播放器开发技术点

SDL

SDL_Init

SDL（Simple DirectMedia Layer）库中的一个函数，用于初始化 SDL 库的各种子系统。

```
int SDL_Init(Uint32 flags);
```

- `flags`：这是一个位掩码，用于指定要初始化的子系统。可以是以下常量的组合：
  - `SDL_INIT_TIMER`：初始化计时器子系统。
  - `SDL_INIT_AUDIO`：初始化音频子系统。
  - `SDL_INIT_VIDEO`：初始化视频子系统（包括事件处理）。
  - `SDL_INIT_JOYSTICK`：初始化操纵杆子系统。
  - `SDL_INIT_HAPTIC`：初始化触觉反馈子系统。
  - `SDL_INIT_GAMECONTROLLER`：初始化游戏控制器子系统。
  - `SDL_INIT_EVENTS`：初始化事件处理子系统。
  - `SDL_INIT_EVERYTHING`：初始化所有可用的子系统。

SDL_CreateWindow

- `title`：窗口的标题，是一个字符串。
- `x`：窗口的初始横坐标位置。可以使用 `SDL_WINDOWPOS_UNDEFINED` 或 `SDL_WINDOWPOS_CENTERED` 来让 SDL 自动选择位置。
- `y`：窗口的初始纵坐标位置。可以使用 `SDL_WINDOWPOS_UNDEFINED` 或 `SDL_WINDOWPOS_CENTERED` 来让 SDL 自动选择位置。
- `w`：窗口的宽度（以像素为单位）。
- `h`：窗口的高度（以像素为单位）。
- `flags`：用于指定窗口的特性。常见的标志包括：
  - `SDL_WINDOW_FULLSCREEN`：创建一个全屏窗口。
  - `SDL_WINDOW_OPENGL`：创建一个支持 OpenGL 的窗口。
  - `SDL_WINDOW_SHOWN`：创建一个默认显示的窗口。
  - `SDL_WINDOW_HIDDEN`：创建一个默认隐藏的窗口。
  - `SDL_WINDOW_BORDERLESS`：创建一个无边框窗口。
  - `SDL_WINDOW_RESIZABLE`：创建一个可调整大小的窗口。

返回值

- 成功时返回指向新创建的窗口对象（`SDL_Window`）的指针。
- 如果创建失败，返回 NULL，并且可以通过调用 `SDL_GetError()` 获取错误信息。

SDL_CreateRenderer

用于创建一个渲染器。渲染器是用于在窗口上绘制图形的对象。通过渲染器，你可以绘制纹理、形状和其他图形元素。

```
SDL_Renderer* SDL_CreateRenderer(SDL_Window* window,
                                 int index,
```

```
                                    Uint32 flags);
```

参数

- `window`：指向要在其上创建渲染器的窗口（`SDL_Window`）。
- `index`：渲染驱动程序的索引。可以设置为 -1 以让 SDL 自动选择最适合的驱动程序。
- `flags`：用于指定渲染器的特性。常见的标志包括：
  - `SDL_RENDERER_SOFTWARE`：使用软件渲染。
  - `SDL_RENDERER_ACCELERATED`：使用硬件加速渲染。
  - `SDL_RENDERER_PRESENTVSYNC`：启用垂直同步。
  - `SDL_RENDERER_TARGETTEXTURE`：支持渲染目标纹理。

返回值

- 成功时返回指向新创建的渲染器对象（`SDL_Renderer`）的指针。
- 如果创建失败，返回 NULL，并且可以通过调用 `SDL_GetError()` 获取错误信息。

SDL_CreateTexture

用于创建一个纹理。纹理是用于在渲染器上绘制图像的对象。你可以从表面、像素数据或其他资源创建纹理。

```
SDL_Texture* SDL_CreateTexture(SDL_Renderer* renderer,
                               Uint32 format,
                               int access,
                               int w,
                               int h);
```

参数

- `renderer`：指向用于渲染此纹理的渲染器（`SDL_Renderer`）。
- `format`：像素格式。常见的像素格式包括：
  - `SDL_PIXELFORMAT_RGBA8888`
  - `SDL_PIXELFORMAT_ARGB8888`
  - `SDL_PIXELFORMAT_YV12`（YUV 格式）
- `access`：纹理访问模式。常见的访问模式包括：
  - `SDL_TEXTUREACCESS_STATIC`：表示纹理的数据不会被改变。
  - `SDL_TEXTUREACCESS_STREAMING`：表示纹理的数据会被频繁更新。
  - `SDL_TEXTUREACCESS_TARGET`：表示纹理可以用作渲染目标。
- `w`：纹理的宽度（以像素为单位）。
- `h`：纹理的高度（以像素为单位）。

返回值

- 成功时返回指向新创建的纹理对象（`SDL_Texture`）的指针。
- 如果创建失败，返回 NULL，并且可以通过调用 `SDL_GetError()` 获取错误信息。

FFmpeg

解码：

AVFormatContext 结构体

- `AVInputFormat* iformat`：输入格式。如果是输入文件，这个指针指向相应的输入格式。
- `AVOutputFormat* oformat`：输出格式。如果是输出文件，这个指针指向相应的输出格式。
- `void* priv_data`：私有数据，用于存储特定格式所需的信息。

- `AVIOContext* pb`：I/O 上下文，用于读取或写入数据。
- `unsigned int nb_streams`：流的数量，即文件中包含的视频、音频、字幕等流的数量。
- `AVStream** streams`：指向流数组的指针，每个流对应一个 `AVStream` 结构体。
- `int64_t duration`：文件的持续时间，以微秒为单位。
- `int bit_rate`：文件的比特率，以比特每秒为单位。
- `char filename[1024]`：文件名或 URL。

`avformat_alloc_context()`

a. 作用：分配并初始化一个 `AVFormatContext` 结构体，用于存储多媒体文件的格式信息。
b. 返回值：返回一个指向新分配的 `AVFormatContext` 的指针。如果分配失败，返回 NULL。

`avformat_open_input()`

c. 作用：打开输入文件并读取头部信息以确定文件格式。该函数会将文件的信息存储在 `AVFormatContext` 中。
d. 参数：
   i. `AVFormatContext** ps`：指向指向 `AVFormatContext` 的指针。
   ii. `const char* url`：文件名或 URL。
   iii. `AVInputFormat* fmt`：输入格式，如果为 NULL，FFmpeg 会自动检测格式。
   iv. `AVDictionary** options`：选项字典，可以为 NULL。
e. 返回值：成功时返回 0，失败时返回负值。

`avformat_find_stream_info()`

f. 作用：读取媒体文件的数据包以获取流的信息，如编解码器、比特率等。这对于后续的解码操作非常重要。
g. 参数：
   i. `AVFormatContext* ic`：指向已打开的 `AVFormatContext` 的指针。
   ii. `AVDictionary** options`：选项字典，可以为 NULL。
h. 返回值：成功时返回非负值，失败时返回负值。

AVCodecContext结构体

`AVCodecContext` 是 FFmpeg 库中用于描述编解码器上下文的核心结构体。它包含了音视频编解码过程中所需的各种参数和状态信息。以下是 `AVCodecContext` 结构体的一些主要成员变量及其用途：

主要成员变量

1. **常规信息**
   a. `const AVClass *av_class`：用于日志和选项系统。
   b. `enum AVMediaType codec_type`：编解码器类型（如音频、视频、字幕等）。
   c. `const struct AVCodec *codec`：指向编解码器的指针。
   d. `char codec_name[32]`：编解码器名称。
2. **编码参数**
   a. `int bit_rate`：比特率。
   b. `int width, height`：视频宽度和高度。
   c. `int gop_size`：I帧之间的帧数（GOP大小）。
   d. `enum AVPixelFormat pix_fmt`：像素格式。
3. **时间基准**
   a. `AVRational time_base`：基本时间单位，用于时间戳计算。

4. **图像相关**
    a. `int framerate_num, framerate_den`：帧率分子和分母。
    b. `int ticks_per_frame`：每帧的时钟滴答数。
5. **音频相关**
    a. `int sample_rate`：采样率。
    b. `enum AVSampleFormat sample_fmt`：采样格式。
    c. `int channels`：通道数。
    d. `uint64_t channel_layout`：通道布局。
6. **缓冲区**
    a. `uint8_t *extradata`：附加数据（如头信息）。
    b. `int extradata_size`：附加数据大小。
7. **编解码器状态**
    a. `void *priv_data`：私有数据，用于编解码器内部使用。
8. **错误处理**
    a. `int err_recognition`：错误识别标志。
9. **其他参数**
    a. 还有许多其他参数，用于控制各种细节，如线程数、最大B帧数、量化参数等。

AVCodecParameters 结构体

用于描述编解码器参数。它包含了与音频、视频、字幕等流相关的各种参数信息。这个结构体在 FFmpeg 中用于传递和存储流的编解码器参数，与 `AVCodecContext` 相比，它更轻量化且专注于参数本身。

avcodec_alloc_context3()

会分配一个未初始化的 `AVCodecContext` 结构体。

avcodec_parameters_to_context()

用于将 `AVCodecParameters` 结构体中的参数复制到 `AVCodecContext` 结构体中。这个函数通常在解码或编码过程中使用，以便将流的参数设置到编解码器上下文中。

avcodec_find_decoder()

寻找合适的解码器

avcodec_open2()

`avcodec_open2` 函数用于打开并初始化指定的编解码器上下文，以便进行音频或视频的编码和解码。

AVPacket 结构体

用于存储编码后的数据包（例如音频或视频帧）。`av_packet_alloc` 函数用于分配并初始化一个 `AVPacket` 结构体。

av_read_frame()

`av_read_frame` 函数用于从 `formatCtx` 指向的格式上下文中读取下一帧数据，并将其存储在 `packet` 中。如果读取成功，函数返回一个非负值，如果读取失败，返回负值。

AVFrame 结构体

用于表示音频或视频帧

AVFrame 主要成员变量

以下是 `AVFrame` 结构体的一些主要成员变量：

- `uint8_t *data[AV_NUM_DATA_POINTERS]`：用于存储帧数据的指针数组。
- `int linesize[AV_NUM_DATA_POINTERS]`：每个数据指针的行大小。
- `int width, height`：视频帧的宽度和高度。
- `int nb_samples`：音频帧中的样本数量。
- `int format`：帧的数据格式（像素格式或样本格式）。
- `int64_t pts`：时间戳（Presentation Time Stamp）。
- `int key_frame`：标记是否为关键帧。
- `enum AVPictureType pict_type`：帧类型（如 I 帧、P 帧、B 帧等）。
- `AVRational sample_aspect_ratio`：样本纵横比。
- `int64_t channel_layout`：音频通道布局。

avcodec_send_packet(videoCodecCtx, packet);

将压缩的数据包（`packet`）发送到解码器（`videoCodecCtx`）进行处理。

avcodec_receive_frame(videoCodecCtx, frame)

从解码器中获取一个解码后的帧（`frame`）。
这个函数实际上执行了解码操作，将压缩数据转化为原始的视频或音频帧

视频帧 播放时间的计算

1. 计算帧的显示时间（`frameTime`）

```
double frameTime = frame->pts * av_q2d(formatCtx->streams[videoStream]->
```

- `frame->pts` 是帧的时间戳 (Presentation Time Stamp, PTS)，表示这帧应该在视频流的哪个时间点显示。
- `time_base` 是时间基，通常表示时间单位，以秒为单位。不同的流（如音频流、视频流）可能有不同的时间基。
- `av_q2d(formatCtx->streams[videoStream]->time_base)` 将时间基转换为双精度浮点数。
- 通过将 `PTS` 与时间基相乘，可以得到帧的显示时间 `frameTime`，以秒为单位。

2. 获取当前时间（`currentTime`）

```
double currentTime = get_time();
```

- `get_time()` 是一个自定义函数（假设如此），用于获取当前系统时间（从播放开始的时间点算起）。
- `currentTime` 是从视频播放开始到现在的时间，以秒为单位。

3. 计算延迟时间（`delay`）

```
double delay = frameTime - (currentTime - videoStartTime);
```

- `videoStartTime` 是视频开始播放的时间点。
- `currentTime - videoStartTime` 计算的是视频从开始播放到当前的播放时间（当前进度）。
- `frameTime - (currentTime - videoStartTime)` 计算当前帧的时间戳与实际播放时间之间的差距，得到的就是 `delay`，即需要等待的时间。

4. 根据延迟调整播放速度

```
if (delay > 0) {
    std::this_thread::sleep_for(std::chrono::duration<double>(delay));
}
```

- 如果 `delay` 为正值，表示当前播放时间比应显示的时间提前，这意味着播放器需要等待 `delay` 秒，以便同步显示。
- `std::this_thread::sleep_for(std::chrono::duration<double>(delay));` 会暂停当前线程一段时间（`delay` 秒），以实现帧的同步显示。

SDL音频解码后启动音频播放

```
// 设置音频参数并启动音频播放
    SDL_AudioSpec wanted_spec, spec;
    wanted_spec.freq = 44100;
    wanted_spec.format = AUDIO_S16SYS;
    wanted_spec.channels = 2;
    wanted_spec.silence = 0;
    wanted_spec.samples = 2048;
    wanted_spec.callback = audio_callback;
    SDL_OpenAudio(&wanted_spec, &spec);
    SDL_PauseAudio(0);
```

1. 设置音频参数

```
SDL_AudioSpec wanted_spec, spec;
```

- `SDL_AudioSpec` 是一个结构体，用于描述音频设备的参数。
- `wanted_spec` 是你期望的音频设备参数，你通过它告诉 SDL 你希望如何配置音频输出。
- `spec` 是 SDL 实际提供的音频设备参数，可能与 `wanted_spec` 稍有不同。

2. 设置期望的音频参数

```
wanted_spec.freq = 44100;
wanted_spec.format = AUDIO_S16SYS;
wanted_spec.channels = 2;
wanted_spec.silence = 0;
wanted_spec.samples = 2048;
wanted_spec.callback = audio_callback;
```

- `wanted_spec.freq = 44100`：设置音频的采样率为 44100 Hz。这是 CD 质量的标准采样率，表示每秒采样 44100 次。
- `wanted_spec.format = AUDIO_S16SYS`：设置音频样本格式为 `AUDIO_S16SYS`，即 16 位有符号整数格式，字节序与系统相同。`S16` 表示 16 位有符号整数，`SYS` 表示使用系统的字节序（大端或小端）。
- `wanted_spec.channels = 2`：设置音频通道数量为 2，表示立体声（左声道和右声道）。
- `wanted_spec.silence = 0`：指定静音的值，这里设为 0，通常是无声数据的值（对 16 位音频来说是 `0x0000`）。

- `wanted_spec.samples = 2048`：设置音频缓冲区的大小，以样本数量为单位。缓冲区越大，延迟越大，但音频播放越流畅。2048 是一个常见的缓冲区大小，通常在音频播放时表现稳定。
- `wanted_spec.callback = audio_callback`：指定回调函数 `audio_callback`，SDL 会调用此函数来填充音频缓冲区。这是音频播放的核心部分，你需要在这个回调函数中处理音频数据。

3. 打开音频设备

```
SDL_OpenAudio(&wanted_spec, &spec);
```

- `SDL_OpenAudio(&wanted_spec, &spec)`：
  - 这个函数打开音频设备，并根据你提供的 `wanted_spec` 参数来初始化设备。SDL 尽量匹配 `wanted_spec` 中的设置，但有时可能会调整一些参数（比如采样率或缓冲区大小）以适应硬件设备。实际使用的参数将保存在 `spec` 中。

4. 开始播放音频

```
SDL_PauseAudio(0);
```

- `SDL_PauseAudio(0)`：
  - 这个函数启动音频播放。`SDL_PauseAudio(1)` 会暂停音频播放，而 `SDL_PauseAudio(0)` 则取消暂停并开始播放音频数据。

5. 回调函数 `audio_callback`

- `audio_callback` 是 SDL 音频子系统在需要音频数据时调用的函数。你需要在这个函数中填充音频缓冲区。例如，将解码后的音频数据传递给音频硬件进行播放。

```
void audio_callback(void* userdata, Uint8* stream, int len) {
    static uint8_t* audio_pos = nullptr;  // 指向当前播放音频数据的指针
    static int audio_len = 0;  // 当前剩余音频数据的长度
    if (audio_len == 0) {
        // 当一个音频数据帧用完时，等待新数据
        std::unique_lock<std::mutex> lock(audioMutex);
        audioCond.wait(lock, [] { return !audioFrameQueue.empty() || qui
        if (quit) return;
        // 获取音频帧并将其放入音频缓冲区
        AVFrame* frame = audioFrameQueue.front();
        audioFrameQueue.pop();
        audio_pos = frame->data[0]; // 音频数据的指针
        audio_len = frame->linesize[0]; // 音频数据的长度
        av_frame_free(&frame); // 释放音频帧
    }
    // 确保音频数据不会超出缓冲区
    len = (len > audio_len) ? audio_len : len;
    SDL_memcpy(stream, audio_pos, len); // 将音频数据复制到输出缓冲区
    audio_pos += len; // 更新音频数据指针
    audio_len -= len; // 减少剩余音频数据长度
}
```

- `audio_callback` 的实现中，你可以从音频数据队列中提取音频数据，填充到 `stream` 缓冲区中，确保音频播放流畅。

SDL 视频播放

```
void video_thread(SDL_Renderer* renderer, SDL_Texture* texture) {
    double videoStartTime = get_time(); // 获取视频开始播放的时间
    while (!quit) {
        std::unique_lock<std::mutex> lock(videoMutex);
        videoCond.wait(lock, [] { return !videoFrameQueue.empty() || qui
        if (quit) break;
        AVFrame* frame = videoFrameQueue.front();
        videoFrameQueue.pop();
        // 计算当前帧的播放时间并调整播放速度
        double frameTime = frame->pts * av_q2d(formatCtx->streams[videoS
        double currentTime = get_time();
        double delay = frameTime - (currentTime - videoStartTime);
        if (delay > 0) {
            std::this_thread::sleep_for(std::chrono::duration<double>(de
        }
        // 获取视频帧的实际尺寸
        int frameWidth = frame->width;
        int frameHeight = frame->height;
        // 更新SDL纹理的尺寸
        if (frameWidth != videoWidth || frameHeight != videoHeight) {
            videoWidth = frameWidth;
            videoHeight = frameHeight;
            SDL_DestroyTexture(texture);
            texture = SDL_CreateTexture(renderer, SDL_PIXELFORMAT_YV12,
        }
        // 将YUV数据更新到SDL纹理中
        SDL_UpdateYUVTexture(texture, nullptr,
            frame->data[0], frame->linesize[0],
            frame->data[1], frame->linesize[1],
            frame->data[2], frame->linesize[2]);

        // 清除渲染器并复制纹理，然后呈现
        SDL_RenderClear(renderer);
        SDL_RenderCopy(renderer, texture, nullptr, nullptr);
        SDL_RenderPresent(renderer);
        av_frame_free(&frame); // 释放视频帧内存
    }
}
```

设计QT界面 选择文件播放

在项目中导入QT使用的QT 库文件

```
#include <QtWidgets/QMainWindow>
#include <QtWidgets/QApplication>
#include <QtWidgets/QFileDialog>
#include "Player.h"
```

```
int main(int argc, char* argv[]) {
    // 播放文件
    QApplication a(argc, argv);

    // 弹出文件选择对话框
    QString file_name = QFileDialog::getOpenFileName(nullptr, QStringLit

    // 如果选择了文件
    if (!file_name.isEmpty()) {
        // 将QString转换为C字符串
        QByteArray byte_array = file_name.toLocal8Bit();
        const char* file_name_cstr = byte_array.data();

        // 创建播放器实例
        Player player(file_name_cstr);
        player.InitializePlayer();
        player.StartPlayer();

        // SDL事件处理
        SDL_Event e;
        bool running = true;
        while (running) {
            while (SDL_PollEvent(&e)) {
                if (e.type == SDL_QUIT) {
                    running = false;
                    player.StopPlayer();
                }
            }
        }
    }
    else {
        // 如果没有选择文件，退出程序
        return 0;
    }

    return a.exec();
}
```

空格暂停功能的实现

在player 类中添加字段 bool paused_ 控制暂停 播放，添加类成员函数TogglePause（）；

(1)在 Player 类中添加一个布尔变量 paused_ 来表示当前的播放状态。你需要在播放线程中根据 paused_ 的状态来决定是否暂停播放

(2)在 main.cpp 的事件循环中，检测空格键的按下并调用 player.TogglePause() 来切换播放状态

(3)修改视频播放线程和音频播放线程逻辑，实现按下空格后，音频和视频画面暂停

2、难点：实现暂停后再次播放后，音频输出正常，视频播放过快：音视频不同步

定位：视频播放线程，视频同步时间delay计算逻辑需要修改

解决：在Plarer类中添加pause_time_、video_pause_time_俩个成员变量，记录暂停时刻，和视频暂停时间

视频同步的计算公式为 delay(延时时间) = frame_time(视频的pts) - (current_time（当前时间） - video_start_time(当前播放时间));

暂停后current_time（当前时间） = GetCurrentSystemTime()- video_pause_time_; 减去暂停时间，实现同步

难点：

1.视频播放过快：SDL 渲染过快：

分析：视频渲染线程 从 视频帧读取 帧 速度过快 ， 因该在此处做限制。
解决：
对与音视频来说，每一帧都有时间戳，音视频时间戳通常是相对时间戳，即相对于视频或频的**起始时间点**而言。如果需要将音视频时间戳转换为绝对时间戳，需要知道视频或音频的起始时间点，并加上相应的偏移量。
等待时间 = 当前帧的播放时间 –（当前时间 – 开始播放时间）
等待时间>0 ,线程进行等待

2.音频播放失真，噪声很大，声音过尖

分析 ：
1. 查看SDL 音频SDL_AudioSpec 设置，采样率，音频数据格式，声道数，音频 缓冲区大小（一帧数据的样本数）设置是都是否正确，
2.解码完放入 队列前的 原始音频帧二进制文件 是否正确，使用Audacity测试，
3.在SDL 处理音视频回调函数 从队列 取出 音频帧 是否正确
4.将队列中取出音频帧中取出所有样本数据放入 buf 缓存中 ， buf 中的数据是否可以正常播放，
3.最后将buf中的数据放入SKD 音频缓存区中stream ， 看stream中是否正确，
解决：
最后发现 .将队列中取出音频帧中取出所有样本数据放入 buf 数组中，buf中的数据不正常，
从队列中AVFrame 中取数据 放入 buf 出现错误，修改了一下AVFrame 的写入方式解决，
2.音频播卡顿问题：
最后将buf中的数据拷贝入SKD 音频缓存区中stream中 ,每一次拷贝的数据最好是len(960样本)（SDLstream 的最大字节数），满足这个条件可以是声音流畅连贯，
解决，buf的大小设置很大，每次调用回调函数，也就是播音频数据，都检查buf的大小是否大于 len，小于，读一个音频帧到buf（2048个样本），最后bug拷贝到stream中，将拷贝过的数据前移，从buf中删除·

AVFrame中 数据存储

平面格式 (Planar Format)

在平面格式中，每个音频通道的数据被存储在独立的平面中。在 AVFrame 中，这种格式的音频数据会存储在 frame->data 的不同索引中。

● **数据存储**：frame->data[0] 存储第一个通道的数据，frame->data[1] 存储第二个通道的数据，等等。
● **数据访问**：每个通道的样本数据在其平面中是连续的。假设有两个通道的音频数据，frame->data[0] 存储第一个通道的所有样本，frame->data[1] 存储第二个通道的所有样本。

```
AVFrame* frame = audioFrameQueue.front();
audioFrameQueue.pop();

// 逐个样本复制数据
for (int i = 0; i < frame->nb_samples; ++i) {
    for (int ch = 0; ch < 2; ++ch) {  // 假设2通道音频
        float* samples = reinterpret_cast<float*>(frame->data[ch]);
        audio_buffer.push_back(samples[i]);
    }
}

```

交错格式 (Interleaved Format)

在交错格式中，音频数据是按通道交替存储的
数据存储：frame->data[0] 存储交错的音频数据，其中每个样本按 [L1, R1, L2, R2, ...] 的方式排列。

```
AVFrame* frame = audioFrameQueue.front();
audioFrameQueue.pop();

int num_channels = 2; // 假设2通道音频
int sample_size = frame->nb_samples;

// 逐个样本复制数据
float* samples = reinterpret_cast<float*>(frame->data[0]);
for (int i = 0; i < sample_size; ++i) {
    for (int ch = 0; ch < num_channels; ++ch) {
        audio_buffer.push_back(samples[i * num_channels + ch]);
    }
}
```

数据访问：每个样本的左声道和右声道数据是交替存储的。

关键区别

1. **存储方式**：
a. **平面格式**：每个通道的数据存储在不同的 frame->data 元素中。
b. **交错格式**：所有通道的数据存储在同一个 frame->data 元素中，按照交替的顺序排列。
2. **数据提取**：
a. **平面格式**：需要逐个通道读取数据，并根据 frame->data 的索引来访问。

b. **交错格式**：需要按通道交替从同一数据块中读取数据。

3.暂停后音视频同步问题

实现暂停后再次播放后，音频输出正常，视频播放过快：音视频不同步

定位：视频播放线程，视频同步时间delay计算逻辑需要修改

解决：在Plarer类中添加pause_time_、video_pause_time_俩个成员变量，记录暂停时刻，和视频暂停时间

视频同步的计算公式为 delay(延时时间) = frame_time(视频的pts) - (current_time（当前时间） - video_start_time(当前播放时间));

暂停后current_time（当前时间） = GetCurrentSystemTime()- video_pause_time_; 减去暂停时间，实现同步