android4.2.2 Camera HAL的结构

时间 2014-04-03 15:38:30 CSDN博客

原文 http://blog.csdn.net/gzzaigcnforever/article/details/22801445

主题 安卓开发

这里单独以preview的控制和数据流来进行相关的camera的调用处理，主要先引入Camera 的HAL层的处理结构。

调用还是先从camera的JNI和HAL两个方面来分析：

step1： 启动预览startPreview()

```
// start preview mode
status_t Camera::startPreview()
{
    ALOGV("startPreview");
    sp <ICamera> c = mCamera;
    if (c == 0) return NO_INIT;
    return c->startPreview();
}
```

这里的mCamera是之前connect请求CameraService建立，该类是匿名的BpCamera直接和CameraService处的CameraClient（该类继承了CameraService的内部类Client，Client继承了BnCamera）进行交互。

step2:调用CameraService侧的CameraClient里的startpreview()

```
status_t CameraClient::startPreviewMode() {
  LOG1("startPreviewMode");
  status_t result = NO_ERROR;

  // if preview has been enabled, nothing needs to be done
  if (mHardware->previewEnabled()) {//使能预览
    return NO_ERROR;
  }
```

```
  if (mPreviewWindow != 0) {
    native_window_set_scaling_mode(mPreviewWindow.get(),
        NATIVE_WINDOW_SCALING_MODE_SCALE_TO_WINDOW);
    native_window_set_buffers_transform(mPreviewWindow.get(),
        mOrientation);
  }
  mHardware->setPreviewWindow(mPreviewWindow);//mPreviewWindow为一个本地窗口ANativeWindow
  result = mHardware->startPreview();

  return result;
}
```

这里出现了一个mPreviewWIndow对象，其类为ANativeWindow，很熟悉的一个应用端的本地窗口。那么这个窗口的初始化过程呢，即这个变量是哪里来的?

step3:探秘本地预览窗口mPreviewWinodw对象

java处:

```
public final void setPreviewDisplay(SurfaceHolder holder) throws IOException {
        if (holder != null) {
            setPreviewDisplay(holder.getSurface());
        } else {
            setPreviewDisplay((Surface)null);
        }
    }
```

这个getSurface()的获取调用如下

```
static sp<Surface> getSurface(JNIEnv* env, jobject surfaceObj) {
  sp<Surface> result(android_view_Surface_getSurface(env, surfaceObj));
  if (result == NULL) {
    /*
     * if this method is called from the WindowManager's process, it means
     * the client is is not remote, and therefore is allowed to have
     * a Surface (data), so we create it here.
     * If we don't have a SurfaceControl, it means we're in a different
     * process.
     */

    SurfaceControl* const control = reinterpret_cast<SurfaceControl*>(
        env->GetIntField(surfaceObj, gSurfaceClassInfo.mNativeSurfaceControl));
    if (control) {
      result = control->getSurface();
      if (result != NULL) {
        result->incStrong(surfaceObj);
        env->SetIntField(surfaceObj, gSurfaceClassInfo.mNativeSurface,
            reinterpret_cast<jint>(result.get()));
      }
    }
  }
  return result;
}
```

看到这里可以回顾到 从Android Bootanimation理解SurfaceFlinger的客户端建立 这一文中，对客户端的一个Surface建立，这里的过程几乎一摸一样，最终返回一个客户端需要的Surface用来绘图使用。

而这个surface最终也将进一步传递到JNI，HAL供实时的预览等。

JNI处:

```
static void android_hardware_Camera_setPreviewDisplay(JNIEnv *env, jobject thiz, jobject jSurfac
{
  ALOGV("setPreviewDisplay");
  sp<Camera> camera = get_native_camera(env, thiz, NULL);
  if (camera == 0) return;

  sp<Surface> surface = NULL;
  if (jSurface != NULL) {
    surface = reinterpret_cast<Surface*>(env->GetIntField(jSurface, fields.surface));
  }
  if (camera->setPreviewDisplay(surface) != NO_ERROR) {
    jniThrowException(env, "java/io/IOException", "setPreviewDisplay failed");
  }
}
```

来到JNI层的实现，获取之前由CameraService创作的Camera对象，该类继承了BpCamera用于进一步和CameraService端的CameraClient进行交互。

step4:CameraService处的响应

```
status_t BnCamera::onTransact(
  uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
{
.....
    case SET_PREVIEW_DISPLAY: {
      ALOGV("SET_PREVIEW_DISPLAY");
      CHECK_INTERFACE(ICamera, data, reply);
      sp<Surface> surface = Surface::readFromParcel(data);
      reply->writeInt32(setPreviewDisplay(surface));
      return NO_ERROR;
    } break;
......
    case START_PREVIEW: {
      ALOGV("START_PREVIEW");
      CHECK_INTERFACE(ICamera, data, reply);
      reply->writeInt32(startPreview());//调用服务端的cameraclient处的函数，为该类的派生类
      return NO_ERROR;
    } break;
}
```

由于之前connect写入的Binder本地实体类对象为CameraClient，则由该类对象的成员函数来实现。

```
status_t CameraClient::setPreviewDisplay(const sp<Surface>& surface) {
  LOG1("setPreviewDisplay(%p) (pid %d)", surface.get(), getCallingPid());

  sp<IBinder> binder(surface != 0 ? surface->asBinder() : 0);
  sp<ANativeWindow> window(surface);
  return setPreviewWindow(binder, window);
}
再调用SetPreviewWindow(),传入的Binder分别为Surface对象和一个ANativeWindow对象window。

status_t CameraClient::setPreviewWindow(const sp<IBinder>& binder,
    const sp<ANativeWindow>& window) {
  Mutex::Autolock lock(mLock);
  status_t result = checkPidAndHardware();
  if (result != NO_ERROR) return result;

  // return if no change in surface.
  if (binder == mSurface) {
    return NO_ERROR;
  }

  if (window != 0) {
```

```
    result = native_window_api_connect(window.get(), NATIVE_WINDOW_API_CAMERA);
    if (result != NO_ERROR) {
      ALOGE("native_window_api_connect failed: %s (%d)", strerror(-result),
          result);
      return result;
    }
  }

  // If preview has been already started, register preview buffers now.
  if (mHardware->previewEnabled()) {
    if (window != 0) {
      native_window_set_scaling_mode(window.get(),
          NATIVE_WINDOW_SCALING_MODE_SCALE_TO_WINDOW);
      native_window_set_buffers_transform(window.get(), mOrientation);
      result = mHardware->setPreviewWindow(window);
    }
  }

  if (result == NO_ERROR) {
    // Everything has succeeded.  Disconnect the old window and remember the
    // new window.
    disconnectWindow(mPreviewWindow);
    mSurface = binder;// This is a binder of Surface or SurfaceTexture.
    mPreviewWindow = window;//获取了预览的数据窗口
  } else {
    // Something went wrong after we connected to the new window, so
    // disconnect here.
    disconnectWindow(window);
  }

  return result;
}
```

调用mHardware这个硬件接口将本地的一个Window窗口传递到HAL层。并将这个windw记录到mPreviewWindow中。

Camera的HAL相关的具体实现结构

到了这里已经非讲不可的是mHardware啦，因为这个接口类将不得不访问HAL层。如最之前的result = mHardware->startPreview();函数。

```
status_t startPreview()
    {
        ALOGV("%s(%s)", __FUNCTION__, mName.string());
        if (mDevice->ops->start_preview)
            return mDevice->ops->start_preview(mDevice);
        return INVALID_OPERATION;
    }
```

这是一个典型的底层设备的调用。因此将和大家分享Camera的HAL层的相关操作。

1.参考当前平台的Camera源码，CameraService启动时会调用Camera的HAL模块，第一次open操作的最终调用如下：

```
int HALCameraFactory::device_open(const hw_module_t* module,
                                       const char* name,
                                       hw_device_t** device)//最先被framework层调用
{
    /*
     * Simply verify the parameters, and dispatch the call inside the
     * HALCameraFactory instance.
     */
```

```
.....
    return gEmulatedCameraFactory.cameraDeviceOpen(atoi(name), device);
}
```

该Camera模块中gEmulatedCameraFactory是一个静态的全局对象。来看该对象的构造过程:

```
HALCameraFactory::HALCameraFactory()
    : mHardwareCameras(NULL),
      mAttachedCamerasNum(0),
      mRemovableCamerasNum(0),
      mConstructedOK(false)
{
  F_LOG;

  LOGD("camera hal version: %s", CAMERA_HAL_VERSION);

  /* Make sure that array is allocated. */
  if (mHardwareCameras == NULL) {
    mHardwareCameras = new CameraHardware*[MAX_NUM_OF_CAMERAS];
    if (mHardwareCameras == NULL) {
      LOGE("%s: Unable to allocate V4L2Camera array for %d entries",
         __FUNCTION__, MAX_NUM_OF_CAMERAS);
      return;
    }
    memset(mHardwareCameras, 0, MAX_NUM_OF_CAMERAS * sizeof(CameraHardware*));
  }

  /* Create the cameras */
  for (int id = 0; id < MAX_NUM_OF_CAMERAS; id++)
  {
    // camera config information
    mCameraConfig[id] = new CCameraConfig(id);//读取camera配置文件.cfg
    if(mCameraConfig[id] == 0)
    {
      LOGW("create CCameraConfig failed");
    }
    else
    {
      mCameraConfig[id]->initParameters();
      mCameraConfig[id]->dumpParameters();
    }

    mHardwareCameras[id] = new CameraHardware(&HAL_MODULE_INFO_SYM.common, mCameraConfig[id]);//
    if (mHardwareCameras[id] == NULL)
    {
      mHardwareCameras--;
      LOGE("%s: Unable to instantiate fake camera class", __FUNCTION__);
      return;
    }
  }

  // check camera cfg
  if (mCameraConfig[0] != NULL)
  {
    mAttachedCamerasNum = mCameraConfig[0]->numberOfCamera();

    if ((mAttachedCamerasNum == 2)
      && (mCameraConfig[1] == NULL))
    {
      return;
    }
  }
```

```
    mConstructedOK = true;
}
```

这个全局对象是新建并初始化CameraHardware对象，而这里的Camera支持数目为2个。CameraHardware表示的是一个完整的摄像头类型，该类继承了camera_device这个结构体类：

```
CameraHardware::CameraHardware(struct hw_module_t* module, CCameraConfig* pCameraCfg)
    : mPreviewWindow(),
      mCallbackNotifier(),
      mCameraConfig(pCameraCfg),
      mIsCameraIdle(true),
      mFirstSetParameters(true),
      mFullSizeWidth(0),
      mFullSizeHeight(0),
      mCaptureWidth(0),
      mCaptureHeight(0),
      mVideoCaptureWidth(0),
      mVideoCaptureHeight(0),
      mUseHwEncoder(false),
      mFaceDetection(NULL),
      mFocusStatus(FOCUS_STATUS_IDLE),
      mIsSingleFocus(false),
      mOriention(0),
      mAutoFocusThreadExit(true),
      mIsImageCaptureIntent(false)
{
    /*
     * Initialize camera_device descriptor for this object.
     */
    F_LOG;

    /* Common header */
    common.tag = HARDWARE_DEVICE_TAG;
    common.version = 0;
    common.module = module;
    common.close = CameraHardware::close;

    /* camera_device fields. */
    ops = &mDeviceOps;
    priv = this;

    // instance V4L2CameraDevice object
    mV4L2CameraDevice = new V4L2CameraDevice(this, &mPreviewWindow, &mCallbackNotifier);//初始化V4l
    if (mV4L2CameraDevice == NULL)
    {
      LOGE("Failed to create V4L2Camera instance");
      return ;
    }

    memset((void*)mCallingProcessName, 0, sizeof(mCallingProcessName));

    memset(&mFrameRectCrop, 0, sizeof(mFrameRectCrop));
    memset((void*)mFocusAreasStr, 0, sizeof(mFocusAreasStr));
    memset((void*)&mLastFocusAreas, 0, sizeof(mLastFocusAreas));

    // init command queue
    OSAL_QueueCreate(&mQueueCommand, CMD_QUEUE_MAX);
    memset((void*)mQueueElement, 0, sizeof(mQueueElement));

    // init command thread
    pthread_mutex_init(&mCommandMutex, NULL);
    pthread_cond_init(&mCommandCond, NULL);
    mCommandThread = new DoCommandThread(this);
    mCommandThread->startThread();
```

```
    // init auto focus thread
    pthread_mutex_init(&mAutoFocusMutex, NULL);
    pthread_cond_init(&mAutoFocusCond, NULL);
    mAutoFocusThread = new DoAutoFocusThread(this);
  }
```

这里对这个CameraHardware对象进行了成员变量的初始化，其中包括camera_device_t结构体的初始化，其中ops是对Camera模块操作的核心所在。

```
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;
    camera_device_ops_t *ops;
    void *priv;
} camera_device_t;
```

这里有出息了一个V4L2CameraDevice对象，真正的和底层内核打交道的地方，基于V4L2的架构实现。

```
V4L2CameraDevice::V4L2CameraDevice(CameraHardware* camera_hal,
                   PreviewWindow * preview_window,
                   CallbackNotifier * cb)
  : mCameraHardware(camera_hal),
.......
{
  LOGV("V4L2CameraDevice construct");

  memset(&mHalCameraInfo, 0, sizeof(mHalCameraInfo));
  memset(&mRectCrop, 0, sizeof(Rect));

  // init preview buffer queue
  OSAL_QueueCreate(&mQueueBufferPreview, NB_BUFFER);//建立10个预览帧
  OSAL_QueueCreate(&mQueueBufferPicture, 2);//建立2个图片帧buffer

  // init capture thread
  mCaptureThread = new DoCaptureThread(this);
  pthread_mutex_init(&mCaptureMutex, NULL);
  pthread_cond_init(&mCaptureCond, NULL);
  mCaptureThreadState = CAPTURE_STATE_PAUSED;
  mCaptureThread->startThread();//启动视频采集

  // init preview thread
  mPreviewThread = new DoPreviewThread(this);
  pthread_mutex_init(&mPreviewMutex, NULL);
  pthread_cond_init(&mPreviewCond, NULL);
  mPreviewThread->startThread();//启动预览

  // init picture thread
  mPictureThread = new DoPictureThread(this);
  pthread_mutex_init(&mPictureMutex, NULL);
  pthread_cond_init(&mPictureCond, NULL);
  mPictureThread->startThread();//启动拍照

  pthread_mutex_init(&mConnectMutex, NULL);
  pthread_cond_init(&mConnectCond, NULL);

  // init continuous picture thread
  mContinuousPictureThread = new DoContinuousPictureThread(this);
  pthread_mutex_init(&mContinuousPictureMutex, NULL);
```

```
  pthread_cond_init(&mContinuousPictureCond, NULL);
  mContinuousPictureThread->startThread();//启动连续拍照
}
```

创建预览mQueueBufferPreview队列，初始化并启动了Camera需要的几个线程：视频采集，预览，拍照，以及连续的拍照等。

通过以上的几个对象构造后Camera的硬件信息维护到了全局类HALCameraFactory的mHardwareCameras[id]成员变量当中。

在客户端的connect，最终调用HAL的cameraDeviceOpen打开真正的设备:

```
int HALCameraFactory::cameraDeviceOpen(int camera_id, hw_device_t** device)
{

.....
  if (!mHardwareCameras[0]->isCameraIdle()
    || !mHardwareCameras[1]->isCameraIdle())
  {
    LOGW("camera device is busy, wait a moment");
    usleep(500000);
  }

  mHardwareCameras[camera_id]->setCameraHardwareInfo(&mHalCameraInfo[camera_id]);

  if (mHardwareCameras[camera_id]->connectCamera(device) != NO_ERROR)//连接camera硬件设备
  {
    LOGE("%s: Unable to connect camera", __FUNCTION__);
    return -EINVAL;
  }

  if (mHardwareCameras[camera_id]->Initialize() != NO_ERROR) //camera硬件设备参数等初始化
  {
    LOGE("%s: Unable to Initialize camera", __FUNCTION__);
    return -EINVAL;
  }

    return NO_ERROR;
}
```

初始化的流程依次调用HAL的HardwareCamera的connectCamera函数，其实这里最终的核心是返回一个camera device给上层调用camera的具体操作:最终将CameraHardware的基类camera_devcie_t返回给device。

```
status_t CameraHardware::connectCamera(hw_device_t** device)
{
  F_LOG;
  status_t res = EINVAL;

  {
    Mutex::Autolock locker(&mCameraIdleLock);
    mIsCameraIdle = false;
  }

  if (mV4L2CameraDevice != NULL)
  {
    res = mV4L2CameraDevice->connectDevice(&mHalCameraInfo);
    if (res == NO_ERROR)
    {
      *device = &common;
......
}
```

mV4L2CameraDevice->connectDevice()，真正的开启V4l2的相关Camera启动，内部通过openCameraDev来实现

```
status_t V4L2CameraDevice::connectDevice(HALCameraInfo * halInfo)
{
  F_LOG;

.....

  // open v4l2 camera device
  int ret = openCameraDev(halInfo);//调用v4l2的camera标准接口
  if (ret != OK)
  {
    return ret;
  }

  memcpy((void*)&mHalCameraInfo, (void*)halInfo, sizeof(HALCameraInfo));

.......
}
```

openCameraDev()函数内部的实现就是V4L2的典型的API流程，通过ioctl来完成对内核Camera视频采集的驱动的控制。该流程可以参考 DM6446的视频前端VPFE驱动之ioctl控制（视频缓存区，CCDC，decoder)解析

```
int V4L2CameraDevice::openCameraDev(HALCameraInfo * halInfo)
{
  F_LOG;

  int ret = -1;
  struct v4l2_input inp;
  struct v4l2_capability cap;

  if (halInfo == NULL)
  {
    LOGE("error HAL camera info");
    return -1;
  }

  // open V4L2 device
  mCameraFd = open(halInfo->device_name, O_RDWR | O_NONBLOCK, 0);
  if (mCameraFd == -1)
  {
  LOGE("ERROR opening %s: %s", halInfo->device_name, strerror(errno));
    return -1;
  }

  // check v4l2 device capabilities
  ret = ioctl (mCameraFd, VIDIOC_QUERYCAP, &cap);
    if (ret < 0)
  {
  LOGE("Error opening device: unable to query device.");
  goto END_ERROR;
    }

    if ((cap.capabilities & V4L2_CAP_VIDEO_CAPTURE) == 0)
  {
  LOGE("Error opening device: video capture not supported.");
  goto END_ERROR;
    }

    if ((cap.capabilities & V4L2_CAP_STREAMING) == 0)
  {
```

```
    LOGE("Capture device does not support streaming i/o");
    goto END_ERROR;
      }

    if (!strcmp((char *)cap.driver, "uvcvideo"))
    {
      mIsUsbCamera = true;
    }

    if (!mIsUsbCamera)
    {
      // uvc do not need to set input
      inp.index = halInfo->device_id;
      if (-1 == ioctl (mCameraFd, VIDIOC_S_INPUT, &inp))
      {
        LOGE("VIDIOC_S_INPUT error!");
        goto END_ERROR;
      }
    }

    // try to support this format: NV21, YUYV
  .....
  }
```

到这里为止一共典型的Camera从应用侧到CameraService再到Camera HAL处的初始化流程基本完成。

总结下的是HAL层建立了一个基于V4L2的V4L2CameraDevice对象来完成和内核视频采集模块的互动，返回一个camera_device_t结构体对象mDevice来为后续对Camera设备的进一步控制。

推荐文章

- 1. Android 整合实现简单易用、功能强大的 RecyclerView
- 2. Android N 混合编译与对热补丁影响解析
- 3. Android Nougat(7.0) 创建自定义通知栏快捷设置
- 4. Android 基础 -- 生命周期和启动模式实践总结
- 5. Android 复杂的多类型列表视图新写法：MultiType
- 6. Android带你解析ScrollView--仿QQ空间标题栏渐变

我来评几句

请输入评论内容...

登录后评论

已发表评论数(0)

相关站点

CSDN博客
＋订阅
热门文章

- 1. Android 整合实现简单易用、功能强大的 RecyclerView
- 2. Android N 混合编译与对热补丁影响解析
- 3. Android 开源项目推荐之「网络请求哪家强」
- 4. 【Android】 多语言切换
- 5. 你应该知道的那些Android小经验

收藏到推刊

[创建推刊]()

收 藏 取消
推刊名(必填) 请填写推刊名
推刊描述
描述不能大于100个字符!
权限设置: 公开 仅自己可见
创建 取消
×

文章纠错

邮箱地址
错误类型 正文不准确
补充信息
提交

网站相关

- 关于我们
- 移动应用
- 建议反馈

关注我们

- 推酷网
- tuicool2012

友情链接

人人都是产品经理 PM256 移动信息化 行晓网 智城外包网 虎嗅 IT耳朵 创媒工场 经理人分享 市场部网 砍柴网 CocoaChina 北风网 云智慧 我赢职场 大数据时代 奇笛网 咕噜网 红联linux Win10之家 鸟哥笔记 爱游戏 投资潮 31会议网 极光推送 Teambition 硅谷网 leangoo ZEALER中国 OpenSNS 小牛学堂 handone Scrum中文网 比戈大牛 又拍云 更多链接>>