[Android audio and video development and upgrade: FFmpeg audio and video codec] 3. Android FFmpeg video decoding and playback

jianshu.com/p/d7c8f49d9ea4

In this article you can learn

> Based on the audio and video decoding process of FFmpeg 4.x, it focuses on how to realize video playback.

foreword

Hi~ I've been waiting for a long time!

This article is very long, because there may be more friends who `JNI` `C/C++` are not very familiar with , so this article `FFmpeg` explains the code used in more detail, fully demonstrates `FFmpeg` the decoding and rendering process of , and encapsulates the decoding process.

In order to facilitate explanation and reading comprehension, the code is explained in blocks, that is to say, the content of the entire class will not be directly posted.

But each part of the code will be marked at the beginning of which file and which class it belongs to. If you want to see the complete code, please check the [Github repository] directly.

This article requires `C/C++` basic knowledge. If you are `C/C++` not familiar with it, you can check my other article: [Getting Started with Android NDK: C++ Basics].

Please read it patiently, I believe you can have a considerable understanding of `FFmpeg` decoding .

1. Introduction to FFmpeg related libraries

In the last article , the `FFmpeg` related libraries are introduced into the `Android` project, there are the following libraries:

library	introduce
avcodec	Audio and video codec core library
avformat	Encapsulation and parsing of audio and video container formats

library	introduce
on the helper	core tool library
swscal	Image format conversion module
swresample	Audio resampling
defilter	Audio and video filter library such as video watermarking, audio voice changing
avdevice	Input and output device library, providing input and output of device data

FFmpeg relies on the above libraries to achieve powerful audio and video encoding, decoding, editing, conversion, acquisition and other capabilities.

2. Introduction to FFMpeg decoding process

In the previous series of articles, the video decoding and playback were implemented using the native hard decoding capabilities `Android` provided by .

In summary, the process is as follows:

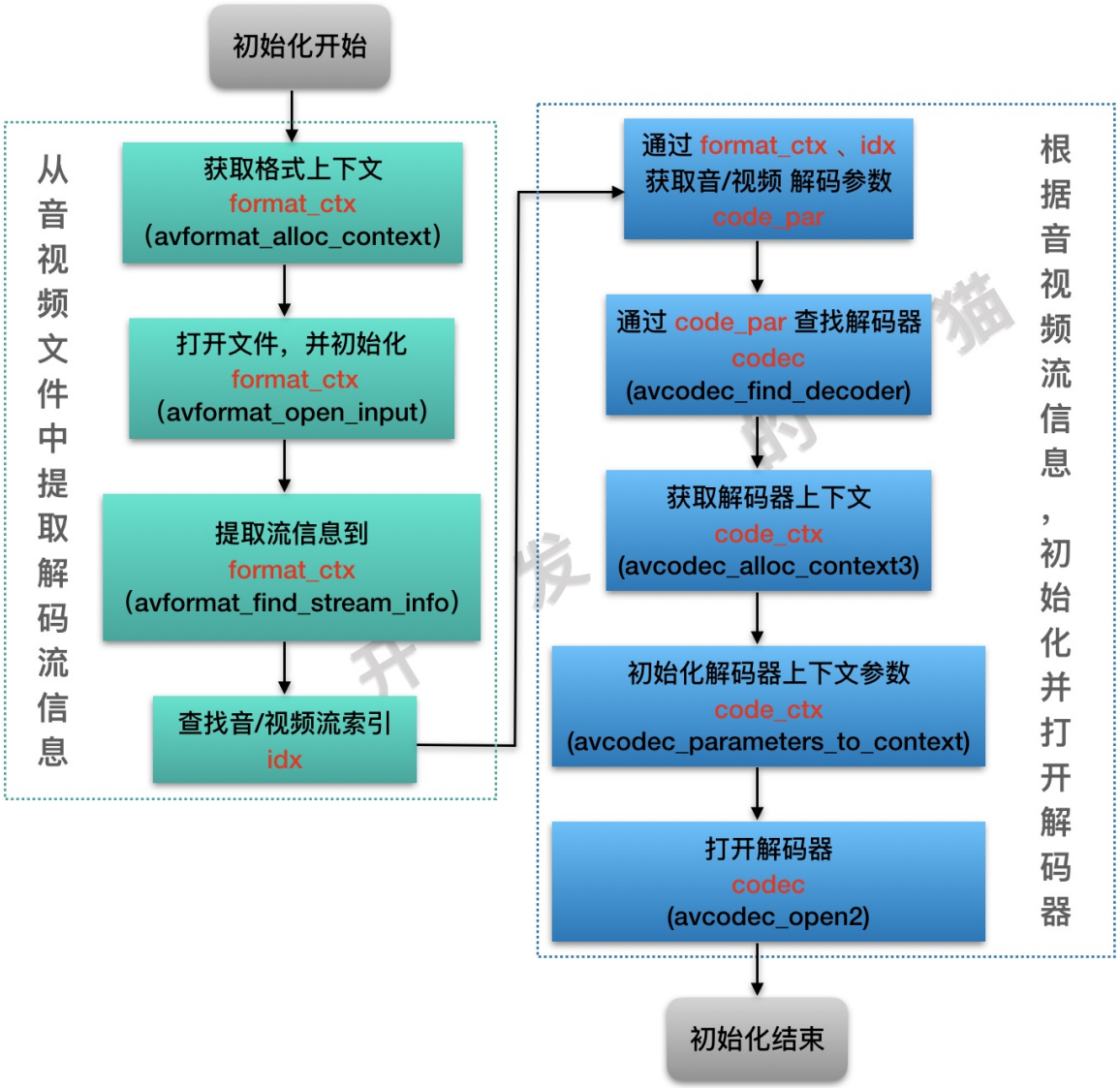
- Initialize the decoder
- Read the encoded data in the `Mp4` file and send it to the decoder for decoding
- Get decoded frame data
- Render a frame to the screen

> `FFmpeg` Decoding is nothing more than the above process, but `FFmpeg` only uses `CPU` the computing power of to decode.

1. FFmpeg initialization

`FFmpeg` The initialization process is relatively trivial compared to `Android` native hard decoding, but the process is fixed and can be applied directly once it is packaged.

First look at the flow chart of initialization



FFmpeg initialization

In fact, it is to initialize a series of parameters according to the format of the file to be decoded.

Among them, several are `结构体` more important, namely `AVFormatContext` (format_ctx), `AVCodecContext` (codec_ctx), `AVCodec` (codec)

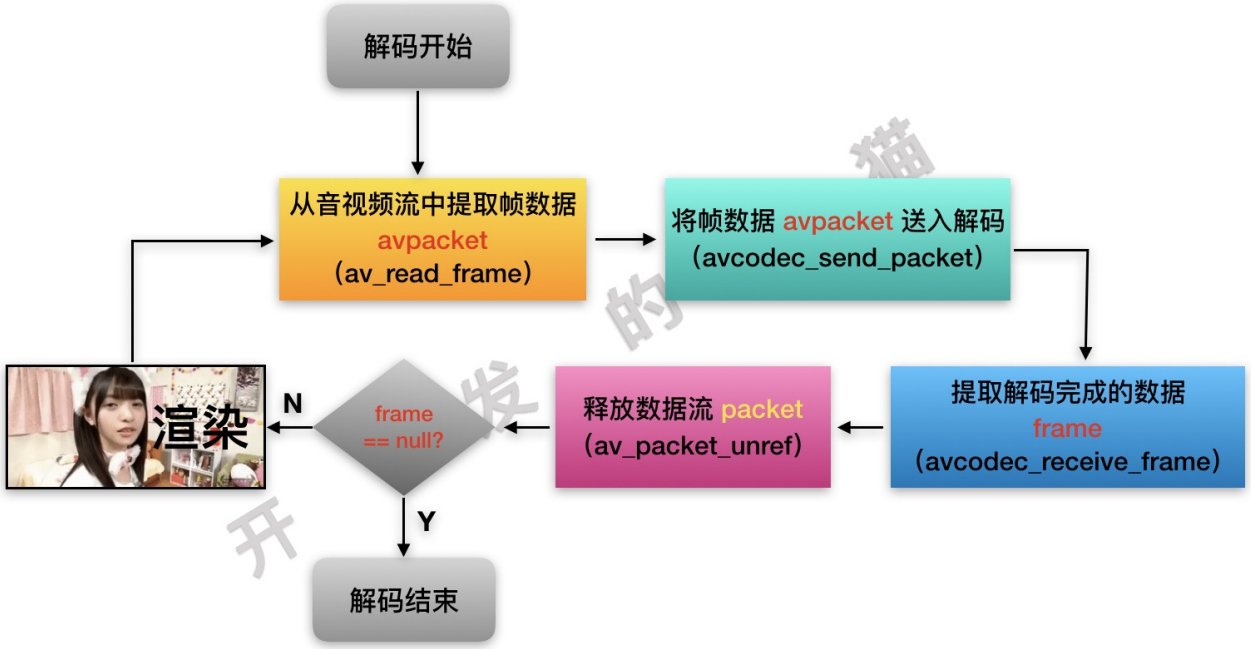
> **Structure** : FFmpeg is developed based on the `C` language , we know that `C` the language is a process-oriented language, that `C++` is classes to encapsulate internal data. However, it `C` provides a structure that can be used to encapsulate data to achieve a class-like effect.

- **AVFormatContext** : It belongs to `avformat` the library and stores the context of the stream data, mainly used for the `封装` sum of audio and video `解封` .
- **AVCodecContext** : It belongs to `avcodec` the library and stores the codec parameter context, which is mainly used for summing audio `编码` and video data `解码` .

- **AVCodec** : belongs to `avcodec` the library , audio and video codec, the real codec executor.

2. FFmpeg decoding loop

Similarly, a flowchart is used to illustrate the specific decoding process:



FFmpeg decoding loop

`FFmpeg` After initialization , the specific data frame decoding can be performed.

As can be seen from the above figure, `FFmpeg` the data is first extracted into one `AVPacket` (avpacket), and then by decoding, the data is decoded into a frame of data that can be rendered, called `AVFrame` (frame).

Similarly, `AVPacket` and `AVFrame` are also two structures, which encapsulate specific data.

3. Encapsulation and decoding class

With the above understanding of the decoding process, you `流程图` can .

According to past experience, since `FFmepg` the initialization and decoding processes are trivial and repetitive tasks, we must encapsulate them for better reuse and expansion.

Decoding process encapsulation

1. Define the decoding state: `decode_state.h`

On the `src/main/cpp/media/decoder` directory, right-click `New` -> `C++ Header File`, enter `decode_state`

```
//decode_state.h

#ifndef LEARNVIDEO_DECODESTATE_H
#define LEARNVIDEO_DECODESTATE_H

enum DecodeState {
    STOP,
    PREPARE,
    START,
    DECODING,
    PAUSE,
    FINISH
};

#endif //LEARNVIDEO_DECODESTATE_H
```

This is an enum that defines the state of the decoder decoding

2. Define the basic functions of the decoder: `i_decoder.h` :

On the `src/main/cpp/media/decoder` directory, right-click `New` -> `C++ Header File`, enter `i_decoder` .

```
// i_decoder.h

#ifndef LEARNVIDEO_I_DECODER_H
#define LEARNVIDEO_I_DECODER_H

class IDecoder {
public:
    virtual void GoOn() = 0;
    virtual void Pause() = 0;
    virtual void Stop() = 0;
    virtual bool IsRunning() = 0;
    virtual long GetDuration() = 0;
    virtual long GetCurPos() = 0;
};
```

This is a pure virtual class, similar `Java` to `interface` (see Getting Started with Android NDK: C++ Basics), which defines the basic methods that the decoder should have.

3. Define a decoder base class `base_decoder` .

On the `src/main/cpp/media/decoder` directory, right click `New` -> `C++ Class` input `base_decoder` , this class is used to encapsulate the most basic process in decoding.

Two files are generated: `base_decoder.h` , `base_decoder.cpp` .

Define the header file: `base_decoder.h`

```
//base_decoder.h

#ifndef LEARNVIDEO_BASEDECODER_H
#define LEARNVIDEO_BASEDECODER_H

#include <jni.h>
#include <string>
#include <thread>
#include "../../utils/logger.h"
#include "i_decoder.h"
#include "decode_state.h"

extern "C" {
#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include <libavutil/frame.h>
#include <libavutil/time.h>
};

class BaseDecoder: public IDecoder {

private:

    const char *TAG = "BaseDecoder";

    //-------------定义解码相关-------------------------------
    // 解码信息上下文
    AVFormatContext *m_format_ctx = NULL;

    // 解码器
    AVCodec *m_codec = NULL;

    // 解码器上下文
    AVCodecContext *m_codec_ctx = NULL;

    // 待解码包
    AVPacket *m_packet = NULL;

    // 最终解码数据
    AVFrame *m_frame = NULL;

    // 当前播放时间
    int64_t m_cur_t_s = 0;

    // 总时长
    long m_duration = 0;

    // 开始播放的时间
    int64_t m_started_t = -1;

    // 解码状态
```

```
    DecodeState m_state = STOP;

    // 数据流索引
    int m_stream_index = -1;

    // 省略其他

    // ......

}
```

> Note: When importing the header file of the `FFmpeg` related library, you need to pay attention to `#include` put `extern "C" {}` in . Because it `FFmpeg` is `C` written in the language, when it is introduced into the `C++` file , it needs to be marked with `C` to compile, otherwise it will cause a compilation error.

In the header file, first declare the relevant variables that `cpp` need to be used, the focus is on the decoding-related structures mentioned in the previous section.

Define methods related to initialization and decoding loops:

```
//base_decoder.h

class BaseDecoder: public IDecoder {

private:

    const char *TAG = "BaseDecoder";

    //-------------定义解码相关-------------------------------
    //省略....

    //-----------------私有方法-------------------------------

    /**
     * 初始化FFMpeg相关的参数
     * @param env jvm环境
     */
    void InitFFMpegDecoder(JNIEnv * env);

    /**
     * 分配解码过程中需要的缓存
     */
    void AllocFrameBuffer();

    /**
     * 循环解码
     */
    void LoopDecode();

    /**
     * 获取当前帧时间戳
     */
    void ObtainTimeStamp();

    /**
     * 解码完成
     * @param env jvm环境
     */
    void DoneDecode(JNIEnv *env);

    /**
     * 时间同步
     */
    void SyncRender();

    // 省略其他

    // ......

}
```

This decoding base class inherits from `i_decoder` and needs to implement the general methods specified in it.

```
//base_decoder.h

class BaseDecoder: public IDecoder {

    //省略其他

    //......

public:

    //--------构造方法和析构方法-------------

    BaseDecoder(JNIEnv *env, jstring path);
    virtual ~BaseDecoder();

    //--------实现基础类方法-----------------

    void GoOn() override;
    void Pause() override;
    void Stop() override;
    bool IsRunning() override;
    long GetDuration() override;
    long GetCurPos() override;
}
```

Define the decoding thread

We know that decoding is a very time-consuming operation. Just like native hard decoding, we need to start a thread to carry the decoding task. Therefore, first define thread-related variables and methods in the header file.

```
//base_decoder.h

class BaseDecoder: public IDecoder {

private:

    //省略其他

    //......

    // --------------------定义线程相关-----------------------------
    // 线程依附的JVM环境
    JavaVM *m_jvm_for_thread = NULL;

    // 原始路径jstring引用，否则无法在线程中操作
    jobject m_path_ref = NULL;

    // 经过转换的路径
    const char *m_path = NULL;

    // 线程等待锁变量
    pthread_mutex_t m_mutex = PTHREAD_MUTEX_INITIALIZER;
    pthread_cond_t m_cond = PTHREAD_COND_INITIALIZER;


    /**
     * 新建解码线程
     */
    void CreateDecodeThread();

    /**
     * 静态解码方法，用于解码线程回调
     * @param that 当前解码器
     */
    static void Decode(std::shared_ptr<BaseDecoder> that);

protected:

    /**
     * 进入等待
     */
    void Wait(long second = 0);

    /**
     * 恢复解码
     */
    void SendSignal();

}
```

Define virtual functions that subclasses need to implement

```
//base_decoder.h

class BaseDecoder: public IDecoder {
protected:

    /**
     * 子类准备回调方法
     * @note 注：在解码线程中回调
     * @param env 解码线程绑定的JVM环境
     */
    virtual void Prepare(JNIEnv *env) = 0;

    /**
     * 子类渲染回调方法
     * @note 注：在解码线程中回调
     * @param frame 视频：一帧YUV数据；音频：一帧PCM数据
     */
    virtual void Render(AVFrame *frame) = 0;

    /**
     * 子类释放资源回调方法
     */
    virtual void Release() = 0;

}
```

Above, the basic structure of the decoding class is defined:

- `FFmpeg` Decode related structure parameters
- Decoder basic method
- decoding thread
- Specifies the methods that subclasses need to implement

4. Implement the basic decoder

`base_decoder.cpp` In , implement the method declared in the header file

Initialize decoding thread

```
// base_decoder.cpp

#include "base_decoder.h"
#include "../../utils/timer.c"

BaseDecoder::BaseDecoder(JNIEnv *env, jstring path) {
    Init(env, path);
    CreateDecodeThread();
}

BaseDecoder::~BaseDecoder() {
    if (m_format_ctx != NULL) delete m_format_ctx;
    if (m_codec_ctx != NULL) delete m_codec_ctx;
    if (m_frame != NULL) delete m_frame;
    if (m_packet != NULL) delete m_packet;
}

void BaseDecoder::Init(JNIEnv *env, jstring path) {
    m_path_ref = env->NewGlobalRef(path);
    m_path = env->GetStringUTFChars(path, NULL);
    //获取JVM虚拟机，为创建线程作准备
    env->GetJavaVM(&m_jvm_for_thread);
}

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

The constructor is very simple, pass in `JNI` the environment variables, and the path of the file to be decoded.

In the `Init` method , because `jstring` is not `C++` a standard type of , you need to convert `jstring` the of type to the type in order to use it. `path` `char`

> **Explanation** : Since `JNIEnv` and `线程` are one-to-one correspondence, that is to say, `Android` in `JNI环境` is bound to a thread, each thread has an independent `JNIEnv` environment and cannot be accessed from each other. So if you want to access it in a new thread `JNIEnv` , you need to create a new one for this thread `JNIEnv` .

At the end of the `Init` method , `env->GetJavaVM(&m_jvm_for_thread)` get the `JavaVM` instance and save to `m_jvm_for_thread` , **this instance is shared by all** , and through it, a new `JNIEnv` environment .

`C++` Creating a thread in is very simple, with just two sentences, you can start a thread:

```
std::thread t(静态方法, 静态方法参数);
t.detach();
```

That is to say, the thread needs a static method as a parameter. After starting, the static method will be called back, and parameters can be passed to the static method.

Also, `CreateDecodeThread` the first code in the method is to create a smart pointer.

> We know that `C++` `new` the pointer object that comes out needs to be `delete` deleted , otherwise there will be a memory leak. The role of smart pointers is to help us achieve memory management.

When the reference count of this pointer reaches 0, it is automatically destroyed. In other words, we do not need to do it manually `delete` .

```
std::shared_ptr<BaseDecoder> that(this);
```

Here is `this` encapsulated into `that` a smart pointer named , so when the decoder is used externally, there is no need to manually release the memory. When the decoding thread exits, it will be automatically destroyed and the destructor will be called.

Encapsulation and decoding process

```
// base_decoder.cpp

void BaseDecoder::Decode(std::shared_ptr<BaseDecoder> 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 decode thread");
        return;
    }

    // 初始化解码器
    that->InitFFMpegDecoder(env);
    // 分配解码帧数据内存
    that->AllocFrameBuffer();
    // 回调子类方法，通知子类解码器初始化完毕
    that->Prepare(env);
    // 进入解码循环
    that->LoopDecode();
    // 退出解码
    that->DoneDecode(env);

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

}
```

In the `base_decoder.h` header file declaration, `Decode` is a static member method.

First, the decoding thread is created `JNIEnv`, and if it fails, the decoding is directly exited.

The above `Decode` method is to call the corresponding method step by step, it is very simple, just look at the comments.

Next, let's look at the content of the specific step-by-step call.

Initialize the decoder

```
void BaseDecoder::InitFFMpegDecoder(JNIEnv * env) {
    //1，初始化上下文
    m_format_ctx = avformat_alloc_context();

    //2，打开文件
    if (avformat_open_input(&m_format_ctx, m_path, NULL, NULL) != 0) {
        LOG_ERROR(TAG, LogSpec(), "Fail to open file [%s]", m_path);
        DoneDecode(env);
        return;
    }

    //3，获取音视频流信息
    if (avformat_find_stream_info(m_format_ctx, NULL) < 0) {
        LOG_ERROR(TAG, LogSpec(), "Fail to find stream info");
        DoneDecode(env);
        return;
    }

    //4，查找编解码器
    //4.1 获取视频流的索引
    int vIdx = -1;//存放视频流的索引
    for (int i = 0; i < m_format_ctx->nb_streams; ++i) {
        if (m_format_ctx->streams[i]->codecpar->codec_type == GetMediaType()) {
            vIdx = i;
            break;
        }
    }
    if (vIdx == -1) {
        LOG_ERROR(TAG, LogSpec(), "Fail to find stream index")
        DoneDecode(env);
        return;
    }
    m_stream_index = vIdx;

    //4.2 获取解码器参数
    AVCodecParameters *codecPar = m_format_ctx->streams[vIdx]->codecpar;

    //4.3 获取解码器
    m_codec = avcodec_find_decoder(codecPar->codec_id);

    //4.4 获取解码器上下文
    m_codec_ctx = avcodec_alloc_context3(m_codec);
    if (avcodec_parameters_to_context(m_codec_ctx, codecPar) != 0) {
        LOG_ERROR(TAG, LogSpec(), "Fail to obtain av codec context");
        DoneDecode(env);
        return;
    }

    //5，打开解码器
    if (avcodec_open2(m_codec_ctx, m_codec, NULL) < 0) {
        LOG_ERROR(TAG, LogSpec(), "Fail to open av codec");
        DoneDecode(env);
```

```
        return;
    }

    m_duration = (long)((float)m_format_ctx->duration/AV_TIME_BASE * 1000);

    LOG_INFO(TAG, LogSpec(), "Decoder init success")
}
```

It seems to be very complicated, but in fact the routines are the same, and you will feel uncomfortable at first, mainly because these methods are procedure-oriented calling methods, which are different from the usual usage habits of object-oriented languages.

for example:

In the above code, the method to open the file is as follows:

```
avformat_open_input(&m_format_ctx, m_path, NULL, NULL);
```

And if it is object-oriented, the code usually looks like this:

```
// 注意：以下为伪代码，仅用于举例说明

m_format_ctx.avformat_open_input(m_path);
```

So how `C` to understand this kind of procedure-oriented call in ?

We know that `m_format_ctx` is a structure that encapsulates specific data, so `avformat_open_input` this method is actually a method of operating this structure. Different method calls are operations on different data in the structure.

For the specific process, please refer to the comments above. Without going into details, it is actually the implementation of the steps in [Initialization Flowchart] in the first section.

There are two things to note:

1. `FFmpeg` The method with the `alloc` word in it usually only initializes the corresponding structure, but the specific parameters and data buffers generally need to be initialized by another method before they can be used.

For example `m_format_ctx` , `m_codec_ctx` :

```
// 创建
m_format_ctx = avformat_alloc_context();
// 初始化流信息
avformat_open_input(&m_format_ctx, m_path, NULL, NULL)

-------------------------------------------------------

// 创建
m_codec_ctx = avcodec_alloc_context3(m_codec);
//初始化具体内容
avcodec_parameters_to_context(m_codec_ctx, codecPar);
```

2. Point 4 about comments in code

We know that audio and video data are usually encapsulated in different tracks, so in order to obtain the correct audio and video data, it is necessary to obtain the corresponding index first.

The data type of audio and video is `GetMediaType()` obtained . The specific implementation is in the subclass, which are:

video: `AVMediaType.AVMEDIA_TYPE_VIDEO`

Audio: `AVMediaType.AVMEDIA_TYPE_AUDIO`

Create data structures to be decoded and decoded

```
// base_decoder.cpp

void BaseDecoder::AllocFrameBuffer() {
    // 初始化待解码和解码数据结构
    // 1）初始化AVPacket，存放解码前的数据
    m_packet = av_packet_alloc();
    // 2）初始化AVFrame，存放解码后的数据
    m_frame = av_frame_alloc();
}
```

Very simple, memory is allocated by two methods for later decoding.

decode loop

```
// base_decoder.cpp

void BaseDecoder::LoopDecode() {
    if (STOP == m_state) { // 如果已被外部改变状态，维持外部配置
        m_state = START;
    }

    LOG_INFO(TAG, LogSpec(), "Start loop decode")
    while(1) {
        if (m_state != DECODING &&
            m_state != START &&
            m_state != STOP) {
            Wait();
            // 恢复同步起始时间，去除等待流失的时间
            m_started_t = GetCurMsTime() - m_cur_t_s;
        }

        if (m_state == STOP) {
            break;
        }

        if (-1 == m_started_t) {
            m_started_t = GetCurMsTime();
        }

        if (DecodeOneFrame() != NULL) {
            SyncRender();
            Render(m_frame);

            if (m_state == START) {
                m_state = PAUSE;
            }
        } else {
            LOG_INFO(TAG, LogSpec(), "m_state = %d" ,m_state)
            if (ForSynthesizer()) {
                m_state = STOP;
            } else {
                m_state = FINISH;
            }
        }
    }
}
```

It can be seen that an `while` infinite , which integrates some time synchronization codes. The synchronization logic is described in detail in the previous hard solution article. For details, please refer to Audio and Video Synchronization .

Without further elaboration, here is only one of the most important methods: `DecodeOneFrame()` .

decode a frame of data

Before looking at the specific code, let's take a look `FFmpeg` at how to implement decoding. There are three methods:

++av_read_frame(m_format_ctx, m_packet)++ :

`m_format_ctx` Read a frame of unpacked data to be decoded from , and store it `m_packet` in ;

++avcodec_send_packet(m_codec_ctx, m_packet)++ :

`m_packet` Send to the decoder for decoding, and the decoded data is stored in `m_codec_ctx` ;

++avcodec_receive_frame(m_codec_ctx, m_frame)++ :

Receive a frame of decoded data and store it `m_frame` in .

```
// base_decoder.cpp

AVFrame* BaseDecoder::DecodeOneFrame() {
    int ret = av_read_frame(m_format_ctx, m_packet);
    while (ret == 0) {
        if (m_packet->stream_index == m_stream_index) {
            switch (avcodec_send_packet(m_codec_ctx, m_packet)) {
                case AVERROR_EOF: {
                    av_packet_unref(m_packet);
                    LOG_ERROR(TAG, LogSpec(), "Decode error: %s",
av_err2str(AVERROR_EOF));
                    return NULL; //解码结束
                }
                case AVERROR(EAGAIN):
                    LOG_ERROR(TAG, LogSpec(), "Decode error: %s",
av_err2str(AVERROR(EAGAIN)));
                    break;
                case AVERROR(EINVAL):
                    LOG_ERROR(TAG, LogSpec(), "Decode error: %s",
av_err2str(AVERROR(EINVAL)));
                    break;
                case AVERROR(ENOMEM):
                    LOG_ERROR(TAG, LogSpec(), "Decode error: %s",
av_err2str(AVERROR(ENOMEM)));
                    break;
                default:
                    break;
            }
            int result = avcodec_receive_frame(m_codec_ctx, m_frame);
            if (result == 0) {
                ObtainTimeStamp();
                av_packet_unref(m_packet);
                return m_frame;
            } else {
                LOG_INFO(TAG, LogSpec(), "Receive frame error result: %d",
av_err2str(AVERROR(result)))
            }
        }
        // 释放packet
        av_packet_unref(m_packet);
        ret = av_read_frame(m_format_ctx, m_packet);
    }
    av_packet_unref(m_packet);
    LOGI(TAG, "ret = %d", ret)
    return NULL;
}
```

Knowing the decoding process, the other is actually handling exceptions, such as:

- When decoding needs to wait, the data is sent to the decoder again, and then the data is retrieved;

- An exception occurs in decoding, read the next frame of data, and then continue decoding;
- If decoding is complete, return empty data `NULL`;

Finally, it is very important that when decoding a frame of data, be sure to call to `av_packet_unref(m_packet);` release memory, otherwise it will cause a memory leak.

After decoding, release resources

After decoding, it is necessary to release all `FFmpeg` related resources and close the decoder.

Another point to note is that during initialization, the `jstring` converted file path should also be released, and the global reference should be deleted.

```
// base_deocder.cpp

void BaseDecoder::DoneDecode(JNIEnv *env) {
    LOG_INFO(TAG, LogSpec(), "Decode done and decoder release")
    // 释放缓存
    if (m_packet != NULL) {
        av_packet_free(&m_packet);
    }
    if (m_frame != NULL) {
        av_frame_free(&m_frame);
    }
    // 关闭解码器
    if (m_codec_ctx != NULL) {
        avcodec_close(m_codec_ctx);
        avcodec_free_context(&m_codec_ctx);
    }
    // 关闭输入流
    if (m_format_ctx != NULL) {
        avformat_close_input(&m_format_ctx);
        avformat_free_context(m_format_ctx);
    }
    // 释放转换参数
    if (m_path_ref != NULL && m_path != NULL) {
        env->ReleaseStringUTFChars((jstring) m_path_ref, m_path);
        env->DeleteGlobalRef(m_path_ref);
    }

    // 通知子类释放资源
    Release();
}
```

Above, the basic structure of the decoder is encapsulated, as long as the specified virtual function is inherited and implemented, the video decoding can be realized.

4. Video playback

video decoder

There are two important points to note here:

1. Video data transcoding

We know that after the video is decoded, the data format is the `YUV` same, and it is required when the screen is displayed `RGBA` . Therefore, in the video decoder, a layer of data conversion needs to be done.

`FFmpeg` The tools in are `SwsContext` used, the conversion method is `sws_scale` , and they are all part of `swresampel` the toolkit.

`sws_scale` It can not only realize the conversion of data format, but also scale the width and height of the screen.

2. Declare the renderer

After conversion, the video frame data becomes `RGBA` , and it can be rendered on the mobile phone screen. There are two methods:

- First, the data is directly rendered through the local window, which cannot be used to re-edit the screen.
- Second, through `OpenGL ES` rendering , the editing of the picture can be realized

> This article uses the former, and the `OpenGL ES` rendering method will be explained separately in the following articles.

Create a new directory `src/main/cpp/decoder/video` and create a new video codec `v_decoder` .

look at the header file `v_decoder.h`

```
// base_decoder.cpp

#ifndef LEARNVIDEO_V_DECODER_H
#define LEARNVIDEO_V_DECODER_H

#include "../base_decoder.h"
#include "../../render/video/video_render.h"
#include <jni.h>
#include <android/native_window_jni.h>
#include <android/native_window.h>

extern "C" {
#include <libavutil/imgutils.h>
#include <libswscale/swscale.h>
};

class VideoDecoder : public BaseDecoder {
private:
    const char *TAG = "VideoDecoder";

    //视频数据目标格式
    const AVPixelFormat DST_FORMAT = AV_PIX_FMT_RGBA;

    //存放YUV转换为RGB后的数据
    AVFrame *m_rgb_frame = NULL;

    uint8_t *m_buf_for_rgb_frame = NULL;

    //视频格式转换器
    SwsContext *m_sws_ctx = NULL;

    //视频渲染器
    VideoRender *m_video_render = NULL;

    //显示的目标宽
    int m_dst_w;
    //显示的目标高
    int m_dst_h;

    /**
     * 初始化渲染器
     */
    void InitRender(JNIEnv *env);

    /**
     * 初始化显示器
     * @param env
     */
    void InitBuffer();

    /**
     * 初始化视频数据转换器
```

```
     */
    void InitSws();

public:
    VideoDecoder(JNIEnv *env, jstring path, bool for_synthesizer = false);
    ~VideoDecoder();
    void SetRender(VideoRender *render);

protected:
    AVMediaType GetMediaType() override {
        return AVMEDIA_TYPE_VIDEO;
    }

    /**
     * 是否需要循环解码
     */
    bool NeedLoopDecode() override;

    /**
     * 准备解码环境
     * 注：在解码线程中回调
     * @param env 解码线程绑定的jni环境
     */
    void Prepare(JNIEnv *env) override;

    /**
     * 渲染
     * 注：在解码线程中回调
     * @param frame 解码RGBA数据
     */
    void Render(AVFrame *frame) override;

    /**
     * 释放回调
     */
    void Release() override;

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

#endif //LEARNVIDEO_V_DECODER_H
```

Next, look at the `v_deocder.cpp` implementation , first look at the initialization related code:

```
// v_deocder.cpp

VideoDecoder::VideoDecoder(JNIEnv *env, jstring path, bool for_synthesizer)
: BaseDecoder(env, path, for_synthesizer) {
}

void VideoDecoder::Prepare(JNIEnv *env) {
    InitRender(env);
    InitBuffer();
    InitSws();
}
```

The constructor is very simple, just pass the relevant parameters `base_decoder` to .

Next is the `Prepare` method . This method is the method that the subclass `base_decoder` specified in the parent class must implement. It is called after the decoder is initialized. Review:

```
// base_decoder.cpp

void BaseDecoder::Decode(std::shared_ptr<BaseDecoder> that) {

    // 省略无关代码...

    that->InitFFMpegDecoder(env);
    that->AllocFrameBuffer();

    //子类初始化方法调用
    that->Prepare(env);

    that->LoopDecode();
    that->DoneDecode(env);

    // 省略无关代码...

}
```

`Prepare` In , the initialization of the renderer is `InitRender` skipped first and will be discussed in detail later.

Take a look at the initialization related to data format conversion.

Store data cache initialization:

```
// base_decoder.cpp

void VideoDecoder::InitBuffer() {
    m_rgb_frame = av_frame_alloc();
    // 获取缓存大小
    int numBytes = av_image_get_buffer_size(DST_FORMAT, m_dst_w, m_dst_h, 1);
    // 分配内存
    m_buf_for_rgb_frame = (uint8_t *) av_malloc(numBytes * sizeof(uint8_t));
    // 将内存分配给RgbFrame，并将内存格式化为三个通道后，分别保存其地址
    av_image_fill_arrays(m_rgb_frame->data, m_rgb_frame->linesize,
                         m_buf_for_rgb_frame, DST_FORMAT, m_dst_w, m_dst_h, 1);
}
```

Initialize a block through the `av_frame_alloc` method `AVFrame`, note that this method does not allocate cache memory;

Then calculate the required memory block size by the `av_image_get_buffer_size` method , where

```
AVPixelFormat DST_FORMAT = AV_PIX_FMT_RGBA

m_dst_w: 为目标画面宽度（即画面显示时的实际宽度，将通过后续渲染器中具体的窗户大小计算得出）
m_dst_h：为目标画面高度（即画面显示时的实际高度，将通过后续渲染器中具体的窗户大小计算得出）
```

Then by `av_malloc` **actually** allocating a piece of memory ;

Finally, `av_image_fill_arrays` by giving the obtained memory `AVFrame` , at this point, the memory allocation is completed.

Data conversion tool initialization

```
// base_decoder.cpp

void VideoDecoder::InitSws() {
    // 初始化格式转换工具
    m_sws_ctx = sws_getContext(width(), height(), video_pixel_format(),
                               m_dst_w, m_dst_h, DST_FORMAT,
                               SWS_FAST_BILINEAR, NULL, NULL, NULL);
}
```

This is very simple, as long as the length, width, format, etc. of the original picture data and the target picture data are passed in.

release related resources

After decoding, the parent class will call the subclass `Release` method to release the related resources in the subclass.

```
// v_deocder.cpp

void VideoDecoder::Release() {
    LOGE(TAG, "[VIDEO] release")
    if (m_rgb_frame != NULL) {
        av_frame_free(&m_rgb_frame);
        m_rgb_frame = NULL;
    }
    if (m_buf_for_rgb_frame != NULL) {
        free(m_buf_for_rgb_frame);
        m_buf_for_rgb_frame = NULL;
    }
    if (m_sws_ctx != NULL) {
        sws_freeContext(m_sws_ctx);
        m_sws_ctx = NULL;
    }
    if (m_video_render != NULL) {
        m_video_render->ReleaseRender();
        m_video_render = NULL;
    }
}
```

Initialization and resource release are complete, leaving the final renderer configuration.

Renderer

As I said above, there are generally two ways to render the picture, so the renderer should be defined first to facilitate later expansion.

Define the video renderer

Create a new directory `src/main/cpp/media/render/video` and create a header file `video_render.h`.

```
#ifndef LEARNVIDEO_VIDEORENDER_H
#define LEARNVIDEO_VIDEORENDER_H

#include <stdint.h>
#include <jni.h>

#include "../../one_frame.h"

class VideoRender {
public:
    virtual void InitRender(JNIEnv *env, int video_width, int video_height, int
*dst_size) = 0;
    virtual void Render(OneFrame *one_frame) = 0;
    virtual void ReleaseRender() = 0;
};

#endif //LEARNVIDEO_VIDEORENDER_H
```

This class is also a pure virtual class, similar `Java` to `interface` .

There are only a few interfaces specified here, namely initialization, rendering, and resource release.

Implement a native window renderer

Create a new directory `src/main/cpp/media/render/video/native_render` and create a header `native_render` class .

`native_render` head File:

```
// native_render.h

#ifndef LEARNVIDEO_NATIVE_RENDER_H
#define LEARNVIDEO_NATIVE_RENDER_H

#include <android/native_window.h>
#include <android/native_window_jni.h>
#include <jni.h>

#include "../video_render.h"
#include "../../../../utils/logger.h"

extern "C" {
#include <libavutil/mem.h>
};

class NativeRender: public VideoRender {
private:
    const char *TAG = "NativeRender";

    // Surface引用，必须使用引用，否则无法在线程中操作
    jobject m_surface_ref = NULL;

    // 存放输出到屏幕的缓存数据
    ANativeWindow_Buffer m_out_buffer;

    // 本地窗口
    ANativeWindow *m_native_window = NULL;

    //显示的目标宽
    int m_dst_w;

    //显示的目标高
    int m_dst_h;

public:
    NativeRender(JNIEnv *env, jobject surface);
    ~NativeRender();
    void InitRender(JNIEnv *env, int video_width, int video_height, int *dst_size)
override ;
    void Render(OneFrame *one_frame) override ;
    void ReleaseRender() override ;
};
```

As you can see, the renderer holds a `Surface` reference , which is something we are very familiar with. In the previous series of articles, it was used for screen rendering.

Another is the local window `ANativeWindow` , as long as **you `Surface` bind to `ANativeWindow` , you can achieve `Surface` rendering** .

Take a look at the implementation of the renderer `native_render.cpp` .

initialization

```
// native_render.cpp

ativeRender::NativeRender(JNIEnv *env, jobject surface) {
    m_surface_ref = env->NewGlobalRef(surface);
}

NativeRender::~NativeRender() {

}

void NativeRender::InitRender(JNIEnv *env, int video_width, int video_height, int
*dst_size) {
    // 初始化窗口
    m_native_window = ANativeWindow_fromSurface(env, m_surface_ref);

    // 绘制区域的宽高
    int windowWidth = ANativeWindow_getWidth(m_native_window);
    int windowHeight = ANativeWindow_getHeight(m_native_window);

    // 计算目标视频的宽高
    m_dst_w = windowWidth;
    m_dst_h = m_dst_w * video_height / video_width;
    if (m_dst_h > windowHeight) {
        m_dst_h = windowHeight;
        m_dst_w = windowHeight * video_width / video_height;
    }
    LOGE(TAG, "windowW: %d, windowH: %d, dstVideoW: %d, dstVideoH: %d",
         windowWidth, windowHeight, m_dst_w, m_dst_h)

    //设置宽高限制缓冲区中的像素数量
    ANativeWindow_setBuffersGeometry(m_native_window, windowWidth,
            windowHeight, WINDOW_FORMAT_RGBA_8888);

    dst_size[0] = m_dst_w;
    dst_size[1] = m_dst_h;
}
```

Focus on the `InitRender` method :

`ANativeWindow_fromSurface` By `Surface` binding to the local window;

The width and height of the displayable area `ANativeWindow_getWidth` `ANativeWindow_getHeight` can be obtained through ; `Surface`

Then, according to the width and height of the original video picture and the width and height `video_width` `video_height` of the realizable area, the picture is scaled, and the width and height of the final displayed picture can be calculated and assigned to the decoder.

> `v_decoder` After the video decoder obtains the width and height of the target image, it can initialize the size of the data conversion buffer.

Finally, complete the initialization by `ANativeWindow_setBuffersGeometry` setting the size of the local window buffer.

render

Two important native methods:

`ANativeWindow_lock` Lock the window and get to the output buffer `m_out_buffer` .

`ANativeWindow_unlockAndPost` Release the window and draw the buffered data to the screen.

```
// native_render.cpp

void NativeRender::Render(OneFrame *one_frame) {
    //锁定窗口
    ANativeWindow_lock(m_native_window, &m_out_buffer, NULL);
    uint8_t *dst = (uint8_t *) m_out_buffer.bits;
    // 获取stride：一行可以保存的内存像素数量*4（即：rgba的位数）
    int dstStride = m_out_buffer.stride * 4;
    int srcStride = one_frame->line_size;

    // 由于window的stride和帧的stride不同，因此需要逐行复制
    for (int h = 0; h < m_dst_h; h++) {
        memcpy(dst + h * dstStride, one_frame->data + h * srcStride, srcStride);
    }
    //释放窗口
    ANativeWindow_unlockAndPost(m_native_window);
}
```

The rendering process looks complicated, mainly because there is `stride` a concept of , which refers to the width of each line of data in a frame.

> For example, the data format here is `RGBA` that the pixels of one line of picture are 8, then the total `stride` width is 8*4 = 32.
> Why do you need to convert? The reason is that the `stride` size may be `stride` inconsistent with the video image data. When the video image data is directly sent to the local window, it may lead to inconsistent data reading and eventually lead to a blurry screen.

Therefore, it is necessary to copy the data line by line (memcpy) according to the local window `dstStride` and video screen data. `srcStride`

renderer call

Finally, let's take a look at the call to the renderer `v_decoder` in

```
// v_decoder.cpp

void VideoDecoder::SetRender(VideoRender *render) {
    this->m_video_render = render;
}

void VideoDecoder::InitRender(JNIEnv *env) {
    if (m_video_render != NULL) {
        int dst_size[2] = {-1, -1};
        m_video_render->InitRender(env, width(), height(), dst_size);

        m_dst_w = dst_size[0];
        m_dst_h = dst_size[1];
        if (m_dst_w == -1) {
            m_dst_w = width();
        }
        if (m_dst_h == -1) {
            m_dst_w = height();
        }
        LOGI(TAG, "dst %d, %d", m_dst_w, m_dst_h)
    } else {
        LOGE(TAG, "Init render error, you should call SetRender first!")
    }
}

void VideoDecoder::Render(AVFrame *frame) {
    sws_scale(m_sws_ctx, frame->data, frame->linesize, 0,
              height(), m_rgb_frame->data, m_rgb_frame->linesize);
    OneFrame * one_frame = new OneFrame(m_rgb_frame->data[0], m_rgb_frame-
>linesize[0], frame->pts, time_base(), NULL, false);
    m_video_render->Render(one_frame);
}
```

First, set the rendering to the video decoder;

The second is to call the renderer's `InitRender` method to initialize the renderer and obtain the width and height of the target screen

Finally, call the renderer `Render` method to render.

Among them, it `OneFrame` is a custom class, which is used to encapsulate the content related to a frame of data .

write player

Above, completed:

1. `基础解码器` The package --> `视频解码器` implementation;
2. `渲染器` Definition --> `本地渲染窗口` Implementation.

In the end, it was necessary to integrate them together and realize playback.

Create a new player in the `src/main/cpp/media` directory `player`, as follows:

```
// player.h

#ifndef LEARNINGVIDEO_PLAYER_H
#define LEARNINGVIDEO_PLAYER_H


#include "decoder/video/v_decoder.h"

class Player {
private:
    VideoDecoder *m_v_decoder;
    VideoRender *m_v_render;

public:
    Player(JNIEnv *jniEnv, jstring path, jobject surface);
    ~Player();

    void play();
    void pause();
};

#endif //LEARNINGVIDEO_PLAYER_H
```

The player holds a video decoder and a video renderer, and a play and pause method.

```
// player.cpp

#include "player.h"
#include "render/video/native_render/native_render.h"

Player::Player(JNIEnv *jniEnv, jstring path, jobject surface) {
    m_v_decoder = new VideoDecoder(jniEnv, path);
    m_v_render = new NativeRender(jniEnv, surface);
    m_v_decoder->SetRender(m_v_render);
}

Player::~Player() {
    // 此处不需要 delete 成员指针
    // 在BaseDecoder中的线程已经使用智能指针，会自动释放
}

void Player::play() {
    if (m_v_decoder != NULL) {
        m_v_decoder->GoOn();
    }
}

void Player::pause() {
    if (m_v_decoder != NULL) {
        m_v_decoder->Pause();
    }
}
```

The code is very simple, which is to associate the decoder with the renderer.

Add the source code to the compilation

Although the writing of each function module has been completed above, the compiler will not automatically add them to the compilation. If you want to add the `C++` code to the compilation, you need to manually configure it in the `CMakeLists.txt` file. The location of the configuration is the `native-lib.cpp` same as , and you can list it later.

```
# CMakeLists.txt

// 省略无关配置
//......

# 配置目标so库编译信息
add_library( # Sets the name of the library.
        native-lib

        # Sets the library as a shared library.
        SHARED

        # Provides a relative path to your source file(s).
        native-lib.cpp

        # 工具
        ${CMAKE_SOURCE_DIR}/utils/logger.h
        ${CMAKE_SOURCE_DIR}/utils/timer.c

        # 播放器
        ${CMAKE_SOURCE_DIR}/media//player.cpp

        # 解码器
        ${CMAKE_SOURCE_DIR}/media//one_frame.h
        ${CMAKE_SOURCE_DIR}/media/decoder/i_decoder.h
        ${CMAKE_SOURCE_DIR}/media/decoder/decode_state.h
        ${CMAKE_SOURCE_DIR}/media/decoder/base_decoder.cpp
        ${CMAKE_SOURCE_DIR}/media/decoder/video/v_decoder.cpp

        # 渲染器
        ${CMAKE_SOURCE_DIR}/media/render/video/video_render.h
        ${CMAKE_SOURCE_DIR}/media/render/video/native_render/native_render.cpp
        )

// 省略无关配置
//......
```

> If the class only has a `.h` header file, only write the `.h` file. If the class has both a header file and `.cpp` an implementation file, only the configuration `.cpp` file is required

It should be noted that when each class is created, it needs to be configured `CMakeLists.txt` in , otherwise, when writing code, it may not be able to import the relevant library header files, and it will not pass the compilation.

Writing the JNI interface

Next, you need to expose the player to the `Java` layer , and then you need to use the `JNI` interface file . `native-lib.cpp`

Before starting to write an `JNI` interface `FFmpegActivity`, write the corresponding interface in :

```
// FFmpegActivity.kt

class FFmpegActivity: AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_ffmpeg_info)
        tv.text = ffmpegInfo()
        initSfv()
    }

    private fun initSfv() {
        sfv.holder.addCallback(object: SurfaceHolder.Callback {
            override fun surfaceChanged(holder: SurfaceHolder, format: Int, width:
Int, height: Int) {

            }

            override fun surfaceDestroyed(holder: SurfaceHolder) {
            }

            override fun surfaceCreated(holder: SurfaceHolder) {
                if (player == null) {
                    player = createPlayer(path, holder.surface)
                    play(player!!)
                }
            }
        })
    }

//------------ JNI 相关接口方法 ----------------------

    private external fun ffmpegInfo(): String

    private external fun createPlayer(path: String, surface: Surface): Int

    private external fun play(player: Int)

    private external fun pause(player: Int)

    companion object {
        init {
            System.loadLibrary("native-lib")
        }
    }
}
```

The interface is simple:

createPlayer(path: String, surface: Surface): Int: Create a player and return the address of the player object

play(player: Int): Play, the parameter is the player object

pause(player: Int): Pause, the parameter is the player object

The creation timing of the player is When `SurfaceView` initialization completes: `surfaceCreated`.

The page layout `xml` is as follows :

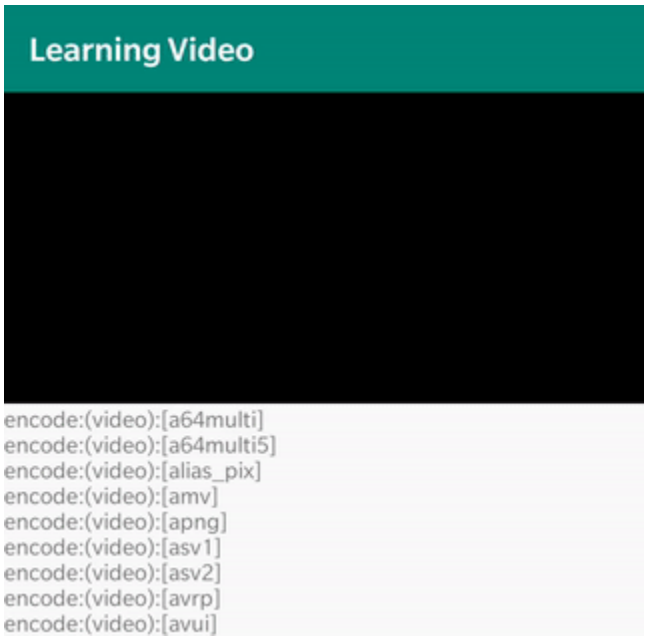
```
<android.support.constraint.ConstraintLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent" android:layout_height="match_parent">
    <ScrollView
            android:layout_width="match_parent"
            android:layout_height="match_parent">
        <LinearLayout
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:orientation="vertical">
            <SurfaceView android:id="@+id/sfv"
                    android:layout_width="match_parent"
                    android:layout_height="200dp" />
            <TextView android:id="@+id/tv"
                    android:layout_width="match_parent"
                    android:layout_height="match_parent"/>
        </LinearLayout>
    </ScrollView>
</android.support.constraint.ConstraintLayout>
```

Next, according to the above three interfaces `JNI`, write the corresponding interface in .

```
// native-lib.cpp

#include <jni.h>
#include <string>
#include <unistd.h>
#include "media/player.h"

extern "C" {

    JNIEXPORT jint JNICALL
    Java_com_cxp_learningvideo_FFmpegActivity_createPlayer(JNIEnv *env,
            jobject  /* this */,
            jstring path,
            jobject surface) {
            Player *player = new Player(env, path, surface);
            return (jint) player;
        }

    JNIEXPORT void JNICALL
    Java_com_cxp_learningvideo_FFmpegActivity_play(JNIEnv *env,
                                                   jobject  /* this */,
                                                   jint player) {
        Player *p = (Player *) player;
        p->play();
    }

    JNIEXPORT void JNICALL
    Java_com_cxp_learningvideo_FFmpegActivity_pause(JNIEnv *env,
                                                   jobject  /* this */,
                                                   jint player) {
        Player *p = (Player *) player;
        p->pause();
    }
}
```

It's very simple, I believe everyone can understand, in fact, it is to initialize a player object pointer, and then return it to the `Java` layer save it. The subsequent play and pause operations are `Java` the layer to pass the player pointer to the `JNI` layer for specific operations.

play video



V. Summary

There is a lot of code, but in fact, if you have read the previous series of articles on native hard solutions, it should be easier to understand.

Finally, a brief summary:

- Initialization: Initialize the decoder according to some functional interfaces `FFmpeg` provided by
    - Input file stream context AVFormatContext
    - Decoder context AVCodecContext
    - Decoder AVCodec
    - Allocate data buffer space AVPacket (store data to be decoded) and AVFrame (store decoded data)
- Decoding: Decode through the decoding interface `FFmpeg` provided by
    - av_read_frame reads the data to be decoded into AVPacket
    - avcodec_send_packet send AVPacket to decoder for decoding
    - avcodec_receive_frame reads the decoded data into AVFrame
- Transcoding and scaling: Convert YUV to RGBA via the transcoding interface `FFmpeg` provided by
    - sws_getContext initializes the conversion tool SwsContext
    - sws_scale performs data transformation

- Rendering: Render video data to the screen through the interface `Android` provided by
    - ANativeWindow_fromSurface binds the Surface to the local window
    - ANativeWindow_getWidth/ANativeWindow_getWidth Get Surface width and height
    - ANativeWindow_setBuffersGeometry sets the screen buffer size
    - ANativeWindow_lock locks the window and gets the display buffer
    - `Stride` Copy (memcpy) data to buffer according to
    - ANativeWindow_unlockAndPost unlocks the window and displays it