Android Camera HAL3中预览preview模式下的数据流

前沿：

为了更好的梳理preview下buffer数据流的操作过程，前一文中对surface下的buffer相关的操作**架构**进行了描述。本文主要以此为基础，重点分析再Camera2Client和Camera3Device下是如何维护并读写这些视频帧缓存的。

1. Camera3Device::convertMetadataListToRequestListLocked函数

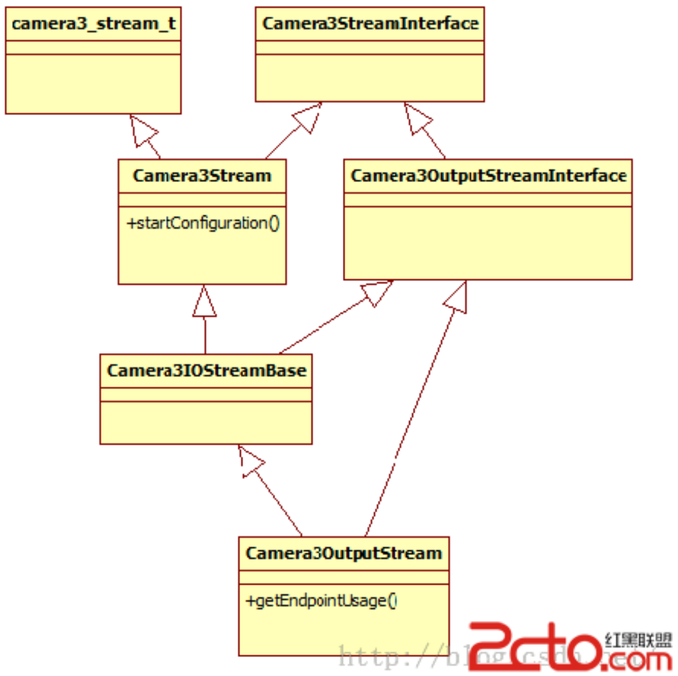
结合上一博文中关于preview的控制流，定位到数据流主要的操作主要是对preview模式下将CameraMetadata mPreviewRequest转换为CaptureRequest的过程之中，回顾到mPreviewRequest是主要包含了当前preview下所需要Camera3Device来操作的OutputStream的index值。

2. Camera3Device::configureStreamsLocked函数

在configureStreamsLocked的函数中，主要关注的是Camera3Device对当前所具有的所有的mInputStreams和mOutputStreams进行Config的操作，分别包括startConfiguration/finishConfiguration两个状态。

2.1 mOutputStreams.editValueAt(i)->startConfiguration()

这里的遍历所有输出stream即最终调用的函数入口为Camera3Stream::startConfiguration()，这里需要先看下Camera3OutputStream的整个结构，出现了Camera3Stream和Camera3IOStreamBase，两者是Input和Output stream所共有的，前者提供的更多的是对buffer的config、get/retrun buffer的操作，后者以维护当前的stream所拥有的buffer数目。另一个支路camera3_stream_t是一个和Camera HAL3底层进行stream信息交互的入口。



startConfiguration函数首先是判断当前stream的状态，对于已经config的不作处理，config的主要操作是getEndpointUsage：

```
status_t Camera3OutputStream::getEndpointUsage(uint32_t *usage) {
    status_t res;
    int32_t u = 0;
    res = mConsumer->query(mConsumer.get(),
            NATIVE_WINDOW_CONSUMER_USAGE_BITS, &u);
    *usage = u;
    return res;
}
```

这里的mConsumer其实就是之前创建的Surface的本体，每一个Stream在建立时createStream，都会传入一个ANativeWIndow类似的Consumer绑定到当前的stream中去。这里主要是完成当前window所管理的buffer的USAGE值，可参看grallo.h中的定义，由Gralloc模块负责指定当前buffer操作是由HW还是SW来完成以及不同的应用场合，在Gralloc模块中不同模块需求的buffer亦会有不同的分配、定义与处理方式：

```
/* buffer will be used as an OpenGL ES texture */
GRALLOC_USAGE_HW_TEXTURE = 0x00000100,
```

```
/* buffer will be used as an OpenGL ES render target */
GRALLOC_USAGE_HW_RENDER = 0x00000200,
/* buffer will be used by the 2D hardware blitter */
GRALLOC_USAGE_HW_2D = 0x00000400,
/* buffer will be used by the HWComposer HAL module */
GRALLOC_USAGE_HW_COMPOSER = 0x00000800,
/* buffer will be used with the framebuffer device */
GRALLOC_USAGE_HW_FB = 0x00001000,
/* buffer will be used with the HW video encoder */
GRALLOC_USAGE_HW_VIDEO_ENCODER = 0x00010000,
/* buffer will be written by the HW camera pipeline */
GRALLOC_USAGE_HW_CAMERA_WRITE = 0x00020000,
/* buffer will be read by the HW camera pipeline */
GRALLOC_USAGE_HW_CAMERA_READ = 0x00040000,
/* buffer will be used as part of zero-shutter-lag queue */
GRALLOC_USAGE_HW_CAMERA_ZSL = 0x00060000,
/* mask for the camera access values */
GRALLOC_USAGE_HW_CAMERA_MASK = 0x00060000,
/* mask for the software usage bit-mask */
GRALLOC_USAGE_HW_MASK = 0x00071F00,
```

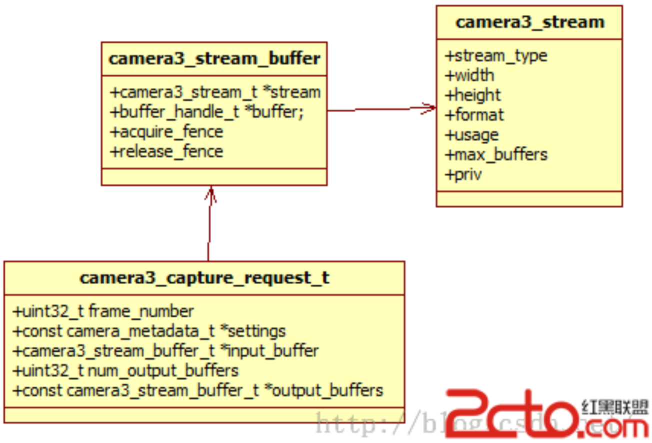
2.2 mHal3Device->ops->configure_streams(mHal3Device, &config);

config是一个camera3_stream_configuration**数据结构**，他记录了一次和HAL3交互的stream的数量，已经当前每一个stream的属性配置相关的信息camer3_stream_t，包括stream中每一个buffer的属性值，stream的类型值等等，提交这些信息供hal3去分析处理。在高通平台中你可以看到，对于每一个stream在HAL3平台下均以一个Channel的形式存在。

?

```
typedef struct camera3_stream_configuration {
    uint32_t num_streams;
    camera3_stream_t **streams;
```

```
}
camera3_stream_configuration_t;
```



stream_type包括：CAMERA3_STREAM_OUTPUT、CAMERA3_STREAM_INPUT、CAMERA3_STREAM_BIDIRECTIONAL。

format主要是指当前buffer支持的像素点存储格式，以HAL_PIXEL_FORMAT_IMPLEMENTATION_DEFINED居多，表明数据格式是由Gralloc模块来决定的。

对于HAL3中对configureStreams接口的实现会放在后续介绍高通平台的实现机制时再做分析。

2.3 Camera3Stream::finishConfiguration

该函数主要执行configureQueueLocked和registerBuffersLocked两个函数

```
status_t Camera3OutputStream::configureQueueLocked() {
    status_t res;

    mTraceFirstBuffer = true;
    if ((res = Camera3IOStreamBase::configureQueueLocked()) != OK) {
        return res;
    }

    ALOG_ASSERT(mConsumer != 0, mConsumer should never be NULL);

    // Configure consumer-side ANativeWindow interface
    res = native_window_api_connect(mConsumer.get(),
```

```
            NATIVE_WINDOW_API_CAMERA);
    if (res != OK) {
        ALOGE(%s: Unable to connect to native window for stream %d,
                __FUNCTION__, mId);
        return res;
    }

    res = native_window_set_usage(mConsumer.get(), camera3_stream::usage);
    if (res != OK) {
        ALOGE(%s: Unable to configure usage %08x for stream %d,
                __FUNCTION__, camera3_stream::usage, mId);
        return res;
    }

    res = native_window_set_scaling_mode(mConsumer.get(),
            NATIVE_WINDOW_SCALING_MODE_SCALE_TO_WINDOW);
    if (res != OK) {
        ALOGE(%s: Unable to configure stream scaling: %s (%d),
                __FUNCTION__, strerror(-res), res);
        return res;
    }

    if (mMaxSize == 0) {
        // For buffers of known size
        res = native_window_set_buffers_dimensions(mConsumer.get(),
                camera3_stream::width, camera3_stream::height);
    }
```

```
else
{
        //
 For buffers with bounded size
        res
 = native_window_set_buffers_dimensions(mConsumer.get(),
                mMaxSize,
1);
    }
    if
(res != OK) {
        ALOGE(%s:
 Unable to configure stream buffer dimensions
                %d
 x %d (maxSize %zu) for
stream %d,
                __FUNCTION__,
 camera3_stream::width, camera3_stream::height,
                mMaxSize,
 mId);
        return
res;
    }
    res
 = native_window_set_buffers_format(mConsumer.get(),
            camera3_stream::format);
    if
(res != OK) {
        ALOGE(%s:
 Unable to configure stream buffer format %#x for
stream %d,
                __FUNCTION__,
 camera3_stream::format, mId);
        return
res;
    }

    int
maxConsumerBuffers;
    res
 = mConsumer->query(mConsumer.get(),
            NATIVE_WINDOW_MIN_UNDEQUEUED_BUFFERS,
 &maxConsumerBuffers);//支持的最大buffer数量
    if
(res != OK) {
        ALOGE(%s:
 Unable to query consumer undequeued
                buffer
 count for
stream %d, __FUNCTION__, mId);
```

```
        return
res;
    }

    ALOGV(%s:
 Consumer wants %d buffers, HAL wants %d, __FUNCTION__,
            maxConsumerBuffers,
 camera3_stream::max_buffers);
    if
(camera3_stream::max_buffers == 0)
 {
        ALOGE(%s:
 Camera HAL requested max_buffer count: %d, requires at least 1,
                __FUNCTION__,
 camera3_stream::max_buffers);
        return
INVALID_OPERATION;
    }

    mTotalBufferCount
 = maxConsumerBuffers + camera3_stream::max_buffers;//至少2个buffer
    mHandoutTotalBufferCount
 = 0;
    mFrameCount
 = 0;
    mLastTimestamp
 = 0;

    res
 = native_window_set_buffer_count(mConsumer.get(),
            mTotalBufferCount);
    if
(res != OK) {
        ALOGE(%s:
 Unable to set buffer count for
stream %d,
                __FUNCTION__,
 mId);
        return
res;
    }

    res
 = native_window_set_buffers_transform(mConsumer.get(),
            mTransform);
    if
(res != OK) {
        ALOGE(%s:
 Unable to configure stream transform to %x: %s (%d),
```

```
                    __FUNCTION__,
 mTransform, strerror(-res), res);
    }

    return
 OK;
 }
```

如果你对SurfaceFlinger的架构熟悉的话，该代码会相对比较好理解。本质是根据当前stream设置的buffer属性，将这些属性值通过ANativeWindow这个接口传递给Consumer侧去维护：

这里重点关注以下几个buffer的相关属性信息：

比如native_window_set_buffer_count是设置当前Window所需要的buffer数目：

总的当前stream下的buffer个数总数为mTotalBufferCount = maxConsumerBuffers + camera3_stream::max_buffers。其中camera3_stream::max_buffer需要的buffer总数由configureStreams时HAL3底层的Device来决定的，高通平台下定义的camera3_stream::max_buffer数为7个，而maxConsumerBuffers指的是在所有buffer被dequeue时还应该保留的处于queue操作的buffer个数，即全dequeue时至少有maxConsumerBuffers个buffer是处于queue状态在被Consumer使用的。通过query NATIVE_WINDOW_MIN_UNDEQUEUED_BUFFERS来完成，一般默认是1个，即每个stream可以认为需要由8个buffer缓存块组成，实际可dequeue的为8个。

比如native_window_set_buffers_transform一般是指定buffer的Consumer，即当前buffer显示的90/180/270°角度。

该过程本质是结合HAL3的底层buffer配置需求，反过来请求Buffer的Consumer端BufferQueueConsumer来设置相关的buffer属性。

registerBuffersLocked是一个比较重要的处理过程：

?

```
status_t
 Camera3Stream::registerBuffersLocked(camera3_device *hal3Device) {
    ATRACE_CALL();

    /**
     *
 >= CAMERA_DEVICE_API_VERSION_3_2:
     *
     *
 camera3_device_t->ops->register_stream_buffers() is not called anc
     *
 be NULL.
     */
    if
 (hal3Device->common.version >= CAMERA_DEVICE_API_VERSION_3_2) {
        ALOGV(%s:
 register_stream_buffers unused as of HAL3.2,
 __FUNCTION__);
```

```
        if
(hal3Device->ops->register_stream_buffers != NULL) {
            ALOGE(%s:
 register_stream_buffers is deprecated in HAL3.2;
                    must
 be set to NULL in camera3_device::ops, __FUNCTION__);
            return
INVALID_OPERATION;
        }
else
{
            ALOGD(%s:
 Skipping NULL check for
deprecated register_stream_buffers, __FUNCTION__);
        }

        return
OK;
    }
else
{
        ALOGV(%s:
 register_stream_buffers using deprecated code path, __FUNCTION__);
    }


    status_t
 res;


    size_t
 bufferCount = getBufferCountLocked();//获取buffer的数量,mTotalBuffe:


    Vector<buffer_handle_t*>
 buffers;

    buffers.insertAt(/*prototype_item*/NULL,
/*index*/0,
 bufferCount);


    camera3_stream_buffer_set
 bufferSet = camera3_stream_buffer_set();

    bufferSet.stream
 = this;//新的bufferset指向camera3_stream_t

    bufferSet.num_buffers
 = bufferCount;//当前stream下的buffer数

    bufferSet.buffers
 = buffers.editArray();


    Vector<camera3_stream_buffer_t>
 streamBuffers;

    streamBuffers.insertAt(camera3_stream_buffer_t(),
/*index*/0,
 bufferCount);
```

```
    // Register all buffers with the HAL. This means getting all the buff
    // from the stream, providing them to the HAL with the
    // register_stream_buffers() method, and then returning them back to
    // stream in the error state, since they won't have valid data.
    //
    // Only registered buffers can be sent to the HAL.

    uint32_t bufferIdx = 0;
    for (; bufferIdx < bufferCount; bufferIdx++) {
        res = getBufferLocked( &streamBuffers.editItemAt(bufferIdx) );//返回de
buffer出来的所有buffer
        if (res != OK) {
            ALOGE(%s: Unable to get buffer %d for registration with HAL,
                    __FUNCTION__, bufferIdx);
            // Skip registering, go straight to cleanup
            break;
        }

        sp<fence> fence = new Fence(streamBuffers[bufferIdx].acquire_fence);
        fence->waitForever(Camera3Stream::registerBuffers);//等待可

        buffers.editItemAt(bufferIdx) = streamBuffers[bufferIdx].buffer;//dequeue
buffer出来的buffer handle
    }
    if (bufferIdx == bufferCount) {
        // Got all buffers, register with HAL
        ALOGV(%s: Registering %zu buffers with camera HAL,
                __FUNCTION__, bufferCount);
        ATRACE_BEGIN(camera3->register_stream_buffers);
        res = hal3Device->ops->register_stream_buffers(hal3Device,
```

```
                          &bufferSet);//buffer绑定并register到hal层
                  ATRACE_END();
              }

              //
          Return all valid buffers to stream, in ERROR state to indicate
              //
          they weren't filled.
              for
         (size_t i = 0;
          i < bufferIdx; i++) {
                  streamBuffers.editItemAt(i).release_fence
          = -1;
                  streamBuffers.editItemAt(i).status
          = CAMERA3_BUFFER_STATUS_ERROR;
                  returnBufferLocked(streamBuffers[i],
         0);//register后进行queue
          buffer
              }

              return
         res;
         }
         </fence></camera3_stream_buffer_t></buffer_handle_t*>
```

a 可以明确看到CAMERA_DEVICE_API_VERSION_3_2的版本才支持这个Device ops接口

b getBufferCountLocked

获取当前stream下的允许的buffer总数

c camera3_stream_buffer_t、camera3_stream_buffer_set和buffer_handle_t

首先需要关注的结构是camera3_stream_buffer_t，用于描述每一个stream下的buffer自身的特性值，其中关键结构是buffer_handle_t值是每一个buffer在不同进程间共享的handle，此外acquire_fence和release_fence用来不同硬件模块对buffer读写时的同步。

camera3_stream_buffer_set是封装了当前stream下所有的buffer的信息：

?

```
typedef
 struct camera3_stream_buffer_set {
    /**
     *
 The stream handle for the stream these buffers belong to
     */
    camera3_stream_t
 *stream;
    /**
     *
```

```
     * The number of buffers in this stream. It is guaranteed to be at le
     * stream->max_buffers.
     */
    uint32_t num_buffers;
    /**
     * The array of gralloc buffer handles for this stream. If the stream
     * is set to HAL_PIXEL_FORMAT_IMPLEMENTATION_DEFINED, the camera HAL
     * should inspect the passed-in buffers to determine any platform-pri
     * pixel format information.
     */
    buffer_handle_t **buffers;

} camera3_stream_buffer_set_t;
```

三个变量分别保存stream的buffer个数，当前这个set集合所属的stream，以及他所包含的所有buffer的handle信息列表。

d getBufferLocked获取当前buffer

?

```
status_t Camera3OutputStream::getBufferLocked(camera3_stream_buffer *buffer
    ATRACE_CALL();
    status_t res;

    if ((res = getBufferPreconditionCheckLocked()) != OK) {
        return res;
    }

    ANativeWindowBuffer* anb;
    int fenceFd;

    /**
     * Release the lock briefly to avoid deadlock for below scenario:
     *
```

```
 Thread 1: StreamingProcessor::startStream -> Camera3Stream::isConf
     *
 This thread acquired StreamingProcessor lock and try to lock Camer
     *
 Thread 2: Camera3Stream::returnBuffer->StreamingProcessor::onFrame
     *
 This thread acquired Camera3Stream lock and bufferQueue lock, and
     *
 StreamingProcessor lock.
     *
 Thread 3: Camera3Stream::getBuffer(). This thread acquired Camera3
     *
 and try to lock bufferQueue lock.
     *
 Then there is circular locking dependency.
     */
    sp
 currentConsumer = mConsumer;
    mLock.unlock();


    res
 = currentConsumer->dequeueBuffer(currentConsumer.get(), &anb, &fen
    mLock.lock();
    if
(res != OK) {
        ALOGE(%s:
 Stream %d: Can't dequeue next output buffer: %s (%d),
                __FUNCTION__,
 mId, strerror(-res), res);
        return
res;
    }


    /**
     *
 FenceFD now owned by HAL except in case of error,
     *
 in which case we reassign it to acquire_fence
     */
    handoutBufferLocked(*buffer,
 &(anb->handle), /*acquireFence*/fenceFd,
                       /*releaseFence*/-1,
 CAMERA3_BUFFER_STATUS_OK, /*output*/true);


    return
OK;
}
</anativewindow>
```

该函数主要是从由ANativeWindow从Consumer端dequeue获取一个buffer，本质上这个过程中首次执行是会有Consumer端去分配一个由实际物理空间的给当前的一个buffer的。

接着执行handoutBufferLocked，填充camera3_stream_buffer这个结构体，其中设置的acquireFence为-1值表明hal3的这个buffer可被Framewrok直接使用，而acquireFence表示HAL3如何想使用这个buffer时需要等待其变为1，因为buffer分配和handler返回不一定是一致同步的。还会切换当前buffer的状态CAMERA3_BUFFER_STATUS_OK。

?

```
void
Camera3IOStreamBase::handoutBufferLocked(camera3_stream_buffer &buf
                                              buffer_handle_t *handle,
                                              int acquireFence,
                                              int releaseFence,
                                              camera3_buffer_status status,
                                              bool output) {
    /**
     * Note that all fences are now owned by HAL.
     */

    // Handing out a raw pointer to this object. Increment internal refcc
    incStrong(this);
    buffer.stream = this;
    buffer.buffer = handle;
    buffer.acquire_fence = acquireFence;
    buffer.release_fence = releaseFence;
    buffer.status = status;

    // Inform tracker about becoming busy
    if (mHandoutTotalBufferCount == 0 && mState != STATE_IN_CONFIG &&
            mState != STATE_IN_RECONFIG) {
        /**
         * Avoid a spurious IDLE->ACTIVE->IDLE transition when using buffers
         * before/after register_stream_buffers during initial configuration
```

```
         *
 or re-configuration.
         *
         *
 TODO: IN_CONFIG and IN_RECONFIG checks only make sense for
<hal3.2
statustracker="">
 statusTracker = mStatusTracker.promote();
        if
(statusTracker != 0)
 {
            statusTracker->markComponentActive(mStatusId);
        }
    }
    mHandoutTotalBufferCount++;//统计dequeuebuffer的数量

    if
(output) {
        mHandoutOutputBufferCount++;
    }
}</hal3.2>
```

e hal3Device->ops->register_stream_buffers(hal3Device,&bufferSet);//buffer绑定并register到hal层

将所属的stream下的所有buffer有关的信息，主要是每个buffer的buffer_handle_t值，交给HAL3层去实现。比如高通HAL3平台每一个Channel对应于Camera3Device端的stream，而每一个stream的buffer在不同的Channel下面却是一个个的stream，这是高通的实现方式。

f 在完成register所有buffer后，设置每一个buffer状态为从CAMERA3_BUFFER_STATUS_OK切换到CAMERA3_BUFFER_STATUS_ERROR表明这个buffer都是可用的，目的在于执行returnBufferLocked是为了将这些因为register而出列的所有buffer再次cancelbuffer操作。

Camera3OutputStream::returnBufferLocked->Camera3IOStreamBase::returnAnyBufferLocked->Camera3OutputStream::returnBufferCheckedLocked

?

```
status_t
 Camera3OutputStream::returnBufferCheckedLocked(//result返回时调用
            const
camera3_stream_buffer &buffer,
            nsecs_t
 timestamp,
            bool
 output,
            /*out*/
            sp<fence>
 *releaseFenceOut) {
```

```
    (void)output;
    ALOG_ASSERT(output,
 Expected output to be true);

    status_t
 res;
    sp<fence>
 releaseFence;

    /**
     *
 Fence management - calculate Release Fence
     */
    if
(buffer.status == CAMERA3_BUFFER_STATUS_ERROR) {
        if
(buffer.release_fence != -1)
 {
            ALOGE(%s:
 Stream %d: HAL should not set release_fence(%d) when
                there
 is an error, __FUNCTION__, mId, buffer.release_fence);
            close(buffer.release_fence);
        }

        /**
         *
 Reassign release fence as the acquire fence in case of error
         */
        releaseFence
 = new
Fence(buffer.acquire_fence);
    }
else
{
        res
 = native_window_set_buffers_timestamp(mConsumer.get(), timestamp);
        if
(res != OK) {
            ALOGE(%s:
 Stream %d: Error setting timestamp: %s (%d),
                  __FUNCTION__,
 mId, strerror(-res), res);
            return
res;
        }

        releaseFence
```

```
 = new
Fence(buffer.release_fence);
    }

    int
anwReleaseFence = releaseFence->dup();

    /**
     *
 Release the lock briefly to avoid deadlock with
     *
 StreamingProcessor::startStream -> Camera3Stream::isConfiguring (t
     *
 thread will go into StreamingProcessor::onFrameAvailable) during
     *
 queueBuffer
     */
    sp
 currentConsumer = mConsumer;
    mLock.unlock();

    /**
     *
 Return buffer back to ANativeWindow
     */
    if
(buffer.status == CAMERA3_BUFFER_STATUS_ERROR) {
        //
 Cancel buffer
        res
 = currentConsumer->cancelBuffer(currentConsumer.get(),
                container_of(buffer.buffer,
 ANativeWindowBuffer, handle),
                anwReleaseFence);//Register
 buffer locked所在的事情，cancelbuffer dequeue的buffer
        if
(res != OK) {
            ALOGE(%s:
 Stream %d: Error cancelling buffer to native
window:
                   %s
 (%d), __FUNCTION__, mId, strerror(-res), res);
        }
    }
else
{
        if
(mTraceFirstBuffer && (stream_type == CAMERA3_STREAM_OUTPUT)) {
            {
```

```
                char
        traceLog[48];
                snprintf(traceLog,
 sizeof(traceLog), Stream %d: first full buffer
,
 mId);
                ATRACE_NAME(traceLog);
            }
            mTraceFirstBuffer
 = false;
        }


        res
 = currentConsumer->queueBuffer(currentConsumer.get(),
                container_of(buffer.buffer,
 ANativeWindowBuffer, handle),
                anwReleaseFence);//queuebuffer，送显ANativeWindowBuf
        if
 (res != OK) {
            ALOGE(%s:
 Stream %d: Error queueing buffer to native
window:
                  %s
 (%d), __FUNCTION__, mId, strerror(-res), res);
        }
    }
    mLock.lock();
    if
 (res != OK) {
        close(anwReleaseFence);
    }


    *releaseFenceOut
 = releaseFence;


    return
res;
}
</anativewindow></fence></fence>
```

该函数对于首次register的处理来说，他处理的buffer均是CAMERA3_BUFFER_STATUS_ERROR，调用了cancelBuffer将所有buffer的状态都还原为free的状态，依次说明目前的buffer均是可用的，之前均不涉及到对buffer的数据流的操作。

3 buffer数据流的dequeue操作

上述步骤2主要是将每一个Stream下全部的buffer信息全部register到下层的HAL3中，为后续对buffer的数据流读

写操作奠定基础。

那么preview模式下我们又是如何去获得一帧完成的视频流的呢?

触发点就是preview模式下的Request，前面提到过一个mPreviewRequest至少包含一个StreamProcessor和一个CallbackProcessor的两路stream，每路stream拥有不同的buffer数量。比如要从HAL3获取一帧图像数据，最简单的思路就是从StreamProcessor下的Outputstream流中下发一个可用的buffer地址，然后HAL3填充下数据，Framework就可以拥有一帧数据了。

根据这个思路，回顾到前一博文中每次会不断的下发一个Request命令包到HAL3中，在这里我们就可以看到这个buffer地址身影。

Camera3Device::RequestThread::threadLoop() 下的部分代码：

```
outputBuffers.insertAt(camera3_stream_buffer_t(), 0,
        nextRequest->mOutputStreams.size());//Streamprocess,Callbac
request.output_buffers = outputBuffers.array();//camera3_stream_buffer_t
for (size_t i = 0; i < nextRequest->mOutputStreams.size(); i++) {
    res = nextRequest->mOutputStreams.editItemAt(i)->
            getBuffer(&outputBuffers.editItemAt(i));//等待获取buffer
    if (res != OK) {
        // Can't get output buffer from gralloc queue - this could be due to
        // abandoned queue or other consumer misbehavior, so not a fatal
        // error
        ALOGE(RequestThread: Can't get output buffer, skipping request:
                %s (%d), strerror(-res), res);
        Mutex::Autolock l(mRequestLock);
        if (mListener != NULL) {
            mListener->notifyError(
                    ICameraDeviceCallbacks::ERROR_CAMERA_REQUEST,
                    nextRequest->mResultExtras);
        }
        cleanUpFailedRequest(request, nextRequest, outputBuffers);
        return true;
    }
    request.num_output_buffers++;//一般一根OutStream对应一个buffer,故总
```

```
        }
```

在这个下发到HAL3的camera3_capture_request中，可以看到 const camera3_stream_buffer_t *output_buffers，下面的代码可以说明这一次的Request的output_buffers是打包了当前Camera3Device所拥有的mOutputStreams。

```
outputBuffers.insertAt(camera3_stream_buffer_t(), 0,
        nextRequest->mOutputStreams.size());//Streamprocess,Callback
```

对于每一个OutputStream他会给她分配一个buffer handle。关注下面的处理代码：

```
nextRequest->mOutputStreams.editItemAt(i)->
                getBuffer(&outputBuffers.editItemAt(i))
```

nextRequest->mOutputStreams.editItemAt(i)是获取一个Camera3OutputStream对象，然后对getBuffer而言传入的是这个Camera3OutputStream所对应的这次buffer的输入位置，这个camera3_stream_buffer是需要从Camera3OutputStream对象中去获取的。

```
status_t Camera3Stream::getBuffer(camera3_stream_buffer *buffer) {
    ATRACE_CALL();
    Mutex::Autolock l(mLock);
    status_t res = OK;

    // This function should be only called when the stream is configured
    if (mState != STATE_CONFIGURED) {
        ALOGE("%s: Stream %d: Can't get buffers if stream is not in CONFIGURED state %d",
                __FUNCTION__, mId, mState);
        return INVALID_OPERATION;
    }

    // Wait for new buffer returned back if we are running into the limit
    if (getHandoutOutputBufferCountLocked() == camera3_stream::max_buffers
        ALOGV("%s:
```

```
 Already dequeued max output buffers (%d), wait for
next returned one.,
                __FUNCTION__,
 camera3_stream::max_buffers);
        res
 = mOutputBufferReturnedSignal.waitRelative(mLock, kWaitForBufferDu
        if
(res != OK) {
            if
(res == TIMED_OUT) {
                ALOGE(%s:
 wait for
output buffer return
timed out after %lldms, __FUNCTION__,
                        kWaitForBufferDuration
 / 1000000LL);
            }
            return
res;
        }
    }


    res
 = getBufferLocked(buffer);
    if
(res == OK) {
        fireBufferListenersLocked(*buffer,
/*acquired*/true,
/*output*/true);
    }


    return
res;
}
```

上述的代码先是检查dequeue了的buffer是否已经达到本stream申请的buffer数目的最大值，如果已经全部dequeue的话就得wait到当前已经有buffer return并且queue操作后，在处理完成后才允许将从buffer队列中再次执行dequeue操作。

随后调用getBufferLocked通过2.2(d）小节可以知道是从buffer队列中获取一个可用的buffer，并填充这个camera3_stream_buffer值。

这样处理完的结果是，下发的Request包含所有模块下的outputstream，同时每个stream都配备了一个camera3_stream_buffer供底层HAL3.0去处理，而这个buffer在Camera3Device模式下，可以是交互的是帧图像数据，可以是参数控制命令，也可以是其他的3A信息，这些不同的信息一般归不同的模块管理，也就是不同的stream来处理。

4 buffer数据流的queue操作

dequeue出来的buffer信息已经随着Request下发到了HAL3层，在Camera3Device架构下，可以使用一个Callback接口将数据从HAL3回传到Camera所在的Framework层。Camera3Device私有继承了一个Callback接口camera3_callback_ops数据结构，分别预留了notify和process_capture_result。前者是用于回调一些shutter已经error等信息，后者以Callback数据流为主，这个回调接口通过device->initialize(camera3_device, this)来完成注册。

```
void
Camera3Device::sProcessCaptureResult(const
camera3_callback_ops *cb,
        const
camera3_capture_result *result) {
    Camera3Device
 *d =
            const_cast<camera3device*>(static_cast<const>(cb));
    d->processCaptureResult(result);
}
</const></camera3device*>
```

返回的buffer所有信息均包含在camera3_capture_result中，该函数的处理过程相对比较复杂，如果只定位queue buffer的入口可直接到returnOutputBuffers中去：

```
void
Camera3Device::returnOutputBuffers(
        const
camera3_stream_buffer_t *outputBuffers, size_t numBuffers,
        nsecs_t
 timestamp) {
    for
(size_t i = 0;
 i < numBuffers; i++)//对每一个buffer所属的stream进行分析
    {
        Camera3Stream
 *stream = Camera3Stream::cast(outputBuffers[i].stream);//该buffer对
        status_t
 res = stream->returnBuffer(outputBuffers[i], timestamp);//Camera3C
        //
 Note: stream may be deallocated at this point, if this buffer was
        //
 the last reference to it.
        if
(res != OK) {
            ALOGE(Can't
return
buffer to its stream: %s (%d),
                strerror(-res),
 res);
        }
```

```
        }
    }
```

因为在下发Request时，每一个buffer均包含所述的stream信息，当buffer数据返回到Framework层时，我们又可以转到Camera3OutPutStream来处理这个return的buffer。

```
status_t Camera3Stream::returnBuffer(const camera3_stream_buffer &buffer,
        nsecs_t timestamp) {
    ATRACE_CALL();
    Mutex::Autolock l(mLock);

    /**
     * TODO: Check that the state is valid first.
     *
     * <hal3.2 addition="" and="" configured.="" in="" in_config="" in_re
     *
     * Do this for getBuffer as well.
     */
    status_t res = returnBufferLocked(buffer, timestamp);//以queue buffer为主
    if (res == OK) {
        fireBufferListenersLocked(buffer, /*acquired*/false, /*output*/true);
        mOutputBufferReturnedSignal.signal();
    }

    return res;
}</hal3.2>
```

在这里看看registerBuffersLocked，参考前面对这个函数他是register完所有的buffer时被调用，在这里其本质处理的buffer状态不在是CAMERA3_BUFFER_STATUS_ERROR，而是CAMERA3_BUFFER_STATUS_OK故执行的是将会queuebuffer的操作。

5 buffer数据真正的被Consumer处理

在queuebuffer的操作时，参考前一博文Android5.1中surface和CpuConsumer下生产者和消费者间的处理框架简述

很容易知道真正的Consumer需要开始工作了，对于preview模式下的当然是由SurfaceFlinger的那套机制去处理。而在Camera2Client和Camera3Device下你还可以看到CPUConsumer的存在，比如：

```
void CallbackProcessor::onFrameAvailable(const BufferItem& /*item*/) {
    Mutex::Autolock l(mInputMutex);
    if (!mCallbackAvailable) {
        mCallbackAvailable = true;
        mCallbackAvailableSignal.signal();//数据callback线程处理
    }
}
```

在这里，你就可以去处理那些处于queue状态的buffer数据，比如这里的Callback将这帧数据上传会APP。

```
bool CallbackProcessor::threadLoop() {
    status_t res;

    {
        Mutex::Autolock l(mInputMutex);
        while (!mCallbackAvailable) {
            res = mCallbackAvailableSignal.waitRelative(mInputMutex,
                    kWaitDuration);
            if (res == TIMED_OUT) return true;
        }
        mCallbackAvailable = false;
    }

    do {
        sp<camera2client> client = mClient.promote();
        if
```

```
(client == 0)
 {
              res
 = discardNewCallback();

         }
else

{

              res
 = processNewCallback(client);//callback
 处理新的一帧

         }

     }
while

(res == OK);


    return

true;

}&lt;/camera2client&gt;
```

?

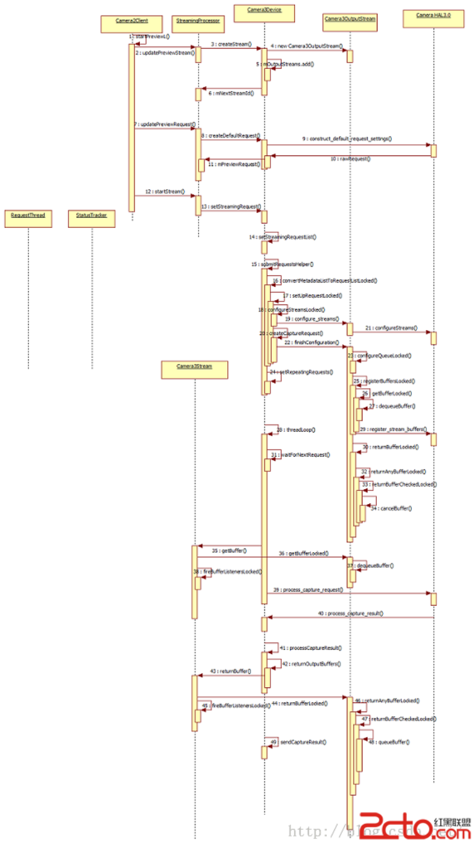
```
l.mRemoteCallback->dataCallback(CAMERA_MSG_PREVIEW_FRAME,

          callbackHeap->mBuffers[heapIdx],
NULL);//处理成API的需求后，回调Preview
frame
```

6 总结

到这里，整个preview预览的视频流基本介绍完毕了，主要框架虽然负责，但仔细看看也就是buffer的queue与dequeue操作，真正的HAL3的实现才是最为复杂的。后续还会简单介绍下整个take picture的过程，数据的回调处理在后续中还会继续分析。

下面贴一图是整个Camera3架构下基于Request和result的处理流序图:



上一篇 android camera接口介绍
下一篇 Android Camera API2中采用CameraMetadata用于从APP到HAL的参数交互

猜你在找

搜狗郭理勇：小而美-Sogou数据库中间件Compass深度	Android Camera HAL3中拍照Capture模式下多模块间的
大数据时代的<集装箱式>架构设计与Docker潮流	android camera HAL 错误数据流处理
大数据系统架构	我心依旧之Android Camera模块FWHAL3探学序
Windows Server 2012 R2 远程桌面管理	Android Camera HAL V3参数传递
Android之数据库详解	Android Camera HAL V3 Vendor Tag及V1V3参数转换

查看评论

暂无评论

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr Angular Cloud Foundry Redis Scala Django Bootstrap