native_getParameters、reconnect、lock、unlock 等。

下面简要介绍拍照的处理流程和 Camera 对数据的处理。

1．拍照的处理流程

为了执行 Camera 的功能，Camera 应用需要完成和 Camera 原生服务的一系列通信，如启动预览、自动对焦、拍照等，以及基于模拟器下 CameraHardwareStub 的实现。Camera 应用调用 Camera 原生接口的过程如图 5-3 所示。

流程说明：

在系统启动时，Android Runtime 会调用 register_android_hardware_Camera()注册 Camera 相关的 navtive 函数到 JNI。当 Camera 应用启动后，首先会通过 Camera 客户端向 Camera 原生服务发出连接 Camera 设备的请求。Camera 原生服务在收到连接 Camera 设备的请求后，创建一个内部类 Client 的对象，调用 HAL 的 CameraHardwareInterface::openCameraHardware()方法打开 Camera 设备，在模拟器环境下，会创建一个 CameraHardwareStub 对象，配置初始化参数。调用 HAL 的 CameraHardwareStub::useOverlay()方法，设置图像的渲染方式为 Overlay。

在 Aurora 中，在默认情况下，预览帧速率为 15 帧/秒，预览帧元数据为 YUV 420，拍照分辨率为 2048×1536 像素，图像质量为 100。参数初始化工作在 QualcommCamera Hardware:: initDefaultParameters()方法中实现。

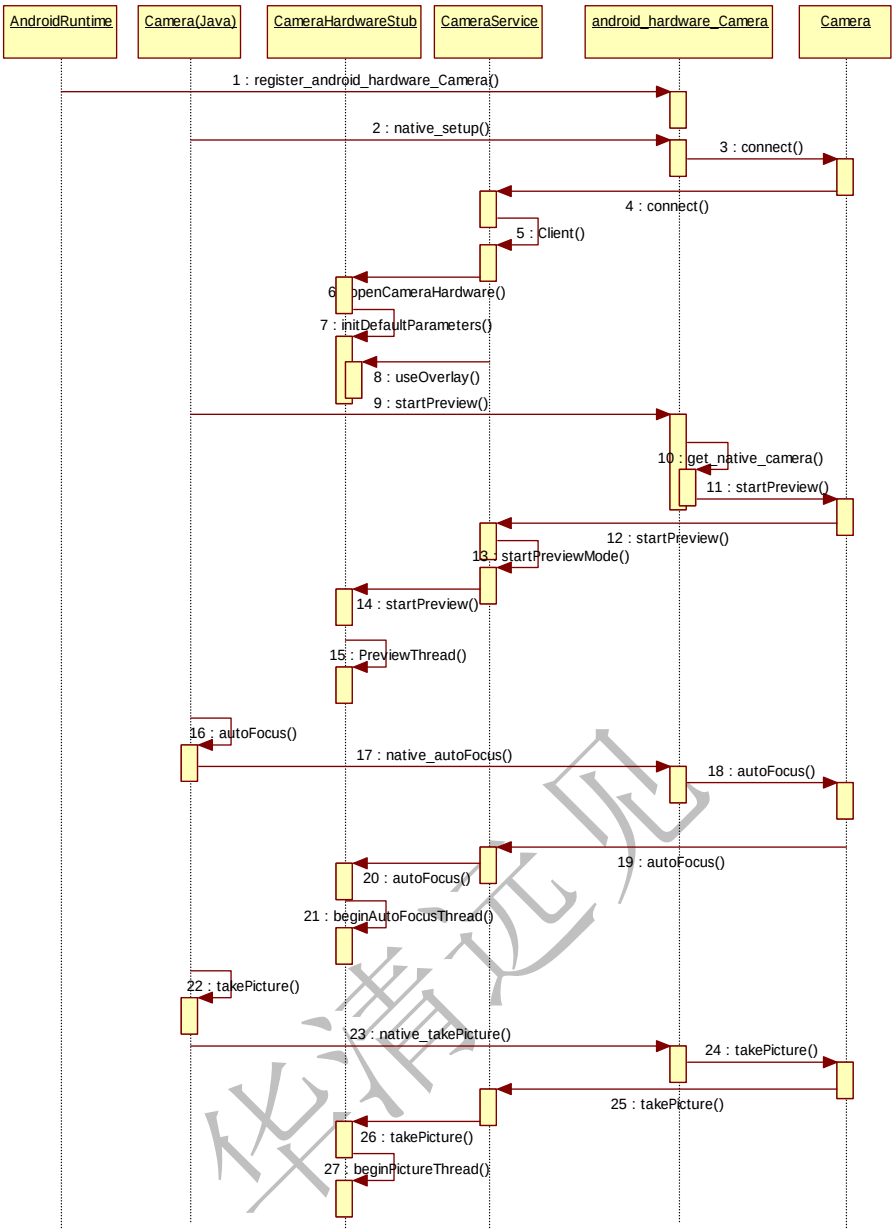

图 5-3　Java 层调用 Camera 原生服务的过程

当 Java 层发出预览请求后，经过 Camera 客户端的转发，Camera 原生服务会调用 HAL 层的 CameraHardwareStub::startPreview()方法创建一个预览线程，在该线程中接收硬件上传的数据。如果是在模拟器上运行，则将 FakeCamera 的数据传给 UI，在原始 Android 代码中，默认的预览分辨率为 176×144 像素（QCIF）。在 Aurora 中，目前支持的预览分辨率为 WVGA、VGA、HVGA、CIF、QVGA、QCIF。

当 Java 层发出自动对焦请求后，经过 Camera 客户端的转发，Camera 原生服务会调用 HAL 层的 CameraHardwareStub:: autoFocus ()方法创建一个自动对焦的线程。

当 Java 层发出拍照请求后，经过 Camera 客户端的转发，Camera 原生服务会调用 HAL 层的 CameraHardwareStub:: takePicture ()方法完成 Camera 的拍照工作，然后通过回调函数传给 UI。

2．数据的处理

当 Camera 原生服务有数据上传给客户端时，在 ICameraClient 的数据回调函数中会收到数据，具体如下：

代码 5-2　客户端的数据回调处理

```
    void CameraService::Client::dataCallback(int32_t msgType, const sp<IMemory>& dataPtr, void* user)
```

```
{
    LOGV("dataCallback(%d)", msgType);
    sp<Client> client=getClientFromCookie(user);     //获取客户端
    if (client==0) {
        return;
    }
    sp<ICameraClient> c=client->mCameraClient;       //获取 ICameraClient
    if (dataPtr==NULL) {       //如果没有数据
        LOGE("Null data returned in data callback");
        if (c != NULL) {
            c->notifyCallback(CAMERA_MSG_ERROR, UNKNOWN_ERROR, 0);
            c->dataCallback(msgType, NULL);
        }
        return;
    }

    switch (msgType) {
        case CAMERA_MSG_PREVIEW_FRAME:           //预览数据帧
            client->handlePreviewData(dataPtr);
            break;
        case CAMERA_MSG_POSTVIEW_FRAME:          //准备预览
            client->handlePostview(dataPtr);
            break;
        case CAMERA_MSG_RAW_IMAGE:               //原始图像
            client->handleRawPicture(dataPtr);
            break;
        case CAMERA_MSG_COMPRESSED_IMAGE:        //压缩图像
            client->handleCompressedPicture(dataPtr);
            break;
        default:
            if (c !=NULL) {
                c->dataCallback(msgType, dataPtr);
            }
            break;
    }
#if DEBUG_CLIENT_REFERENCES
    if (client->getStrongCount()==1) {
LOGE("++++++++++++++++ (DATA CALLBACK) THIS WILL CAUSE A LOCKUP!");
        client->printRefs();
    }
#endif
}
```

当客户端的数据回调函数收到的消息类型为 CAMERA_MSG_PREVIEW_FRAME 时，客户端通过 handlePreviewData()方法进行处理；当消息类型为 CAMERA_MSG_ POSTVIEW_FRAME 时，客户端通过 handlePostview()方法进行处理，然后向上层应用传递 CAMERA_MSG_PREVIEW_FRAME 消息；当消息类型为 CAMERA_MSG_RAW_IMAGE 时，客户端通过 handleRawPicture()方法进行处理，将拍摄的图像渲染在屏幕上；当消息类型为 CAMERA_MSG_COMPRESSED_IMAGE 时，客户端通过 handleCompressedPicture()方法进行处理，将 JPG 数据传递给上层应用。

在 handlePreviewData()方法中，会将收到的数据进行屏幕渲染，下面是 handlePreviewData()方法的实现：

代码 5-3　预览数据处理

```
void CameraService::Client::handlePreviewData(const sp<IMemory>& mem)
{
    ssize_t offset;
    size_t size;
//获取偏移地址和大小，格式为 YUV
    sp<IMemoryHeap> heap=mem->getMemory(&offset, &size);
#if DEBUG_HEAP_LEAKS && 0 // debugging
```

```
    if (gWeakHeap==NULL) {
        if (gWeakHeap != heap) {
LOGD("SETTING PREVIEW HEAP");
            heap->trackMe(true, true);
            gWeakHeap=heap;
        }
    }
#endif
#if DEBUG_DUMP_PREVIEW_FRAME_TO_FILE
    {
        if (debug_frame_cnt++==DEBUG_DUMP_PREVIEW_FRAME_TO_FILE) {
            dump_to_file("/data/preview.yuv",  //调试用
                        (uint8_t *)heap->base() + offset, size);
        }
    }
#endif
    if (!mUseOverlay)
    {
        Mutex::Autolock surfaceLock(mSurfaceLock);
        if (mSurface !=NULL) {
            mSurface->postBuffer(offset);  //传递给 Surface
        }
    }
    int flags=mPreviewCallbackFlag;
    if (!(flags & FRAME_CALLBACK_FLAG_ENABLE_MASK)) {
        LOGV("frame callback is diabled");
        return;
    }
    sp<ICameraClient> c=mCameraClient;
    if ((c==NULL) || (mPreviewCallbackFlag & FRAME_CALLBACK_FLAG_ONE_SHOT_MASK)) {
        LOGV("Disable preview callback");
        mPreviewCallbackFlag &= ~(FRAME_CALLBACK_FLAG_ONE_SHOT_MASK
FRAME_CALLBACK_FLAG_COPY_OUT_MASK|FRAME_CALLBACK_FLAG_ENABLE_MASK);
        if (mUseOverlay)
            mHardware->disableMsgType(CAMERA_MSG_PREVIEW_FRAME);
    }

    if (flags & FRAME_CALLBACK_FLAG_COPY_OUT_MASK) {
        LOGV("frame is copied");
        copyFrameAndPostCopiedFrame(c, heap, offset, size);    //将数据存入预览缓冲
    } else {
        LOGV("frame is forwarded");
        c->dataCallback(CAMERA_MSG_PREVIEW_FRAME, mem);        //向上层传递消息
    }
}
```

当上层收到数据时，执行保持数据的方法为 ImageCapture:: storeImage()。

5.1.2 Camera 的 HAL 接口

在 Linux 系统中，硬件平台驱动，以及其他需要商业保护的部分通常都会通过 HAL 来封装，在 Aurora 中也如此。

目前产业界的 Camera 传感器主要有两种类型：电荷耦合设备（CCD，Charge Couple Device）和互补金属氧化物半导体（CMOS，Complementary Metal Oxide Semiconductor）。其中 CCD 主要应用在高档的 DC、DV 和高档移动终端上，图像质量较好，但驱动模组比较复杂；而 CMOS 则门槛较低，工艺更加成熟，成本较低，外围电路也比较简单，很多厂商已将驱动和信号处理的图像信号处理器（ISP，Image Signal Processor）集成到驱动模组中。因此 CMOS 传感器在移动终端中应用广泛。如图 5-4 所示为 Camera 进行图像编码的底层过程。

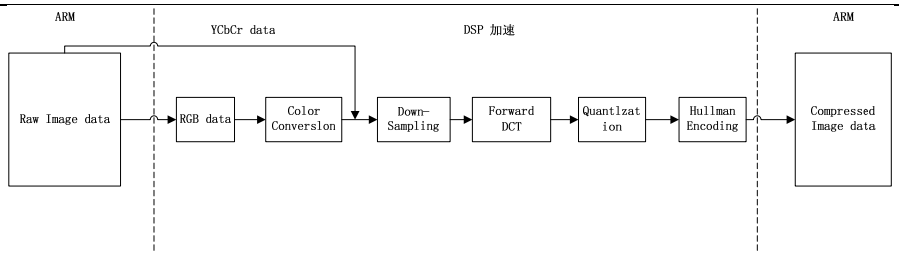

图 5-4　图像编码过程

与 Camera 传感器通信，需涉及 I^2C 驱动；进行文件保存，需涉及 GPIO 驱动和 PMIC 驱动；利用 aDSP 解码，就需要用到 VFE aDSP 驱动。

在这些平台驱动之上，在 Android 的较新版本中，通常会封装一层 HAL，可以为开发者提供更加抽象的接口，同时也可以保护硬件厂商的利益。

在 Android 中，为了实现 Camera 的 HAL 封装，必须继承 CameraHardwareInterface.h 定义的 CameraHardwareInterface 接口。

在 Android 原始代码中，提供了 CameraHardwareStub.cpp 和 QualcommCamera Hardware.cpp 两种实现，CameraHardwareStub 提供了基于模拟器的 FakeCamera 实现。QualcommCameraHardware 实现了对真实物理设备的 HAL 封装。下面首先介绍 CameraHardwareStub 的实现。

在 CameraHardwareStub 中，为了保证系统的流畅运行，在发起预览时，会将 Camera 预览放置在一个单独的预览线程 CameraHardwareStub::previewThread() 中，并在线程的事件循环中周期性调用 previewThread，从底层提取数据。CameraHardwareStub::previewThread()的实现如下：

代码 5-4　预览线程的事件循环处理

```
int CameraHardwareStub::previewThread()
{
    mLock.lock();
        int previewFrameRate=mParameters.getPreviewFrameRate();    //帧速率
        ssize_t offset=mCurrentPreviewFrame * mPreviewFrameSize;   //偏移量
        sp<MemoryHeapBase> heap=mPreviewHeap;
        FakeCamera* fakeCamera=mFakeCamera;
        sp<MemoryBase> buffer=mBuffers[mCurrentPreviewFrame]; //数据缓冲
    mLock.unlock();
    if (buffer != 0) {
              //计算帧延迟
        int delay=(int)(1000000.0f / float(previewFrameRate));
        void *base=heap->base();
              //用假数据填充当前帧
        uint8_t *frame=((uint8_t *)base) + offset;
        fakeCamera->getNextFrameAsYuv422(frame);  //获取数据缓冲
              //通知客户端有新帧到来
        if (mMsgEnabled & CAMERA_MSG_PREVIEW_FRAME)
            mDataCb(CAMERA_MSG_PREVIEW_FRAME, buffer, mCallbackCookie);
        mCurrentPreviewFrame=(mCurrentPreviewFrame + 1) % kBufferCount;
        usleep(delay);
    }
    return NO_ERROR;
}
```

在 CameraHardwareStub 中，目前共有 4 个缓冲用于 Camera 数据的交替存放，这些数据被放置在一个名为 mPreviewHeap 的预览内存堆中。具体实现如下：

代码 5-5　Camera 堆的初始化

```
void CameraHardwareStub::initHeapLocked()
{
    int picture_width, picture_height;
    mParameters.getPictureSize(&picture_width, &picture_height);//拍摄大小
```

```
        //MMAP 映射，存在拍摄时的原始数据
        mRawHeap=new MemoryHeapBase(picture_width * 2 * picture_height);
        int preview_width, preview_height;
        mParameters.getPreviewSize(&preview_width, &preview_height);//预览大小
        LOGD("initHeapLocked: preview size=%dx%d", preview_width, preview_height);
             //强制设置为 yuv422
        int how_big=preview_width * preview_height * 2;
        if (how_big==mPreviewFrameSize)
           return;
        mPreviewFrameSize=how_big;
             //构建一个新 MMAP 堆用于进程间共享
        mPreviewHeap=new MemoryHeapBase(mPreviewFrameSize * kBufferCount);
             //为每帧都构建一个 IMemory
        for (int i=0; i < kBufferCount; i++) {
          mBuffers[i]=new MemoryBase(mPreviewHeap, i * mPreviewFrameSize, mPreviewFrameSize);
        }

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

模拟器默认配置下，预览分辨率为 176×144 像素，帧速率为 15fps,预览数据格式为 YUV422 SP（仅支持），照片编码格式为 JPEG（仅支持）。

出于某种需求，需要将元数据格式 YUV422SP 转换为 RGB565，yuv420sp2rgb.c 文件给出的方法如下：

代码 5-6　YUV422sp 转换为 RGB565 的实现

```
static public void decodeYUV420SP(int[] rgb, byte[] yuv420sp, int width, int height) {
    final int frameSize=width * height;

    for (int j=0, yp=0; j<height; j++) {
        int uvp=frameSize + (j>>1) * width, u=0, v=0;
        for (int i=0; i<width; i++, yp++) {
            int y=(0xff & ((int) yuv420sp[yp])) - 16;
            if (y<0) y=0;
            if ((i & 1)==0) {
                v=(0xff & yuv420sp[uvp++]) - 128;
                u=(0xff & yuv420sp[uvp++]) - 128;
            }

            int y1192=1192 * y;
            int r=(y1192+1634 * v);
            int g=(y1192-833 * v-400*u);
            int b=(y1192+2066 * u);

            if (r<0) r=0; else if (r>262143) r=262143;
            if (g<0) g=0; else if (g>262143) g=262143;
            if (b<0) b=0; else if (b>262143) b=262143;

            rgb[yp]=0xff000000 | ((r<<6) & 0xff0000) | ((g>>2) & 0xff00) | ((b>>10) & 0xff);
        }
    }
}
```

如果希望在预览时就做这样的转换，编码的转换最好放置在一个单独的线程中处理。RGB565 向 YUV422SP 的转换参考 FakeCamera.cpp 文件。

在进行自动对焦和拍照时，CameraHardwareStub 同样会发起一个新的线程进行处理，相关的线程为 autoFocusThread、pictureThread。在 QualcommCameraHardware 中，则没有发起新线程进行处理。

在进行拍照时，为了向上层传递假图片，CameraHardwareStub 的处理方法如下：

代码 5-7　CameraHardwareStub 拍摄照片的过程

```
int CameraHardwareStub::pictureThread()
```

```
    {
        if (mMsgEnabled & CAMERA_MSG_SHUTTER)
            mNotifyCb(CAMERA_MSG_SHUTTER,0,0,mCallbackCookie);
        if (mMsgEnabled & CAMERA_MSG_RAW_IMAGE) {
            int w,h;
            mParameters.getPictureSize(&w, &h);//获得图片大小
            sp<MemoryBase> mem=new MemoryBase(mRawHeap,0,w * 2 * h);
            FakeCamera cam(w,h);
            cam.getNextFrameAsYuv422((uint8_t *)mRawHeap->base());//获取数据缓冲
            mDataCb(CAMERA_MSG_RAW_IMAGE, mem, mCallbackCookie);
        }

        if (mMsgEnabled & CAMERA_MSG_COMPRESSED_IMAGE) {
            sp<MemoryHeapBase> heap=new MemoryHeapBase(kCannedJpegSize);
            sp<MemoryBase> mem=new MemoryBase(heap,0,kCannedJpegSize);
            memcpy(heap->base(),kCannedJpeg, kCannedJpegSize);//填充假图片数据
            mDataCb(CAMERA_MSG_COMPRESSED_IMAGE,mem,mCallbackCookie);
        }
        return NO_ERROR;
    }
```

当然在真实终端环境下，基于 CameraHardwareInterface 的实现必须通过和 Camera 驱动进行通信来获得真实数据。其实现和 CameraHardwareStub 存在着差异。下面为 QualcommCameraHardware 公开的 JPEG 编码过程：

代码 5-8　QualcommCameraHardware 编码的过程

```
    unsigned char QualcommCameraHardware::native_jpeg_encode (
      void *pDim,
      int pmemThumbnailfd,
      int pmemSnapshotfd,
      unsigned char *thumbnail_buf,
      unsigned char *main_img_buf,   //存放拍摄的元数据缓冲
      void *pCrop)
    {
      char jpegFileName[256]={0};
      static int snapshotCntr=0;

      cam_ctrl_dimension_t *dimension=(cam_ctrl_dimension_t *)pDim;
      common_crop_t *cropInfo=(common_crop_t *)pCrop;

      sprintf(jpegFileName,"snapshot_%d.jpg", ++snapshotCntr);//获得文件名

    #ifndef SURF8K
      LOGV("native_jpeg_encode   ,   current   jpeg   main   img   quality   =%d",   mParameters.getJpeg
MainimageQuality());
    //设置图像质量
        if (! LINK_jpeg_encoder_setMainImageQuality(mParameters.getJpegMainimageQuality())) {
          LOGE("native_jpeg_encode set jpeg main image quality :%d@%s: jpeg_encoder_encode failed.\n",
            __LINE__, __FILE__);
          return FALSE;
      }
    #endif
      //调用 libmmcamera.so 中的 jpeg_encoder_encode 进行解码
      if ( !LINK_jpeg_encoder_encode(jpegFileName, dimension,
                            thumbnail_buf, pmemThumbnailfd,
          main_img_buf, pmemSnapshotfd, cropInfo)) {
          LOGV("native_jpeg_encode:%d@%s: jpeg_encoder_encode failed.\n", __LINE__, __FILE__);
        return FALSE;
      }
      return TRUE;
    }
```

在 Qualcomm 的实现中，底层的细节被封装在名为 libmmcamera.so 和 libmmcamera_ target.so 的共享库中，代码并没有公开。

5.1.3 Camera 参数的设置

在 Aurora 中，目前提供了对白平衡（Write Balance）、效果（Effect）、亮度（brightness）、反冲带（antibanding）、iso、夜间模式、旋转、luma、质量、帧速率等参数的默认支持。下面以白平衡为例进行介绍。

在进一步了解白平衡前，需要首先了解黑体和色温的概念。

根据基尔霍夫辐射定律，在热平衡状态的物体所辐射的能量与吸收的能量之比与物体本身物性无关，只与波长和温度有关。而辐射出去的能量在各个波段是不同的，具有一定的谱分布。这种谱分布与物体本身的特性及其温度有关，称之为热辐射。

如果入射的能量能全部被吸收，则既没有反射，也没有透射，这种理想物体就是黑体。

英国著名物理学家 Kelvin 认为，假定某一黑体物质，能够将落在其上的所有热量吸收，而没有损失，同时又能够将热量生成的能量全部以“光”的形式释放出来，它便会因受到热力的高低而变成不同的颜色。颜色成分与该黑体所受的热力温度是相对应的，该温度即色温。

当黑体受到的热力相当于 500℃～550℃时，就会变成暗红色，达到 1050℃～1150℃时，就变成黄色，温度继续升高会呈现蓝色。

由于人眼具有独特的适应性，使我们有的时候不能发现色温的变化。比如在钨丝灯下待久了，并不会觉得钨丝灯下的白纸偏红，如果突然把日光灯改为钨丝灯照明，就会有白纸的颜色偏红的感觉，但这种意识也只能够持续短暂的时间。

与人眼不同，数码相机所采用的感光材料不具备这种适应能力，如果摄像机的色彩调整同景物照明的色温不一致就会发生偏色。

白平衡就是针对不同色温条件，通过调整数码相机内部的色彩电路，使拍摄出来的影像抵消偏色，更接近人眼的视觉习惯。白平衡可以简单地理解为在任意色温条件下，数码相机所拍摄的标准白色经过电路的调整，使之成像后仍然为白色。

所谓自动白平衡，是指依赖数码相机里的测色温系统，测出红光和蓝光的相对比例。再依据此数据调整曝光，产生红、绿、蓝电信号的增益。自动白平衡最大的优势是简单、快捷。

白平衡是图像处理领域的一个重要概念，主要用来解决色彩还原和色调处理的一系列问题。

在 Aurora 中，拍照参数的设置都是通过 Camera 驱动来进行的，Camera 驱动的实现遵循 Linux 驱动的统一模型，为了设置参数，需要通过调用 ioctl 来进行。下面是 Aurora 中对白平衡参数的设置：

代码 5-9　白平衡参数的设置

```
void QualcommCameraHardware::setSensorWBLighting(int camfd, const char* lighting)
{
     int ioctlRetVal=TRUE, lightingValue=1;
     struct msm_ctrl_cmd_t ctrlCmd;
     ctrlCmd.timeout_ms=5000;                                  //超时时间
     ctrlCmd.type=CAMERA_SET_PARM_WB;                          //参数类型
     ctrlCmd.length=sizeof(uint32_t);
       ctrlCmd.value=NULL;
   for (int i=1; i<MAX_WBLIGHTING_EFFECTS; i++) {
     if (! strcmp(wb_lighting[i], lighting)) {
              LOGV("In setSensorWBLighting : Match : %s : %d ", lighting, i);
              lightingValue=i;
              ctrlCmd.value=(void *)&lightingValue;      //白平衡值
              break;
          }
     }
   if(ctrlCmd.value!=NULL)
{
```

```
            //向 Camera 驱动发送命令
        if((ioctlRetVal=ioctl(camfd, MSM_CAM_IOCTL_CTRL_COMMAND, &ctrlCmd))<0) {
        LOGV("setSensorWBLighting : ioctl failed. ioctl return value is %d \n", ioctlRetVal);
      }
    }
    else
    {
        LOGV(" setSensorWBLighting : No match found for %s ", lighting);
    }
}
```

其他 Camera 参数的设置和白平衡类似，相关的实现在 QualcommCameraHardware.cpp 文件中均可找到，这里就不再过多叙述了。

5.2 重要数据结构

为了显示图像，需要对图像进行解码，在智能终端中，具体的解码过程通常都需要借助 aDSP 来实现。如图 5-5 所示为图像的解码过程。

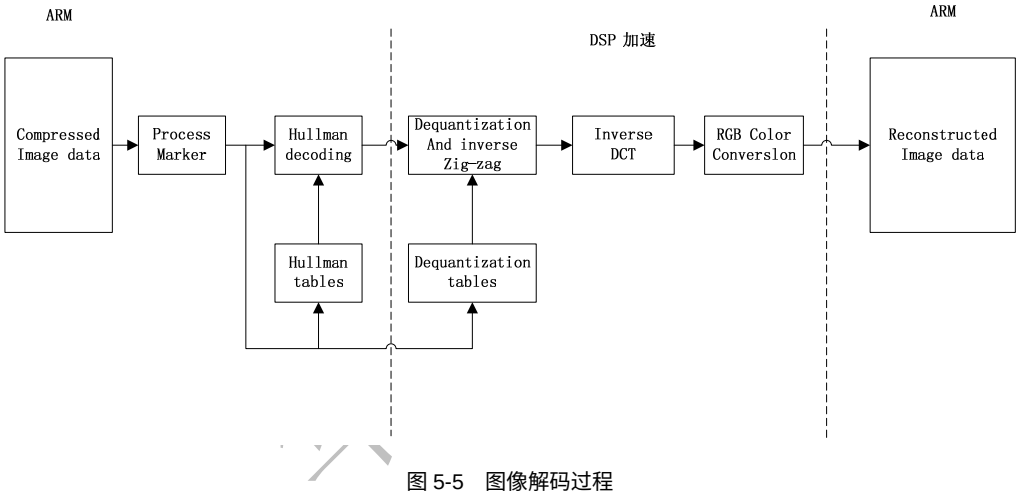

图 5-5　图像解码过程

在 Android 中，图像的解码均是基于 ImageDecoder 基类的，为了进行解码，需要通过 BitmapFactory:: decodeStream()方法对输入流进行处理。下面是 BitmapFactory:: decodeStream()方法的实现：

代码 5-10　BitmapFactory:: decodeStream()的实现

```
public static Bitmap decodeStream(InputStream is, Rect outPadding, Options opts) {
        if (is==null) {
            return null;
        }
        if (!is.markSupported()) {
            is=new BufferedInputStream(is, 16 * 1024);//创建一个缓冲
        }
        is.mark(1024);
        Bitmap  bm;
        if (is instanceof AssetManager.AssetInputStream) {
            return null;
        } else {
            try {
                bm=new Bitmap(is);  //进行解码
    } catch (IOException e) {
                return null;
            }
```

```
        }
        return finishDecode(bm, outPadding, opts);
    }
    private static Bitmap finishDecode(Bitmap bm, Rect outPadding, Options opts) {
        if (bm==null || opts==null) {
            return bm;
        }
        final int density=opts.inDensity;
        if (density==0) {
            return bm;
        }
        bm.setDensity(density); //设置密度
        final int targetDensity=opts.inTargetDensity;
        if (targetDensity==0 || density==targetDensity
                || density==opts.inScreenDensity) {
            return bm;
        }
        byte[] np=bm.getNinePatchChunk();
        final boolean isNinePatch=false; //np != null && NinePatch.isNinePatchChunk(np);
        if (opts.inScaled || isNinePatch) {
            float scale=targetDensity / (float)density;  //伸缩因子
            final Bitmap oldBitmap=bm;
            bm=Bitmap.createScaledBitmap(oldBitmap, (int) (bm.getWidth() * scale + 0.5f),
(int) (bm.getHeight() * scale + 0.5f), true);//创建伸缩图像
            oldBitmap.recycle();

            if (isNinePatch) {
                bm.setNinePatchChunk(np);
            }
            bm.setDensity(targetDensity);     //设置密度
        }
        return bm;
    }
```

最终的解码工作则是通过 SGL 引擎来进行的，关于 SGL 引擎的更多内容，可以参考 8.2.2 节 Skia 图像渲染。

联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路银海大厦 A 座 8 层，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-22193762

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218