# ToDO

Camera2 数据流从framework到Hal源码分析

<https://www.jianshu.com/p/ecb1be82e6a8>

CS架构

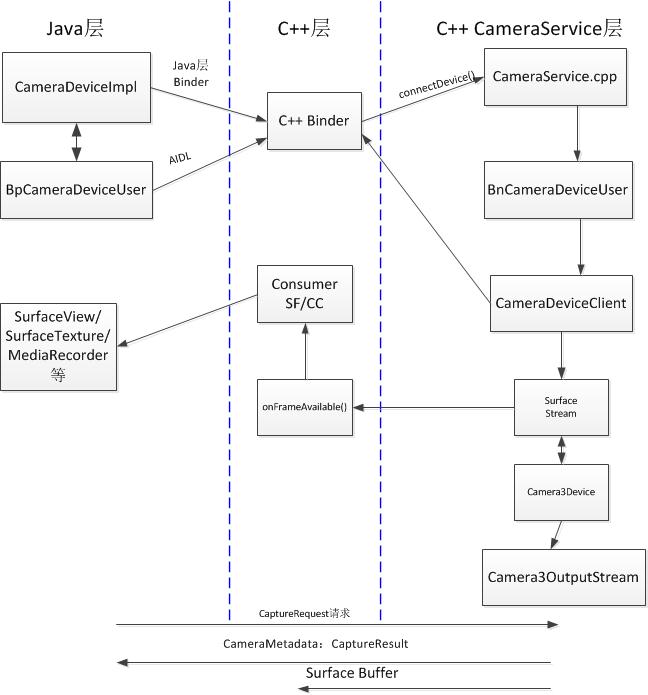
<https://www.androidos.net.cn/android/2.3.7_r1/xref/frameworks/base/services/camera/libcameraservice>

<https://blog.csdn.net/kangear/article/details/8239330>

# Camera API2.0简介

## 架构图

### 数据从framework传递到上层

在framework层会创建一个Surface，而Surface持有一个ANativeWindow接口，ANativeWindow负责管理Buffer的创建与共享在Consumer端ANativeWindow创建Buffer，在Surface通过dequeuebuffer获取buffer handle到本地进行共享，在数据填充完后通过quenebuffer告诉Consumer当前buffer可用，这样就形成了一个buffer生成与消费的关系。这种模式是通过建立不同类型的Consumer，然后在Native层建立一个BufferQueue,并将这个BufferQueue的IGraphicBufferConsumer用于构建CPUConsumer,将IGraphicBufferProducter通过createStream给CameraDevice增加一个Stream。即是说我们是通过buffer从低向上传递数据。

Java层要想与C++层的CameraService层进行通信，都是通过Java层的IPC Binder进制进行的，主要包括ICameraService.aidl以及ICameraDeviceUser.aidl两个接口来实现，其会在Java层维护一个CameraDeviceImpl即Camera设备的代理，而CameraService以及CameraDeviceImpl的初始化会在此文的第二，第三节进行分析。而Java层对Camera的具体操作的操作流程大致为，Java层通过Device代理发送一个CaptureRequest，而C++层进行相应的处理，再调用相应的回调来通知Java相应的处理结果，并将相应的Capture数据保存在Surface Buffer里，这样Java层在回调函数中可以对数据进行相应的处理。

## Java层的CameraService的实现和应用

从Camera API2开始，Camera的实现方式有所不同，最主要的区别是不再使用JNI来调用本地代码，从而获得本地CameraService，并实现其C/S模式的通信，而是直接在Java层通过Java层的IPC Binder机制来获取Java层的CameraService的代理对象，从而直接在Java层获取本地的CameraService与Camera Device进行相应的通信。

相应的代码及目录：

ICameraService.aidl:frameworks/base/core/java/android/hardware

CameraService.cpp:frameworks/av/services/camera/libcameraservice

CameraManager.java:frameworks/base/core/java/android/hardware/camera2

获取CameraService的核心代码如下：

//CameraManager.java

prvate void connectCameraServiceLocked(){

if(mCameraService != null)return;

//获取Binder

IBinder cameraServiceBinder = ServiceManager.getService(CAMERA\_SERVICE\_BINDER\_NAME);

if(cameraServiceBinder == null){

return;

}

try{

cameraServiceBinder.linkToDeath(this,/\*flags\*/ 0);

}catch(RemoteException e){

return;

}

ICameraService cameraServiceRaw = ICameraService.Stub.asInterface(cameraServiceBinder);

//根据cameraServiceRaw 创建CameraService实例

ICameraService cameraService = CameraServiceDecorator.newInstance(cameraServiceRaw);

...

try{

//添加监听

cameraService.addListener(this);

//赋值给mCameraService的全局变量

mCameraService = cameraService;

}catch(CameraRuntimeException e){

...

}

}

由代码可知，通过Java层的Binder从ServiceManager里获取了一个Java层的CameraService实例，在打开Camera的流程中，会通过此CameraService(Native的CameraService)与Camera通信，而其中的通信通过ICameraDeviceUser来实现，接下来分析ICameraDeviceUser的实现。

## ICameraDeviceUser.aidl的通信实现

Java层与C++ CameraService层之间的通信，通过封装了一个CameraDeviceUser来实现，它只是在Java层使用了AIDL技术来实现Client，即在Java层维护了一个CameraDevice，这样的好处是响应速度更快，因为这样不需要通过每次进入Native层来完成通信，而可以通过Java层的IPC Binder机制即可完成。即API2.0通过AIDL实现一个接口ICameraDeviceUser，从而在Java层维护一个Camera proxy，之后的通信都是通过此代理CameraDeviceImpl来实现。

相关代码及目录：

ICameraDeviceUser.aidl:frameworks/base/core/java/android/hardware/camera2

ICameraDeviceUser.cpp:frameworks/av/camera/camera2

CameraDeviceImpl.java:frameworks/base/core/java/android/hardware/camera2/impl

获取Camera Device的Java层代理的核心代码如下：

//CameraManager.java

private CameraDevice openCameraDeviceUserAsync(...){

//初始化Camera Java层代理对象

CameraDevice device = null;

try{

synchronized(mLock){

//初始化ICameraDeviceUser

ICameraDeviceUser cameraUser = null;

//初始化具体的CameraDevice代理

android.hardware.camera2.impl.CameraDeviceImpl deviceImpl = new android.hardware.

camera2.impl.CameraDeviceImpl(cameraId,callback,handler,characteristics);

BinderHolder holder = new BinderHolder();

ICameraDeviceCallbacks callbacks = deviceImpl.getCallbacks();

...

try{

//如果支持HAL3.2+的devices

if(supportsCamera2ApiLocked(cameraId)){

//获取CameraService

ICameraService cameraService = CameraManagerGlobal.get().getCameraService();

...

//连接设备

cameraService.connectDevice(callbacks,id,mContextgetOpPackageName()

,USE\_CALLING\_UID,holder);

//通过Binder获得打开的Camera设备返回的Camera代理

cameraUser = ICameraDeviceUser.Stub.asInterface(holder.getBinder());

}else{//否则用遗产API

cameraUser = CameraDeviceUserShim.connectBinderShim(callbacks,id);

}

}catch(...){

...

}

//包装代理对象

deviceImpl.setRemoteDevice(cameraUser);

device = deviceImpl;

}

}catch(...){

...

}

返回Camera代理

return device;

}

由代码可知，首先获取CameraService，然后通过它来开启Camera，而开启成功后，C++层会返回一个Camera device代理对象，此处即为ICameraDeviceUser，所以在Java层对其进行相应的封装，变成一个CameraDeviceImpl对象，此后，只要需要对Camera进行操作，都会调用CameraDeviceImpl对象的相关方法，并通过ICameraDeviceUser以及Java IPC Binder来与本地的Camera device进行通信，至此，Camera API2.0的框架就分析结束了，具体的操作，如Camera的初始化，preview,capture等流程的分析

## surface和CpuConsumer下生产者和消费者间的处理框架简述

如果对SurfaceFlinger架构的工作原理较为熟悉的话，本文阅读起来会相对容易些。之所以撰写本文是因为在阅读Camera HAL3的实现过程中大量的出现了类似与SurfaceFlinger的工作模式。本文将以CallbackProcessor模块的为入口，和大家进行分享。

### Preview模块Surface与SurfaceFlinger的基础知识

开发过Android Camera模块的人，基本都应该熟悉实时采集的视频如果要显示到手机屏上，对应的Framework层就应该构建一个Surface对象，这对所有Android版本都是通用的。该Surface本质是一个在APP端的ANativeWindow，是需要从SurfaceFlinger端的Gralloc模块获取Buffer的，然后填充帧图像数据并送显示的过程。详细内容可以参考之前的专栏我心所向之Android4.2关于SurfaceFlinger相关的内容。

由于目前Android系统版本升级过快，在5.0以上的版本中SurfaceFlinger部分框架发生了变化(目前还没有深入去研读过)，了解到的是：

一方面他去掉了SurfaceTexture/SurfaceTextureClient等内容，增强了Surface的功能。

另一方面BufferQueue的使用也更加简单与明了，Producer与Consumer的关系也更加明确，BufferQueue不再是以前的消费者角色，转而成为ProducerBufferQueue和Consumer相互关联的桥梁。

此外，如果你在深入的话，会了解到Gralloc模块中非framebuff的缓存管理与共享不在是那个Ashmem匿名共享内存，而是出现了一种新的内存管理机制/dev/ION/，后面会和大家分享他部分的工作机制。

对于SurfaceFlinger与Camera Preview Surface而言，ANativeWindow作为两者的共性负责管理和维护Buffer的创建与共享，在SurfaceFlinger端ANativeWindow创建buffer，在Surface侧通过dequeuebuffer获取buffer handle到本地进行共享，在填充完数据后通过queuebuffer告诉SurfaceFlinger当前buffer可用，这样就形成了一个buffer生产与消费的关系，基于这种原理我们在Camera3Device中除了看到正常的Preview Surface之外还看到了其他模块中出现了Surface的创建。

### CallbackProcessor::updateStream() 创建Surface

sp<IGraphicBufferProducer> producer;

sp<IGraphicBufferConsumer> consumer;

BufferQueue::createBufferQueue(&producer, &consumer);//BufferQueueProducer与BufferQueueConsumer

mCallbackConsumer = new CpuConsumer(consumer, kCallbackHeapCount);

mCallbackConsumer->setFrameAvailableListener(this);//当前CallbackProcessor继承于CpuConsumer::FrameAvailableListener

mCallbackConsumer->setName(String8("Camera2Client::CallbackConsumer"));

mCallbackWindow = new Surface(producer);//用于queue操作，这里直接进

上述代码看上去很简单，但就是他最终形成了一个bufferstream Product和Consumer的处理，这种方式也为Camera2Client下出现多路stream，多路数据流存在奠定了基础，下面我们来简单的描述他的工作机制。

#### 2.1 BufferQueue::createBufferQueue

Sadf

void BufferQueue::createBufferQueue(sp<IGraphicBufferProducer>\* outProducer,

sp<IGraphicBufferConsumer>\* outConsumer,

const sp<IGraphicBufferAlloc>& allocator) {

LOG\_ALWAYS\_FATAL\_IF(outProducer == NULL,

"BufferQueue: outProducer must not be NULL");

LOG\_ALWAYS\_FATAL\_IF(outConsumer == NULL,

"BufferQueue: outConsumer must not be NULL");

sp<BufferQueueCore> core(new BufferQueueCore(allocator));

LOG\_ALWAYS\_FATAL\_IF(core == NULL,

"BufferQueue: failed to create BufferQueueCore");

sp<IGraphicBufferProducer> producer(new BufferQueueProducer(core));//本地Bn的BufferQueueProducer

LOG\_ALWAYS\_FATAL\_IF(producer == NULL,

"BufferQueue: failed to create BufferQueueProducer");

sp<IGraphicBufferConsumer> consumer(new BufferQueueConsumer(core));//本地Bn的BufferQueueConsumer

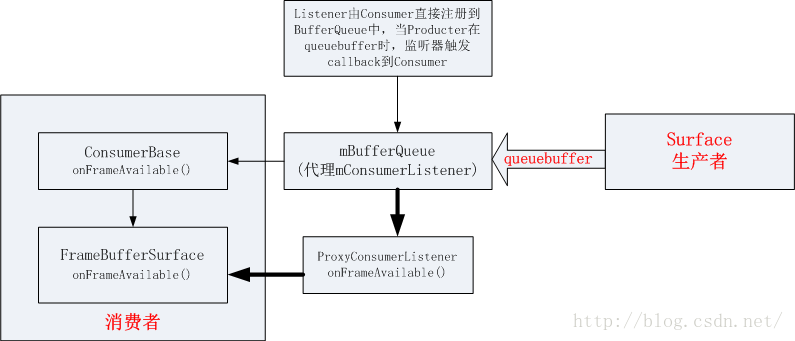
LOG\_ALWAYS\_FATAL\_IF(consumer == NULL,

"BufferQueue: failed to create BufferQueueConsumer");

\*outProducer = producer;

\*outConsumer = consumer;

当前Android5.1版本中可以看到BufferQueue已经改变了原先的角色，成为了一个用于创建一组操作BufferQueue的接口，实现了BufferQueueProducer与BufferQueueConsumer的创建。更本质说现在BufferQueue的工作量已经很小，基本由Surface来全盘操作。这里通过BufferQueueCore会将BufferQueueProducer与BufferQueueConsumer绑定在一起，BufferQueueProducer可以说是替代了旧版本Android4.2中的SurfaceTextureClient，而BufferQueueConsumer则直接是SurfaceTexture以及BufferQueue的替代者。下图是在Android4.2.2中Surface和SurfaceFlinger间基于Bn/BpSurfaceTexture的多进程间buffer的queue操作机制



#### CpuConsumer

在2.1中本质上是创建了BufferQueueProducer和BufferQueueConsumer，一般是成对出现。

这个CpuConsumer的地位本质是和SurfaceFlinger是一样的，主要角色就是来处理已经带有帧数据的buffer块，所以整个的处理机制都是类型的，其中两者均为继承一个ConsumerBase类，用于实现对buffer块的处理。

CpuConsumer::CpuConsumer(const sp<IGraphicBufferConsumer>& bq,

uint32\_t maxLockedBuffers, bool controlledByApp) :

ConsumerBase(bq, controlledByApp),

mMaxLockedBuffers(maxLockedBuffers),

mCurrentLockedBuffers(0)

{

// Create tracking entries for locked buffers

mAcquiredBuffers.insertAt(0, maxLockedBuffers);

mConsumer->setConsumerUsageBits(GRALLOC\_USAGE\_SW\_READ\_OFTEN);

mConsumer->setMaxAcquiredBufferCount(maxLockedBuffers);

}

#### mCallbackConsumer->setFrameAvailableListener(this)

这里所完成的过程是需要Consumer去将一个listener加入到Productor端去，便于在帧数据可用时，可以监听到并告知Consumer去做数据的处理，Product需要在数据可用时触发这个listener的onFrameAvailable，从而让数据从Productor转到Consumer则，数据处理应该是在同一进程的而不是跨进程。

如果作为Consumer如CPUConsumer没有使用setFrameAvailableListener将自己加入到listener中去，会由ConsumerBase的接口onFrameAvailable来替代完成。

#### mCallbackWindow = new Surface(producer)

这里是建立一个Surface，也就是类型于Preview模式下创建的Surface，两者的本质是一样的。在Android5.1中，Surface已经拥有了绝对的控制权，Preview模式下的Surface是跨进程的和SF进行交互，故构造函数传入的sp<IGraphicBufferProducer>& bufferProducer参数一般是一个BufferQueue中BufferQueueProducer侧匿名的binder BpGraphicBufferProducer。对于CPUConsumer而言，两者是处于同一进程之中的，Surface主要行驶如下的操作，而在旧版本中这些主要由SurfaceTextureClient来管理的：

ANativeWindow::setSwapInterval = hook\_setSwapInterval;

ANativeWindow::dequeueBuffer = hook\_dequeueBuffer;

ANativeWindow::cancelBuffer = hook\_cancelBuffer;

ANativeWindow::queueBuffer = hook\_queueBuffer;

ANativeWindow::query = hook\_query;

ANativeWindow::perform = hook\_perform;

ANativeWindow::dequeueBuffer\_DEPRECATED = hook\_dequeueBuffer\_DEPRECATED;

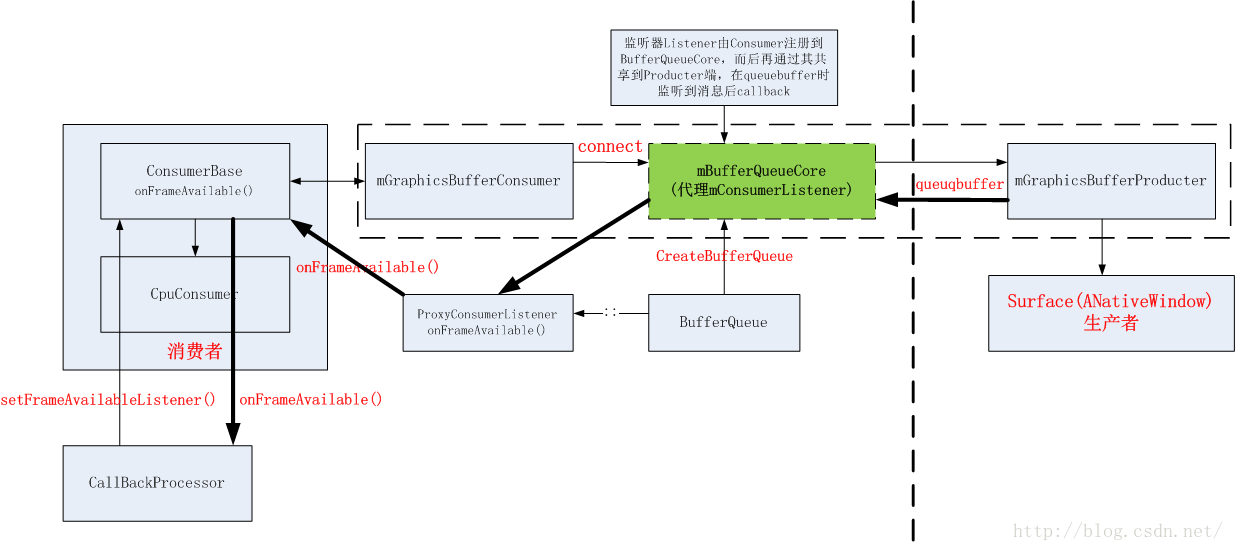
ANativeWindow::cancelBuffer\_DEPRECATED = hook\_cancelBuffer\_DEPRECATED;

ANativeWindow::lockBuffer\_DEPRECATED = hook\_lockBuffer\_DEPRECATED;

ANativeWindow::queueBuffer\_DEPRECATED = hook\_queueBuffer\_DEPRECATED;

一般对Buffer的操作都先是基于Struct ANativeWindow这个本地的窗口类来处理，再是通过hook到实际的surface再交由GraphicBufferProducer来完成的。

### Android5.1下Surface的queuebuffer操作逻辑



对比上一图可知，当前CPUConsumer模式下，从queuebuffer的处理过程来看更加体现出Productor和Consumer这种设计模式，整个架构代码看起来也更明了，这也是谷歌不断升级系统去冗存精的道理吧。整个用户在调用这种逻辑是，只需几句代码就可以获得对一个buffer块的读与写的操作接口，具体后续在分析Camera HAL3的数据流时可以很深刻的看到。

总的来说Surface在Camera3Device的架构下，与HAL3相组合是充当了消费者，与CPUCOnsumer或者Surfaceflinger来说就是生产者。或者说surface充当了buffer信息的传输与过渡。



# Camera2 HAL分析

## Camera HAL的初始化

Camera HAL的初始加载是在Native的CameraService初始化流程中的，而CameraService初始化是在Main\_mediaServer.cpp的main方法开始的

Camera HAL的初始加载是在Native的CameraService初始化流程中的，而CameraService初始化是在Main\_mediaServer.cpp的main方法开始的：

//Main\_mediaServer.cpp

int main(int argc \_\_unused, char\*\* argv){

…

sp<ProcessState> proc(ProcessState::self());

//获取ServieManager

sp<IServiceManager> sm = defaultServiceManager();

ALOGI("ServiceManager: %p", sm.get());

AudioFlinger::instantiate();

//初始化media服务

MediaPlayerService::instantiate();

//初始化资源管理服务

ResourceManagerService::instantiate();

//初始化Camera服务

CameraService::instantiate();

//初始化音频服务

AudioPolicyService::instantiate();

SoundTriggerHwService::instantiate();

//初始化Radio服务

RadioService::instantiate();

registerExtensions();

//开始线程池

ProcessState::self()->startThreadPool();

IPCThreadState::self()->joinThreadPool();

}

其中，CameraService继承自BinderService，instantiate也是在BinderService中定义的，此方法就是调用publish方法，所以来看publish方法：

// BinderService.h

static status\_t publish(bool allowIsolated = false) {

sp<IServiceManager> sm(defaultServiceManager());

//将服务添加到ServiceManager

return sm->addService(String16(SERVICE::getServiceName()),new SERVICE(), allowIsolated);

}

这里，将会把CameraService服务加入到ServiceManager进行管理。

而在前面的文章android6.0源码分析之Camera API2.0简介中，需要通过Java层的IPC Binder来获取此CameraService对象，在此过程中会初始CameraService的sp类型的对象，而对于sp，此处不做过多的分析，具体的可以查看深入理解Android卷Ⅰ中的第五章中的相关内容。此处，在CameraService的构造时，会调用CameraService的onFirstRef方法

/CameraService.cpp

void CameraService::onFirstRef()

{

BnCameraService::onFirstRef();

...

camera\_module\_t \*rawModule;

//根据CAMERA\_HARDWARE\_MODULE\_ID(字符串camera)来获取camera\_module\_t对象

int err = hw\_get\_module(CAMERA\_HARDWARE\_MODULE\_ID,

(const hw\_module\_t \*\*)&rawModule);

//创建CameraModule对象

mModule = new CameraModule(rawModule);

//模块初始化

err = mModule->init();

...

//通过Module获取Camera的数量

mNumberOfCameras = mModule->getNumberOfCameras();

mNumberOfNormalCameras = mNumberOfCameras;

//初始化闪光灯

mFlashlight = new CameraFlashlight(\*mModule, \*this);

status\_t res = mFlashlight->findFlashUnits();

int latestStrangeCameraId = INT\_MAX;

for (int i = 0; i < mNumberOfCameras; i++) {

//初始化CameraID

String8 cameraId = String8::format("%d", i);

struct camera\_info info;

bool haveInfo = true;

//获取Camera信息

status\_t rc = mModule->getCameraInfo(i, &info);

...

//如果Module版本高于2.4，找出冲突的设备参数

if (mModule->getModuleApiVersion() >= CAMERA\_MODULE\_API\_VERSION\_2\_4 && haveInfo) {

cost = info.resource\_cost;

conflicting\_devices = info.conflicting\_devices;

conflicting\_devices\_length = info.conflicting\_devices\_length;

}

//将冲突设备加入冲突set集中

std::set<String8> conflicting;

for (size\_t i = 0; i < conflicting\_devices\_length; i++) {

conflicting.emplace(String8(conflicting\_devices[i]));

}

...

}

//如果Module的API大于2.1，则设置回调

if (mModule->getModuleApiVersion() >= CAMERA\_MODULE\_API\_VERSION\_2\_1) {

mModule->setCallbacks(this);

}

//若大于2.2，则设置供应商的Tag

if (mModule->getModuleApiVersion() >= CAMERA\_MODULE\_API\_VERSION\_2\_2) {

setUpVendorTags();

}

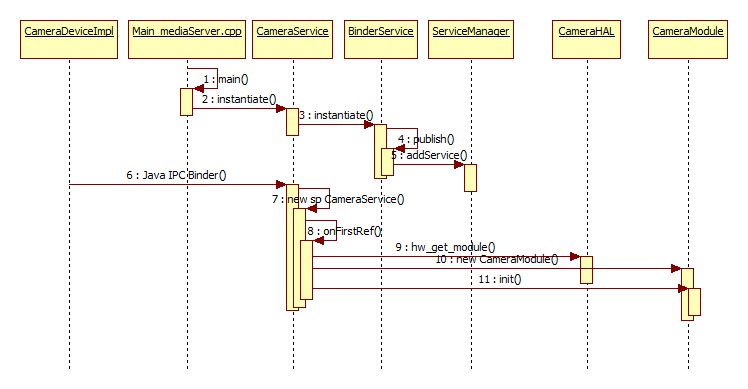
//将此服务注册到CameraDeviceFactory

CameraDeviceFactory::registerService(this);

CameraService::pingCameraServiceProxy();

}

onFirstRef方法中，首先会通过HAL框架的hw\_get\_module来获取CameraModule对象，然后会对其进行相应的初始化，并会进行一些参数的设置，如camera的数量，闪光灯的初始化，以及回调函数的设置等，到这里，Camera2 HAL的模块就初始化结束了，下面给出初始化时序图：



## open流程分析

通过阅读android6.0源码发现，它提供了高通的Camera实现，并且提供了高通的Camera库，也实现了高通的Camera HAL的相应接口，对于高通的Camera，它在后台会有一个守护进程daemon，daemon是介于应用和驱动之间翻译ioctl的中间层(委托处理)。本节将以Camera中的open流程为例，来分析Camera HAL的工作过程，在应用对硬件发出open请求后，会通过Camera HAL来发起open请求，而Camera HAL的open入口在QCamera2Hal.cpp进行了定义：

/QCamera2Hal.cpp

camera\_module\_t HAL\_MODULE\_INFO\_SYM = {

//它里面包含模块的公共方法信息

common: camera\_common,

get\_number\_of\_cameras: qcamera::QCamera2Factory::get\_number\_of\_cameras,

get\_camera\_info: qcamera::QCamera2Factory::get\_camera\_info,

set\_callbacks: qcamera::QCamera2Factory::set\_callbacks,

get\_vendor\_tag\_ops: qcamera::QCamera3VendorTags::get\_vendor\_tag\_ops,

open\_legacy: qcamera::QCamera2Factory::open\_legacy,

set\_torch\_mode: NULL,

init : NULL,

reserved: {0}

};

static hw\_module\_t camera\_common = {

tag: HARDWARE\_MODULE\_TAG,

module\_api\_version: CAMERA\_MODULE\_API\_VERSION\_2\_3,

hal\_api\_version: HARDWARE\_HAL\_API\_VERSION,

id: CAMERA\_HARDWARE\_MODULE\_ID,

name: "QCamera Module",

author: "Qualcomm Innovation Center Inc",

//它的方法数组里绑定了open接口

methods: &qcamera::QCamera2Factory::mModuleMethods,

dso: NULL,

reserved: {0}

};

struct hw\_module\_methods\_t QCamera2Factory::mModuleMethods = {

//open方法的绑定

open: QCamera2Factory::camera\_device\_open,

};

Camera HAL层的open入口其实就是camera\_device\_open方法：

// QCamera2Factory.cpp

int QCamera2Factory::camera\_device\_open(const struct hw\_module\_t \*module, const char \*id,

struct hw\_device\_t \*\*hw\_device){

...

return gQCamera2Factory->cameraDeviceOpen(atoi(id), hw\_device);

}

它调用了cameraDeviceOpen方法，而其中的hw\_device就是最后要返回给应用层的CameraDeviceImpl在Camera HAL层的对象，继续分析cameraDeviceOpen方法：

// QCamera2Factory.cpp

int QCamera2Factory::cameraDeviceOpen(int camera\_id, struct hw\_device\_t \*\*hw\_device){

...

//Camera2采用的Camera HAL版本为HAL3.0

if ( mHalDescriptors[camera\_id].device\_version == CAMERA\_DEVICE\_API\_VERSION\_3\_0 ) {

//初始化QCamera3HardwareInterface对象，这里构造函数里将会进行configure\_streams以及

//process\_capture\_result等的绑定

QCamera3HardwareInterface \*hw = new QCamera3HardwareInterface(

mHalDescriptors[camera\_id].cameraId, mCallbacks);

//通过QCamera3HardwareInterface来打开Camera

rc = hw->openCamera(hw\_device);

...

} else if (mHalDescriptors[camera\_id].device\_version == CAMERA\_DEVICE\_API\_VERSION\_1\_0) {

//HAL API为2.0

QCamera2HardwareInterface \*hw = new QCamera2HardwareInterface((uint32\_t)camera\_id);

rc = hw->openCamera(hw\_device);

...

} else {

...

}

return rc;

}

此方法有两个关键点：一个是QCamera3HardwareInterface对象的创建，它是用户空间与内核空间进行交互的接口；另一个是调用它的openCamera方法来打开Camera，下面将分别进行分析。

### QCamera3HardwareInterface构造函数分析

在它的构造函数里面有一个关键的初始化，即mCameraDevice.ops = &mCameraOps，它会定义Device操作的接口：

//QCamera3HWI.cpp

camera3\_device\_ops\_t QCamera3HardwareInterface::mCameraOps = {

initialize: QCamera3HardwareInterface::initialize,

//配置流数据的相关处理

configure\_streams: QCamera3HardwareInterface::configure\_streams,

register\_stream\_buffers: NULL,

construct\_default\_request\_settings:

QCamera3HardwareInterface::construct\_default\_request\_settings,

//处理结果的接口

process\_capture\_request:

QCamera3HardwareInterface::process\_capture\_request,

get\_metadata\_vendor\_tag\_ops: NULL,

dump: QCamera3HardwareInterface::dump,

flush: QCamera3HardwareInterface::flush,

reserved: {0},

};

其中，会在configure\_streams中配置好流的处理handle：

//QCamera3HWI.cpp

int QCamera3HardwareInterface::configure\_streams(const struct camera3\_device \*device,

camera3\_stream\_configuration\_t \*stream\_list){

//获得QCamera3HardwareInterface对象

QCamera3HardwareInterface \*hw =reinterpret\_cast<QCamera3HardwareInterface \*>(device->priv);

...

//调用它的configureStreams进行配置

int rc = hw->configureStreams(stream\_list);

..

return rc;

}

1

2

3

4

5

6

7

8

9

10

11

继续追踪configureStream方法：

//QCamera3HWI.cpp

int QCamera3HardwareInterface::configureStreams(camera3\_stream\_configuration\_t \*streamList){

...

//初始化Camera版本

al\_version = CAM\_HAL\_V3;

...

//开始配置stream

...

//初始化相关Channel为NULL

if (mMetadataChannel) {

delete mMetadataChannel;

mMetadataChannel = NULL;

}

if (mSupportChannel) {

delete mSupportChannel;

mSupportChannel = NULL;

}

if (mAnalysisChannel) {

delete mAnalysisChannel;

mAnalysisChannel = NULL;

}

//创建Metadata Channel，并对其进行初始化

mMetadataChannel = new QCamera3MetadataChannel(mCameraHandle->camera\_handle,

mCameraHandle->ops, captureResultCb,&gCamCapability[mCameraId]->padding\_info,

CAM\_QCOM\_FEATURE\_NONE, this);

...

//初始化

rc = mMetadataChannel->initialize(IS\_TYPE\_NONE);

...

//如果h/w support可用，则创建分析stream的Channel

if (gCamCapability[mCameraId]->hw\_analysis\_supported) {

mAnalysisChannel = new QCamera3SupportChannel(mCameraHandle->camera\_handle,

mCameraHandle->ops,&gCamCapability[mCameraId]->padding\_info,

CAM\_QCOM\_FEATURE\_PP\_SUPERSET\_HAL3,CAM\_STREAM\_TYPE\_ANALYSIS,

&gCamCapability[mCameraId]->analysis\_recommended\_res,this);

...

}

bool isRawStreamRequested = false;

//清空stream配置信息

memset(&mStreamConfigInfo, 0, sizeof(cam\_stream\_size\_info\_t));

//为requested stream分配相关的channel对象

for (size\_t i = 0; i < streamList->num\_streams; i++) {

camera3\_stream\_t \*newStream = streamList->streams[i];

uint32\_t stream\_usage = newStream->usage;

mStreamConfigInfo.stream\_sizes[mStreamConfigInfo.num\_streams].width = (int32\_t)newStream-

>width;

mStreamConfigInfo.stream\_sizes[mStreamConfigInfo.num\_streams].height = (int32\_t)newStream-

>height;

if ((newStream->stream\_type == CAMERA3\_STREAM\_BIDIRECTIONAL||newStream->usage &

GRALLOC\_USAGE\_HW\_CAMERA\_ZSL) &&newStream->format ==

HAL\_PIXEL\_FORMAT\_IMPLEMENTATION\_DEFINED && jpegStream){

mStreamConfigInfo.type[mStreamConfigInfo.num\_streams] = CAM\_STREAM\_TYPE\_SNAPSHOT;

mStreamConfigInfo.postprocess\_mask[mStreamConfigInfo.num\_streams] =

CAM\_QCOM\_FEATURE\_NONE;

} else if(newStream->stream\_type == CAMERA3\_STREAM\_INPUT) {

} else {

switch (newStream->format) {

//为非zsl streams查找他们的format

...

}

}

if (newStream->priv == NULL) {

//为新的stream构造Channel

switch (newStream->stream\_type) {//分类型构造

case CAMERA3\_STREAM\_INPUT:

newStream->usage |= GRALLOC\_USAGE\_HW\_CAMERA\_READ;

newStream->usage |= GRALLOC\_USAGE\_HW\_CAMERA\_WRITE;//WR for inplace algo's

break;

case CAMERA3\_STREAM\_BIDIRECTIONAL:

...

break;

case CAMERA3\_STREAM\_OUTPUT:

...

break;

default:

break;

}

//根据前面的得到的stream的参数类型以及format分别对各类型的channel进行构造

if (newStream->stream\_type == CAMERA3\_STREAM\_OUTPUT ||

newStream->stream\_type == CAMERA3\_STREAM\_BIDIRECTIONAL) {

QCamera3Channel \*channel = NULL;

switch (newStream->format) {

case HAL\_PIXEL\_FORMAT\_IMPLEMENTATION\_DEFINED:

/\* use higher number of buffers for HFR mode \*/

...

//创建Regular Channel

channel = new QCamera3RegularChannel(mCameraHandle->camera\_handle,

mCameraHandle->ops, captureResultCb,&gCamCapability[mCameraId]-

>padding\_info,this,newStream,(cam\_stream\_type\_t)mStreamConfigInfo.type[

mStreamConfigInfo.num\_streams],mStreamConfigInfo.postprocess\_mask[

mStreamConfigInfo.num\_streams],mMetadataChannel,numBuffers);

...

newStream->max\_buffers = channel->getNumBuffers();

newStream->priv = channel;

break;

case HAL\_PIXEL\_FORMAT\_YCbCr\_420\_888:

//创建YWV Channel

...

break;

case HAL\_PIXEL\_FORMAT\_RAW\_OPAQUE:

case HAL\_PIXEL\_FORMAT\_RAW16:

case HAL\_PIXEL\_FORMAT\_RAW10:

//创建Raw Channel

...

break;

case HAL\_PIXEL\_FORMAT\_BLOB:

//创建QCamera3PicChannel

...

break;

default:

break;

}

} else if (newStream->stream\_type == CAMERA3\_STREAM\_INPUT) {

newStream->max\_buffers = MAX\_INFLIGHT\_REPROCESS\_REQUESTS;

} else {

}

for (List<stream\_info\_t\*>::iterator it=mStreamInfo.begin();it != mStreamInfo.end();

it++) {

if ((\*it)->stream == newStream) {

(\*it)->channel = (QCamera3Channel\*) newStream->priv;

break;

}

}

} else {

}

if (newStream->stream\_type != CAMERA3\_STREAM\_INPUT)

mStreamConfigInfo.num\_streams++;

}

}

if (isZsl) {

if (mPictureChannel) {

mPictureChannel->overrideYuvSize(zslStream->width, zslStream->height);

}

} else if (mPictureChannel && m\_bIs4KVideo) {

mPictureChannel->overrideYuvSize(videoWidth, videoHeight);

}

//RAW DUMP channel

if (mEnableRawDump && isRawStreamRequested == false){

cam\_dimension\_t rawDumpSize;

rawDumpSize = getMaxRawSize(mCameraId);

mRawDumpChannel = new QCamera3RawDumpChannel(mCameraHandle->camera\_handle,

mCameraHandle->ops,rawDumpSize,&gCamCapability[mCameraId]->padding\_info,

this, CAM\_QCOM\_FEATURE\_NONE);

...

}

//进行相关Channel的配置

...

/\* Initialize mPendingRequestInfo and mPendnigBuffersMap \*/

for (List<PendingRequestInfo>::iterator i = mPendingRequestsList.begin();

i != mPendingRequestsList.end(); i++) {

clearInputBuffer(i->input\_buffer);

i = mPendingRequestsList.erase(i);

}

mPendingFrameDropList.clear();

// Initialize/Reset the pending buffers list

mPendingBuffersMap.num\_buffers = 0;

mPendingBuffersMap.mPendingBufferList.clear();

mPendingReprocessResultList.clear();

return rc;

}

此方法内容比较多，只抽取其中核心的代码进行说明，它首先会根据HAL的版本来对stream进行相应的配置初始化，然后再根据stream类型对stream\_list的stream创建相应的Channel，主要有QCamera3MetadataChannel，QCamera3SupportChannel等，然后再进行相应的配置，其中QCamera3MetadataChannel在后面的处理capture request的时候会用到，这里就不做分析，而Camerametadata则是Java层和CameraService之间传递的元数据，见android6.0源码分析之Camera API2.0简介中的Camera2架构图，至此，QCamera3HardwareInterface构造结束，与本文相关的就是配置了mCameraDevice.ops。

### openCamera分析

本节主要分析Module是如何打开Camera的，openCamera的代码如下：

//QCamera3HWI.cpp

int QCamera3HardwareInterface::openCamera(struct hw\_device\_t \*\*hw\_device){

int rc = 0;

if (mCameraOpened) {//如果Camera已经被打开，则此次打开的设备为NULL，并且打开结果为PERMISSION\_DENIED

\*hw\_device = NULL;

return PERMISSION\_DENIED;

}

//调用openCamera方法来打开

rc = openCamera();

//打开结果处理

if (rc == 0) {

//获取打开成功的hw\_device\_t对象

\*hw\_device = &mCameraDevice.common;

} else

\*hw\_device = NULL;

}

return rc;

}

它调用了openCamera()方法来打开Camera:

// QCamera3HWI.cpp

int QCamera3HardwareInterface::openCamera()

{

...

//打开camera，获取mCameraHandle

mCameraHandle = camera\_open((uint8\_t)mCameraId);

...

mCameraOpened = true;

//注册mm-camera-interface里的事件处理,其中camEctHandle为事件处理Handle

rc = mCameraHandle->ops->register\_event\_notify(mCameraHandle->camera\_handle,camEvtHandle

,(void \*)this);

return NO\_ERROR;

}

它调用camera\_open方法来打开Camera，并且向CameraHandle注册了Camera 时间处理的Handle–camEvtHandle，首先分析camera\_open方法，这里就将进入高通的Camera的实现了，而Mm\_camera\_interface.c是高通提供的相关操作的接口，接下来分析高通Camera的camera\_open方法：

//Mm\_camera\_interface.c

mm\_camera\_vtbl\_t \* camera\_open(uint8\_t camera\_idx)

{

int32\_t rc = 0;

mm\_camera\_obj\_t\* cam\_obj = NULL;

/\* opened already 如果已经打开\*/

if(NULL != g\_cam\_ctrl.cam\_obj[camera\_idx]) {

/\* Add reference \*/

g\_cam\_ctrl.cam\_obj[camera\_idx]->ref\_count++;

pthread\_mutex\_unlock(&g\_intf\_lock);

return &g\_cam\_ctrl.cam\_obj[camera\_idx]->vtbl;

}

cam\_obj = (mm\_camera\_obj\_t \*)malloc(sizeof(mm\_camera\_obj\_t));

...

/\* initialize camera obj \*/

memset(cam\_obj, 0, sizeof(mm\_camera\_obj\_t));

cam\_obj->ctrl\_fd = -1;

cam\_obj->ds\_fd = -1;

cam\_obj->ref\_count++;

cam\_obj->my\_hdl = mm\_camera\_util\_generate\_handler(camera\_idx);

cam\_obj->vtbl.camera\_handle = cam\_obj->my\_hdl; /\* set handler \*/

//mm\_camera\_ops里绑定了相关的操作接口

cam\_obj->vtbl.ops = &mm\_camera\_ops;

pthread\_mutex\_init(&cam\_obj->cam\_lock, NULL);

pthread\_mutex\_lock(&cam\_obj->cam\_lock);

pthread\_mutex\_unlock(&g\_intf\_lock);

//调用mm\_camera\_open方法来打开camera

rc = mm\_camera\_open(cam\_obj);

pthread\_mutex\_lock(&g\_intf\_lock);

...

//结果处理，并返回

...

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

由代码可知，这里将会初始化一个mm\_camera\_obj\_t对象，其中，ds\_fd为socket fd，而mm\_camera\_ops则绑定了相关的接口，最后调用mm\_camera\_open来打开Camera，首先来看看mm\_camera\_ops绑定了哪些方法：

//Mm\_camera\_interface.c

static mm\_camera\_ops\_t mm\_camera\_ops = {

.query\_capability = mm\_camera\_intf\_query\_capability,

//注册事件通知的方法

.register\_event\_notify = mm\_camera\_intf\_register\_event\_notify,

.close\_camera = mm\_camera\_intf\_close,

.set\_parms = mm\_camera\_intf\_set\_parms,

.get\_parms = mm\_camera\_intf\_get\_parms,

.do\_auto\_focus = mm\_camera\_intf\_do\_auto\_focus,

.cancel\_auto\_focus = mm\_camera\_intf\_cancel\_auto\_focus,

.prepare\_snapshot = mm\_camera\_intf\_prepare\_snapshot,

.start\_zsl\_snapshot = mm\_camera\_intf\_start\_zsl\_snapshot,

.stop\_zsl\_snapshot = mm\_camera\_intf\_stop\_zsl\_snapshot,

.map\_buf = mm\_camera\_intf\_map\_buf,

.unmap\_buf = mm\_camera\_intf\_unmap\_buf,

.add\_channel = mm\_camera\_intf\_add\_channel,

.delete\_channel = mm\_camera\_intf\_del\_channel,

.get\_bundle\_info = mm\_camera\_intf\_get\_bundle\_info,

.add\_stream = mm\_camera\_intf\_add\_stream,

.link\_stream = mm\_camera\_intf\_link\_stream,

.delete\_stream = mm\_camera\_intf\_del\_stream,

//配置stream的方法

.config\_stream = mm\_camera\_intf\_config\_stream,

.qbuf = mm\_camera\_intf\_qbuf,

.get\_queued\_buf\_count = mm\_camera\_intf\_get\_queued\_buf\_count,

.map\_stream\_buf = mm\_camera\_intf\_map\_stream\_buf,

.unmap\_stream\_buf = mm\_camera\_intf\_unmap\_stream\_buf,

.set\_stream\_parms = mm\_camera\_intf\_set\_stream\_parms,

.get\_stream\_parms = mm\_camera\_intf\_get\_stream\_parms,

.start\_channel = mm\_camera\_intf\_start\_channel,

.stop\_channel = mm\_camera\_intf\_stop\_channel,

.request\_super\_buf = mm\_camera\_intf\_request\_super\_buf,

.cancel\_super\_buf\_request = mm\_camera\_intf\_cancel\_super\_buf\_request,

.flush\_super\_buf\_queue = mm\_camera\_intf\_flush\_super\_buf\_queue,

.configure\_notify\_mode = mm\_camera\_intf\_configure\_notify\_mode,

//处理capture的方法

.process\_advanced\_capture = mm\_camera\_intf\_process\_advanced\_capture

};

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

接着分析mm\_camera\_open方法：

//Mm\_camera.c

int32\_t mm\_camera\_open(mm\_camera\_obj\_t \*my\_obj){

...

do{

n\_try--;

//根据设备名字，打开相应的设备驱动fd

my\_obj->ctrl\_fd = open(dev\_name, O\_RDWR | O\_NONBLOCK);

if((my\_obj->ctrl\_fd >= 0) || (errno != EIO) || (n\_try <= 0 )) {

break;

}

usleep(sleep\_msec \* 1000U);

}while (n\_try > 0);

...

//打开domain socket

n\_try = MM\_CAMERA\_DEV\_OPEN\_TRIES;

do {

n\_try--;

my\_obj->ds\_fd = mm\_camera\_socket\_create(cam\_idx, MM\_CAMERA\_SOCK\_TYPE\_UDP);

usleep(sleep\_msec \* 1000U);

} while (n\_try > 0);

...

//初始化锁

pthread\_mutex\_init(&my\_obj->msg\_lock, NULL);

pthread\_mutex\_init(&my\_obj->cb\_lock, NULL);

pthread\_mutex\_init(&my\_obj->evt\_lock, NULL);

pthread\_cond\_init(&my\_obj->evt\_cond, NULL);

//开启线程，它的线程体在mm\_camera\_dispatch\_app\_event方法中

mm\_camera\_cmd\_thread\_launch(&my\_obj->evt\_thread,

mm\_camera\_dispatch\_app\_event,

(void \*)my\_obj);

mm\_camera\_poll\_thread\_launch(&my\_obj->evt\_poll\_thread,

MM\_CAMERA\_POLL\_TYPE\_EVT);

mm\_camera\_evt\_sub(my\_obj, TRUE);

return rc;

...

}

由代码可知，它会打开Camera的设备文件，然后开启dispatch\_app\_event线程，线程方法体mm\_camera\_dispatch\_app\_event方法代码如下：

//Mm\_camera.c

static void mm\_camera\_dispatch\_app\_event(mm\_camera\_cmdcb\_t \*cmd\_cb,void\* user\_data){

mm\_camera\_cmd\_thread\_name("mm\_cam\_event");

int i;

mm\_camera\_event\_t \*event = &cmd\_cb->u.evt;

mm\_camera\_obj\_t \* my\_obj = (mm\_camera\_obj\_t \*)user\_data;

if (NULL != my\_obj) {

pthread\_mutex\_lock(&my\_obj->cb\_lock);

for(i = 0; i < MM\_CAMERA\_EVT\_ENTRY\_MAX; i++) {

if(my\_obj->evt.evt[i].evt\_cb) {

//调用camEvtHandle方法

my\_obj->evt.evt[i].evt\_cb(

my\_obj->my\_hdl,

event,

my\_obj->evt.evt[i].user\_data);

}

}

pthread\_mutex\_unlock(&my\_obj->cb\_lock);

}

}

最后会调用mm-camera-interface中注册好的事件处理evt\_cb，它就是在前面注册好的camEvtHandle：

//QCamera3HWI.cpp

void QCamera3HardwareInterface::camEvtHandle(uint32\_t /\*camera\_handle\*/,mm\_camera\_event\_t \*evt,

void \*user\_data){

//获取QCamera3HardwareInterface接口指针

QCamera3HardwareInterface \*obj = (QCamera3HardwareInterface \*)user\_data;

if (obj && evt) {

switch(evt->server\_event\_type) {

case CAM\_EVENT\_TYPE\_DAEMON\_DIED:

camera3\_notify\_msg\_t notify\_msg;

memset(&notify\_msg, 0, sizeof(camera3\_notify\_msg\_t));

notify\_msg.type = CAMERA3\_MSG\_ERROR;

notify\_msg.message.error.error\_code = CAMERA3\_MSG\_ERROR\_DEVICE;

notify\_msg.message.error.error\_stream = NULL;

notify\_msg.message.error.frame\_number = 0;

obj->mCallbackOps->notify(obj->mCallbackOps, &notify\_msg);

break;

case CAM\_EVENT\_TYPE\_DAEMON\_PULL\_REQ:

pthread\_mutex\_lock(&obj->mMutex);

obj->mWokenUpByDaemon = true;

//开启process\_capture\_request

obj->unblockRequestIfNecessary();

pthread\_mutex\_unlock(&obj->mMutex);

break;

default:

break;

}

} else {

}

}

由代码可知，它会调用QCamera3HardwareInterface的unblockRequestIfNecessary来发起结果处理请求：

//QCamera3HWI.cpp

void QCamera3HardwareInterface::unblockRequestIfNecessary()

{

// Unblock process\_capture\_request

//开启process\_capture\_request

pthread\_cond\_signal(&mRequestCond);

}

在初始化QCamera3HardwareInterface对象的时候，就绑定了处理Metadata的回调captureResultCb方法：它主要是对数据源进行相应的处理，而具体的capture请求的结果处理还是由process\_capture\_request来进行处理的，而这里会调用方法unblockRequestIfNecessary来触发process\_capture\_request方法执行，而在Camera框架中，发起请求时会启动一个RequestThread线程，在它的threadLoop方法中，会不停的调用process\_capture\_request方法来进行请求的处理，而它最后会回调Camera3Device中的processCaptureResult方法来进行结果处理：

//Camera3Device.cpp

void Camera3Device::processCaptureResult(const camera3\_capture\_result \*result) {

...

{

...

if (mUsePartialResult && result->result != NULL) {

if (mDeviceVersion >= CAMERA\_DEVICE\_API\_VERSION\_3\_2) {

...

if (isPartialResult) {

request.partialResult.collectedResult.append(result->result);

}

} else {

camera\_metadata\_ro\_entry\_t partialResultEntry;

res = find\_camera\_metadata\_ro\_entry(result->result,

ANDROID\_QUIRKS\_PARTIAL\_RESULT, &partialResultEntry);

if (res != NAME\_NOT\_FOUND &&partialResultEntry.count > 0 &&

partialResultEntry.data.u8[0] ==ANDROID\_QUIRKS\_PARTIAL\_RESULT\_PARTIAL) {

isPartialResult = true;

request.partialResult.collectedResult.append(

result->result);

request.partialResult.collectedResult.erase(

ANDROID\_QUIRKS\_PARTIAL\_RESULT);

}

}

if (isPartialResult) {

// Fire off a 3A-only result if possible

if (!request.partialResult.haveSent3A) {

//处理3A结果

request.partialResult.haveSent3A =processPartial3AResult(frameNumber,

request.partialResult.collectedResult,request.resultExtras);

}

}

}

...

//查找camera元数据入口

camera\_metadata\_ro\_entry\_t entry;

res = find\_camera\_metadata\_ro\_entry(result->result,

ANDROID\_SENSOR\_TIMESTAMP, &entry);

if (shutterTimestamp == 0) {

request.pendingOutputBuffers.appendArray(result->output\_buffers,

result->num\_output\_buffers);

} else {

重要的分析//返回处理的outputbuffer

returnOutputBuffers(result->output\_buffers,

result->num\_output\_buffers, shutterTimestamp);

}

if (result->result != NULL && !isPartialResult) {

if (shutterTimestamp == 0) {

request.pendingMetadata = result->result;

request.partialResult.collectedResult = collectedPartialResult;

} else {

CameraMetadata metadata;

metadata = result->result;

//发送Capture结构，即调用通知回调

sendCaptureResult(metadata, request.resultExtras,

collectedPartialResult, frameNumber, hasInputBufferInRequest,

request.aeTriggerCancelOverride);

}

}

removeInFlightRequestIfReadyLocked(idx);

} // scope for mInFlightLock

if (result->input\_buffer != NULL) {

if (hasInputBufferInRequest) {

Camera3Stream \*stream =

Camera3Stream::cast(result->input\_buffer->stream);

重要的分析//返回处理的inputbuffer

res = stream->returnInputBuffer(\*(result->input\_buffer));

} else {}

}

}

分析returnOutputBuffers方法，inputbuffer的runturnInputBuffer方法流程类似：

//Camera3Device.cpp

void Camera3Device::returnOutputBuffers(const camera3\_stream\_buffer\_t \*outputBuffers, size\_t

numBuffers, nsecs\_t timestamp) {

for (size\_t i = 0; i < numBuffers; i++)

{

Camera3Stream \*stream = Camera3Stream::cast(outputBuffers[i].stream);

status\_t res = stream->returnBuffer(outputBuffers[i], timestamp);

...

}

}

方法里调用了returnBuffer方法：

//Camera3Stream.cpp

status\_t Camera3Stream::returnBuffer(const camera3\_stream\_buffer &buffer,nsecs\_t timestamp) {

//返回buffer

status\_t res = returnBufferLocked(buffer, timestamp);

if (res == OK) {

fireBufferListenersLocked(buffer, /\*acquired\*/false, /\*output\*/true);

mOutputBufferReturnedSignal.signal();

}

return res;

}

再继续看returnBufferLocked,它调用了returnAnyBufferLocked方法，而returnAnyBufferLocked方法又调用了returnBufferCheckedLocked方法，现在分析returnBufferCheckedLocked：

// Camera3OutputStream.cpp

status\_t Camera3OutputStream::returnBufferCheckedLocked(const camera3\_stream\_buffer &buffer,

nsecs\_t timestamp,bool output,/\*out\*/sp<Fence> \*releaseFenceOut) {

...

// Fence management - always honor release fence from HAL

sp<Fence> releaseFence = new Fence(buffer.release\_fence);

int anwReleaseFence = releaseFence->dup();

if (buffer.status == CAMERA3\_BUFFER\_STATUS\_ERROR) {

// Cancel buffer

res = currentConsumer->cancelBuffer(currentConsumer.get(),

container\_of(buffer.buffer, ANativeWindowBuffer, handle),

anwReleaseFence);

...

} else {

...

res = currentConsumer->queueBuffer(currentConsumer.get(),

container\_of(buffer.buffer, ANativeWindowBuffer, handle),

anwReleaseFence);

...

}

...

return res;

}

由代码可知，如果Buffer没有出现状态错误，它会调用currentConsumer的queueBuffer方法，而具体的Consumer则是在应用层初始化Camera时进行绑定的，典型的Consumer有SurfaceTexture，ImageReader等，而在Native层中，它会调用BufferQueueProducer的queueBuffer方法：

#### queueBuffer

f

/ BufferQueueProducer.cpp

status\_t BufferQueueProducer::queueBuffer(int slot,

const QueueBufferInput &input, QueueBufferOutput \*output) {

...

//初始化Frame可用的监听器

sp<IConsumerListener> frameAvailableListener;

sp<IConsumerListener> frameReplacedListener;

int callbackTicket = 0;

BufferItem item;

{ // Autolock scope

...

const sp<GraphicBuffer>& graphicBuffer(mSlots[slot].mGraphicBuffer);

Rect bufferRect(graphicBuffer->getWidth(), graphicBuffer->getHeight());

Rect croppedRect;

crop.intersect(bufferRect, &croppedRect);

...

//如果队列为空

if (mCore->mQueue.empty()) {

mCore->mQueue.push\_back(item);

frameAvailableListener = mCore->mConsumerListener;

} else {

//否则，不为空，对Buffer进行处理，并获取FrameAvailableListener监听

BufferQueueCore::Fifo::iterator front(mCore->mQueue.begin());

if (front->mIsDroppable) {

if (mCore->stillTracking(front)) {

mSlots[front->mSlot].mBufferState = BufferSlot::FREE;

mCore->mFreeBuffers.push\_front(front->mSlot);

}

\*front = item;

frameReplacedListener = mCore->mConsumerListener;

} else {

mCore->mQueue.push\_back(item);

frameAvailableListener = mCore->mConsumerListener;

}

}

mCore->mBufferHasBeenQueued = true;

mCore->mDequeueCondition.broadcast();

output->inflate(mCore->mDefaultWidth, mCore->mDefaultHeight,mCore->mTransformHint,

static\_cast<uint32\_t>(mCore->mQueue.size()));

// Take a ticket for the callback functions

callbackTicket = mNextCallbackTicket++;

mCore->validateConsistencyLocked();

} // Autolock scope

...

{

...

if (frameAvailableListener != NULL) {

//回调SurfaceTexture中定义好的监听IConsumerListener的onFrameAvailable方法来对数据进行处理

frameAvailableListener->onFrameAvailable(item);

} else if (frameReplacedListener != NULL) {

frameReplacedListener->onFrameReplaced(item);

}

++mCurrentCallbackTicket;

mCallbackCondition.broadcast();

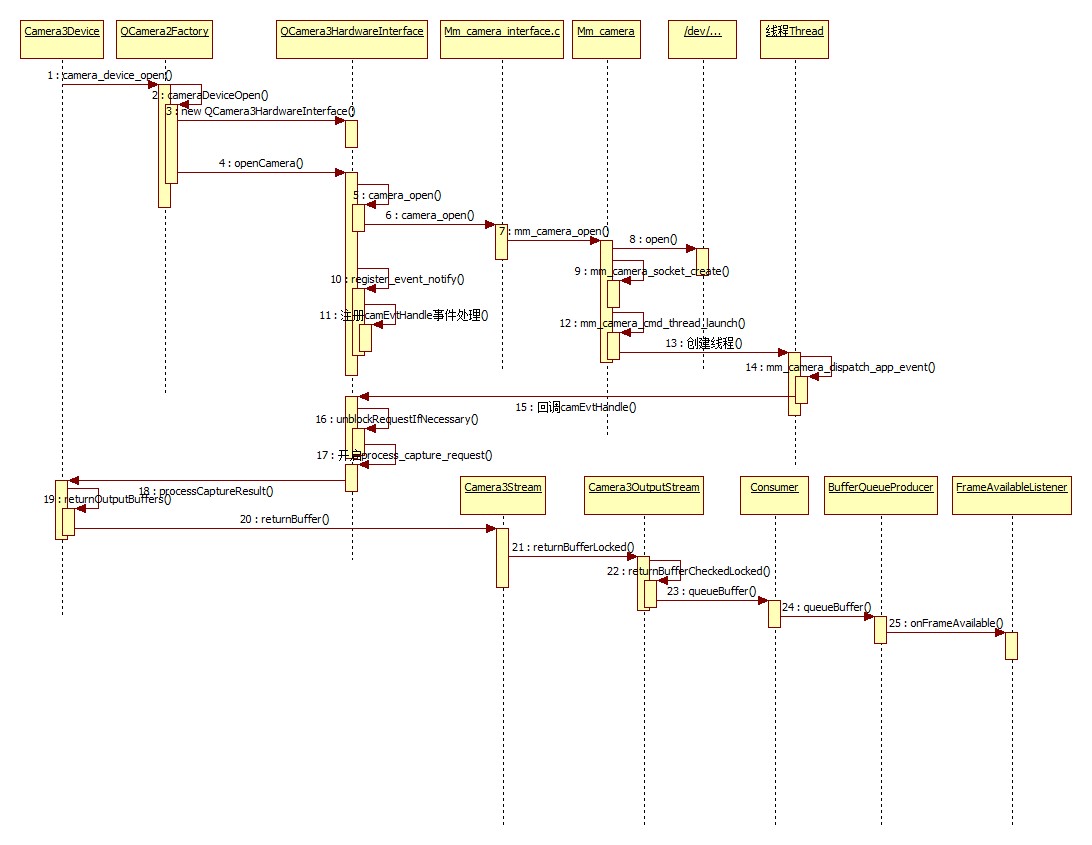
}

return NO\_ERROR;

}

由代码可知，它最后会调用Consumer的回调FrameAvailableListener的onFrameAvailable方法，到这里，就比较清晰为什么我们在写Camera应用，为其初始化Surface时，我们需要重写FrameAvailableListener了，因为在此方法里面，会进行结果的处理，至此，Camera HAL的Open流程就分析结束了。下面给出流程的时序图

#### 时序图



# Camera2初始化流程分析

## Camera2初始化的应用层流程分析

Camera2的初始化流程与Camera1.0有所区别，本文将就Camera2的内置应用来分析Camera2.0的初始化过程。Camera2.0首先启动的是CameraActivity，而它继承自QuickActivity，在代码中你会发现没有重写OnCreate等生命周期方法，因为此处采用的是模板方法的设计模式，在QuickActivity中的onCreate方法调用的是onCreateTasks等方法，所以要看onCreate方法就只须看onCreateTasks方法即可：

### onCreateTasks

//CameraActivity.java

@Override

public void onCreateTasks(Bundle state) {

Profile profile = mProfiler.create("CameraActivity.onCreateTasks")

.start();

...

mOnCreateTime = System.currentTimeMillis();

mAppContext = getApplicationContext();

mMainHandler = new MainHandler(this, getMainLooper());

…

try {

//初始化OneCameraOpener对象

①mOneCameraOpener = OneCameraModule.provideOneCameraOpener(

mFeatureConfig, mAppContext,mActiveCameraDeviceTracker,

ResolutionUtil.getDisplayMetrics(this));

mOneCameraManager = OneCameraModule.provideOneCameraManager();

} catch (OneCameraException e) {...}

…

//建立模块信息

②ModulesInfo.setupModules(mAppContext, mModuleManager, mFeatureConfig);

…

//进行初始化

③mCurrentModule.init(this, isSecureCamera(), isCaptureIntent());

…

}

如代码所示，重要的有以上三点，先看第一点：

//OneCameraModule.java

public static OneCameraOpener provideOneCameraOpener(OneCameraFeatureConfig

featureConfig, Context context, ActiveCameraDeviceTracker

activeCameraDeviceTracker,DisplayMetrics displayMetrics)

throws OneCameraException {

//创建OneCameraOpener对象

Optional<OneCameraOpener> manager = Camera2OneCameraOpenerImpl.create(

featureConfig, context, activeCameraDeviceTracker, displayMetrics);

if (!manager.isPresent()) {

manager = LegacyOneCameraOpenerImpl.create();

}

...

return manager.get();

}

13

14

它调用Camera2OneCameraOpenerImpl的create方法来获得一个OneCameraOpener对象，以供CameraActivity之后的操作使用，继续看create方法：

//Camera2OneCameraOpenerImpl.java

public static Optional<OneCameraOpener> create(OneCameraFeatureConfig

featureConfig, Context context, ActiveCameraDeviceTracker

activeCameraDeviceTracker, DisplayMetrics displayMetrics) {

...

CameraManager cameraManager;

try {

cameraManager = AndroidServices.instance().provideCameraManager();

} catch (IllegalStateException ex) {...}

//新建一个Camera2OneCameraOpenerImpl对象

OneCameraOpener oneCameraOpener = new Camera2OneCameraOpenerImpl(

featureConfig, context, cameraManager,

activeCameraDeviceTracker, displayMetrics);

return Optional.of(oneCameraOpener);

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

很明显，它首先获取一个cameraManger对象，然后根据这个cameraManager对象来新创建了一个Camera2OneCameraOpenerImpl对象，所以第一步主要是为了获取一个OneCameraOpener对象，它的实现为Camera2OneCameraOpenerImpl类。

继续看第二步，ModulesInfo.setupModules:

// ModulesInfo.java

public static void setupModules(Context context, ModuleManager moduleManager,

OneCameraFeatureConfig config) {

Resources res = context.getResources();

int photoModuleId = context.getResources().getInteger(

R.integer.camera\_mode\_photo);

//注册Photo模块

registerPhotoModule(moduleManager, photoModuleId,

SettingsScopeNamespaces.PHOTO,config.isUsingCaptureModule());

//计算你还Photo模块设置为默认的模块

moduleManager.setDefaultModuleIndex(photoModuleId);

//注册Videa模块

registerVideoModule(moduleManager, res.getInteger(

R.integer.camera\_mode\_video),SettingsScopeNamespaces.VIDEO);

if (PhotoSphereHelper.hasLightCycleCapture(context)) {//开启闪光

//注册广角镜头

registerWideAngleModule(moduleManager, res.getInteger(

R.integer.camera\_mode\_panorama),SettingsScopeNamespaces

.PANORAMA);

//注册光球模块

registerPhotoSphereModule(moduleManager,res.getInteger(

R.integer.camera\_mode\_photosphere),

SettingsScopeNamespaces.PANORAMA);

}

//若需重新聚焦

if (RefocusHelper.hasRefocusCapture(context)) {

//注册重聚焦模块

registerRefocusModule(moduleManager, res.getInteger(

R.integer.camera\_mode\_refocus),

SettingsScopeNamespaces.REFOCUS);

}

//如果有色分离模块

if (GcamHelper.hasGcamAsSeparateModule(config)) {

//注册色分离模块

registerGcamModule(moduleManager, res.getInteger(

R.integer.camera\_mode\_gcam),SettingsScopeNamespaces.PHOTO,

config.getHdrPlusSupportLevel(OneCamera.Facing.BACK));

}

int imageCaptureIntentModuleId = res.getInteger(

R.integer.camera\_mode\_capture\_intent);

registerCaptureIntentModule(moduleManager,

imageCaptureIntentModuleId,SettingsScopeNamespaces.PHOTO,

config.isUsingCaptureModule());

}

代码根据配置信息，进行一系列模块的注册，其中PhotoModule和VideoModule被注册，而其他的module则是根据配置来进行的，因为打开Camera应用，既可以拍照片也可以拍视频，此处，只分析PhoneModule的注册：

### PhoneModule

/ ModulesInfo.java

private static void registerPhotoModule(ModuleManager moduleManager, final

int moduleId, final String namespace,

final boolean enableCaptureModule) {

//向ModuleManager注册PhotoModule模块

moduleManager.registerModule(new ModuleManager.ModuleAgent() {

@Override

public int getModuleId() {

return moduleId;

}

@Override

public boolean requestAppForCamera() {

return !enableCaptureModule;

}

@Override

public String getScopeNamespace() {

return namespace;

}

@Override

public ModuleController createModule(AppController app, Intent

intent) {

Log.v(TAG, "EnableCaptureModule = " + enableCaptureModule);

//创建ModuleController

return enableCaptureModule ? new CaptureModule(app)

: new PhotoModule(app);

}

});

}

由代码可知，它最终是由ModuleManager来新建一个CaptureModule实例，而CaptureModule其实实现了ModuleController ，即创建了一个CaptureModule模式下的ModuleController对象，而真正的CaptureModule的具体实现为ModuleManagerImpl。

至此，前两步已经获得了OneCameraOpener以及新建了ModuleController，并进行了注册，接下来分析第三步

### mCurrentModule.init(this, isSecureCamera(), isCaptureIntent()):

//CaptureModule.java

public void init(CameraActivity activity, boolean isSecureCamera, boolean

isCaptureIntent) {

...

HandlerThread thread = new HandlerThread("CaptureModule.mCameraHandler");

thread.start();

mCameraHandler = new Handler(thread.getLooper());

//获取第一步中创建的OneCameraOpener对象

mOneCameraOpener = mAppController.getCameraOpener();

try {

//获取前面创建的OneCameraManager对象

mOneCameraManager = OneCameraModule.provideOneCameraManager();

} catch (OneCameraException e) {

Log.e(TAG, "Unable to provide a OneCameraManager. ", e);

}

`...

//新建CaptureModule的UI

mUI = new CaptureModuleUI(activity, mAppController.

getModuleLayoutRoot(), mUIListener);

//设置预览状态的监听

mAppController.setPreviewStatusListener(mPreviewStatusListener);

synchronized (mSurfaceTextureLock) {

//获取SurfaceTexture

mPreviewSurfaceTexture = mAppController.getCameraAppUI()

.getSurfaceTexture();

}

}

首先获取前面创建的OneCameraOpener对象以及OneCameraManager对象，然后再设置预览状态监听，这里主要分析预览状态的监听：

### PreviewStatusListener

/CaptureModule.java

private final PreviewStatusListener mPreviewStatusListener = new

PreviewStatusListener() {

...

@Override

public void onSurfaceTextureAvailable(SurfaceTexture surface,

int width, int height) {

updatePreviewTransform(width, height, true);

synchronized (mSurfaceTextureLock) {

mPreviewSurfaceTexture = surface;

}

//打开Camera

reopenCamera();

}

@Override

public boolean onSurfaceTextureDestroyed(SurfaceTexture surface) {

Log.d(TAG, "onSurfaceTextureDestroyed");

synchronized (mSurfaceTextureLock) {

mPreviewSurfaceTexture = null;

}

//关闭Camera

closeCamera();

return true;

}

@Override

public void onSurfaceTextureSizeChanged(SurfaceTexture surface,

int width, int height) {

//更新预览尺寸

updatePreviewBufferSize();

}

...

};

由代码可知，当SurfaceTexture的状态变成可用的时候，会调用reopenCamera()方法来打开Camera，所以继续分析reopenCamera()方法：

#### reopenCamera

dsa

/CaptureModule.java

private void reopenCamera() {

if (mPaused) {

return;

}

AsyncTask.THREAD\_POOL\_EXECUTOR.execute(new Runnable() {

@Override

public void run() {

closeCamera();

if(!mAppController.isPaused()) {

//开启Camera并开始预览

openCameraAndStartPreview();

}

}

});

}

它采用异步任务的方法，开启一个异步线程来进行启动操作，首先关闭打开的Camera，然后如果AppController不处于暂停状态，则打开Camera并启动Preview操作，所以继续分析openCameraAndStartPreview方法：

#### openCameraAndStartPreview

//CaptureModule.java

private void openCameraAndStartPreview() {

...

if (mOneCameraOpener == null) {

Log.e(TAG, "no available OneCameraManager, showing error dialog");

//释放CameraOpenCloseLock锁

mCameraOpenCloseLock.release();

mAppController.getFatalErrorHandler().onGenericCameraAccessFailure();

guard.stop("No OneCameraManager");

return;

}

// Derive objects necessary for camera creation.

MainThread mainThread = MainThread.create();

//查找需要打开的CameraId

CameraId cameraId = mOneCameraManager.findFirstCameraFacing(

mCameraFacing);

...

//打开Camera

mOneCameraOpener.open(cameraId, captureSetting, mCameraHandler,

mainThread, imageRotationCalculator, mBurstController,

mSoundPlayer,new OpenCallback() {

@Override

public void onFailure() {

//进行失败的处理

...

}

@Override

public void onCameraClosed() {

...

}

@Override

public void onCameraOpened(@Nonnull final OneCamera camera) {

Log.d(TAG, "onCameraOpened: " + camera);

mCamera = camera;

if (mAppController.isPaused()) {

onFailure();

return;

}

...

mMainThread.execute(new Runnable() {

@Override

public void run() {

//通知UI，Camera状态变化

mAppController.getCameraAppUI().onChangeCamera();

//使能拍照按钮

mAppController.getButtonManager().enableCameraButton();

}

});

//至此，Camera打开成功，开始预览

camera.startPreview(new Surface(getPreviewSurfaceTexture()),

new CaptureReadyCallback() {

@Override

public void onSetupFailed() {

...

}

@Override

public void onReadyForCapture() {

//释放锁

mCameraOpenCloseLock.release();

mMainThread.execute(new Runnable() {

@Override

public void run() {

...

onPreviewStarted();

...

onReadyStateChanged(true);

//设置CaptureModule为Capture准备的状态监听

mCamera.setReadyStateChangedListener(

CaptureModule.this);

mUI.initializeZoom(mCamera.getMaxZoom());

mCamera.setFocusStateListener(

CaptureModule.this);

}

});

}

});

}

}, mAppController.getFatalErrorHandler());

guard.stop("mOneCameraOpener.open()");

}

}

首先，它主要会调用Camera2OneCameraOpenerImpl的open方法来打开Camera，并定义了开启的回调函数，对开启结束后的结果进行处理，如失败则释放mCameraOpenCloseLock，并暂停mAppController，如果打开成功，通知UI成功，并开启Camera的Preview，并且定义了Preview的各种回调操作，这里主要分析Open过程，所以继续分析：

//Camera2OneCameraOpenerImpl.java

@Override

public void open(

...

mActiveCameraDeviceTracker.onCameraOpening(cameraKey);

//打开Camera，此处调用框架层的CameraManager类的openCamera，进入frameworks层

mCameraManager.openCamera(cameraKey.getValue(),

new CameraDevice.StateCallback() {

private boolean isFirstCallback = true;

@Override

...

@Override

public void onOpened(CameraDevice device) {

//第一次调用此回调

if (isFirstCallback) {

isFirstCallback = false;

try {

CameraCharacteristics characteristics = mCameraManager

.getCameraCharacteristics(device.getId());

...

//创建OneCamera对象

OneCamera oneCamera = OneCameraCreator.create(device,

characteristics, mFeatureConfig, captureSetting,

mDisplayMetrics, mContext, mainThread,

imageRotationCalculator, burstController, soundPlayer,

fatalErrorHandler);

if (oneCamera != null) {

//如果oneCamera不为空，则回调onCameraOpened，后面将做分析

openCallback.onCameraOpened(oneCamera);

} else {

...

openCallback.onFailure();

}

} catch (CameraAccessException e) {

openCallback.onFailure();

} catch (OneCameraAccessException e) {

Log.d(TAG, "Could not create OneCamera", e);

openCallback.onFailure();

}

}

}

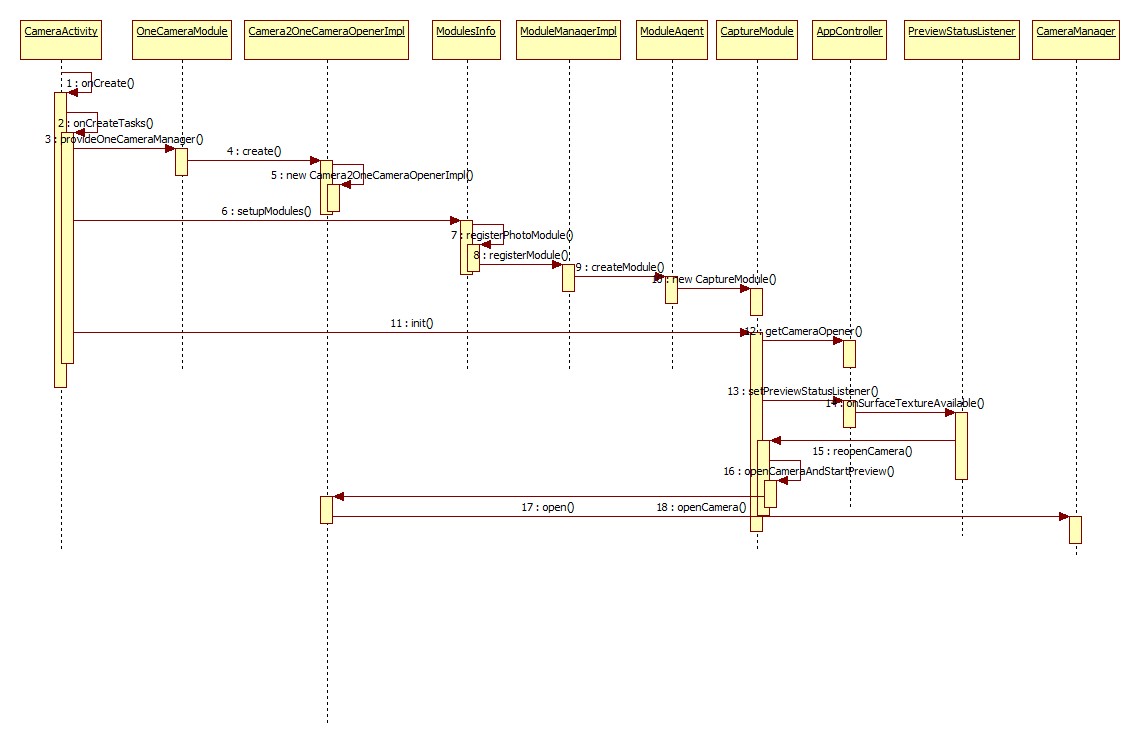
}, handler);

...

}

至此，Camera的初始化流程中应用层的分析就差不多了，下一步将会调用CameraManager的openCamera方法来进入框架层，并进行Camera的初始化，下面将应用层的初始化时序图：

### 初始化时序图



## Camera2初始化的框架层流程分析

由上面的分析可知，将由应用层进入到框架层处理，将会调用CameraManager的openCamera方法，并且定义了CameraDevice的状态回调函数，具体的回调操作此处不做分析，继续跟踪openCamera()方法：

//CameraManager.java(frameworks/base/core/java/android/hardware/camera2)

@RequiresPermission(android.Manifest.permission.CAMERA)

public void openCamera(@NonNull String cameraId,@NonNull final

CameraDevice.StateCallback callback, @Nullable Handler handler)

throws CameraAccessException {

...

openCameraDeviceUserAsync(cameraId, callback, handler);

}

1

2

3

4

5

6

7

8

由代码可知，此处与Camera1.0有明显不同，Camera1.0是通过一个异步的线程以及JNI来调用android\_hardware\_camera.java里面的native\_setup方法来连接Camera，其使用的是C++的Binder来与CameraService进行通信的，而此处则不一样，它直接使用的是Java层的Binder来进行通信，先看openCameraDeviceUserAsync代码:

//CameraManager.java(frameworks/base/core/java/android/hardware/camera2)

private CameraDevice openCameraDeviceUserAsync(String cameraId,

CameraDevice.StateCallback callback, Handler handler)

throws CameraAccessException {

CameraCharacteristics characteristics = getCameraCharacteristics(

cameraId);

CameraDevice device = null;

try {

synchronized (mLock) {

ICameraDeviceUser cameraUser = null;

//初始化一个CameraDevice对象

android.hardware.camera2.impl.CameraDeviceImpl deviceImpl =

new android.hardware.camera2.impl.CameraDeviceImpl(cameraId,

callback, handler, characteristics);

BinderHolder holder = new BinderHolder();

//获取回调

ICameraDeviceCallbacks callbacks = deviceImpl.getCallbacks();

int id = Integer.parseInt(cameraId);

try {

if (supportsCamera2ApiLocked(cameraId)) {

//通过Java层的Binder获取CameraService

ICameraService cameraService = CameraManagerGlobal.get()

.getCameraService();

...

//通过CameraService连接Camera设备

cameraService.connectDevice(callbacks, id, mContext

.getOpPackageName(), USE\_CALLING\_UID, holder);

//获取连接成功的CameraUser对象，它用来与CameraService通信

cameraUser = ICameraDeviceUser.Stub.asInterface(

holder.getBinder());

} else {

//使用遗留的API

cameraUser = CameraDeviceUserShim.connectBinderShim(

callbacks, id);

}

} catch (CameraRuntimeException e) {

...

} catch (RemoteException e) {

...

//将其包装成DeviceImpl对象，供应用层使用

deviceImpl.setRemoteDevice(cameraUser);

device = deviceImpl;

}

} catch (NumberFormatException e) {

...

} catch (CameraRuntimeException e) {

throw e.asChecked();

}

return device;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

此方法的目的是通过CameraService来连接并获取CameraDevice对象，该对象用来与Camera进行通信操作。代码首先通过Java层的Binder机制获取CameraService，然后调用其connectDevice方法来连接CaneraDevice，最后Camera返回的是CameraDeviceUser对象，而接着将其封装成Jav层CameraDevice对象，而之后所有与Camera的通信都通过CameraDevice的接口来进行。接下来分析一下Native层下的CameraDevice的初始化过程：

//CameraService.cpp，其中device为输出对象

status\_t CameraService::connectDevice(const sp<ICameraDeviceCallbacks>& cameraCb,int cameraId,

const String16& clientPackageName,int clientUid,/\*out\*/sp<ICameraDeviceUser>& device) {

status\_t ret = NO\_ERROR;

String8 id = String8::format("%d", cameraId);

sp<CameraDeviceClient> client = nullptr;

ret = connectHelper<ICameraDeviceCallbacks,CameraDeviceClient>(cameraCb, id,

CAMERA\_HAL\_API\_VERSION\_UNSPECIFIED, clientPackageName, clientUid, API\_2, false, false,

/\*out\*/client);//client为输出对象

...

device = client;

return NO\_ERROR;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

Native层的connectDevice方法就是调用了connectHelper方法，所以继续分析connectHelper：

//CameraService.h

template<class CALLBACK, class CLIENT>

status\_t CameraService::connectHelper(const sp<CALLBACK>& cameraCb, const String8& cameraId,

int halVersion, const String16& clientPackageName, int clientUid,

apiLevel effectiveApiLevel, bool legacyMode, bool shimUpdateOnly,

/\*out\*/sp<CLIENT>& device) {

status\_t ret = NO\_ERROR;

String8 clientName8(clientPackageName);

int clientPid = getCallingPid();

...

sp<CLIENT> client = nullptr;

{

...

//如果有必要，给FlashLight关闭设备的机会

mFlashlight->prepareDeviceOpen(cameraId);

//获取CameraId

int id = cameraIdToInt(cameraId);

...

//获取Device的版本，此处为Device3

int deviceVersion = getDeviceVersion(id, /\*out\*/&facing);

sp<BasicClient> tmp = nullptr;

//获取client对象

if((ret = makeClient(this, cameraCb, clientPackageName, cameraId, facing, clientPid,

clientUid, getpid(), legacyMode, halVersion, deviceVersion, effectiveApiLevel,

/\*out\*/&tmp)) != NO\_ERROR) {

return ret;

}

client = static\_cast<CLIENT\*>(tmp.get());

//调用client的初始化函数来初始化模块

if ((ret = client->initialize(mModule)) != OK) {

ALOGE("%s: Could not initialize client from HAL module.", \_\_FUNCTION\_\_);

return ret;

}

sp<IBinder> remoteCallback = client->getRemote();

if (remoteCallback != nullptr) {

remoteCallback->linkToDeath(this);

}

} // lock is destroyed, allow further connect calls

//将client赋值给输出Device

device = client;

return NO\_ERROR;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

CameraService根据Camera的相关参数来获取一个client，如makeClient方法，然后再调用client的initialize来进行初始化，首先看makeClient：

//CameraService.cpp

status\_t CameraService::makeClient(const sp<CameraService>& cameraService,

const sp<IInterface>& cameraCb, const String16& packageName, const String8& cameraId,

int facing, int clientPid, uid\_t clientUid, int servicePid, bool legacyMode,

int halVersion, int deviceVersion, apiLevel effectiveApiLevel,

/\*out\*/sp<BasicClient>\* client) {

//将字符串的CameraId转换成整形

int id = cameraIdToInt(cameraId);

...

if (halVersion < 0 || halVersion == deviceVersion) {//判断Camera HAL版本是否和Device的版本相同

switch(deviceVersion) {

case CAMERA\_DEVICE\_API\_VERSION\_1\_0:

if (effectiveApiLevel == API\_1) { // Camera1 API route

sp<ICameraClient> tmp = static\_cast<ICameraClient\*>(cameraCb.get());

\*client = new CameraClient(cameraService, tmp, packageName, id, facing,

clientPid, clientUid, getpid(), legacyMode);

} else { // Camera2 API route

ALOGW("Camera using old HAL version: %d", deviceVersion);

return -EOPNOTSUPP;

}

break;

case CAMERA\_DEVICE\_API\_VERSION\_2\_0:

case CAMERA\_DEVICE\_API\_VERSION\_2\_1:

case CAMERA\_DEVICE\_API\_VERSION\_3\_0:

case CAMERA\_DEVICE\_API\_VERSION\_3\_1:

case CAMERA\_DEVICE\_API\_VERSION\_3\_2:

case CAMERA\_DEVICE\_API\_VERSION\_3\_3:

if (effectiveApiLevel == API\_1) { // Camera1 API route

sp<ICameraClient> tmp = static\_cast<ICameraClient\*>(cameraCb.get());

\*client = new Camera2Client(cameraService, tmp, packageName, id, facing,

clientPid, clientUid, servicePid, legacyMode);

} else { // Camera2 API route

sp<ICameraDeviceCallbacks> tmp =

static\_cast<ICameraDeviceCallbacks\*>(cameraCb.get());

\*client = new CameraDeviceClient(cameraService, tmp, packageName, id,

facing, clientPid, clientUid, servicePid);

}

break;

default:

// Should not be reachable

ALOGE("Unknown camera device HAL version: %d", deviceVersion);

return INVALID\_OPERATION;

}

} else {

// A particular HAL version is requested by caller. Create CameraClient

// based on the requested HAL version.

if (deviceVersion > CAMERA\_DEVICE\_API\_VERSION\_1\_0 &&

halVersion == CAMERA\_DEVICE\_API\_VERSION\_1\_0) {

// Only support higher HAL version device opened as HAL1.0 device.

sp<ICameraClient> tmp = static\_cast<ICameraClient\*>(cameraCb.get());

\*client = new CameraClient(cameraService, tmp, packageName, id, facing,

clientPid, clientUid, servicePid, legacyMode);

} else {

// Other combinations (e.g. HAL3.x open as HAL2.x) are not supported yet.

ALOGE("Invalid camera HAL version %x: HAL %x device can only be"

" opened as HAL %x device", halVersion, deviceVersion,

CAMERA\_DEVICE\_API\_VERSION\_1\_0);

return INVALID\_OPERATION;

}

}

return NO\_ERROR;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

其中就是创建一个Client对象，由于此处分析的是Camera API2.0，其HAL的版本是3.0+，而Device的版本则其Device的版本即为3.0+，所以会创建一个CameraDeviceClient对象，至此，makeClient已经创建了client对象，并返回了，接着看它的初始化：

//CameraDeviceClient.cpp

status\_t CameraDeviceClient::initialize(CameraModule \*module)

{

ATRACE\_CALL();

status\_t res;

//调用Camera2ClientBase的初始化函数来初始化CameraModule模块

res = Camera2ClientBase::initialize(module);

if (res != OK) {

return res;

}

String8 threadName;

//初始化FrameProcessor

mFrameProcessor = new FrameProcessorBase(mDevice);

threadName = String8::format("CDU-%d-FrameProc", mCameraId);

mFrameProcessor->run(threadName.string());

//并注册监听，监听的实现就在CameraDeviceClient类中

mFrameProcessor->registerListener(FRAME\_PROCESSOR\_LISTENER\_MIN\_ID,

FRAME\_PROCESSOR\_LISTENER\_MAX\_ID, /\*listener\*/this,/\*sendPartials\*/true);

return OK;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

它会调用Camera2ClientBase的initialize方法来初始化，并且会初始化一个FrameProcessor来进行帧处理，主要是回调每一帧的ExtraResult到应用中，也就是3A相关的数据信息。而Camera1.0中各种Processor模块，即将数据打包处理后再返回到应用的模块都已经不存在，而Camera2.0中将由MediaRecorder、SurfaceView、ImageReader等来直接处理，总体来说效率更好。继续看initialize：

// Camera2ClientBase.cpp

template <typename TClientBase>

status\_t Camera2ClientBase<TClientBase>::initialize(CameraModule \*module) {

...

//调用Device的initialie方法

res = mDevice->initialize(module);

...

res = mDevice->setNotifyCallback(this);

return OK;

}

1

2

3

4

5

6

7

8

9

10

代码就是调用了Device的initialize方法，此处的Device是在Camera2ClientBase的构造函数中创建的：

// Camera2ClientBase.cpp

template <typename TClientBase>

Camera2ClientBase<TClientBase>::Camera2ClientBase(const sp<CameraService>& cameraService,

const sp<TCamCallbacks>& remoteCallback,const String16& clientPackageName,

int cameraId,int cameraFacing,int clientPid,uid\_t clientUid,int servicePid):

TClientBase(cameraService, remoteCallback, clientPackageName,cameraId, cameraFacing,

clientPid, clientUid, servicePid),mSharedCameraCallbacks(remoteCallback),

mDeviceVersion(cameraService->getDeviceVersion(cameraId))

{

...

mInitialClientPid = clientPid;

mDevice = CameraDeviceFactory::createDevice(cameraId);

...

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

目前Camera API是2.0，而Device的API已经是3.0+了，继续看CameraDeviceFactory的createDevice方法：

// CameraDeviceFactory.cpp

sp<CameraDeviceBase> CameraDeviceFactory::createDevice(int cameraId) {

sp<CameraService> svc = sService.promote();

...

int deviceVersion = svc->getDeviceVersion(cameraId, /\*facing\*/NULL);

sp<CameraDeviceBase> device;

switch (deviceVersion) {

case CAMERA\_DEVICE\_API\_VERSION\_2\_0:

case CAMERA\_DEVICE\_API\_VERSION\_2\_1:

device = new Camera2Device(cameraId);

break;

case CAMERA\_DEVICE\_API\_VERSION\_3\_0:

case CAMERA\_DEVICE\_API\_VERSION\_3\_1:

case CAMERA\_DEVICE\_API\_VERSION\_3\_2:

case CAMERA\_DEVICE\_API\_VERSION\_3\_3:

device = new Camera3Device(cameraId);

break;

default:

ALOGE("%s: Camera %d: Unknown HAL device version %d",

\_\_FUNCTION\_\_, cameraId, deviceVersion);

device = NULL;

break;

}

return device;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

很显然，它将会创建一个Camera3Device对象，所以，Device的initialize就是调用了Camera3Device的initialize方法：

// Camera3Device.cpp

status\_t Camera3Device::initialize(CameraModule \*module)

{

...

camera3\_device\_t \*device;

//打开Camera HAL层的Deivce

res = module->open(deviceName.string(),

reinterpret\_cast<hw\_device\_t\*\*>(&device));

...

//交叉检查Device的版本

if (device->common.version < CAMERA\_DEVICE\_API\_VERSION\_3\_0) {

SET\_ERR\_L("Could not open camera: "

"Camera device should be at least %x, reports %x instead",

CAMERA\_DEVICE\_API\_VERSION\_3\_0,

device->common.version);

device->common.close(&device->common);

return BAD\_VALUE;

}

...

//调用回调函数来进行初始化，即调用打开Device的initialize方法来进行初始化

res = device->ops->initialize(device, this);

...

//启动请求队列线程

mRequestThread = new RequestThread(this, mStatusTracker, device, aeLockAvailable);

res = mRequestThread->run(String8::format("C3Dev-%d-ReqQueue", mId).string());

if (res != OK) {

SET\_ERR\_L("Unable to start request queue thread: %s (%d)",

strerror(-res), res);

device->common.close(&device->common);

mRequestThread.clear();

return res;

}

...

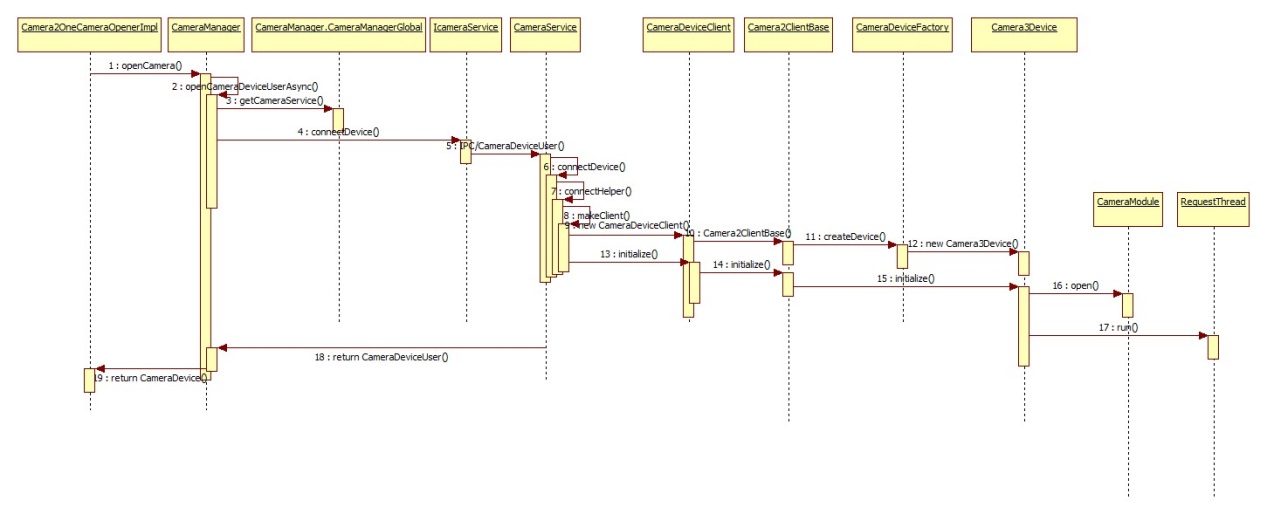
//返回初始成功

return OK;

}

1

首先，会依赖HAL框架打开并获得相应的Device对象，具体的流程请参考android6.0源码分析之Camera2 HAL分析，然后再回调此对象的initialize方法进行初始化，最后再启动RequestThread等线程，并返回initialize成功。至此Camera API2.0下的初始化过程就分析结束了。框架层的初始化时序图如下：



# Preview(预览)流程分析

<https://blog.csdn.net/yangzhihuiguming/article/details/51800587>

## Camera2 preview的应用层流程分析

preview流程都是从startPreview开始的，所以来看startPreview方法的代码：

### startPreview

//OneCameraImpl.java

@Override

public void startPreview(Surface previewSurface, CaptureReadyCallback listener) {

mPreviewSurface = previewSurface;

//根据Surface以及CaptureReadyCallback回调来建立preview环境

setupAsync(mPreviewSurface, listener);

}‘

这其中有一个比较重要的回调CaptureReadyCallback，先分析setupAsync方法：

### setupAsync

//OneCameraImpl.java

private void setupAsync(final Surface previewSurface, final CaptureReadyCallback listener) {

mCameraHandler.post(new Runnable() {

@Override

public void run() {

//建立preview环境

setup(previewSurface, listener);

}

});

}

1

这里通过CameraHandler来post一个Runnable对象，它只会调用Runnable的run方法，它仍然属于UI线程，并没有创建新的线程。所以，继续分析setup方法：

### setup

// OneCameraImpl.java

private void setup(Surface previewSurface, final CaptureReadyCallback listener) {

try {

if (mCaptureSession != null) {

mCaptureSession.abortCaptures();

mCaptureSession = null;

}

List<Surface> outputSurfaces = new ArrayList<Surface>(2);

outputSurfaces.add(previewSurface);

outputSurfaces.add(mCaptureImageReader.getSurface());

//创建CaptureSession会话来与Camera Device发送Preview请求

mDevice.createCaptureSession(outputSurfaces, new CameraCaptureSession.StateCallback() {

@Override

public void onConfigureFailed(CameraCaptureSession session) {

//如果配置失败，则回调CaptureReadyCallback的onSetupFailed方法

listener.onSetupFailed();

}

@Override

public void onConfigured(CameraCaptureSession session) {

mCaptureSession = session;

mAFRegions = ZERO\_WEIGHT\_3A\_REGION;

mAERegions = ZERO\_WEIGHT\_3A\_REGION;

mZoomValue = 1f;

mCropRegion = cropRegionForZoom(mZoomValue);

//调用repeatingPreview来启动preview

boolean success = repeatingPreview(null);

if (success) {

//若启动成功，则回调CaptureReadyCallback的onReadyForCapture，表示准备拍照成功

listener.onReadyForCapture();

} else {

//若启动失败，则回调CaptureReadyCallback的onSetupFailed，表示preview建立失败

listener.onSetupFailed();

}

}

@Override

public void onClosed(CameraCaptureSession session) {

super.onClosed(session);

}

}, mCameraHandler);

} catch (CameraAccessException ex) {

Log.e(TAG, "Could not set up capture session", ex);

listener.onSetupFailed();

}

}

6

47

首先，调用Device的createCaptureSession方法来创建一个会话，并定义了会话的状态回调CameraCaptureSession.StateCallback()，其中，当会话创建成功，则会回调onConfigured()方法,在其中，首先调用repeatingPreview来启动preview，然后处理preview的结果并调用先前定义的CaptureReadyCallback来通知用户进行Capture操作。先分析repeatingPreview方法：

### repeatingPreview

// OneCameraImpl.java

private boolean repeatingPreview(Object tag) {

try {

//通过CameraDevice对象创建一个CaptureRequest的preview请求

CaptureRequest.Builder builder = mDevice.createCaptureRequest(

CameraDevice.TEMPLATE\_PREVIEW);

//添加预览的目标Surface

builder.addTarget(mPreviewSurface);

//设置预览模式

builder.set(CaptureRequest.CONTROL\_MODE, CameraMetadata.CONTROL\_MODE\_AUTO);

addBaselineCaptureKeysToRequest(builder);

//利用会话发送请求，mCaptureCallback为

mCaptureSession.setRepeatingRequest(builder.build(), mCaptureCallback,mCameraHandler);

Log.v(TAG, String.format("Sent repeating Preview request, zoom = %.2f", mZoomValue));

return true;

} catch (CameraAccessException ex) {

Log.e(TAG, "Could not access camera setting up preview.", ex);

return false;

}

}

首先调用CameraDeviceImpl的createCaptureRequest方法创建类型为TEMPLATE\_PREVIEW 的CaptureRequest，然后调用CameraCaptureSessionImpl的setRepeatingRequest方法将此请求发送出去：

### setRepeatingRequest

//CameraCaptureSessionImpl.java

@Override

public synchronized int setRepeatingRequest(CaptureRequest request, CaptureCallback callback,

Handler handler) throws CameraAccessException {

if (request == null) {

throw new IllegalArgumentException("request must not be null");

} else if (request.isReprocess()) {

throw new IllegalArgumentException("repeating reprocess requests are not supported");

}

checkNotClosed();

handler = checkHandler(handler, callback);

...

//将此请求添加到待处理的序列里

return addPendingSequence(mDeviceImpl.setRepeatingRequest(request,createCaptureCallbackProxy(

handler, callback), mDeviceHandler));

}

### CaptureReadyCallback

至此应用层的preview的请求流程分析结束，继续分析其结果处理，如果preview开启成功，则会回调CaptureReadyCallback的onReadyForCapture方法，现在分析CaptureReadyCallback回调：

//CaptureModule.java

new CaptureReadyCallback() {

@Override

public void onSetupFailed() {

mCameraOpenCloseLock.release();

Log.e(TAG, "Could not set up preview.");

mMainThread.execute(new Runnable() {

@Override

public void run() {

if (mCamera == null) {

Log.d(TAG, "Camera closed, aborting.");

return;

}

mCamera.close();

mCamera = null;

}

});

}

@Override

public void onReadyForCapture() {

mCameraOpenCloseLock.release();

mMainThread.execute(new Runnable() {

@Override

public void run() {

Log.d(TAG, "Ready for capture.");

if (mCamera == null) {

Log.d(TAG, "Camera closed, aborting.");

return;

}

//

onPreviewStarted();

onReadyStateChanged(true);

mCamera.setReadyStateChangedListener(CaptureModule.this);

mUI.initializeZoom(mCamera.getMaxZoom());

mCamera.setFocusStateListener(CaptureModule.this);

}

});

}

}

### onReadyForCapture

根据前面的分析，预览成功后会回调onReadyForCapture方法，它主要是通知主线程的状态改变，并设置Camera的ReadyStateChangedListener的监听，其回调方法如下：

//CaptureModule.java

@Override

public void onReadyStateChanged(boolean readyForCapture) {

if (readyForCapture) {

mAppController.getCameraAppUI().enableModeOptions();

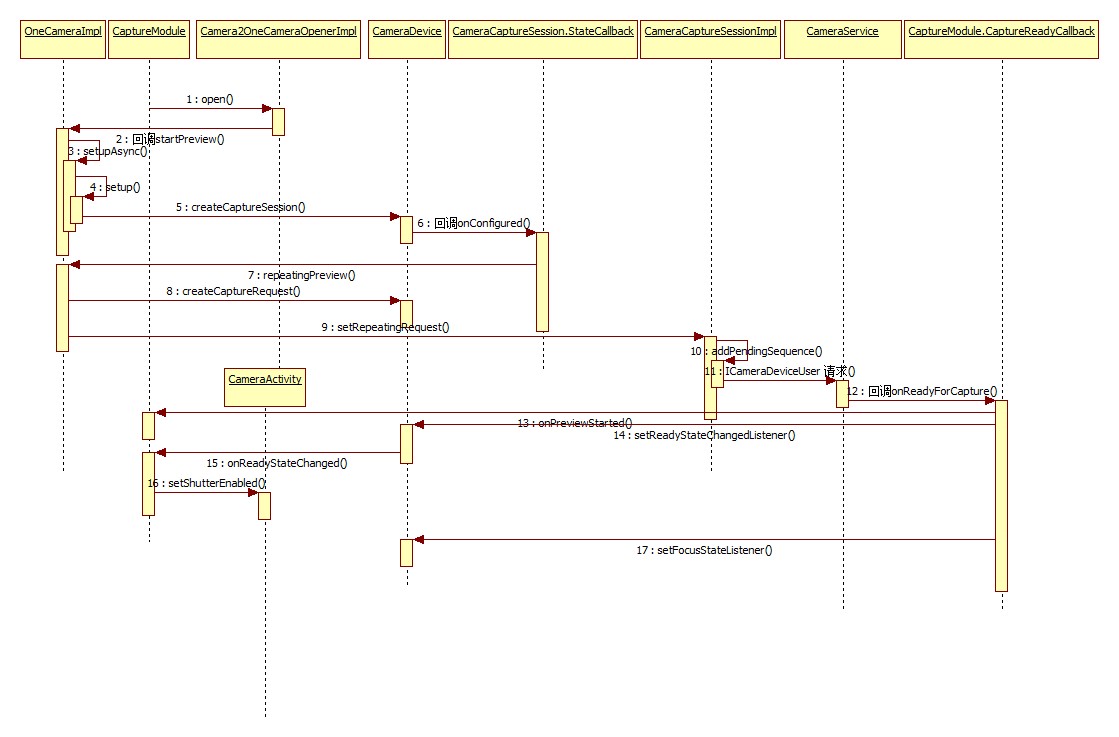
}

mAppController.setShutterEnabled(readyForCapture);

}

1

如代码所示，当其状态变成准备好拍照，则将会调用CameraActivity的setShutterEnabled方法，即使能快门按键，此时也就是说预览成功结束，可以按快门进行拍照了，所以，到这里，应用层的preview的流程基本分析完毕，下图是应用层的关键调用的流程时序图：



## Native层流程分析

分析Preview的Native的代码真是费了九牛二虎之力，若有分析不正确之处，请各位大神指正，在第一小节的后段最后会调用CameraDeviceImpl的setRepeatingRequest方法来提交请求，而在android6.0源码分析之Camera API2.0简介中，分析了Camera2框架Java IPC通信使用了CameraDeviceUser来进行通信，所以看Native层的ICameraDeviceUser的onTransact方法来处理请求的提交：

### onTransact

//ICameraDeviceUser.cpp

status\_t BnCameraDeviceUser::onTransact(uint32\_t code, const Parcel& data, Parcel\* reply,

uint32\_t flags){

switch(code) {

…

//请求提交

case SUBMIT\_REQUEST: {

CHECK\_INTERFACE(ICameraDeviceUser, data, reply);

// arg0 = request

sp<CaptureRequest> request;

if (data.readInt32() != 0) {

request = new CaptureRequest();

request->readFromParcel(const\_cast<Parcel\*>(&data));

}

// arg1 = streaming (bool)

bool repeating = data.readInt32();

// return code: requestId (int32)

reply->writeNoException();

int64\_t lastFrameNumber = -1;

//将实现BnCameraDeviceUser的对下岗的submitRequest方法代码写入Binder

reply->writeInt32(submitRequest(request, repeating, &lastFrameNumber));

reply->writeInt32(1);

reply->writeInt64(lastFrameNumber);

return NO\_ERROR;

} break;

...

}

CameraDeviceClientBase继承了BnCameraDeviceUser类，所以CameraDeviceClientBase相当于IPC Binder中的client，所以会调用其submitRequest方法，此处，至于IPC Binder通信原理不做分析，其参照其它资料：

### submitRequest

//CameraDeviceClient.cpp

status\_t CameraDeviceClient::submitRequest(sp<CaptureRequest> request,bool streaming,

/\*out\*/int64\_t\* lastFrameNumber) {

List<sp<CaptureRequest> > requestList;

requestList.push\_back(request);

return submitRequestList(requestList, streaming, lastFrameNumber);

}

简单的调用，继续分析submitRequestList：

// CameraDeviceClient

status\_t CameraDeviceClient::submitRequestList(List<sp<CaptureRequest> > requests,bool streaming,

int64\_t\* lastFrameNumber) {

...

//Metadata链表

List<const CameraMetadata> metadataRequestList;

...

for (List<sp<CaptureRequest> >::iterator it = requests.begin(); it != requests.end(); ++it) {

sp<CaptureRequest> request = \*it;

...

//初始化Metadata数据

CameraMetadata metadata(request->mMetadata);

...

//设置Stream的容量

Vector<int32\_t> outputStreamIds;

outputStreamIds.setCapacity(request->mSurfaceList.size());

//循环初始化Surface

for (size\_t i = 0; i < request->mSurfaceList.size(); ++i) {

sp<Surface> surface = request->mSurfaceList[i];

if (surface == 0) continue;

sp<IGraphicBufferProducer> gbp = surface->getIGraphicBufferProducer();

int idx = mStreamMap.indexOfKey(IInterface::asBinder(gbp));

...

int streamId = mStreamMap.valueAt(idx);

outputStreamIds.push\_back(streamId);

}

//更新数据

metadata.update(ANDROID\_REQUEST\_OUTPUT\_STREAMS, &outputStreamIds[0],

outputStreamIds.size());

if (request->mIsReprocess) {

metadata.update(ANDROID\_REQUEST\_INPUT\_STREAMS, &mInputStream.id, 1);

}

metadata.update(ANDROID\_REQUEST\_ID, &requestId, /\*size\*/1);

loopCounter++; // loopCounter starts from 1

//压栈

metadataRequestList.push\_back(metadata);

}

mRequestIdCounter++;

if (streaming) {

//预览会走此条通道

res = mDevice->setStreamingRequestList(metadataRequestList, lastFrameNumber);

if (res != OK) {

...

} else {

mStreamingRequestList.push\_back(requestId);

}

} else {

//Capture等走此条通道

res = mDevice->captureList(metadataRequestList, lastFrameNumber);

if (res != OK) {

...

}

}

if (res == OK) {

return requestId;

}

return res;

}

setStreamingRequestList和captureList方法都调用了submitRequestsHelper方法，只是他们的repeating参数一个ture,一个为false，而本节分析的preview调用的是setStreamingRequestList方法，并且API2.0下Device的实现为Camera3Device，所以看它的submitRequestsHelper实现：

// Camera3Device.cpp

status\_t Camera3Device::submitRequestsHelper(const List<const CameraMetadata> &requests,

bool repeating,/\*out\*/int64\_t \*lastFrameNumber) {

...

RequestList requestList;

//在这里面会进行CaptureRequest的创建，并调用configureStreamLocked进行stream的配置，主要是设置了一个回调captureResultCb，即后面要分析的重要的回调

res = convertMetadataListToRequestListLocked(requests, /\*out\*/&requestList);

...

if (repeating) {

//眼熟不，这个方法名和应用层中CameraDevice的setRepeatingRequests一样

res = mRequestThread->setRepeatingRequests(requestList, lastFrameNumber);

} else {

//不需重复，即repeating为false时，调用此方法来讲请求提交

res = mRequestThread->queueRequestList(requestList, lastFrameNumber);

}

...

return res;

}

从代码可知，在Camera3Device里创建了要给RequestThread线程，调用它的setRepeatingRequests或者queueRequestList方法来将应用层发送过来的Request提交，继续看setRepeatingRequests方法：

### setRepeatingRequests

// Camera3Device.cpp

status\_t Camera3Device::RequestThread::setRepeatingRequests(const RequestList &requests,

/\*out\*/int64\_t \*lastFrameNumber) {

Mutex::Autolock l(mRequestLock);

if (lastFrameNumber != NULL) {

\*lastFrameNumber = mRepeatingLastFrameNumber;

}

mRepeatingRequests.clear();

//将其插入mRepeatingRequest链表

mRepeatingRequests.insert(mRepeatingRequests.begin(),

requests.begin(), requests.end());

unpauseForNewRequests();

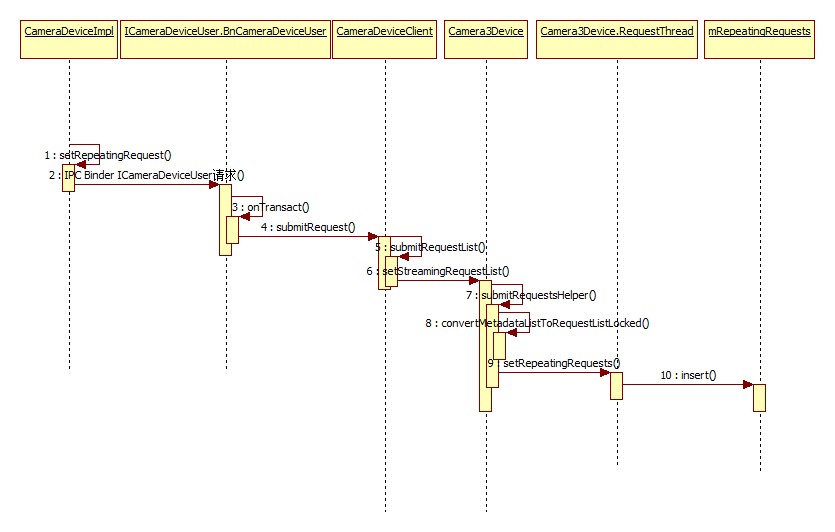
mRepeatingLastFrameNumber = NO\_IN\_FLIGHT\_REPEATING\_FRAMES;

return OK;

}

至此，Native层的preview过程基本分析结束，下面的工作将会交给Camera HAL层来处理，先给出Native层的调用时序图

### 时序图：



## preview的CameraHAL层流程分析

本节将不再对Camera的HAL层的初始化以及相关配置进行分析，只对preview等相关流程中的frame metadata的处理流程进行分析，具体的CameraHAL分析请参考android6.0源码分析之Camera2 HAL分析.在第二小节的submitRequestsHelper方法中调用convertMetadataListToRequestListLocked的时候会进行CaptureRequest的创建，并调用configureStreamLocked进行stream的配置，主要是设置了一个回调captureResultCb，所以Native层在request提交后，会回调此captureResultCb方法，首先分析captureResultCb：

### captureResultCb

// QCamera3HWI.cpp

void QCamera3HardwareInterface::captureResultCb(mm\_camera\_super\_buf\_t \*metadata\_buf,

camera3\_stream\_buffer\_t \*buffer, uint32\_t frame\_number)

{

if (metadata\_buf) {

if (mBatchSize) {

//批处理模式，但代码也是循环调用handleMetadataWithLock方法

handleBatchMetadata(metadata\_buf, true /\* free\_and\_bufdone\_meta\_buf \*/);

} else { /\* mBatchSize = 0 \*/

pthread\_mutex\_lock(&mMutex);

//处理元数据

handleMetadataWithLock(metadata\_buf, true /\* free\_and\_bufdone\_meta\_buf \*/);

pthread\_mutex\_unlock(&mMutex);

}

} else {

pthread\_mutex\_lock(&mMutex);

handleBufferWithLock(buffer, frame\_number);

pthread\_mutex\_unlock(&mMutex);

}

return;

}

一种是通过循环来进行元数据的批处理，另一种是直接进行元数据的处理，但是批处理最终也是循环调用handleMetadataWithLock来处理：

### handleMetadataWithLock

sdf

// QCamera3HWI.cpp

void QCamera3HardwareInterface::handleMetadataWithLock(mm\_camera\_super\_buf\_t \*metadata\_buf,

bool free\_and\_bufdone\_meta\_buf){

...

//Partial result on process\_capture\_result for timestamp

if (urgent\_frame\_number\_valid) {

...

for (List<PendingRequestInfo>::iterator i =mPendingRequestsList.begin();

i != mPendingRequestsList.end(); i++) {

...

if (i->frame\_number == urgent\_frame\_number &&i->bUrgentReceived == 0) {

camera3\_capture\_result\_t result;

memset(&result, 0, sizeof(camera3\_capture\_result\_t));

i->partial\_result\_cnt++;

i->bUrgentReceived = 1;

//提取3A数据

result.result =translateCbUrgentMetadataToResultMetadata(metadata);

...

//对Capture Result进行处理

mCallbackOps->process\_capture\_result(mCallbackOps, &result);

//释放camera\_metadata\_t

free\_camera\_metadata((camera\_metadata\_t \*)result.result);

break;

}

}

}

...

for (List<PendingRequestInfo>::iterator i = mPendingRequestsList.begin();

i != mPendingRequestsList.end() && i->frame\_number <= frame\_number;) {

camera3\_capture\_result\_t result;

memset(&result, 0, sizeof(camera3\_capture\_result\_t));

...

if (i->frame\_number < frame\_number) {

//清空数据结构

camera3\_notify\_msg\_t notify\_msg;

memset(&notify\_msg, 0, sizeof(camera3\_notify\_msg\_t));

//定义消息类型

notify\_msg.type = CAMERA3\_MSG\_SHUTTER;

notify\_msg.message.shutter.frame\_number = i->frame\_number;

notify\_msg.message.shutter.timestamp = (uint64\_t)capture\_time (urgent\_frame\_number -

i->frame\_number) \* NSEC\_PER\_33MSEC;

//调用回调通知应用层发生CAMERA3\_MSG\_SHUTTER消息

mCallbackOps->notify(mCallbackOps, &notify\_msg);

...

CameraMetadata dummyMetadata;

//更新元数据

dummyMetadata.update(ANDROID\_SENSOR\_TIMESTAMP,

&i->timestamp, 1);

dummyMetadata.update(ANDROID\_REQUEST\_ID,

&(i->request\_id), 1);

//得到元数据释放结果

result.result = dummyMetadata.release();

} else {

camera3\_notify\_msg\_t notify\_msg;

memset(&notify\_msg, 0, sizeof(camera3\_notify\_msg\_t));

// Send shutter notify to frameworks

notify\_msg.type = CAMERA3\_MSG\_SHUTTER;

...

//从HAL中获得Metadata

result.result = translateFromHalMetadata(metadata,

i->timestamp, i->request\_id, i->jpegMetadata, i->pipeline\_depth,

i->capture\_intent);

saveExifParams(metadata);

if (i->blob\_request) {

...

if (enabled && metadata->is\_tuning\_params\_valid) {

//将Metadata复制到文件

dumpMetadataToFile(metadata->tuning\_params, mMetaFrameCount, enabled,

"Snapshot",frame\_number);

}

mPictureChannel->queueReprocMetadata(metadata\_buf);

} else {

// Return metadata buffer

if (free\_and\_bufdone\_meta\_buf) {

mMetadataChannel->bufDone(metadata\_buf);

free(metadata\_buf);

}

}

}

...

}

}

其中，首先会调用回调的process\_capture\_result方法来对Capture Result进行处理，然后会调用回调的notify方法来发送一个CAMERA3\_MSG\_SHUTTER消息，而process\_capture\_result所对应的实现其实就是Camera3Device的processCaptureResult方法，先分析processCaptureResult：

//Camera3Device.cpp

void Camera3Device::processCaptureResult(const camera3\_capture\_result \*result) {

...

//对于HAL3.2+,如果HAL不支持partial，当metadata被包含在result中时，它必须将partial\_result设置为1

...

{

Mutex::Autolock l(mInFlightLock);

ssize\_t idx = mInFlightMap.indexOfKey(frameNumber);

...

InFlightRequest &request = mInFlightMap.editValueAt(idx);

if (result->partial\_result != 0)

request.resultExtras.partialResultCount = result->partial\_result;

// 检查结果是否只有partial metadata

if (mUsePartialResult && result->result != NULL) {

if (mDeviceVersion >= CAMERA\_DEVICE\_API\_VERSION\_3\_2) {//HAL版本高于3.2

if (result->partial\_result > mNumPartialResults || result->partial\_result < 1) {

//Log显示错误

return;

}

isPartialResult = (result->partial\_result < mNumPartialResults);

if (isPartialResult) {

//将结果加入到请求的结果集中

request.partialResult.collectedResult.append(result->result);

}

} else {//低于3.2

...

}

if (isPartialResult) {

// Fire off a 3A-only result if possible

if (!request.partialResult.haveSent3A) {

request.partialResult.haveSent3A =processPartial3AResult(frameNumber,

request.partialResult.collectedResult,request.resultExtras);

}

}

}

...

if (result->result != NULL && !isPartialResult) {

if (shutterTimestamp == 0) {

request.pendingMetadata = result->result;

request.partialResult.collectedResult = collectedPartialResult;

} else {

CameraMetadata metadata;

metadata = result->result;

//发送Capture Result

sendCaptureResult(metadata, request.resultExtras, collectedPartialResult,

frameNumber, hasInputBufferInRequest,request.aeTriggerCancelOverride);

}

}

//结果处理好了，将请求移除

removeInFlightRequestIfReadyLocked(idx);

} // scope for mInFlightLock

...

}

由代码可知，它会处理局部的或者全部的metadata数据，最后如果result不为空，且得到的是请求处理的全部数据，则会调用sendCaptureResult方法来将请求结果发送出去：

//Camera3Device.cpp

void Camera3Device::sendCaptureResult(CameraMetadata &pendingMetadata,CaptureResultExtras

&resultExtras,CameraMetadata &collectedPartialResult,uint32\_t frameNumber,bool reprocess,

const AeTriggerCancelOverride\_t &aeTriggerCancelOverride) {

if (pendingMetadata.isEmpty())//如果数据为空，直接返回

return;

...

CaptureResult captureResult;

captureResult.mResultExtras = resultExtras;

captureResult.mMetadata = pendingMetadata;

//更新metadata

if (captureResult.mMetadata.update(ANDROID\_REQUEST\_FRAME\_COUNT(int32\_t\*)&frameNumber, 1)

!= OK) {

SET\_ERR("Failed to set frame# in metadata (%d)",frameNumber);

return;

} else {

...

}

// Append any previous partials to form a complete result

if (mUsePartialResult && !collectedPartialResult.isEmpty()) {

captureResult.mMetadata.append(collectedPartialResult);

}

//排序

captureResult.mMetadata.sort();

// Check that there's a timestamp in the result metadata

camera\_metadata\_entry entry = captureResult.mMetadata.find(ANDROID\_SENSOR\_TIMESTAMP);

...

overrideResultForPrecaptureCancel(&captureResult.mMetadata, aeTriggerCancelOverride);

// 有效的结果，将其插入Buffer

List<CaptureResult>::iterator queuedResult =mResultQueue.insert(mResultQueue.end(),

CaptureResult(captureResult));

...

mResultSignal.signal();

}

最后，它将Capture Result插入了结果队列，并释放了结果的信号量，所以到这里，Capture Result处理成功，下面分析前面的notify发送CAMERA3\_MSG\_SHUTTER消息：

//Camera3Device.cpp

void Camera3Device::notify(const camera3\_notify\_msg \*msg) {

NotificationListener \*listener;

{

Mutex::Autolock l(mOutputLock);

listener = mListener;

}

...

switch (msg->type) {

case CAMERA3\_MSG\_ERROR: {

notifyError(msg->message.error, listener);

break;

}

case CAMERA3\_MSG\_SHUTTER: {

notifyShutter(msg->message.shutter, listener);

break;

}

default:

SET\_ERR("Unknown notify message from HAL: %d",

msg->type);

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

它调用了notifyShutter方法：

// Camera3Device.cpp

void Camera3Device::notifyShutter(const camera3\_shutter\_msg\_t &msg,

NotificationListener \*listener) {

...

// Set timestamp for the request in the in-flight tracking

// and get the request ID to send upstream

{

Mutex::Autolock l(mInFlightLock);

idx = mInFlightMap.indexOfKey(msg.frame\_number);

if (idx >= 0) {

InFlightRequest &r = mInFlightMap.editValueAt(idx);

// Call listener, if any

if (listener != NULL) {

//调用监听的notifyShutter法国法

listener->notifyShutter(r.resultExtras, msg.timestamp);

}

...

//将待处理的result发送到Buffer

sendCaptureResult(r.pendingMetadata, r.resultExtras,

r.partialResult.collectedResult, msg.frame\_number,

r.hasInputBuffer, r.aeTriggerCancelOverride);

returnOutputBuffers(r.pendingOutputBuffers.array(),

r.pendingOutputBuffers.size(), r.shutterTimestamp);

r.pendingOutputBuffers.clear();

removeInFlightRequestIfReadyLocked(idx);

}

}

...

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

首先它会通知listener preview成功，最后会调用sendCaptureResult将结果加入到结果队列。它会调用listener的notifyShutter方法，此处的listener其实是CameraDeviceClient类，所以会调用CameraDeviceClient类的notifyShutter方法：

//CameraDeviceClient.cpp

void CameraDeviceClient::notifyShutter(const CaptureResultExtras& resultExtras,nsecs\_t timestamp) {

// Thread safe. Don't bother locking.

sp<ICameraDeviceCallbacks> remoteCb = getRemoteCallback();

if (remoteCb != 0) {

//调用应用层的回调(CaptureCallback的onCaptureStarted方法)

remoteCb->onCaptureStarted(resultExtras, timestamp);

}

}

1

2

3

4

5

6

7

8

9

此处的ICameraDeviceCallbacks对应的是Java层的CameraDeviceImpl.java中的内部类CameraDeviceCallbacks，所以会调用它的onCaptureStarted方法：

//CameraDeviceImpl.java

@Override

public void onCaptureStarted(final CaptureResultExtras resultExtras, final long timestamp) {

int requestId = resultExtras.getRequestId();

final long frameNumber = resultExtras.getFrameNumber();

final CaptureCallbackHolder holder;

synchronized(mInterfaceLock) {

if (mRemoteDevice == null) return; // Camera already closed

// Get the callback for this frame ID, if there is one

holder = CameraDeviceImpl.this.mCaptureCallbackMap.get(requestId);

...

// Dispatch capture start notice

holder.getHandler().post(new Runnable() {

@Override

public void run() {

if (!CameraDeviceImpl.this.isClosed()) {

holder.getCallback().onCaptureStarted(CameraDeviceImpl.this,holder.getRequest(

resultExtras.getSubsequenceId()),timestamp, frameNumber);

}

}

});

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

它会调用OneCameraImpl.java中的mCaptureCallback的onCaptureStarted方法：

//OneCameraImpl.java

//Common listener for preview frame metadata.

private final CameraCaptureSession.CaptureCallback mCaptureCallback =

new CameraCaptureSession.CaptureCallback() {

@Override

public void onCaptureStarted(CameraCaptureSession session,CaptureRequest request,

long timestamp,long frameNumber) {

if (request.getTag() == RequestTag.CAPTURE&& mLastPictureCallback != null) {

mLastPictureCallback.onQuickExpose();

}

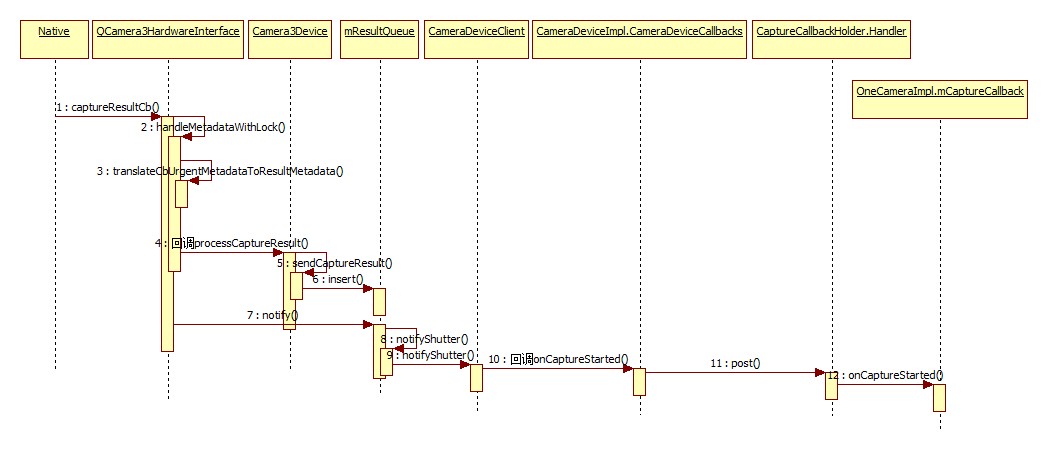
}

…

}

注意：Capture,preview以及autoFocus都是使用的这个回调，而Capture调用的时候，其RequestTag为CAPTURE，而autoFocus的时候为TAP\_TO\_FOCUS,而preview请求时没有对RequestTag进行设置，所以回调到onCaptureStarted方法时，不需要进行处理，但是到此时，preview已经启动成功，可以进行预览了，其数据都在buffer里。所以到此时，preview的流程全部分析结束，下面给出HAL层上的流程

### 时序图



# Capture流程分析

https://blog.csdn.net/yangzhihuiguming/article/details/51822384

# video流程分析

[无。。。](https://blog.csdn.net/yangzhihuiguming/article/details/51837084)

# Camera API2.0的应用

无。。。

https://blog.csdn.net/wangjicong\_215/article/details/73278757

# 案例

## [Android-Camera2Video的demo源码和调试心得](https://www.cnblogs.com/lihaiping/p/6142512.html)

<https://www.cnblogs.com/lihaiping/p/6142512.html>

mPreviewSize = chooseOptimalSize(map.getOutputSizes(SurfaceTexture.class),

1280, 720, cameraLargest);

# REF

[Android5.1中surface和CpuConsumer下生产者和消费者间的处理框架简述](https://blog.csdn.net/gzzaigcnforever/article/details/49025297)