[Android audio and video development and upgrade: FFmpeg audio and video codec] Fourth, Android FFmpeg + OpenSL ES audio decoding playback

jianshu.com/p/28fc978721b4

In this article you can learn

> This article introduces how to use `FFmpeg` for audio decoding, focusing on how to use `OpenSL ES` to achieve audio rendering and playback in the DNK layer.

1. Audio decoding

In the previous article , I introduced `FFmepg` the playback process of , and abstracted the decoding process framework, integrated the common points of the video and audio decoding processes, and formed the `BaseDecoder` class. `BaseDecoder` The video decoding subclass is implemented by inheriting from `VideoDeocder` and integrated into `Player` to realize the playback and rendering of the video.

This article uses the already defined decoding base class `BaseDecoder` to implement the audio decoding subclass `AudioDecoder` .

Implement the audio decoding subclass

First, let's see what needs to be implemented to implement audio decoding.

Define the decoding process

`a_decoder.h` We define the required member variables and process methods through the header file .

i. Member variable definition

```
//a_decoder.h

class AudioDecoder: public BaseDecoder {
private:

    const char *TAG = "AudioDecoder";

    // 音频转换器
    SwrContext *m_swr = NULL;

    // 音频渲染器
    AudioRender *m_render = NULL;

    // 输出缓冲
    uint8_t *m_out_buffer[1] = {NULL};

    // 重采样后，每个通道包含的采样数
    // acc默认为1024，重采样后可能会变化
    int m_dest_nb_sample = 1024;

    // 重采样以后，一帧数据的大小
    size_t m_dest_data_size = 0;

    //......
}
```

Among them, `SwrContext` is the audio conversion tool `FFmpeg` provided by , located `swresample` in , which can be used to convert the sample rate, the number of decoding channels, the number of sample bits, etc. Used here to convert audio data to `双通道立体` sound , unity `采样位数` .

`AudioRender` is a custom audio renderer, which will be introduced later.

Other variables need to be used together in audio conversion, such as conversion output buffer, buffer size, and number of samples.

ii. Define member methods

```
//a_decoder.h

class AudioDecoder: public BaseDecoder {
private:

    // 省略成员变量......

    /**
     * 初始化转换工具
     */
    void InitSwr();

    /**
     * 初始化输出缓冲
     */
    void InitOutBuffer();

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

    /**
     * 释放缓冲区
     */
    void ReleaseOutBuffer();

    /**
     * 采样格式：16位
     */
    AVSampleFormat GetSampleFmt() {
        return AV_SAMPLE_FMT_S16;
    }

    /**
     * 目标采样率
     */
    int GetSampleRate(int spr) {
        return AUDIO_DEST_SAMPLE_RATE; //44100Hz
    }

public:
    AudioDecoder(JNIEnv *env, const jstring path, bool forSynthesizer);
    ~AudioDecoder();

    void SetRender(AudioRender *render);

protected:
    void Prepare(JNIEnv *env) override;
    void Render(AVFrame *frame) override;
    void Release() override;
```

```
    bool NeedLoopDecode() override {
        return true;
    }

    AVMediaType GetMediaType() override {
        return AVMEDIA_TYPE_AUDIO;
    }

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

The above code is not complicated, it is some initialization related methods, as well `BaseDecoder` as the implementation of the abstract methods defined in .

Focus on these two methods:

```
/**
 * 采样格式：16位
 */
AVSampleFormat GetSampleFmt() {
    return AV_SAMPLE_FMT_S16;
}

/**
 * 目标采样率
 */
int GetSampleRate(int spr) {
    return AUDIO_DEST_SAMPLE_RATE; //44100Hz
}
```

The first thing to know is that the purpose of these two methods is to be compatible with future encodings.

We know that the sampling rate and sampling bits of the audio are unique to the audio data, and each audio may be different, so when playing or re-encoding, the data is usually converted to a fixed specification, so that it can be played normally or Recode.

> 播放和编码的配置也稍有不同，这里，采样位数是 16 位，采样率使用 44100 。

接下来，看看具体的实现。

实现解码流程

```
// a_decoder.cpp

AudioDecoder::AudioDecoder(JNIEnv *env, const jstring path, bool forSynthesizer) :
BaseDecoder(
        env, path, forSynthesizer) {

}

void AudioDecoder::~AudioDecoder() {
    if (m_render != NULL) {
        delete m_render;
    }
}

void AudioDecoder::SetRender(AudioRender *render) {
    m_render = render;
}

void AudioDecoder::Prepare(JNIEnv *env) {
    InitSwr();
    InitOutBuffer();
    InitRender();
}

//省略其他....
```

i. 初始化

重点看 `Prepare` 方法，这个方法在基类 `BaseDecoder` 初始化完解码器以后，就会调用。

在 `Prepare` 方法中，依次调用了：

`InitSwr()`，初始化转换器

`InitOutBuffer()`，初始化输出缓冲

`InitRender()`，初始化渲染器

下面具体解析如何配置初始化参数。

`SwrContext` 配置：

```
// a_decoder.cpp

void AudioDecoder::InitSwr() {

    // codec_cxt() 为解码上下文，从父类 BaseDecoder 中获取
    AVCodecContext *codeCtx = codec_cxt();

    //初始化格式转换工具
    m_swr = swr_alloc();

    // 配置输入/输出通道类型
    av_opt_set_int(m_swr, "in_channel_layout",  codeCtx->channel_layout, 0);

    // 这里 AUDIO_DEST_CHANNEL_LAYOUT = AV_CH_LAYOUT_STEREO，即 立体声
    av_opt_set_int(m_swr, "out_channel_layout", AUDIO_DEST_CHANNEL_LAYOUT, 0);

    // 配置输入/输出采样率
    av_opt_set_int(m_swr, "in_sample_rate", codeCtx->sample_rate, 0);
    av_opt_set_int(m_swr, "out_sample_rate", GetSampleRate(codeCtx->sample_rate), 0);

    // 配置输入/输出数据格式
    av_opt_set_sample_fmt(m_swr, "in_sample_fmt",  codeCtx->sample_fmt, 0);
    av_opt_set_sample_fmt(m_swr, "out_sample_fmt", GetSampleFmt(),  0);

    swr_init(m_swr);
}
```

初始化很简单，首先调用 FFmpeg 的 `swr_alloc` 方法，分配内存，得到一个转化工具 `m_swr` ，接着调用对应的方法，设置输入和输出的音频数据参数。

输入输出参数的设置，也可通过一个统一的方法 `swr_alloc_set_opts` 设置，具体可以参看该接口注释。

输出缓冲配置：

```
// a_decoder.cpp

void AudioDecoder::InitOutBuffer() {
    // 重采样后一个通道采样数
    m_dest_nb_sample = (int)av_rescale_rnd(ACC_NB_SAMPLES, GetSampleRate(codec_cxt()->sample_rate),
                                           codec_cxt()->sample_rate, AV_ROUND_UP);
    // 重采样后一帧数据的大小
    m_dest_data_size = (size_t)av_samples_get_buffer_size(
            NULL, AUDIO_DEST_CHANNEL_COUNTS,
            m_dest_nb_sample, GetSampleFmt(), 1);

    m_out_buffer[0] = (uint8_t *) malloc(m_dest_data_size);
}

void AudioDecoder::InitRender() {
    m_render->InitRender();
}
```

在转换音频数据之前，我们需要一个数据缓冲区来存储转换后的数据，因此需要知道转换后的音频数据有多大，并以此来分配缓冲区。

影响数据缓冲大小的因素有三个，分别是：`采样个数`、`通道数`、`采样位数`。

采样个数计算

我们知道 `AAC` 一帧数据包含采样个数是 1024 个。如果对一帧音频数据进行重采样的话，那么采样个数就会发生变化。

如果采样率变大，那么采样个数会变多；采样率变小，则采样个数变少。并且成比例关系。

计算方式如下：【`目标采样个数` = `原采样个数` * (`目标采样率` / `原采样率`)】

`FFmpeg` 提供了 `av_rescale_rnd` 用于计算这种缩放关系，优化了计算益处问题。

`FFmpeg` 提供了 `av_samples_get_buffer_size` 方法来帮我们计算这个缓存区的大小。只需提供计算出来的`目标采样个数`、`通道数`、`采样位数`。

得到缓存大小后，通过 `malloc` 分配内存。

ii. 渲染

```
// a_decoder.cpp

void AudioDecoder::Render(AVFrame *frame) {
    // 转换，返回每个通道的样本数
    int ret = swr_convert(m_swr, m_out_buffer, m_dest_data_size/2,
                          (const uint8_t **) frame->data, frame->nb_samples);
    if (ret > 0) {
        m_render->Render(m_out_buffer[0], (size_t) m_dest_data_size);
    }
}
```

父类 `BaseDecoder` 解码数据后，回调子类渲染方法 `Render` 。在渲染之前，调用 `swr_convert` 方法，转换音频数据。

以下为接口原型：

```
/**
 * out：输出缓冲区
 * out_count：输出数据单通道采样个数
 * in：待转换原音频数据
 * in_count：原音频单通道采样个数
 */
int swr_convert(struct SwrContext *s, uint8_t **out, int out_count,
                                const uint8_t **in , int in_count);
```

最后调用渲染器 `m_render` 渲染播放。

iii.释放资源

```
// a_decoder.cpp

void AudioDecoder::Release() {
    if (m_swr != NULL) {
        swr_free(&m_swr);
    }
    if (m_render != NULL) {
        m_render->ReleaseRender();
    }
    ReleaseOutBuffer();
}

void AudioDecoder::ReleaseOutBuffer() {
    if (m_out_buffer[0] != NULL) {
        free(m_out_buffer[0]);
        m_out_buffer[0] = NULL;
    }
}
```

解码完毕，退出播放的时候，需要将转换器、输出缓冲区释放。

二、接入 OpenSL ES

在 `Android` 上播放音频，通常使用的 `AudioTrack` ，但是在 `NDK` 层，没有提供直接的类，需要通过 `NDK` 调用 `Java` 层的方式，回调实现播放。相对来说比较麻烦，效率也比较低。

在 `NDK` 层，提供另一种播放音频的方法： `OpenSL ES` 。

什么是 OpenSL ES

> OpenSL ES (Open Sound Library for Embedded Systems)是无授权费、跨平台、针对嵌入式系统精心优化的硬件音频加速API。它为嵌入式移动多媒体设备上的本地应用程序开发者提供标准化，高性能，低响应时间的音频功能实现方法，并实现软/硬件音频性能的直接跨平台部署，降低执行难度。

OpenSL ES 提供哪些功能

OpenSL 主要提供了录制和播放的功能，本文主讲播放功能。

播放源支持 `PCM` 、 `sdcard资源` 、 `res/assets资源` 、 `网络资源` 。

我们使用的 `FFmpeg` 解码，所以播放源是 `PCM` 。

OpenSL ES 状态机

OpenSL ES 是基于 `C` 语言开发的库，但是其接口是使用了面向对象的编程思想编写的，它的接口不能直接调用，而是要经过对象创建、初始化后，通过对象来调用。

Object 和 Interface

OpenSL ES 提供了一系列 `Object` ，它们拥有一些基础操作方法，比如 Realize，Resume，GetState，Destroy，GetInterface等。

> 一个 `Object` 拥有一个或多个 `Interface` 方法，但是一个 `Intefcace` 只属于一个 `Obejct` 。

想要调用 `Object` 中的 `Interface` 方法，必须要通过 `Object` 的 `GetInterface` 先获取到接口 `Interface` ，再通过获取到的 `Interface` 来调用。

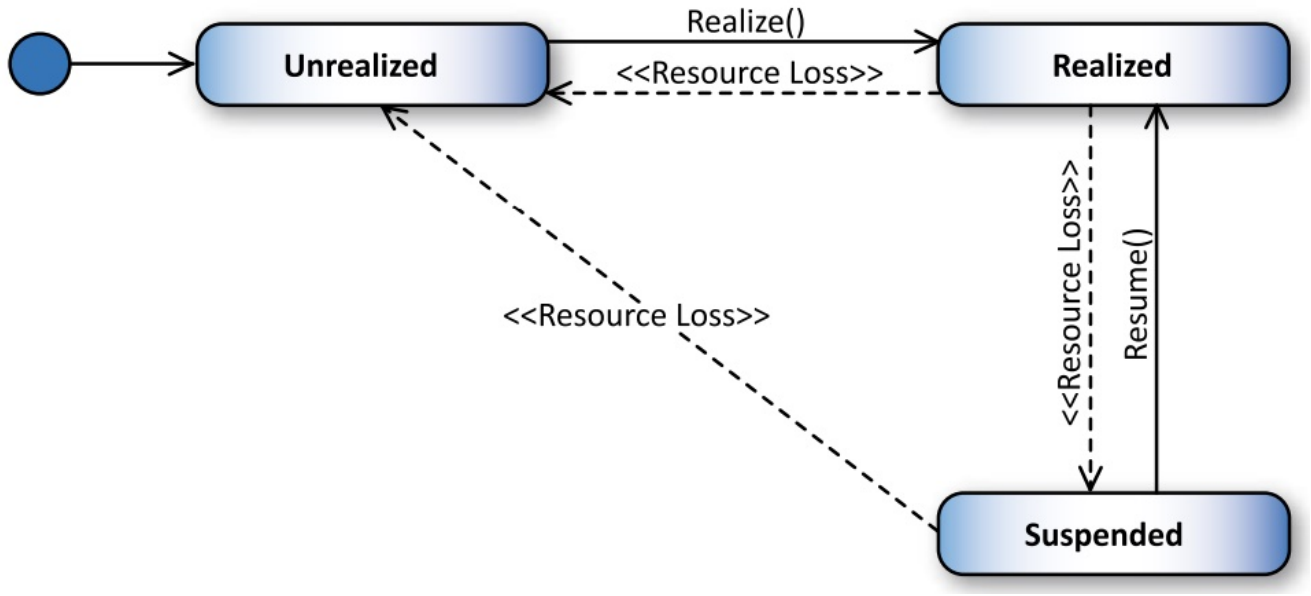
比如：

```
// 创建引擎
SLObjectItf m_engine_obj = NULL;
SLresult result = slCreateEngine(&m_engine_obj, 0, NULL, 0, NULL, NULL);

// 初始化引擎
result = (*m_engine_obj)->Realize(m_engine_obj, SL_BOOLEAN_FALSE);

// 获取引擎接口
SLEngineItf m_engine = NULL;
result = (*m_engine_obj)->GetInterface(m_engine_obj, SL_IID_ENGINE, &m_engine);
```

可以看到，`Object` 需要经过创建、初始化之后才能使用，这就是 `OpenSL ES` 中的状态机机制。

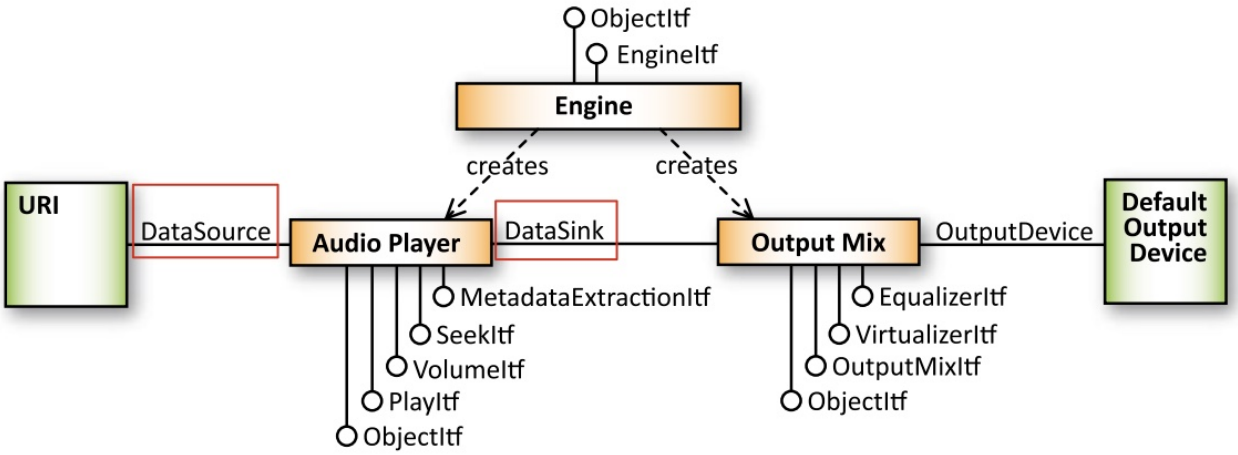


OpenSL ES 状态机

> `Object` 被创建后，进入 `Unrealized` 状态，调用 `Realize()` 方法以后会分配相关的内存资源，进入 `Realized` 状态，这时 `Object` 的 `Interface` 方法才能被获取和使用。
>
> 在后续执行过程中，如果出现错误，`Object` 会进入 `Suspended` 状态。调用 `Resume()` 可以恢复到 `Realized` 状态。

OpenSL ES 播放初始化配置

来看一张官方的播放流程图



OpenSL ES 播放流程

这张图非常清晰的展示了 `OpenSL ES` 是如何运作的。

`OpenSL ES` 播放需要的两个核心是 `Audio Player` 和 `Output Mix` ，即 `播放起` 和 `混音器` ，而这两个都是由 `OpenSL ES` 的引擎 `Engine` 创建（creates）出来的。

所以，整个初始化流程可以总结为：

> 通过 Engine 创建 `Output Mix/混音器` ，并将 `混音器` 作为参数，在创建 `Audio Player/播放器` 时，绑定给 `Audio Player` 作为输出。

DataSource 和 DataSink

在创建 `Audio Player` 的时候，需要给其设置 `数据源` 和 `输出目标` ，这样播放器才知道，如何获取播放数据、将数据输出到哪里进行播放。

这就需要用到 `OpenSL ES` 的两个结构体 `DataSource` 和 `DataSink` 。

```
typedef struct SLDataSource_ {
    void *pLocator;
    void *pFormat;
} SLDataSource;

typedef struct SLDataSink_ {
    void *pLocator;
    void *pFormat;
} SLDataSink;
```

其中，
SLDataSource pLocator 有以下几种类型：

```
SLDataLocator_Address
SLDataLocator_BufferQueue
SLDataLocator_IODevice
SLDataLocator_MIDIBufferQueue
SLDataLocator_URI
```

播放 `PCM` 使用的是 `SLDataLocator_BufferQueue` 缓冲队列。

SLDataSink pLocator 一般为 `SL_DATALOCATOR_OUTPUTMIX` 。

另外一个参数 `pFormat` 为数据的格式。

实现渲染流程

在接入 `OpenSL ES` 之前，先定义好上文提到的音频渲染接口，方便规范和拓展。

```
// audio_render.h

class AudioRender {
public:
    virtual void InitRender() = 0;
    virtual void Render(uint8_t *pcm, int size) = 0;
    virtual void ReleaseRender() = 0;
    virtual ~AudioRender() {}
};
```

在 `CMakeList.txt` 中，打开 `OpenSL ES` 支持

```
# CMakeList.txt

# 省略其他...

# 指定编译目标库时，cmake要链接的库
target_link_libraries(

        native-lib

        avutil
        swresample
        avcodec
        avfilter
        swscale
        avformat
        avdevice

        -landroid

        # 打开opensl es支持
        OpenSLES

        # Links the target library to the log library
        # included in the NDK.
        ${log-lib} )
```

初始化

i. 定义成员变量

先定义需要用到的引擎、混音器、播放器、以及缓冲队列接口、音量调节接口等。

```
// opensl_render.h

class OpenSLRender: public AudioRender {

private:

    // 引擎接口
    SLObjectItf m_engine_obj = NULL;
    SLEngineItf m_engine = NULL;

    //混音器
    SLObjectItf m_output_mix_obj = NULL;
    SLEnvironmentalReverbItf m_output_mix_evn_reverb = NULL;
    SLEnvironmentalReverbSettings m_reverb_settings =
SL_I3DL2_ENVIRONMENT_PRESET_DEFAULT;

    //pcm播放器
    SLObjectItf m_pcm_player_obj = NULL;
    SLPlayItf m_pcm_player = NULL;
    SLVolumeItf m_pcm_player_volume = NULL;

    //缓冲器队列接口
    SLAndroidSimpleBufferQueueItf m_pcm_buffer;

    //省略其他......
}
```

ii. 定义相关成员方法

```
// opensl_render.h

class OpenSLRender: public AudioRender {

private:

    // 省略成员变量...

    // 创建引擎
    bool CreateEngine();

    // 创建混音器
    bool CreateOutputMixer();

    // 创建播放器
    bool CreatePlayer();

    // 开始播放渲染
    void StartRender();

    // 音频数据压入缓冲队列
    void BlockEnqueue();

    // 检查是否发生错误
    bool CheckError(SLresult result, std::string hint);

    // 数据填充通知接口，后续会介绍这个方法的作用
    void static sReadPcmBufferCbFun(SLAndroidSimpleBufferQueueItf bufferQueueItf,
void *context);

public:
    OpenSLRender();
    ~OpenSLRender();

    void InitRender() override;
    void Render(uint8_t *pcm, int size) override;
    void ReleaseRender() override;
```

iii. 实现初始化流程

```
// opensl_render.cpp

OpenSLRender::OpenSLRender() {
}

OpenSLRender::~OpenSLRender() {
}

void OpenSLRender::InitRender() {
    if (!CreateEngine()) return;
    if (!CreateOutputMixer()) return;
    if (!CreatePlayer()) return;
}

// 省略其他......
```

创建引擎

```
// opensl_render.cpp

bool OpenSLRender::CreateEngine() {
    SLresult result = slCreateEngine(&m_engine_obj, 0, NULL, 0, NULL, NULL);
    if (CheckError(result, "Engine")) return false;

    result = (*m_engine_obj)->Realize(m_engine_obj, SL_BOOLEAN_FALSE);
    if (CheckError(result, "Engine Realize")) return false;

    result = (*m_engine_obj)->GetInterface(m_engine_obj, SL_IID_ENGINE, &m_engine);
    return !CheckError(result, "Engine Interface");
}
```

创建混音器

```
// opensl_render.cpp

bool OpenSLRender::CreateOutputMixer() {
    SLresult result = (*m_engine)->CreateOutputMix(m_engine, &m_output_mix_obj, 1,
NULL, NULL);
    if (CheckError(result, "Output Mix")) return false;

    result = (*m_output_mix_obj)->Realize(m_output_mix_obj, SL_BOOLEAN_FALSE);
    if (CheckError(result, "Output Mix Realize")) return false;

    return true;
}
```

按照前面状态机的机制，先创建引擎对象 `m_engine_obj` 、然后 `Realize` 初始化，然后再通过 `GetInterface` 方法，获取到引擎接口 `m_engine` 。

创建播放器

```
// opensl_render.cpp

bool OpenSLRender::CreatePlayer() {

    //【1.配置数据源 DataSource】----------------------

    //配置PCM格式信息
    SLDataLocator_AndroidSimpleBufferQueue android_queue =
{SL_DATALOCATOR_ANDROIDSIMPLEBUFFERQUEUE, SL_QUEUE_BUFFER_COUNT};
    SLDataFormat_PCM pcm = {
            SL_DATAFORMAT_PCM,//播放pcm格式的数据
            (SLuint32)2,//2个声道（立体声）
            SL_SAMPLINGRATE_44_1,//44100hz的频率
            SL_PCMSAMPLEFORMAT_FIXED_16,//位数 16位
            SL_PCMSAMPLEFORMAT_FIXED_16,//和位数一致就行
            SL_SPEAKER_FRONT_LEFT | SL_SPEAKER_FRONT_RIGHT,//立体声（前左前右）
            SL_BYTEORDER_LITTLEENDIAN//结束标志
    };
    SLDataSource slDataSource = {&android_queue, &pcm};

    //【2.配置输出 DataSink】----------------------

    SLDataLocator_OutputMix outputMix = {SL_DATALOCATOR_OUTPUTMIX, m_output_mix_obj};
    SLDataSink slDataSink = {&outputMix, NULL};

    const SLInterfaceID ids[3] = {SL_IID_BUFFERQUEUE, SL_IID_EFFECTSEND,
SL_IID_VOLUME};
    const SLboolean req[3] = {SL_BOOLEAN_TRUE, SL_BOOLEAN_TRUE, SL_BOOLEAN_TRUE};

    //【3.创建播放器】----------------------

    SLresult result = (*m_engine)->CreateAudioPlayer(m_engine, &m_pcm_player_obj,
&slDataSource, &slDataSink, 3, ids, req);
    if (CheckError(result, "Player")) return false;

    //初始化播放器
    result = (*m_pcm_player_obj)->Realize(m_pcm_player_obj, SL_BOOLEAN_FALSE);
    if (CheckError(result, "Player Realize")) return false;

    //【4.获取播放器接口】----------------------

    //得到接口后调用，获取Player接口
    result = (*m_pcm_player_obj)->GetInterface(m_pcm_player_obj, SL_IID_PLAY,
&m_pcm_player);
    if (CheckError(result, "Player Interface")) return false;

    //获取音量接口
    result =  (*m_pcm_player_obj)->GetInterface(m_pcm_player_obj, SL_IID_VOLUME,
&m_pcm_player_volume);
    if (CheckError(result, "Player Volume Interface")) return false;

    //【5. 获取缓冲队列接口】----------------------
```

```
    //注册回调缓冲区，获取缓冲队列接口
    result = (*m_pcm_player_obj)->GetInterface(m_pcm_player_obj, SL_IID_BUFFERQUEUE,
&m_pcm_buffer);
    if (CheckError(result, "Player Queue Buffer")) return false;

    //注册缓冲接口回调
    result = (*m_pcm_buffer)->RegisterCallback(m_pcm_buffer, sReadPcmBufferCbFun,
this);
    if (CheckError(result, "Register Callback Interface")) return false;

    LOGI(TAG, "OpenSL ES init success")

    return true;
}
```

播放器的初始化比较麻烦一些，不过都是根据前面介绍的初始化流程，按部就班。

配置数据源、输出器、以及初始化后，获取播放接口、音量调节接口等。

要注意的是最后一步，即代码中的第【5】。

数据源为 `缓冲队列` 的时候，需要获取一个缓冲接口，用于将数据填入缓冲区。

那么什么时候填充数据呢？这就是最后注册回调接口的作用。

我们需要注册一个回调函数到播放器中，当播放器中的数据播放完，就会回调这个方法，告诉我们：数据播完啦，要填充新的数据了。

> `sReadPcmBufferCbFun` 是一个静态方法，可以推测出，`OpenSL ES` 播放音频内部是一个独立的线程，这个线程不断的读取缓冲区的数据，进行渲染，并在数据渲染完了以后，通过这个回调接口通知我们填充新数据。

实现播放

> 启动 `OpenSL ES` 渲染很简单，只需调用播放器的播放接口，并且往缓冲区压入一帧数据，就可以启动渲染流程。

如果是播放一个 `sdcard` 的 `pcm` 文件，那只要在回调方法 `sReadPcmBufferCbFun` 中读取一帧数据填入即可。

但是，在我们这里没有那么简单，还记得我们的 `BaseDeocder` 中启动了一个解码线程吗？而 `OpenSL ES` 渲染也是一个独立的线程，因此，在这里变成两个线程的数据同步问题。

> 当然了，也可以将 `FFmpeg` 做成一个简单的解码模块，在 `OpenSL ES` 的渲染线程实现解码播放，处理起来就会简单得多。

为了解码流程的统一，这里将会采用两个独立线程。

i. 开启播放等待

上面已经提到，播放和解码是两个所以数据需要同步，因此，在初始化为 `OpenSL` 以后，不能马上开始进入播放状态，而是要等待解码数据第一帧，才能开始播放。

这里，通过线程的等待方式，等待数据。

在前面的 `InitRender` 方法中，首先初始化了 `OpenSL` ，在这方法的最后，我们让播放进入等待状态。

```
// opensl_render.cpp

OpenSLRender::OpenSLRender() {
}

OpenSLRender::~OpenSLRender() {
}

void OpenSLRender::InitRender() {
    if (!CreateEngine()) return;
    if (!CreateOutputMixer()) return;
    if (!ConfigPlayer()) return;

    // 开启线程，进入播放等待
    std::thread t(sRenderPcm, this);
    t.detach();
}

void OpenSLRender::sRenderPcm(OpenSLRender *that) {
    that->StartRender();
}

void OpenSLRender::StartRender() {
    while (m_data_queue.empty()) {
        WaitForCache();
    }
    (*m_pcm_player)->SetPlayState(m_pcm_player, SL_PLAYSTATE_PLAYING);
    sReadPcmBufferCbFun(m_pcm_buffer, this);
}

/**
 * 线程进入等待
 */
void OpenSLRender::WaitForCache() {
    pthread_mutex_lock(&m_cache_mutex);
    pthread_cond_wait(&m_cache_cond, &m_cache_mutex);
    pthread_mutex_unlock(&m_cache_mutex);
}

/**
 * 通知线程恢复执行
 */
void OpenSLRender::SendCacheReadySignal() {
    pthread_mutex_lock(&m_cache_mutex);
    pthread_cond_signal(&m_cache_cond);
    pthread_mutex_unlock(&m_cache_mutex);
}
```

最后的 `StartRender()` 方法是真正被线程执行的方法，进入该方法，首先判断数据缓冲队列是否有数据，没有则进入等待，直到数据到来。

其中，`m_data_queue` 是自定义的数据缓冲队列，如下:

```
// opensl_render.h

class OpenSLRender: public AudioRender {

private:
    /**
     * 封装 PCM 数据，主要用于实现数据内存的释放
     */
    class PcmData {
    public:
        PcmData(uint8_t *pcm, int size) {
            this->pcm = pcm;
            this->size = size;
        }
        ~PcmData() {
            if (pcm != NULL) {
                //释放已使用的内存
                free(pcm);
                pcm = NULL;
                used = false;
            }
        }
        uint8_t *pcm = NULL;
        int size = 0;
        bool used = false;
    };

    // 数据缓冲列表
    std::queue<PcmData *> m_data_queue;

    // 省略其他...
}
```

ii. 数据同步与播放

接下来，就来看看如何尽心数据同步与播放。

初始化 `OpenSL` 的时候，在最后注册了播放回调接口 `sReadPcmBufferCbFun` ，首先来看看它的实现。

```
// opensl_render.cpp

void OpenSLRender::sReadPcmBufferCbFun(SLAndroidSimpleBufferQueueItf bufferQueueItf,
void *context) {
    OpenSLRender *player = (OpenSLRender *)context;
    player->BlockEnqueue();
}

void OpenSLRender::BlockEnqueue() {
    if (m_pcm_player == NULL) return;

    // 先将已经使用过的数据移除
    while (!m_data_queue.empty()) {
        PcmData *pcm = m_data_queue.front();
        if (pcm->used) {
            m_data_queue.pop();
            delete pcm;
        } else {
            break;
        }
    }

    // 等待数据缓冲
    while (m_data_queue.empty() && m_pcm_player != NULL) {// if m_pcm_player is NULL,
stop render
        WaitForCache();
    }

    PcmData *pcmData = m_data_queue.front();
    if (NULL != pcmData && m_pcm_player) {
        SLresult result = (*m_pcm_buffer)->Enqueue(m_pcm_buffer, pcmData->pcm,
(SLuint32) pcmData->size);
        if (result == SL_RESULT_SUCCESS) {
            // 只做已经使用标记，在下一帧数据压入前移除
            // 保证数据能正常使用，否则可能会出现破音
            pcmData->used = true;
        }
    }
}
```

当 `StartRender()` 等待到缓冲数据的到来时，就会通过以下方法启动播放

```
(*m_pcm_player)->SetPlayState(m_pcm_player, SL_PLAYSTATE_PLAYING);
sReadPcmBufferCbFun(m_pcm_buffer, this);
```

这时候，经过一层层调用，最后调用的是 `BlockEnqueue()` 方法。

在这个方法中，

首先，将 `m_data_queue` 中已经使用的数据先删除，回收资源；

接着，判断是否还有未播放的缓冲数据，没有则进入等待；

最后，通过 `(*m_pcm_buffer)->Enqueue()` 方法，将数据压入 `OpenSL` 队列。

注：在接下来的播放过程中，`OpenSL` 只要播放完数据，就会自动回调 `sReadPcmBufferCbFun` 重新进入以上的播放流程。

压入数据，开启播放

以上是整个播放的流程，最后还有关键的一点，来开启这个播放流程，那就是 `AudioRender` 定义的渲染播放接口 `void Render(uint8_t *pcm, int size)`。

```
// opensl_render.cpp

void OpenSLRender::Render(uint8_t *pcm, int size) {
    if (m_pcm_player) {
        if (pcm != NULL && size > 0) {
            // 只缓存两帧数据，避免占用太多内存，导致内存申请失败，播放出现杂音
            while (m_data_queue.size() >= 2) {
                SendCacheReadySignal();
                usleep(20000);
            }

            // 将数据复制一份，并压入队列
            uint8_t *data = (uint8_t *) malloc(size);
            memcpy(data, pcm, size);

            PcmData *pcmData = new PcmData(pcm, size);
            m_data_queue.push(pcmData);

            // 通知播放线程推出等待，恢复播放
            SendCacheReadySignal();
        }
    } else {
        free(pcm);
    }
}
```

其实很简单，就是把解码得到的数据压入队列，并且发送数据缓冲准备完毕信号，通知播放线程可以进入播放了。

这样，就完成了整个流程，总结一下：

1. 初始化 `OpenSL` ，开启「开始播放等待线程」，并进入播放等待；
2. 将数据压入缓冲队列，通知播放线程恢复执行，进入播放；
3. 开启播放时，将 `OpenSL` 设置为播放状态，并压入一帧数据；
4. `OpenSL` 播放完一帧数据后，自动回调通知继续压入数据；
5. 解码线程不断压入数据到缓冲队列；

6. In the next process, the "OpenSL ES playback thread" and the "FFMpeg decoding thread" will be executed at the same time, repeating "2 to 5", and in the case of insufficient data buffer, the "playing thread" will wait for the "decoding thread" to press After entering the data, continue to execute until the playback is completed, and both parties exit the thread.

3. Integrated playback

In the above, the relevant functions of the `OpenSL ES` player , and `AudioRander` the interface defined in is implemented, as long as it `AudioDecoder` is called correctly in .

How to call has also been introduced in the first section, and now you only need to integrate them `Player` into , you can achieve audio playback.

In the player, add audio decoders and renderers:

```
//player.h

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

    // 新增音频解码和渲染器
    AudioDecoder *m_a_decoder;
    AudioRender *m_a_render;

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

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

Instantiate the audio decoder and renderer:

```
// player.cpp

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);

    // 实例化音频解码器和渲染器
    m_a_decoder = new AudioDecoder(jniEnv, path, false);
    m_a_render = new OpenSLRender();
    m_a_decoder->SetRender(m_a_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();
        m_a_decoder->GoOn();
    }
}

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