# 【Android 音视频开发打怪升级：FFmpeg音视频编解码篇】七、Android FFmpeg 视频编码 - 简书



## write in front

This article is the last article in the audio and video series, and the one that has been delayed for the longest time (lazy cancer attack-_-!!), and finally made up my mind to fill in the hole.

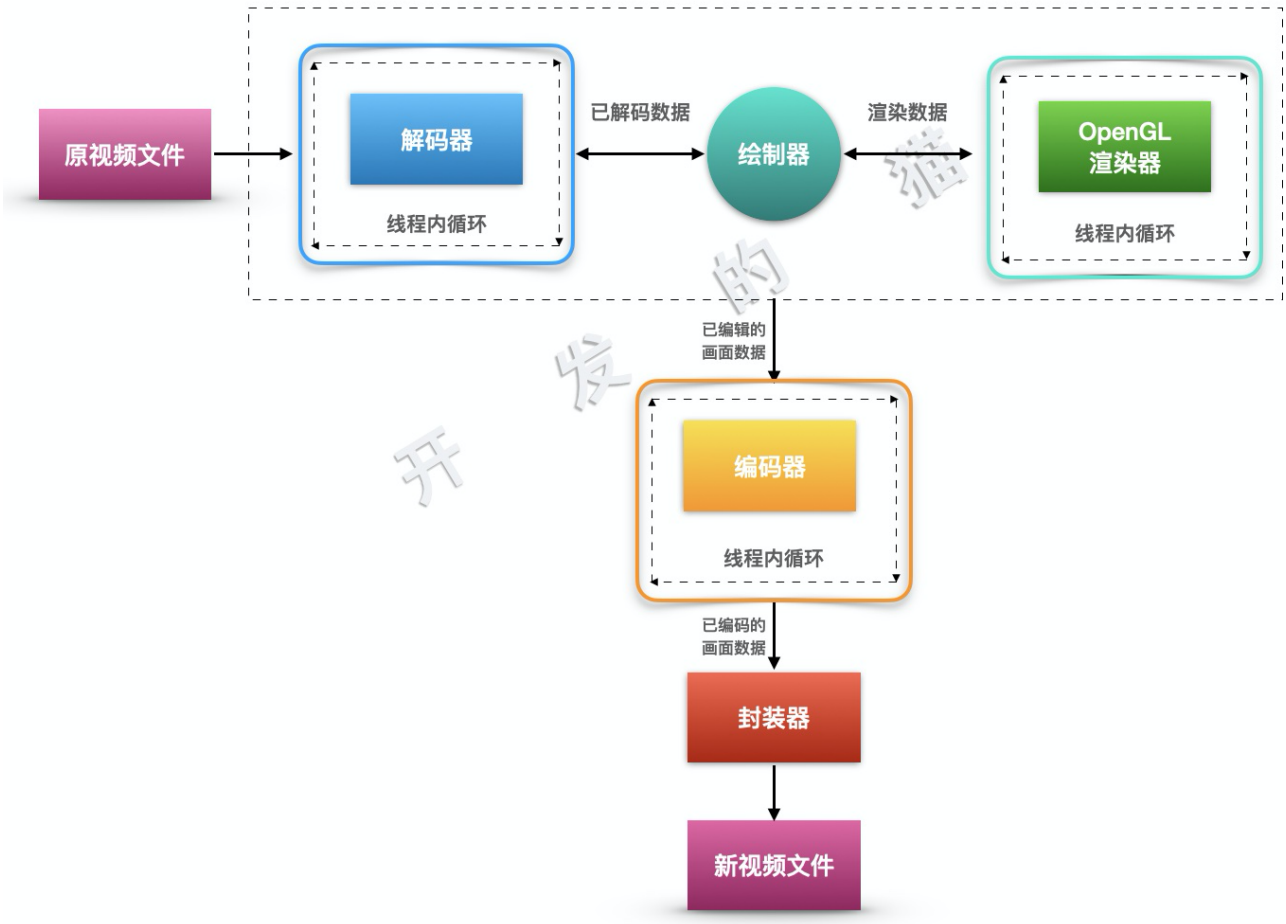
Without further ado, let's get into the text immediately.

In the previous article, I introduced how to decapsulate and repackage audio and video files. This process does not involve decoding and encoding of audio and video, that is, there is no editing of audio and video, which cannot meet daily development needs.

Therefore, this article will fill in the gaps in the editorial process and bring this series to a close.

## 1. Description of the overall process

In the previous articles, we have done a good job `解码器` , `OpenGL 渲染器` so, when coding, in addition to `编码器` needing , we also need to integrate the previous content. The following is a brief description with a picture:




## module

First of all, it can be noticed that this process has three major modules, which `独立又互相关联` are threads, responsible for:

- Original video decoding
- OpenGL screen rendering
- target video encoding

## data flow

See how the video data flows:

1. After the original video is `解码器` decoded , `YUV` data is obtained, and after format conversion, it becomes `RGB` data .
2. `解码器` `RGB` Pass data to `绘制器` , waiting for `OpenGL 渲染器` use .
3. `OpenGL 渲染器` Through the internal thread loop, when appropriate, call the `绘制器` render screen.

4. After the picture is drawn, the `OpenGL` rendered (edited) picture is obtained and sent to `编码器` for encoding.

5. Finally, write the encoded data to a local file.

> **illustrate:**
>
> This article will mainly talk about the `编码` knowledge . Since the whole process involves these two knowledge points introduced earlier, we will reuse the previously packaged tools and make some adaptations in some special places according to the needs of encoding `解码` . `OpenGL 渲染`
>
> Therefore, when it comes to decoding and OpenGL, you can post the adapted code. For details, you can check the previous article, or check the source code directly .

## 2. Compilation and introduction of x264 so library

Since `x264` is based on `GPL` the open source protocol, and the `FFmpeg` default is based on `LGPL` the protocol, `x264` when , due to `GPL` the contagious nature of , our code must also be open source, you can `OpenH264` use instead.

Here is still used `x264` to learn the relevant coding process.

In addition, due to space limitations, this article will not introduce `x264` the compilation of , and will write another article to introduce it.

The introduction of the x264 so library is the same as other so introductions. For details, please refer to the previous article, or check the CMakeList.txt in the source code.

> `FFmpeg` The h264 decoder is already built in, so if it's just decoding, it doesn't need to be imported `x264` .

## 3. Encapsulating the encoder

The encoding process is very similar to the decoding process, in fact, it is the inverse process of decoding, so the entire code framework process and the decoder are `BaseDecoder` basically the same.

### definition `BaseEncoder`

```
// BaseEncoder.h

class BaseEncoder: public IEncoder {
private:

    // 编码格式 ID
    AVCodecID m_codec_id;

    // 线程依附的JVM环境
    JavaVM *m_jvm_for_thread = NULL;

    // 编码器
    AVCodec *m_codec = NULL;

    // 编码上下文
    AVCodecContext *m_codec_ctx = NULL;

    // 编码数据包
    AVPacket *m_encoded_pkt = NULL;

    // 写入Mp4的输入流索引
    int m_encode_stream_index = 0;

    // 原数据时间基
    AVRational m_src_time_base;

    // 缓冲队列
    std::queue<OneFrame *> m_src_frames;

    // 操作数据锁
    std::mutex m_frames_lock;

    // 状态回调
    IEncodeStateCb *m_state_cb = NULL;

    bool Init();

    /**
     * 循环拉去已经编码的数据，直到没有数据或者编码完毕
     * @return true 编码结束；false 编码未完成
     */
    bool DrainEncode();

    /**
     * 编码一帧数据
     * @return 错误信息
     */
    int EncodeOneFrame();

    // 新建编码线程
    void CreateEncodeThread();
```

```
    // 解码静态方法，给线程调用
    static void Encode(std::shared_ptr<BaseEncoder> that);

    void OpenEncoder();

    // 循环编码
    void LoopEncode();

    void DoRelease();

    // 省略一些非重点代码(具体请查看源码)
    // .......

protected:

    // Mp4 封装器
    Mp4Muxer *m_muxer = NULL;

//-------------子类需要复写的方法 begin-----------
    // 初始化编码参数（上下文）
    virtual void InitContext(AVCodecContext *codec_ctx) = 0;

    // 配置Mp4 混淆通道信息
    virtual int ConfigureMuxerStream(Mp4Muxer *muxer, AVCodecContext *ctx) = 0;

    // 处理一帧数据
    virtual AVFrame* DealFrame(OneFrame *one_frame) = 0;

    // 释放资源
    virtual void Release() = 0;

    virtual const char *const LogSpec() = 0;
//-------------子类需要复写的方法 end-----------

public:
    BaseEncoder(JNIEnv *env, Mp4Muxer *muxer, AVCodecID codec_id);

    // 压入一帧待编码数据（由外部调用）
    void PushFrame(OneFrame *one_frame) override ;

    // 判断是否缓冲数据过多，用于控制缓冲队列大小
    bool TooMuchData() override {
        return m_src_frames.size() > 100;
    }

    // 设置编码状态监听器
    void SetStateReceiver(IEncodeStateCb *cb) override {
        this->m_state_cb = cb;
    }
};
```

The definition of the encoder is not complicated, it is nothing more than the encoder `m_codec`, decoding context `m_codec_id`, etc. that need to be used for encoding, and encapsulating the corresponding function method to split several steps in the encoding process. Here are the main points:

Control encoding buffer queue size

> Since the encoding speed is much lower than the decoding speed during the encoding process, it is necessary to control the size of the buffer queue to avoid the accumulation of a large amount of data, resulting in content overflow or failure to apply for memory.

Timestamp conversion

> Timestamp conversion has been explained in the previous article, please refer to the previous article for details. In a word, since the time base of the original video and the target video are different, it is necessary to convert the time stamp to ensure that the time after encoding and saving is normal.

Make sure the MP4 track index is correct

> MP4 has two tracks of audio and video, which need to be well matched when writing, see the code for details `m_encode_stream_index`.

### accomplish `BaseEncoder`

initialization

```
// BaseEncoder.cpp

BaseEncoder::BaseEncoder(JNIEnv *env, Mp4Muxer *muxer, AVCodecID codec_id)
: m_muxer(muxer),
m_codec_id(codec_id) {
    if (Init()) {
        env->GetJavaVM(&m_jvm_for_thread);
        CreateEncodeThread();
    }
}

bool BaseEncoder::Init() {
    // 1. 查找编码器
    m_codec = avcodec_find_encoder(m_codec_id);
    if (m_codec == NULL) {
        LOGE(TAG, "Fail to find encoder, code id is %d", m_codec_id)
        return false;
    }
    // 2. 分配编码上下文
    m_codec_ctx = avcodec_alloc_context3(m_codec);
    if (m_codec_ctx == NULL) {
        LOGE(TAG, "Fail to alloc encoder context")
        return false;
    }

    // 3. 初始化编码数据包
    m_encoded_pkt = av_packet_alloc();
    av_init_packet(m_encoded_pkt);

    return true;
}

void BaseEncoder::CreateEncodeThread() {
    // 使用智能指针，线程结束时，自动删除本类指针
    std::shared_ptr<BaseEncoder> that(this);
    std::thread t(Encode, that);
    t.detach();
}
```

Encoding takes two parameters, `m_muxer` and `m_codec_id`, both: MP4 mixer and encoding format ID.

Among them, the encoding format ID is set according to the needs of audio and video, such as video H264 is: `AV_CODEC_ID_H264`, audio AAC is: `AV_CODEC_ID_AAC`.

Next, call the `Init()` method :

1. Find encoder by encoding format ID
2. Allocate coding context
3. Initialize the encoded packet

Finally, create an encoding thread.

## Encapsulation encoding process

```
// BaseEncoder.cpp

void BaseEncoder::Encode(std::shared_ptr<BaseEncoder> that) {
    JNIEnv * env;

    //将线程附加到虚拟机，并获取env
    if (that->m_jvm_for_thread->AttachCurrentThread(&env, NULL) != JNI_OK) {
        LOG_ERROR(that->TAG, that->LogSpec(), "Fail to Init encode thread");
        return;
    }

    that->OpenEncoder(); // 1
    that->LoopEncode();  // 2
    that->DoRelease();   // 3

    //解除线程和jvm关联
    that->m_jvm_for_thread->DetachCurrentThread();

}
```

The process is very similar to decoding.

Step 1, turn on the encoder

```
// BaseEncoder.cpp

void BaseEncoder::OpenEncoder() {
    // 调用子类方法，根据音频和视频的不同，初始化编码上下文
    InitContext(m_codec_ctx);

    int ret = avcodec_open2(m_codec_ctx, m_codec, NULL);
    if (ret < 0) {
        LOG_ERROR(TAG, LogSpec(), "Fail to open encoder : %d", m_codec);
        return;
    }

    m_encode_stream_index = ConfigureMuxerStream(m_muxer, m_codec_ctx);
}
```

Step 2, start the encoding loop

There are only two core methods of encoding:

`avcodec_send_frame` : Send data to the encoding queue

`avcodec_receive_packet` : Receive encoded data

### The encoding process mainly has 5 steps:

1. Get the data to be decoded from the buffer queue
2. Hand over raw data to subclasses for processing (audio and video are processed according to their own needs)
3. Encode by `avcodec_send_frame` sending the data to the encoder
4. Extract the encoded data

Another point, point 5, is **to resend the data** .

It is necessary to explain the `双循环` coding : in addition to the outermost `while(tue)` loop, there is also a `while (m_src_frames.size() > 0)` loop .

`在缓冲队列有数据，并且 FFmpeg 内部编码队列未满` In the case of , it will continue to `FFmpeg` send data to until it finds that the `FFmpeg` code returns `AVERROR(EAGAIN)` , indicating that the internal queue is full, and the encoded data needs to be extracted first, that is, the `DrainEncode()` method .

### One more thing to note: how to interpret that all data has been sent to the encoder?

> Here `one_frame->line_size` is .
>
> When listening to the decoder's notification that the decoding is completed, set an empty frame `OneFrame` data `line_size` to `0` , and push it into the buffer queue.
>
> `BaseEncoder` When you get this empty data frame `FFmpeg` , `avcodec_send_frame()` send a `NULL` data to , and it `FFmpeg` will automatically end encoding.

Please see the following code for details:

```
// BaseEncoder.cpp

void BaseEncoder::LoopEncode() {
    if (m_state_cb != NULL) {
        m_state_cb->EncodeStart();
    }
    while (true) {
        if (m_src_frames.size() == 0) {
            Wait();
        }
        while (m_src_frames.size() > 0) {
            // 1. 获取待解码数据
            m_frames_lock.lock();
            OneFrame *one_frame = m_src_frames.front();
            m_src_frames.pop();
            m_frames_lock.unlock();

            AVFrame *frame = NULL;
            if (one_frame->line_size != 0) {
                m_src_time_base = one_frame->time_base;
                // 2. 子类处理数据
                frame = DealFrame(one_frame);
                delete one_frame;
                if (m_state_cb != NULL) {
                    m_state_cb->EncodeSend();
                }
                if (frame == NULL) {
                    continue;
                }
            } else { //如果数据长度为0，说明编码已经结束，压入空frame，使编码器进入结束状态
                delete one_frame;
            }
            // 3. 将数据发送到编码器
            int ret = avcodec_send_frame(m_codec_ctx, frame);
            switch (ret) {
                case AVERROR_EOF:
                    LOG_ERROR(TAG, LogSpec(), "Send frame finish [AVERROR_EOF]")
                    break;
                case AVERROR(EAGAIN): //编码编码器已满，先取出已编码数据，再尝试发送数据
                    while (ret == AVERROR(EAGAIN)) {
                        LOG_ERROR(TAG, LogSpec(), "Send frame error[EAGAIN]: %s",
av_err2str(AVERROR(EAGAIN)));
                        // 4. 将编码好的数据榨干
                        if (DrainEncode()) return; //编码结束
                        // 5. 重新发送数据
                        ret = avcodec_send_frame(m_codec_ctx, frame);
                    }
                    break;
                case AVERROR(EINVAL):
                    LOG_ERROR(TAG, LogSpec(), "Send frame error[EINVAL]: %s",
av_err2str(AVERROR(EINVAL)));
                    break;
```

```
                case AVERROR(ENOMEM):
                    LOG_ERROR(TAG, LogSpec(), "Send frame error[ENOMEM]: %s",
av_err2str(AVERROR(ENOMEM)));
                    break;
                default:
                    break;
            }
            if (ret != 0) break;
        }

        if (DrainEncode()) break; //编码结束
    }
}
```

Next, look at the `DrainEncode()` methods :

```
// BaseEncoder.cpp

bool BaseEncoder::DrainEncode() {
    int state = EncodeOneFrame();
    while (state == 0) {
        state = EncodeOneFrame();
    }
    return state == AVERROR_EOF;
}

int BaseEncoder::EncodeOneFrame() {
    int state = avcodec_receive_packet(m_codec_ctx, m_encoded_pkt);
    switch (state) {
        case AVERROR_EOF: //解码结束
            LOG_ERROR(TAG, LogSpec(), "Encode finish")
            break;
        case AVERROR(EAGAIN): //编码还未完成，待会再来
            LOG_INFO(TAG, LogSpec(), "Encode error[EAGAIN]: %s",
av_err2str(AVERROR(EAGAIN)));
            break;
        case AVERROR(EINVAL):
            LOG_ERROR(TAG, LogSpec(),  "Encode error[EINVAL]: %s",
av_err2str(AVERROR(EINVAL)));
            break;
        case AVERROR(ENOMEM):
            LOG_ERROR(TAG, LogSpec(), "Encode error[ENOMEM]: %s",
av_err2str(AVERROR(ENOMEM)));
            break;
        default: // 成功获取到一帧编码好的数据，写入 MP4
            //将视频pts/dts转换为容器pts/dts
            av_packet_rescale_ts(m_encoded_pkt, m_src_time_base,
                                 m_muxer->GetTimeBase(m_encode_stream_index));
            if (m_state_cb != NULL) {
                m_state_cb->EncodeFrame(m_encoded_pkt->data);
                long cur_time = (long)(m_encoded_pkt->pts*av_q2d(m_muxer-
>GetTimeBase(m_encode_stream_index))*1000);
                m_state_cb->EncodeProgress(cur_time);
            }
            m_encoded_pkt->stream_index = m_encode_stream_index;
            m_muxer->Write(m_encoded_pkt);
            break;
    }
    av_packet_unref(m_encoded_pkt);
    return state;
}
```

The same is a `while` cycle , according to the status of the received data to determine whether to end the cycle.

The main logic is `EncodeOneFrame()` in , and the data that `avcodec_receive_packet()` has been encoded in get is obtained . If the method returns to indicate that the acquisition is successful, the data can be written into . `FFmpeg` `0` `MP4`

`EncodeOneFrame()` What is returned is `avcodec_receive_packet` the return value of , then when it is `0` , loop to get the next frame of data until the return value is `AVERROR(EAGAIN)` or `AVERROR_EOF` , that is: no data or the end of encoding.

**In this way, through the above several cycles, the data is continuously inserted into the encoder, and the data is pulled until all data encoding is completed, and the encoding is ended.**

Step 3, end coding and release resources

After coding, you need to release related resources

```
// BaseEncoder.cpp

void BaseEncoder::DoRelease() {
    if (m_encoded_pkt != NULL) {
        av_packet_free(&m_encoded_pkt);
        m_encoded_pkt = NULL;
    }
    if (m_codec_ctx != NULL) {
        avcodec_close(m_codec_ctx);
        avcodec_free_context(&m_codec_ctx);
    }
    // 调用子类方法，释放子类资源
    Release();

    if (m_state_cb != NULL) {
        m_state_cb->EncodeFinish();
    }
}
```

## Encapsulated Video Encoder

The video encoder inherits from the base encoder defined above `BaseEncoder` .

```
// VideoEncoder.h

class VideoEncoder: public BaseEncoder {
private:

    const char * TAG = "VideoEncoder";

    // 视频格式转化工具
    SwsContext *m_sws_ctx = NULL;

    // 一阵 YUV 数据
    AVFrame *m_yuv_frame = NULL;

    // 目标视频宽高
    int m_width = 0, m_height = 0;

    void InitYUVFrame();

protected:

    const char *const LogSpec() override {
        return "视频";
    };

    void InitContext(AVCodecContext *codec_ctx) override;
    int ConfigureMuxerStream(Mp4Muxer *muxer, AVCodecContext *ctx) override;
    AVFrame* DealFrame(OneFrame *one_frame) override;
    void Release() override;

public:
    VideoEncoder(JNIEnv *env, Mp4Muxer *muxer, int width, int height);

};
```

Implementation:

## 1. Construction method:

```
// VideoEncoder.cpp

VideoEncoder::VideoEncoder(JNIEnv *env, Mp4Muxer *muxer, int width, int height)
: BaseEncoder(env, muxer, AV_CODEC_ID_H264),
m_width(width),
m_height(height) {
    m_sws_ctx = sws_getContext(width, height, AV_PIX_FMT_RGBA,
                               width, height, AV_PIX_FMT_YUV420P, SWS_FAST_BILINEAR,
                               NULL, NULL, NULL);
}
```

Here, according to the width and height of the target output video, the original format (RGBA data output by OpenGL)/target format (YUV), the format converter is initialized, which is exactly the opposite process of decoding.

**2. Coding parameter initialization:**

2.1 Initialize context and subclass internal data, mainly configure `宽高` , , `码率` , `帧率` , `时间基` etc. of encoded video.

Another important parameter is `qmin` and `qmax` , whose value ranges from [0 to 51], is used to configure the quality of the encoded picture. The larger the value, the lower the picture quality and the smaller the video file. You can configure it according to your needs.

There is `InitYUVFrame()` also the `YUV` data memory space needed to apply for transcoding.

```
// VideoEncoder.cpp

void VideoEncoder::InitContext(AVCodecContext *codec_ctx) {

    codec_ctx->bit_rate = 3*m_width*m_height;

    codec_ctx->width = m_width;
    codec_ctx->height = m_height;

    //把1秒钟分成fps个单位
    codec_ctx->time_base = {1, ENCODE_VIDEO_FPS};
    codec_ctx->framerate = {ENCODE_VIDEO_FPS, 1};

    //画面组大小
    codec_ctx->gop_size = 50;
    //没有B帧
    codec_ctx->max_b_frames = 0;

    codec_ctx->pix_fmt = AV_PIX_FMT_YUV420P;

    codec_ctx->thread_count = 8;

    av_opt_set(codec_ctx->priv_data, "preset", "ultrafast", 0);
    av_opt_set(codec_ctx->priv_data, "tune", "zerolatency", 0);

    //这是量化范围设定，其值范围为0~51，
    //越小质量越高，需要的比特率越大，0为无损编码
    codec_ctx->qmin = 28;
    codec_ctx->qmax = 50;

    //全局的编码信息
    codec_ctx->flags |= AV_CODEC_FLAG_GLOBAL_HEADER;

    InitYUVFrame();

    LOGI(TAG, "Init codec context success")
}

void VideoEncoder::InitYUVFrame() {
    //设置YUV输出空间
    m_yuv_frame = av_frame_alloc();
    m_yuv_frame->format = AV_PIX_FMT_YUV420P;
    m_yuv_frame->width = m_width;
    m_yuv_frame->height = m_height;
    //分配空间
    int ret = av_frame_get_buffer(m_yuv_frame, 0);
    if (ret < 0) {
        LOGE(TAG, "Fail to get yuv frame buffer");
    }
}
```

2.2 According to the decoder information, write the corresponding MP4 track information.

```
// VideoEncoder.cpp

int VideoEncoder::ConfigureMuxerStream(Mp4Muxer *muxer, AVCodecContext *ctx) {
    return muxer->AddVideoStream(ctx);
}
```

### 3. Processing data

Remember the subclass data processing method defined by the parent class?

The video encoder needs to convert the `OpenGL` output to `RGBA` data into `YUV` data before it can be fed into the encoder for encoding.

```
// VideoEncoder.cpp

AVFrame* VideoEncoder::DealFrame(OneFrame *one_frame) {
    uint8_t *in_data[AV_NUM_DATA_POINTERS] = { 0 };
    in_data[0] = one_frame->data;
    int src_line_size[AV_NUM_DATA_POINTERS] = { 0 };
    src_line_size[0] = one_frame->line_size;

    int h = sws_scale(m_sws_ctx, in_data, src_line_size, 0, m_height,
                      m_yuv_frame->data, m_yuv_frame->linesize);
    if (h <= 0) {
        LOGE(TAG, "转码出错");
        return NULL;
    }

    m_yuv_frame->pts = one_frame->pts;

    return m_yuv_frame;
}
```

### 4. Release subclass resources

After the encoding is over, the parent class calls back the subclass method and method resource to notify `Mp4Muxer` the end of video channel writing.

```
// VideoEncoder.cpp

void VideoEncoder::Release() {
    if (m_yuv_frame != NULL) {
        av_frame_free(&m_yuv_frame);
        m_yuv_frame = NULL;
    }
    if (m_sws_ctx != NULL) {
        sws_freeContext(m_sws_ctx);
        m_sws_ctx = NULL;
    }
    // 结束视频通道数据写入
    m_muxer->EndVideoStream();
}
```

### encapsulated audio encoder

The basic video of the audio encoder is the same, but the parameter configuration is different, just look at the implementation directly.

General audio parameter configuration: bit rate, encoding format, number of channels, etc.

Focus on the `InitFrame()` method . Here, you need to `av_samples_get_buffer_size()` calculate the memory size used to save the target frame data through the number of channels, encoding format, etc., with the help of the method.

```
// AudioEncoder.cpp

AudioEncoder::AudioEncoder(JNIEnv *env, Mp4Muxer *muxer)
: BaseEncoder(env, muxer, AV_CODEC_ID_AAC) {

}

void AudioEncoder::InitContext(AVCodecContext *codec_ctx) {
    codec_ctx->codec_type = AVMEDIA_TYPE_AUDIO;
    codec_ctx->sample_fmt = ENCODE_AUDIO_DEST_FORMAT;
    codec_ctx->sample_rate = ENCODE_AUDIO_DEST_SAMPLE_RATE;
    codec_ctx->channel_layout = ENCODE_AUDIO_DEST_CHANNEL_LAYOUT;
    codec_ctx->channels = ENCODE_AUDIO_DEST_CHANNEL_COUNTS;
    codec_ctx->bit_rate = ENCODE_AUDIO_DEST_BIT_RATE;

    InitFrame();
}

void AudioEncoder::InitFrame() {
    m_frame = av_frame_alloc();
    m_frame->nb_samples = 1024;
    m_frame->format = ENCODE_AUDIO_DEST_FORMAT;
    m_frame->channel_layout = ENCODE_AUDIO_DEST_CHANNEL_LAYOUT;

    int size = av_samples_get_buffer_size(NULL, ENCODE_AUDIO_DEST_CHANNEL_COUNTS,
m_frame->nb_samples,
                                          ENCODE_AUDIO_DEST_FORMAT, 1);
    uint8_t *frame_buf = (uint8_t *) av_malloc(size);
    avcodec_fill_audio_frame(m_frame, ENCODE_AUDIO_DEST_CHANNEL_COUNTS,
ENCODE_AUDIO_DEST_FORMAT,
                             frame_buf, size, 1);
}

int AudioEncoder::ConfigureMuxerStream(Mp4Muxer *muxer, AVCodecContext *ctx) {
    return muxer->AddAudioStream(ctx);
}

AVFrame* AudioEncoder::DealFrame(OneFrame *one_frame) {
    m_frame->pts = one_frame->pts;
    memcpy(m_frame->data[0], one_frame->data, 4096);
    memcpy(m_frame->data[1], one_frame->ext_data, 4096);
    return m_frame;
}

void AudioEncoder::Release() {
    m_muxer->EndAudioStream();
}
```

Finally, `DealFrame` it is necessary `one_frame` to copy the left and right channel data saved in to `m_frame` the memory of the application, and return it `父类` to encoder for encoding.

## Fourth, get the video data rendered by OpenGL

We know that after the video data is edited by OpenGL, it cannot be directly sent to the encoder for encoding, and it needs to be obtained by `OpenGL` the `glReadPixels` method of .

Let's transform the original definition `OpenGLRender` to achieve.

For the complete code, please check the project source code .

In the rendering `Render()` method , add the method of acquiring the picture:

```
// OpenGLRender.cpp

void OpenGLRender::Render() {
    if (RENDERING == m_state) {
        m_drawer_proxy->Draw();
        m_egl_surface->SwapBuffers();

        if (m_need_output_pixels && m_pixel_receiver != NULL) {//输出画面rgba
            m_need_output_pixels = false;
            Render(); //再次渲染最新的画面

            size_t size = m_window_width * m_window_height * 4 * sizeof(uint8_t);

            uint8_t *rgb = (uint8_t *) malloc(size);
            if (rgb == NULL) {
                realloc(rgb, size);
                LOGE(TAG, "内存分配失败： %d", rgb)
            }
            glReadPixels(0, 0, m_window_width, m_window_height, GL_RGBA,
GL_UNSIGNED_BYTE, rgb);

            // 将数据发送出去
            m_pixel_receiver->ReceivePixel(rgb);
        }
    }
}
```

Add a request method to notify `OpenGLRender` to send data output:

```
// OpenGLRender.cpp

void OpenGLRender::RequestRgbaData() {
    m_need_output_pixels = true;
}
```

> The principle is very simple. After the decoder decodes a frame of data and sends it to `OpenGL` render , it immediately informs `OpenGLRender` to send the picture.

Of course, you also need to define a receiver:

```
// OpenGLPixelReceiver.h

class OpenGLPixelReceiver {
public:
    virtual void ReceivePixel(uint8_t *rgba) = 0;
};
```

## 5. MP4 wrapper

The content of this part is basically the repackage of the repackage `FFRepack` tool , which will not be repeated here. Please check the previous article or the source code.

```
// Mp4Muxer.cpp

void Mp4Muxer::Init(JNIEnv *env, jstring path) {
    const char *u_path = env->GetStringUTFChars(path, NULL);

    int len = strlen(u_path);
    m_path = new char[len];
    strcpy(m_path, u_path);

    //新建输出上下文
    avformat_alloc_output_context2(&m_fmt_ctx, NULL, NULL, m_path);

    // 释放引用
    env->ReleaseStringUTFChars(path, u_path);
}

int Mp4Muxer::AddVideoStream(AVCodecContext *ctx) {
    int stream_index = AddStream(ctx);
    m_video_configured = true;
    Start();
    return stream_index;
}

int Mp4Muxer::AddAudioStream(AVCodecContext *ctx) {
    int stream_index = AddStream(ctx);
    m_audio_configured = true;
    Start();
    return stream_index;
}

int Mp4Muxer::AddStream(AVCodecContext *ctx) {
    AVStream *video_stream = avformat_new_stream(m_fmt_ctx, NULL);
    avcodec_parameters_from_context(video_stream->codecpar, ctx);
    video_stream->codecpar->codec_tag = 0;
    return video_stream->index;
}

void Mp4Muxer::Start() {
    if (m_video_configured && m_audio_configured) {
        av_dump_format(m_fmt_ctx, 0, m_path, 1);
        //打开文件输入
        int ret = avio_open(&m_fmt_ctx->pb, m_path, AVIO_FLAG_WRITE);
        if (ret < 0) {
            LOGE(TAG, "Open av io fail")
            return;
        } else {
            LOGI(TAG, "Open av io: %s", m_path)
        }
        //写入头部信息
        ret = avformat_write_header(m_fmt_ctx, NULL);
        if (ret < 0) {
            LOGE(TAG, "Write header fail")
```

```
            return;
        } else {
            LOGI(TAG, "Write header success")
        }
    }
}

void Mp4Muxer::Write(AVPacket *pkt) {
    int ret = av_interleaved_write_frame(m_fmt_ctx, pkt);
//    uint64_t time = uint64_t (pkt->pts*av_q2d(GetTimeBase(pkt-
>stream_index))*1000);
//    LOGE(TAG, "Write one frame pts: %lld, ret = %s", time , av_err2str(ret))
}

void Mp4Muxer::EndAudioStream() {
    LOGI(TAG, "End audio stream")
    m_audio_end = true;
    Release();
}

void Mp4Muxer::EndVideoStream() {
    LOGI(TAG, "End video stream")
    m_video_end = true;
    Release();
}

void Mp4Muxer::Release() {
    if (m_video_end && m_audio_end) {
        if (m_fmt_ctx) {
            //写入文件尾部
            av_write_trailer(m_fmt_ctx);

            //关闭输出IO
            avio_close(m_fmt_ctx->pb);

            //释放资源
            avformat_free_context(m_fmt_ctx);

            m_fmt_ctx = NULL;
        }
        delete [] m_path;
        LOGI(TAG, "Muxer Release")
        if (m_mux_finish_cb) {
            m_mux_finish_cb->OnMuxFinished();
        }
    }
}
```

## 6. Integration call

With the definition and packaging of the above tools, plus the previous decoder and renderer, everything is ready, just owe Dongfeng!

We need to integrate them together and connect the entire [decoding--editing--encoding--write to MP4] process.

Define the synth `Synthesizer`.

### initialization

---

```
// Synthesizer.cpp

// 这里直接写死视频宽高了， 需要根据自己的需求动态配置
static int WIDTH = 1920;
static int HEIGHT = 1080;

Synthesizer::Synthesizer(JNIEnv *env, jstring src_path, jstring dst_path) {

    // 封装器
    m_mp4_muxer = new Mp4Muxer();
    m_mp4_muxer->Init(env, dst_path);
    m_mp4_muxer->SetMuxFinishCallback(this);

    // --------------------------视频配置--------------------------
    // 【视频编码器】
    m_v_encoder = new VideoEncoder(env, m_mp4_muxer, WIDTH, HEIGHT);
    m_v_encoder->SetStateReceiver(this);

    // 【绘制器】
    m_drawer_proxy = new DefDrawerProxyImpl();
    VideoDrawer *drawer = new VideoDrawer();
    m_drawer_proxy->AddDrawer(drawer);

    // 【OpenGL 渲染器】
    m_gl_render = new OpenGLRender(env, m_drawer_proxy);
    // 设置离屏渲染画面宽高
    m_gl_render->SetOffScreenSize(WIDTH, HEIGHT);
    // 接收经过（编辑）渲染的画面数据
    m_gl_render->SetPixelReceiver(this);

    // 【视频解码器】
    m_video_decoder = new VideoDecoder(env, src_path, true);
    m_video_decoder->SetRender(drawer);

    // 监听解码状态
    m_video_decoder->SetStateReceiver(this);

    //--------------------------音频配置--------------------------
    // 【音频编码器】
    m_a_encoder = new AudioEncoder(env, m_mp4_muxer);
    // 监听编码状态
    m_a_encoder->SetStateReceiver(this);

    // 【音频解码器】
    m_audio_decoder = new AudioDecoder(env, src_path, true);
    // 监听解码状态
    m_audio_decoder->SetStateReceiver(this);
}
```

It can be seen that the decoding process is almost the same as before, and the three differences are:

1. The decoder needs to be told that this is a synthesis process, no need to add time synchronization after decoding.
2. OpenGL rendering is off-screen rendering, you need to set the rendering size
3. The audio does not need to be rendered into OpenSL, it can be sent directly to the encoding.

### start up

After the initialization is completed, the decoder enters the waiting period and needs an external trigger to enter the cyclic decoding process.

```
// Synthesizer.cpp

void Synthesizer::Start() {
    m_video_decoder->GoOn();
    m_audio_decoder->GoOn();
}
```

When `BaseDecoder` the `GoOn()`, the entire [Decoding-->Encoding] process will be started.

What glues them together is the state callback method of the decoder `DecodeOneFrame()`.

```
// Synthesizer.cpp

bool Synthesizer::DecodeOneFrame(IDecoder *decoder, OneFrame *frame) {
    if (decoder == m_video_decoder) {
        // 等待上一帧画面数据压入编码缓冲队列
        while (m_cur_v_frame) {
            av_usleep(2000); // 2ms
        }
        m_cur_v_frame = frame;
        m_gl_render->RequestRgbaData();
        return m_v_encoder->TooMuchData();
    } else {
        m_cur_a_frame = frame;
        m_a_encoder->PushFrame(frame);
        return m_a_encoder->TooMuchData();
    }
}

void Synthesizer::ReceivePixel(uint8_t *rgba) {
    OneFrame *rgbFrame = new OneFrame(rgba, m_cur_v_frame->line_size,
                                      m_cur_v_frame->pts, m_cur_v_frame->time_base);
    m_v_encoder->PushFrame(rgbFrame);
    // 清空上一帧数据信息
    m_cur_v_frame = NULL;
}
```

After receiving a frame of data from the decoder,

- If it is audio data, directly push the `BaseDecoder` data `PushFrame()` into the queue through the method of .
- If it is video data, save the current frame data information and notify `OpenGLRender` to send the picture data. After receiving the picture data in the `ReceivePixel()` method , pass the data `PushFrame()` to the video encoder.

Until the decoding is completed, in the `DecodeFinish()` method , push the empty data frame to inform the encoder to end the encoding.

```
// Synthesizer.cpp

void Synthesizer::DecodeFinish(IDecoder *decoder) {
    // 编码结束，压入一帧空数据，通知编码器结束编码
    if (decoder == m_video_decoder) {
        m_v_encoder->PushFrame(new OneFrame(NULL, 0, 0, AVRational{1, 25}, NULL));
    } else {
        m_a_encoder->PushFrame(new OneFrame(NULL, 0, 0, AVRational{1, 25}, NULL));
    }
}

void Synthesizer::EncodeFinish() {
    LOGI("Synthesizer", "EncodeFinish ...");
}

void Synthesizer::OnMuxFinished() {
    LOGI("Synthesizer", "OnMuxFinished ...");
    m_gl_render->Stop();

    if (m_mp4_muxer != NULL) {
        delete m_mp4_muxer;
    }
    m_drawer_proxy = NULL;
}
```

At this point, the whole process is complete! ! !

**This series of articles is finally over, and finally I have filled the hole and sprinkled flowers for myself~hahaha~** 🎉🎉🎉🎉🎉🎉🎉🎉🎉