ffmpeg学习经验

书籍推荐

首先阅读了两本书籍，《FFmpeg入门详解》《FFmpeg音视频开发基础与实战》

音视频基础概念

视频基础

- 像素（Pixel）：视频的最小显示单元，包含亮度（Y）和色度（U/V）信息
- 分辨率：视频画面的尺寸（如 1920×1080），决定画面清晰度
- 帧率（FPS）：每秒显示的画面数（如 25FPS、30FPS），影响画面流畅度
- 码率（Bitrate）：单位时间内视频数据的体积（如 1Mbps），码率越高画质越好、文件越大
- 色彩空间：
    - YUV：视频编码常用格式（Y 是亮度，U/V 是色度），比 RGB 更节省带宽（如 YUV420P）

- RGB：显示设备常用格式（红 / 绿 / 蓝三通道）

- 时间基（Time Base）：FFmpeg 中表示时间的单位（如1/25表示每帧持续 1/25 秒），用于时间戳计算
- PTS/DTS：
  - PTS（显示时间戳）：帧的显示时间
  - DTS（解码时间戳）：帧的解码时间（I/P/B 帧存在顺序差异时，PTS≠DTS）

音频基础

- 采样率（Sample Rate）：每秒采集的音频样本数（如 44.1kHz、48kHz），决定音频频率范围
- 采样位深：每个音频样本的比特数（如 16bit），决定音频动态范围
- 声道数：音频的发声通道数（如单声道、立体声）
- 音频帧：音频编码 / 解码的基本单元（通常包含固定时长的样本，如 20ms）

音视频编解码原理

压缩编码的核心目标

通过去除冗余信息（空间冗余、时间冗余、视觉 / 听觉冗余），在保证画质 / 音质的前提下，减小文件体积

视频编码核心概念

- 帧类型：
  - I 帧（关键帧）：不依赖其他帧，可独立解码，体积大、解码快
  - P 帧（预测帧）：参考前一个 I/P 帧编码，体积小
  - B 帧（双向预测帧）：参考前后帧编码，压缩率最高，但需要缓存帧、延迟大
  - GOP（图像组）：连续的视频帧序列（如 I-P-P-B-P），GOP 越大压缩率越高、Seek 越慢
- 编码标准：
  - H.264/AVC：目前最常用的视频编码标准，兼容性强
  - H.265/HEVC：压缩效率比 H.264 高 50%，但编码复杂度更高
  - AV1：开源免费的新一代编码标准，压缩率优于 H.265
- NALU（网络抽象层单元）：视频码流的基本封装单元，包含 SPS/PPS（参数集）、Slice（帧分片）等数据
  - SPS（序列参数集）：描述视频序列的全局参数（分辨率、帧率）
  - PPS（图像参数集）：描述单帧的编码参数（量化矩阵、滤波参数）

音频编码核心概念

- 编码标准：
  - AAC：目前最常用的音频编码格式，音质好、压缩率高（如 MP4 常用）
  - MP3：传统音频格式，兼容性强但压缩效率低于 AAC
  - Opus：新一代开源格式，兼顾低延迟和高音质（适合直播）
- 音频压缩原理：
  - 去除听觉冗余（人耳对某些频率不敏感）
  - 利用心理声学模型（掩蔽效应：强音掩盖弱音）

FFmpeg 中的编码核心组件

- AVCodec：编码器 / 解码器的抽象接口（如avcodec_find_encoder(AV_CODEC_ID_H264)）
- AVCodecContext：编码器 / 解码器的上下文，存储编码参数（分辨率、码率等）
- AVFrame：存储原始音视频数据（未编码）
- AVPacket：存储编码后的压缩数据（FFmpeg 中码流的基本单元）

H.264 编解码基础

H.264 核心技术

- 帧内预测：利用同一帧内相邻像素的相关性，通过预测模型（如水平 / 垂直 / 直流预测）生成 “预测块”，仅编码 “原始块与预测块的残差”，减少空间冗余
- 帧间预测：对 P/B 帧，通过运动估计找到前 / 后参考帧中的匹配块，仅编码 “运动矢量（匹配块的位置偏移）+ 残差”，减少时间冗余
  - 运动矢量：描述当前块相对于参考块的位移
  - 宏块 / CTU：H.264 将帧划分为 16×16 宏块（后续扩展为 64×64 CTU），作为预测和编码的基本单元
- 变换编码：对残差数据进行 DCT（离散余弦变换），将空间域数据转换为频率域，再通过量化丢弃高频细节（人眼对高频不敏感）
- 熵编码：对量化后的数据用 CAVLC（上下文自适应可变长编码） 或 CABAC（上下文自适应二进制算术编码） 编码，进一步压缩数据体积（CABAC 压缩率更高，但复杂度也更高）

H.264 码流结构

H.264 码流由 **NALU（网络抽象层单元）** 组成，关键 NALU 类型：

- SPS（序列参数集）：描述视频序列的全局信息（分辨率、帧率、档次 / 级别）
- PPS（图像参数集）：描述单帧的编码参数（熵编码类型、量化参数）

- IDR 帧（即时解码刷新帧）：特殊的 I 帧，其后的帧不再参考 IDR 帧之前的内容，用于视频的随机访问（Seek）
- Slice（片）：将一帧划分为多个片，每个片独立编码 / 解码，某一片损坏不影响其他片，提升容错性

H.264 档次与级别

可以理解为给编码能力定规矩的两个维度

- 档次（Profile）：定义编码工具集（如 Baseline 仅支持 I/P 帧，Main 支持 B 帧，High 支持更复杂的预测）
- 级别（Level）：限制码率、分辨率、帧率的组合（如 Level 4.0 支持 1920×1080@30FPS）

ACC 编解码基础

AAC 核心技术

- 心理声学模型：利用人耳的掩蔽效应（强音会掩盖同频率附近的弱音），丢弃人耳无法感知的音频细节，减少听觉冗余
- 子带滤波：通过 MDCT（改进的离散余弦变换） 将音频信号划分为多个子带，对不同子带分配不同的量化精度（人耳敏感的频段分配更高精度）
- 熵编码：用 Huffman 编码 或 算术编码 压缩量化后的数据

AAC 常见格式

- AAC-LC（低复杂度）：最常用的 AAC 格式，平衡压缩率和音质（如 MP4、直播中常用）
- HE-AAC（高效 AAC）：通过 SBR（频谱带宽扩展） 技术，在低码率下实现接近 AAC-LC 的音质（适合移动设备）
- AAC-ELD（增强低延迟）：延迟低于 20ms，适合实时通信（如视频通话）

AAC 码流结构

AAC 码流以 ADTS（音频数据传输流） 为封装单元，每个 ADTS 帧包含：

- 头部：描述采样率、声道数、帧长度等信息
- 数据：编码后的音频样本

最简单的基于 FFmpeg 和 SDL2.0 的视频播放器

代码块

```
/**
 * 最简单的基于FFmpeg的视频播放器 2
 * Simplest FFmpeg Player 2
```

```
 * 本程序实现了视频文件的解码和显示(支持HEVC，H.264，MPEG2等)。
 * 是最简单的FFmpeg视频解码方面的教程。
 * 通过学习本例子可以了解FFmpeg的解码流程。
 * This software is a simplest video player based on FFmpeg.
 * Suitable for beginner of FFmpeg.
 *
 */

/*
这段代码是一个基于 FFmpeg 和 SDL2.0 的极简视频播放器实现，核心作用是完成视频文件的解码、格式转换和显示
*/

#include <stdio.h>

#if defined(_MSC_VER) && _MSC_VER >= 1900
 // 适配 VS 新版本中 __iob_func 到 __acrt_iob_func 的变更
extern "C" {
        FILE* __cdecl __iob_func(unsigned i) {
                return __acrt_iob_func(i);
        }
}
#endif

#define __STDC_CONSTANT_MACROS

#ifdef _WIN32
//Windows
extern "C"
{
#include "libavcodec/avcodec.h"      // FFmpeg 编解码核心库
#include "libavformat/avformat.h"    // FFmpeg 格式处理库 （打开文件、解析流）
#include "libswscale/swscale.h"      // FFmpeg 图像格式转换库 （如YUV 转 RGB）
#include "libavutil/imgutils.h"      // FFmpeg 图像工具库 （分配图像缓冲区）
#include "SDL2/SDL.h"                // SDL2库 （创建窗口、图像渲染）
};
#else
//Linux...
#ifdef __cplusplus
extern "C"
{
#endif
#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include <libswscale/swscale.h>
#include <SDL2/SDL.h>
#include <libavutil/imgutils.h>
```

```
#ifdef __cplusplus
};
#endif
#endif

//Output YUV420P data as a file
// 控制是否输出 YUV420P 原始文件：0 为不输出，1 为输出到 output.yuv
#define OUTPUT_YUV420P 0

int main(int argc, char* argv[])
{
        AVFormatContext        *pFormatCtx;        // FFmpeg 格式上下文 （存储文件整体信息）
        int                                i, videoindex;      // i用于循环 videoindex用于标记视频流索引
        AVCodecContext        *pCodecCtx;         // 编解码器上下文 （用于存储解码器参数）
        AVCodec                        *pCodec;            // 编解码器示例
        AVFrame        *pFrame,*pFrameYUV;         // pFrame解码后原始帧 pFrameYUV 格式转换后的 YUV420P帧
        unsigned char *out_buffer;          // 存储YUV420P 帧的缓冲区
        AVPacket *packet;                   // 存储编码后的数据包
        int y_size;                         // Y分量数据大小（宽 × 高）
        int ret, got_picture;               // ret：函数调用返回值；got_picture：标记是否成功解码出一帧
        struct SwsContext *img_convert_ctx;  // 图像格式转换上下文

        char filepath[]="bigbuckbunny_480x272.h265";  // 待播放视频文件路径
        //SDL---------------------------
        int screen_w=0,screen_h=0;         // 窗口宽高
        SDL_Window *screen;                // SDL 窗口实例
        SDL_Renderer* sdlRenderer;         // SDL 渲染器（控制绘制）
        SDL_Texture* sdlTexture;           // SDL 纹理（存储待渲染的图像数据）
        SDL_Rect sdlRect;                  // 渲染区域

        FILE *fp_yuv;                      // 输出YUV文件的文件指针

        av_register_all();                 // 注册 FFmpeg 所有可能的格式和编解码器
        avformat_network_init();           // 初始化网络模块（支持网络流，本地文件也需要调用）
        pFormatCtx = avformat_alloc_context();  // 分配格式上下文内存 后续所有文件信息都存在这里

        // 打开视频文件，读取文件头信息并填充在pFormatCtx
        if(avformat_open_input(&pFormatCtx,filepath,NULL,NULL)!=0){
                printf("Couldn't open input stream.\n");
                return -1;
```

```
90          }
91          // 解析文件中的流信息（视频流、音频流等），填充到pFormatCtx->streams数组
92          if(avformat_find_stream_info(pFormatCtx,NULL)<0){
93                  printf("Couldn't find stream information.\n");
94                  return -1;
95          }
96          // 查找视频流索引：遍历所有流，找到类型为 AVMEDIA_TYPE_VIDEO 的流，记录其索引 videoindex
97          videoindex=-1;
98          for(i=0; i<pFormatCtx->nb_streams; i++)
99                  if(pFormatCtx->streams[i]->codec->codec_type==AVMEDIA_TYPE_VIDEO){
100                         videoindex=i;
101                         break;
102                 }
103         if(videoindex==-1){
104                 printf("Didn't find a video stream.\n");
105                 return -1;
106         }
107
108         // 获取视频流的编解码器上下文
109         pCodecCtx=pFormatCtx->streams[videoindex]->codec;
110         // 查找对应解码器
111         pCodec=avcodec_find_decoder(pCodecCtx->codec_id);
112         if(pCodec==NULL){
113                 printf("Codec not found.\n");
114                 return -1;
115         }
116         // 打开解码器
117         if(avcodec_open2(pCodecCtx, pCodec,NULL)<0){
118                 printf("Could not open codec.\n");
119                 return -1;
120         }
121
122         pFrame=av_frame_alloc();  // 分配原始解码帧内存（存储解码后的原始像素格式）
123         pFrameYUV=av_frame_alloc(); // 分配 YUV420P 帧内存（用于 SDL 渲染）
124         // 分配 YUV420P 帧的缓冲区：av_image_get_buffer_size 计算 YUV420P 格式所需的缓冲区大小（宽 × 高 ×1.5，因 Y:U:V=4:1:1）
125         out_buffer=(unsigned char *)av_malloc(av_image_get_buffer_size(AV_PIX_FMT_YUV420P,  pCodecCtx->width, pCodecCtx->height,1));
126         // av_image_fill_arrays 将缓冲区与 pFrameYUV 的数据指针（data）和行跨度（linesize）绑定，后续格式转换直接写入该缓冲区
127         av_image_fill_arrays(pFrameYUV->data, pFrameYUV->linesize,out_buffer,
128                 AV_PIX_FMT_YUV420P,pCodecCtx->width, pCodecCtx->height,1);
129         // AVPacket 存储从文件读取的编码数据（一帧或多帧的压缩数据）
130         packet=(AVPacket *)av_malloc(sizeof(AVPacket));
```

```
131         //Output Info-----------------------------
132         printf("--------------- File Information ----------------\n");
133         av_dump_format(pFormatCtx,0,filepath,0);
134         printf("-------------------------------------------------\n");
135         // 创建图像格式转换上下文：sws_getContext 定义转换规则
136         img_convert_ctx = sws_getContext(pCodecCtx->width, pCodecCtx->height,
    pCodecCtx->pix_fmt,
137                 pCodecCtx->width, pCodecCtx->height, AV_PIX_FMT_YUV420P,
    SWS_BICUBIC, NULL, NULL, NULL);
138 
139 #if OUTPUT_YUV420P
140     fp_yuv=fopen("output.yuv","wb+");
141 #endif
142 
143         // 初始化 SDL：启用视频（窗口）、音频（本代码未用）、定时器（用于延时）模块
144         if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER)) {
145                 printf( "Could not initialize SDL - %s\n", SDL_GetError());
146                 return -1;
147         }
148 
149         // 窗口宽高设为视频帧宽高
150         screen_w = pCodecCtx->width;
151         screen_h = pCodecCtx->height;
152         //SDL 2.0 Support for multiple windows
153         // 创建 SDL 窗口
154         screen = SDL_CreateWindow("Simplest ffmpeg player's Window",
    SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED,
155                 screen_w, screen_h,
156                 SDL_WINDOW_OPENGL);
157 
158         if(!screen) {
159                 printf("SDL: could not create window -
    exiting:%s\n",SDL_GetError());
160                 return -1;
161         }
162 
163         // 创建渲染器，绑定窗口
164         sdlRenderer = SDL_CreateRenderer(screen, -1, 0);
165         //IYUV: Y + U + V  (3 planes)
166         //YV12: Y + V + U  (3 planes)
167         // 创建 SDL 纹理：格式为 IYUV（即 YUV420P），流式访问（可动态更新数据）
168         sdlTexture = SDL_CreateTexture(sdlRenderer, SDL_PIXELFORMAT_IYUV,
    SDL_TEXTUREACCESS_STREAMING,pCodecCtx->width,pCodecCtx->height);
169 
170         sdlRect.x=0;
171         sdlRect.y=0;
172         // 渲染区域设置为整个窗口
```

```
173         sdlRect.w=screen_w;
174         sdlRect.h=screen_h;
175 
176         //SDL End----------------------
177         // 循环读取文件中的数据包
178         while(av_read_frame(pFormatCtx, packet)>=0){
179                 if(packet->stream_index==videoindex){  // 只处理视频流的数据包
180                         // 解码视频帧：avcodec_decode_video2 是旧版解码接口（新版用 send_packet/receive_frame）
181                         ret = avcodec_decode_video2(pCodecCtx, pFrame, &got_picture, packet);
182                         if(ret < 0){
183                                 printf("Decode Error.\n");
184                                 return -1;
185                         }
186                         if(got_picture){ // 成功解码出一帧
187                                 // 图像格式转换：将原始解码帧转换为 YUV420P 帧
188                                 sws_scale(img_convert_ctx, (const unsigned char* const*)pFrame->data, pFrame->linesize, 0, pCodecCtx->height,
189                                                 pFrameYUV->data, pFrameYUV->linesize);
190 
191 #if OUTPUT_YUV420P
192                                 y_size=pCodecCtx->width*pCodecCtx->height;
193                                 fwrite(pFrameYUV->data[0],1,y_size,fp_yuv);    //Y
194                                 fwrite(pFrameYUV->data[1],1,y_size/4,fp_yuv);  //U
195                                 fwrite(pFrameYUV->data[2],1,y_size/4,fp_yuv);  //V
196 #endif
197                                 //SDL---------------------------
198 #if 0
199                                 SDL_UpdateTexture( sdlTexture, NULL, pFrameYUV->data[0], pFrameYUV->linesize[0] );
200 #else
201                                 // 更新 YUV 三个分量（完整渲染）
202                                 // 更新到sdlTexture中
203                                 SDL_UpdateYUVTexture(sdlTexture, &sdlRect,
204                                 pFrameYUV->data[0], pFrameYUV->linesize[0],
205                                 pFrameYUV->data[1], pFrameYUV->linesize[1],
206                                 pFrameYUV->data[2], pFrameYUV->linesize[2]);
207 #endif
208                                 // 清空渲染器缓存
209                                 SDL_RenderClear( sdlRenderer );
210                                 // 将纹理复制到渲染器
211                                 SDL_RenderCopy( sdlRenderer, sdlTexture, NULL, &sdlRect);
```

```
212                                 // 显示渲染结果（刷新窗口）
213                                 SDL_RenderPresent( sdlRenderer );
214                                 //SDL End----------------------
215                                 //Delay 40ms
216                                 // 延时 40ms，控制播放帧率（约 25 帧/秒）
217                                 SDL_Delay(40);
218                         }
219                 }
220                 av_free_packet(packet);
221         }
222         //flush decoder
223         //FIX: Flush Frames remained in Codec
224         // flush decoder：处理解码器缓存中未输出的帧
225         while (1) {
226                 ret = avcodec_decode_video2(pCodecCtx, pFrame, &got_picture,
    packet);
227                 if (ret < 0)
228                         break;
229                 if (!got_picture)
230                         break;
231                 sws_scale(img_convert_ctx, (const unsigned char* const*)pFrame-
    >data, pFrame->linesize, 0, pCodecCtx->height,
232                         pFrameYUV->data, pFrameYUV->linesize);
233 #if OUTPUT_YUV420P
234                 int y_size=pCodecCtx->width*pCodecCtx->height;
235                 fwrite(pFrameYUV->data[0],1,y_size,fp_yuv);    //Y
236                 fwrite(pFrameYUV->data[1],1,y_size/4,fp_yuv);  //U
237                 fwrite(pFrameYUV->data[2],1,y_size/4,fp_yuv);  //V
238 #endif
239                 //SDL---------------------------
240                 SDL_UpdateTexture( sdlTexture, &sdlRect, pFrameYUV->data[0],
    pFrameYUV->linesize[0] );
241                 SDL_RenderClear( sdlRenderer );
242                 SDL_RenderCopy( sdlRenderer, sdlTexture,  NULL, &sdlRect);
243                 SDL_RenderPresent( sdlRenderer );
244                 //SDL End----------------------
245                 //Delay 40ms
246                 SDL_Delay(40);
247         }
248
249         sws_freeContext(img_convert_ctx);
250
251 #if OUTPUT_YUV420P
252     fclose(fp_yuv);
253 #endif
254
255         // 资源释放
```

```
256         SDL_Quit();
257
258         av_frame_free(&pFrameYUV);
259         av_frame_free(&pFrame);
260         avcodec_close(pCodecCtx);
261         avformat_close_input(&pFormatCtx);
262
263         return 0;
264 }
```

最简单的基于 FFmpeg的视频编码器

YUV编码为H.264

代码块

```
1  #define _CRT_SECURE_NO_WARNINGS  // 禁用VS的C运行时安全警告（针对fopen等函数）
2
3  #include <stdio.h>               // 标准输入输出头文件
4  #include <stdlib.h>              // 标准库头文件（内存分配、退出等）
5  #include <string.h>              // 字符串处理头文件
6  #include <iostream>              // C++输入输出流头文件
7
8  // FFmpeg头文件（使用extern "C"避免C++名称修饰问题）
9  extern "C" {
10 #include "libavutil/opt.h"       // FFmpeg选项设置头文件
11 #include "libavutil/imgutils.h" // FFmpeg图像工具头文件
12 #include "libavcodec/avcodec.h" // FFmpeg编解码核心头文件
13 #include "libavformat/avformat.h" // FFmpeg格式处理头文件
14 #include "libavutil/frame.h"     // FFmpeg帧处理头文件
15 #include "libavutil/mem.h"       // FFmpeg内存管理头文件
16 #include "libavutil/error.h"     // FFmpeg错误处理头文件
17 #include "libavutil/rational.h" // FFmpeg有理数（时间基）处理头文件
18 }
19
20 // 错误处理函数：将FFmpeg错误码转换为可读字符串并打印
21 void print_error(const char* msg, int errnum) {
22     char err_buf[AV_ERROR_MAX_STRING_SIZE] = { 0 };  // 定义错误信息缓冲区
23     av_strerror(errnum, err_buf, sizeof(err_buf));   // 将错误码转换为字符串
24     fprintf(stderr, "%s: %s\n", msg, err_buf);       // 输出错误信息到标准错误流
25 }
26
27 // 刷新编码器：处理编码器中缓冲的剩余帧
28 int flush_encoder(AVFormatContext* fmt_ctx, AVCodecContext* enc_ctx, int
   stream_idx) {
29     int ret = 0;                                     // 函数返回值初始化
```

```
30      AVPacket* pkt = av_packet_alloc();                  // 分配数据包结构
31      if (!pkt) {                                         // 检查内存分配是否成功
32          fprintf(stderr, "Failed to allocate packet\n"); // 输出错误信息
33          return AVERROR(ENOMEM);                         // 返回内存分配错误码
34      }
35
36      // 检查编码器是否支持延迟（是否有缓冲帧需要刷新）
37      if (!(enc_ctx->codec->capabilities & AV_CODEC_CAP_DELAY)) {
38          av_packet_free(&pkt);                           // 释放数据包
39          return 0;                                       // 返回成功
40      }
41      /*
42      当编码器支持延迟时，会在内部缓冲一些数据，不会立即输出编码后的数据包，主要原因为：
43      帧间预测需要参考帧：
44      I帧：关键帧，不需要参考其他帧
45      P帧：前向预测帧，需要参考前面的I/P帧
46      B帧：双向预测帧，需要参考前面和后面的帧
47      为了编码B帧，编码器需要先看到后面的帧，因此必须缓存当前帧，等待后续帧到达
48      */
49
50      while (1) {                                         // 无限循环直到刷新完成
51          // 编码器停止接收新的帧，开始输出内部缓存的所有编码结果
52          ret = avcodec_send_frame(enc_ctx, NULL);        // 发送NULL帧表示刷新编码器
53          if (ret == AVERROR_EOF) break;                  // 如果返回EOF，表示刷新完成
54
55          if (ret < 0) {                                  // 检查发送是否出错
56              print_error("Error sending flush frame", ret); // 打印错误信息
57              av_packet_free(&pkt);                       // 释放数据包
58              return ret;                                 // 返回错误码
59          }
60
61          ret = avcodec_receive_packet(enc_ctx, pkt);     // 接收编码后的数据包
62          if (ret == AVERROR(EAGAIN) || ret == AVERROR_EOF) // 需要更多输入或已结束
63              continue;                                   // 继续循环
64
65          if (ret < 0) {                                  // 检查接收是否出错
66              print_error("Error receiving flush packet", ret); // 打印错误信息
67              av_packet_free(&pkt);                       // 释放数据包
68              return ret;                                 // 返回错误码
69          }
70
71          // 将数据包的时间戳从编码器时间基转换为流的时间基
72          // 时间同步
73          av_packet_rescale_ts(pkt, enc_ctx->time_base, fmt_ctx-
    >streams[stream_idx]->time_base);
74          // 数据归属
75          pkt->stream_index = stream_idx;                 // 设置数据包所属的流索引
```

```
76
77          // 打印刷新的帧信息
78          printf("Flush Encoder: Succeed to encode 1 frame!\tsize:%d\n", pkt->size);
79
80          ret = av_interleaved_write_frame(fmt_ctx, pkt); // 将数据包写入输出文件
81          av_packet_unref(pkt);                           // 解引用数据包（释放引用）
82
83          if (ret < 0) {                                  // 检查写入是否出错
84              print_error("Error writing flush packet", ret); // 打印错误信息
85              av_packet_free(&pkt);                       // 释放数据包
86              return ret;                                 // 返回错误码
87          }
88      }
89
90      av_packet_free(&pkt);                               // 释放数据包
91      return 0;                                           // 返回成功
92  }
93
94  int main() {
95      // 定义变量：格式上下文指针（管理输入/输出格式）
96      AVFormatContext* fmt_ctx = NULL;
97      // 定义变量：编码器上下文指针（管理编码器参数和状态）
98      AVCodecContext* codec_ctx = NULL;
99      // 定义变量：编码器指针（指向具体的编码器实现）
100     const AVCodec* codec = NULL;
101     // 定义变量：视频流指针（表示输出文件中的视频流）
102     AVStream* video_stream = NULL;
103     // 定义变量：帧指针（存储未编码的原始视频数据）
104     AVFrame* frame = NULL;
105     // 定义变量：数据包指针（存储编码后的压缩数据）
106     AVPacket* pkt = NULL;
107     // 定义变量：YUV数据缓冲区指针（存储从文件读取的原始YUV数据）
108     uint8_t* picture_buf = NULL;
109     // 定义变量：输入文件指针（读取YUV文件）
110     FILE* in_file = NULL;
111
112     int ret = 0;                                        // 函数调用返回值
113     int frame_count = 0;                                // 已编码帧数计数器
114     int y_size = 0;                                     // YUV数据大小
115
116     // 配置参数：输入YUV文件路径
117     const char* input_file = "../ds_480x272.yuv";
118     // 配置参数：输出编码文件路径
119     const char* output_file = "ds.h264";
120     // 配置参数：输入视频宽度
121     int in_width = 480;
```

```
122     // 配置参数：输入视频高度
123     int in_height = 272;
124     // 配置参数：要编码的帧数
125     int frame_num = 100;
126     // 配置参数：目标比特率（400kbps）
127     int bit_rate = 400000;
128     // 配置参数：GOP大小（关键帧间隔）
129     int gop_size = 250;
130 
131     // 打开输入YUV文件（二进制读取模式）
132     in_file = fopen(input_file, "rb");
133     if (!in_file) {                                         // 检查文件是否成功打开
134         fprintf(stderr, "Could not open input file '%s'\n", input_file); // 输出
    错误信息
135         ret = -1;                                           // 设置返回错误码
136         goto cleanup;                                       // 跳转到资源清理部分
137     }
138 
139     // 创建输出格式上下文：根据输出文件名推断格式（如.h264对应H.264格式）
140     ret = avformat_alloc_output_context2(&fmt_ctx, NULL, NULL, output_file);
141     if (ret < 0) {                                          // 检查格式上下文创建是否成
    功
142         print_error("Could not create output context", ret); // 打印错误信息
143         goto cleanup;                                       // 跳转到资源清理部分
144     }
145 
146     // 根据输出格式查找合适的编码器（从格式上下文获取视频编码器ID）
147     codec = avcodec_find_encoder(fmt_ctx->oformat->video_codec);
148     if (!codec) {                                           // 检查编码器是否找到
149         fprintf(stderr, "Could not find encoder\n");  // 输出错误信息
150         ret = -1;                                           // 设置返回错误码
151         goto cleanup;                                       // 跳转到资源清理部分
152     }
153 
154     // 创建新的视频流：添加到格式上下文
155     video_stream = avformat_new_stream(fmt_ctx, NULL);
156     if (!video_stream) {                                    // 检查流创建是否成功
157         fprintf(stderr, "Could not create video stream\n"); // 输出错误信息
158         ret = -1;                                           // 设置返回错误码
159         goto cleanup;                                       // 跳转到资源清理部分
160     }
161 
162     // 分配编码器上下文：基于找到的编码器
163     codec_ctx = avcodec_alloc_context3(codec);
164     if (!codec_ctx) {                                       // 检查编码器上下文创建是否
    成功
165         fprintf(stderr, "Could not allocate codec context\n"); // 输出错误信息
```

```
166             ret = -1;                                       // 设置返回错误码
167             goto cleanup;                                   // 跳转到资源清理部分
168         }
169
170         // 设置编码器参数：编码器ID（与找到的编码器匹配）
171         codec_ctx->codec_id = codec->id;
172         // 设置编码器参数：媒体类型为视频
173         codec_ctx->codec_type = AVMEDIA_TYPE_VIDEO;
174         // 设置编码器参数：像素格式为YUV420P（大多数编码器支持的格式）
175         codec_ctx->pix_fmt = AV_PIX_FMT_YUV420P;
176         // 设置编码器参数：视频宽度
177         codec_ctx->width = in_width;
178         // 设置编码器参数：视频高度
179         codec_ctx->height = in_height;
180         // 设置编码器参数：目标比特率（影响视频质量和文件大小）
181         codec_ctx->bit_rate = bit_rate;
182         // 设置编码器参数：GOP大小（关键帧间隔，影响Seek性能）
183         codec_ctx->gop_size = gop_size;
184         // 设置编码器参数：时间基（1/25秒，表示每帧持续时间）
185         codec_ctx->time_base.num = 1;
186         codec_ctx->time_base.den = 25;
187         // 设置编码器参数：帧率（25fps）
188         codec_ctx->framerate.num = 25;
189         codec_ctx->framerate.den = 1;
190
191         // 如果输出格式需要全局头（如MP4格式），设置编码器标志
192         if (fmt_ctx->oformat->flags & AVFMT_GLOBALHEADER) {
193             codec_ctx->flags |= AV_CODEC_FLAG_GLOBAL_HEADER;
194         }
195
196         // 如果是H.264编码器，设置特定参数
197         if (codec_ctx->codec_id == AV_CODEC_ID_H264) {
198             // 设置编码预设为slow（质量更好，编码速度较慢）
199             av_opt_set(codec_ctx->priv_data, "preset", "slow", 0);
200             // 设置为零延迟模式（适合实时编码，无B帧）
201             av_opt_set(codec_ctx->priv_data, "tune", "zerolatency", 0);
202         }
203         // 如果是H.265编码器，设置特定参数
204         else if (codec_ctx->codec_id == AV_CODEC_ID_HEVC) {
205             // 设置编码预设为ultrafast（编码速度最快）
206             av_opt_set(codec_ctx->priv_data, "preset", "ultrafast", 0);
207             // 设置为零延迟模式
208             av_opt_set(codec_ctx->priv_data, "tune", "zero-latency", 0);
209         }
210
211         // 打开编码器：应用参数并初始化编码器
212         ret = avcodec_open2(codec_ctx, codec, NULL);
```

```
    if (ret < 0) {                                          // 检查编码器是否成功打开
        print_error("Could not open codec", ret);           // 打印错误信息
        goto cleanup;                                       // 跳转到资源清理部分
    }

    // 将编码器参数复制到流的codecpar结构中（用于写入文件头）
    ret = avcodec_parameters_from_context(video_stream->codecpar, codec_ctx);
    if (ret < 0) {                                          // 检查参数复制是否成功
        print_error("Could not copy codec parameters", ret); // 打印错误信息
        goto cleanup;                                       // 跳转到资源清理部分
    }
    // 设置流的时间基与编码器一致（确保时间戳同步）
    video_stream->time_base = codec_ctx->time_base;

    // 打印输出格式信息（调试用，显示编码器、分辨率等信息）
    av_dump_format(fmt_ctx, 0, output_file, 1);

    // 如果不是无文件格式（如RTMP流），打开输出文件IO
    if (!(fmt_ctx->oformat->flags & AVFMT_NOFILE)) {
        // 打开输出文件的IO上下文（只写模式）
        /*
        avio_open函数：
        1.打开指定的输出文件
        2.创建一个AVIOContext对象（IO上下文）
        3.将这个IO上下文赋值为格式上下文的pb字段
        4.后续的写操作都会通过这个IO上下文完成
        */
        ret = avio_open(&fmt_ctx->pb, output_file, AVIO_FLAG_WRITE);
        if (ret < 0) {                                      // 检查IO打开是否成功
            print_error("Could not open output file", ret); // 打印错误信息
            goto cleanup;                                   // 跳转到资源清理部分
        }
    }

    // 写入文件头：包含编码器信息、流信息等
    ret = avformat_write_header(fmt_ctx, NULL);
    if (ret < 0) {                                          // 检查文件头写入是否成功
        print_error("Error writing header", ret);           // 打印错误信息
        goto cleanup;                                       // 跳转到资源清理部分
    }

    // 分配帧结构：用于存储未编码的原始视频数据
    frame = av_frame_alloc();
    if (!frame) {                                           // 检查帧结构分配是否成功
        fprintf(stderr, "Could not allocate frame\n"); // 输出错误信息
        ret = -1;                                           // 设置返回错误码
        goto cleanup;                                       // 跳转到资源清理部分
```

```
260         }
261         // 设置帧的像素格式（与编码器一致）
262         frame->format = codec_ctx->pix_fmt;
263         // 设置帧的宽度（与编码器一致）
264         frame->width = codec_ctx->width;
265         // 设置帧的高度（与编码器一致）
266         frame->height = codec_ctx->height;
267 
268         // 为帧分配数据缓冲区（按32字节对齐，优化性能）
269         ret = av_frame_get_buffer(frame, 32);
270         if (ret < 0) {                                      // 检查缓冲区分配是否成功
271             print_error("Could not allocate frame buffer", ret); // 打印错误信息
272             goto cleanup;                                   // 跳转到资源清理部分
273         }
274 
275         // 分配数据包结构：用于存储编码后的压缩数据
276         pkt = av_packet_alloc();
277         if (!pkt) {                                         // 检查数据包分配是否成功
278             fprintf(stderr, "Could not allocate packet\n"); // 输出错误信息
279             ret = -1;                                       // 设置返回错误码
280             goto cleanup;                                   // 跳转到资源清理部分
281         }
282 
283         // 计算YUV420P格式一帧数据的大小（Y: width*height, U/V: 各1/4）
284         y_size = codec_ctx->width * codec_ctx->height;
285         // 分配YUV数据缓冲区（存储从文件读取的原始数据）
286         picture_buf = (uint8_t*)av_malloc(y_size * 3 / 2);
287         if (!picture_buf) {                                 // 检查缓冲区分配是否成功
288             fprintf(stderr, "Could not allocate picture buffer\n"); // 输出错误信息
289             ret = -1;                                       // 设置返回错误码
290             goto cleanup;                                   // 跳转到资源清理部分
291         }
292 
293         // 编码主循环：处理每一帧
294         for (int i = 0; i < frame_num; i++) {
295             // 从输入文件读取一帧YUV数据（YUV420P格式总大小为width*height*3/2）
296             size_t read_size = fread(picture_buf, 1, y_size * 3 / 2, in_file);
297             if (read_size != y_size * 3 / 2) {              // 检查是否读取到完整的帧数
    据
298                 fprintf(stderr, "Warning: Not enough data for frame %d\n", i); //
    输出警告
299                 break;                                      // 退出循环
300             }
301 
302             // 确保帧缓冲区可写（处理可能的引用计数，避免覆盖共享数据）
303             ret = av_frame_make_writable(frame);
304             if (ret < 0) {                                  // 检查帧是否成功设为可写
```

```
305             print_error("Could not make frame writable", ret); // 打印错误信息
306             goto cleanup;                                      // 跳转到资源清理部分
307         }
308 
309         // 填充Y分量数据（亮度通道）
310         memcpy(frame->data[0], picture_buf, y_size);
311         // 填充U分量数据（蓝色色度通道）
312         memcpy(frame->data[1], picture_buf + y_size, y_size / 4);
313         // 填充V分量数据（红色色度通道）
314         memcpy(frame->data[2], picture_buf + y_size * 5 / 4, y_size / 4);
315 
316         // 设置帧的显示时间戳（PTS）：用于同步和排序
317         frame->pts = i;
318 
319         // 将帧发送到编码器进行编码
320         ret = avcodec_send_frame(codec_ctx, frame);
321         if (ret < 0) {                                         // 检查发送是否出错
322             print_error("Error sending frame to encoder", ret); // 打印错误信息
323             break;                                             // 退出循环
324         }
325 
326         // 循环接收编码后的数据包（一个帧可能生成多个数据包）
327         while (ret >= 0) {
328             // 从编码器接收编码后的数据包
329             ret = avcodec_receive_packet(codec_ctx, pkt);
330             // 如果需要更多输入或编码器已刷新完成，退出循环
331             if (ret == AVERROR(EAGAIN) || ret == AVERROR_EOF)
332                 break;
333 
334             if (ret < 0) {                                     // 检查接收是否出错
335                 print_error("Error receiving packet", ret); // 打印错误信息
336                 goto cleanup;                                  // 跳转到资源清理部分
337             }
338 
339             // 将数据包的时间戳从编码器时间基转换为流的时间基
340             av_packet_rescale_ts(pkt, codec_ctx->time_base, video_stream-
    >time_base);
341             // 设置数据包所属的流索引（多流文件中标识所属流）
342             pkt->stream_index = video_stream->index;
343 
344             // 打印编码信息：帧数和数据包大小
345             printf("Encoded frame: %d\tsize:%d\n", frame_count++, pkt->size);
346 
347             // 将数据包写入输出文件（交错写入，保证格式正确性）
348             ret = av_interleaved_write_frame(fmt_ctx, pkt);
349             // 解引用数据包（释放缓冲区引用，避免内存泄漏）
350             av_packet_unref(pkt);
```

```
351
352             if (ret < 0) {                                  // 检查写入是否出错
353                 print_error("Error writing packet", ret); // 打印错误信息
354                 goto cleanup;                               // 跳转到资源清理部分
355             }
356         }
357     }
358
359     // 刷新编码器：处理编码器中缓冲的剩余帧
360     ret = flush_encoder(fmt_ctx, codec_ctx, video_stream->index);
361     if (ret < 0) {                                          // 检查刷新是否成功
362         fprintf(stderr, "Flushing encoder failed\n"); // 输出错误信息
363         goto cleanup;                                       // 跳转到资源清理部分
364     }
365
366     // 写入文件尾：更新文件索引、时长等信息
367     av_write_trailer(fmt_ctx);
368     // 打印成功信息
369     printf("Encoding completed successfully!\n");
370
371 cleanup: // 资源清理标签
372     // 释放YUV数据缓冲区
373     av_free(picture_buf);
374     // 释放数据包结构
375     av_packet_free(&pkt);
376     // 释放帧结构
377     av_frame_free(&frame);
378     // 释放编码器上下文
379     avcodec_free_context(&codec_ctx);
380
381     // 如果格式上下文已分配
382     if (fmt_ctx) {
383         // 如果不是无文件格式，关闭IO上下文
384         if (!(fmt_ctx->oformat->flags & AVFMT_NOFILE)) {
385             avio_closep(&fmt_ctx->pb);
386         }
387         // 释放格式上下文
388         avformat_free_context(fmt_ctx);
389     }
390
391     // 如果输入文件已打开，关闭文件
392     if (in_file) {
393         fclose(in_file);
394     }
395
396     // 返回执行结果：错误返回-1，成功返回0
397     return ret < 0 ? -1 : 0;
```

```
398    }
```

Windows下FFmpeg二次开发环境配置

1.下载预编译库

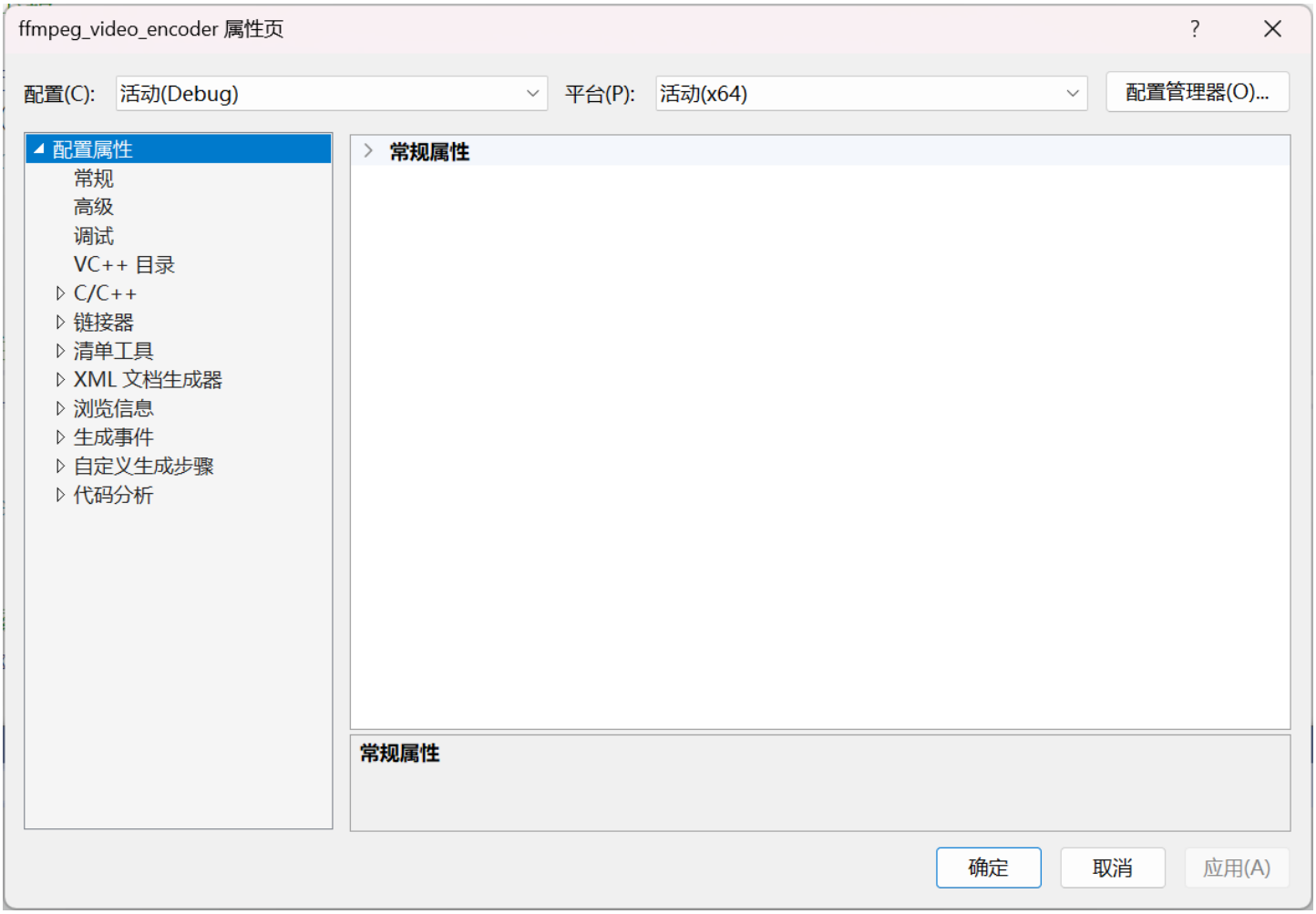
https://github.com/BtbN/FFmpeg-Builds/releases

从官网下载包含dll和开发文件的版本，如ffmpeg-n7.1-latest-win64-gpl-shared.zip

然后解压文件，解压后应该包含以下文件夹及对应的文件

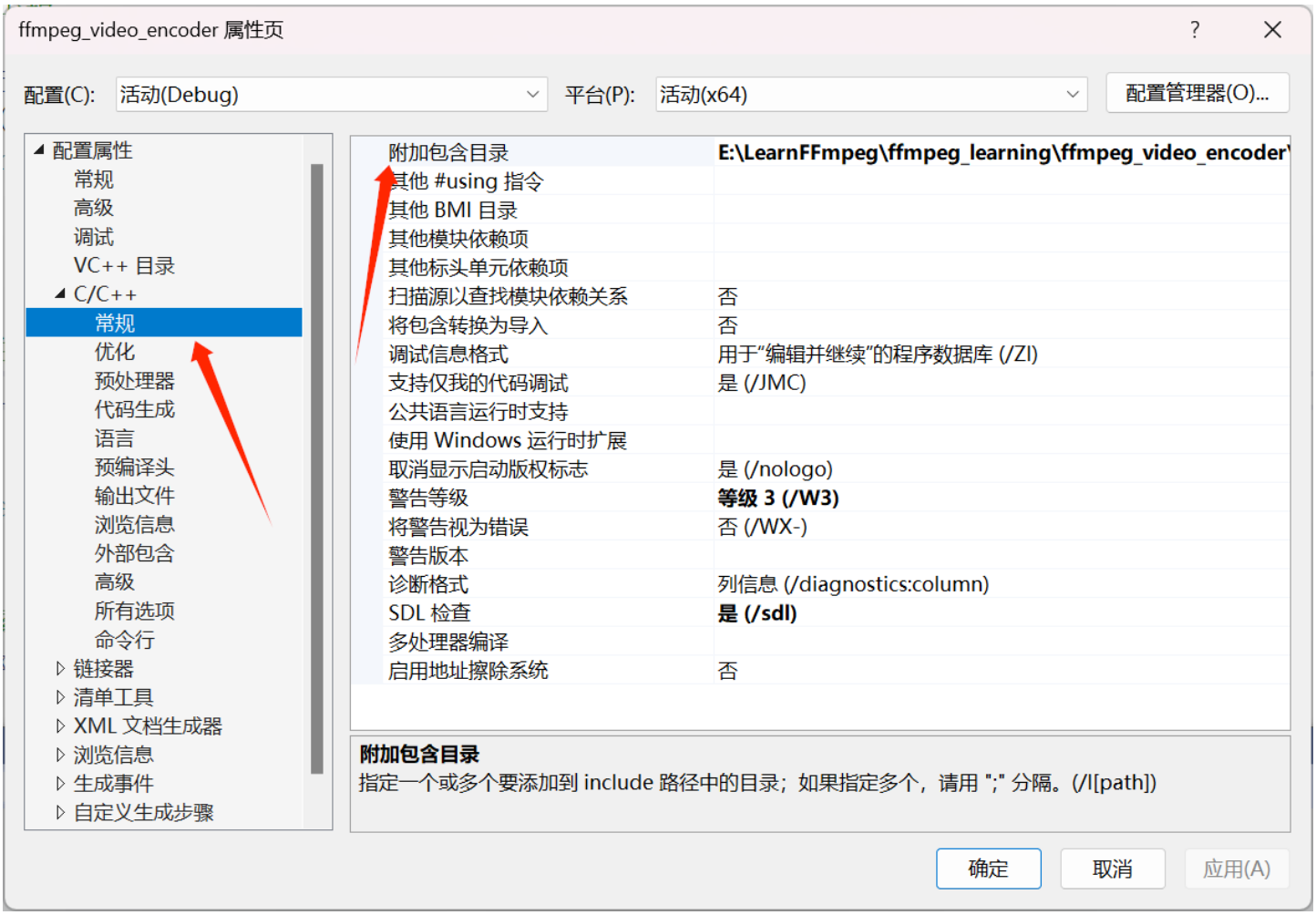
2.配置Visual Studio项目

右键项目 -> 属性 -> 配置属性



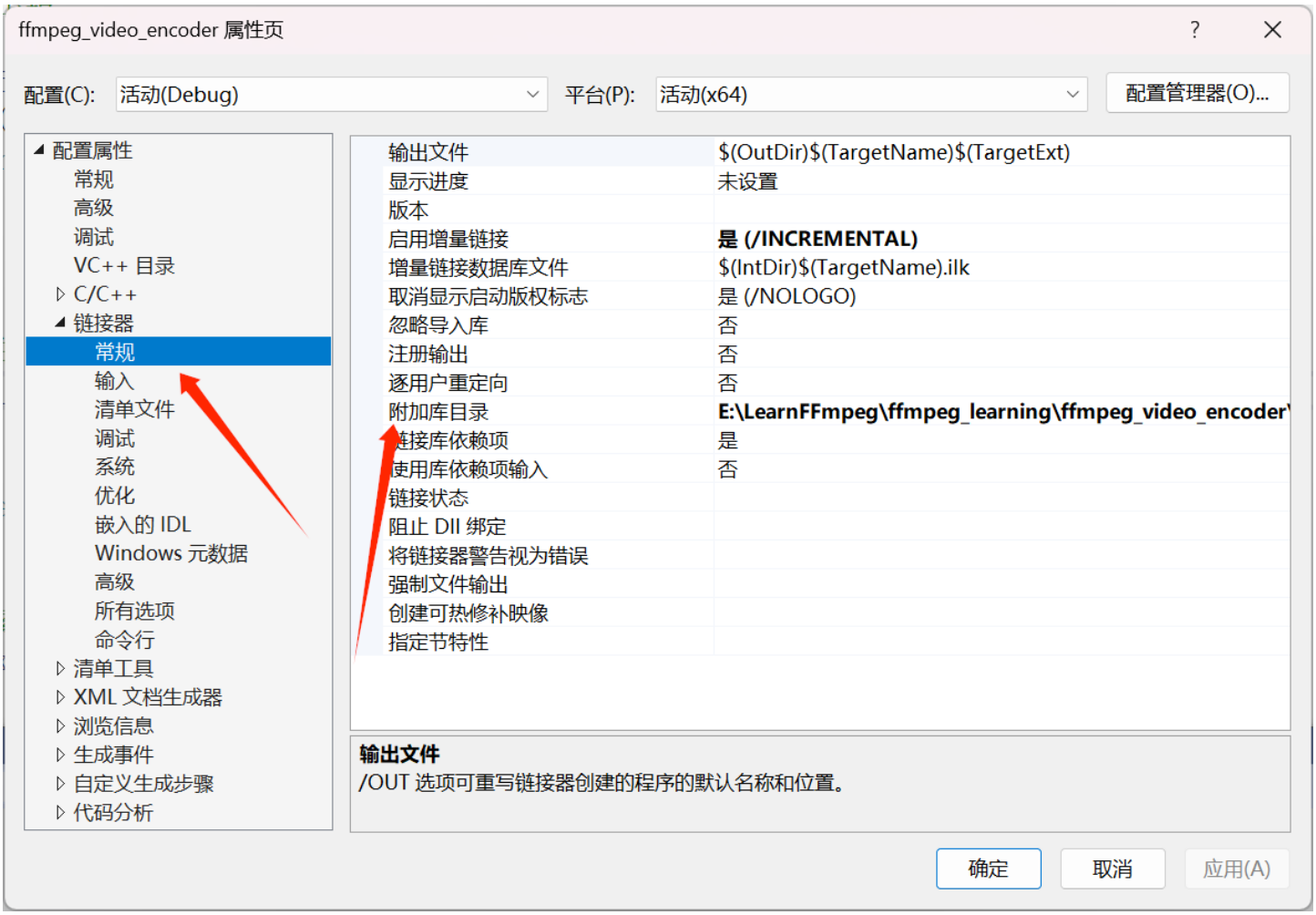
找到C/C++这一栏

附加包含目录

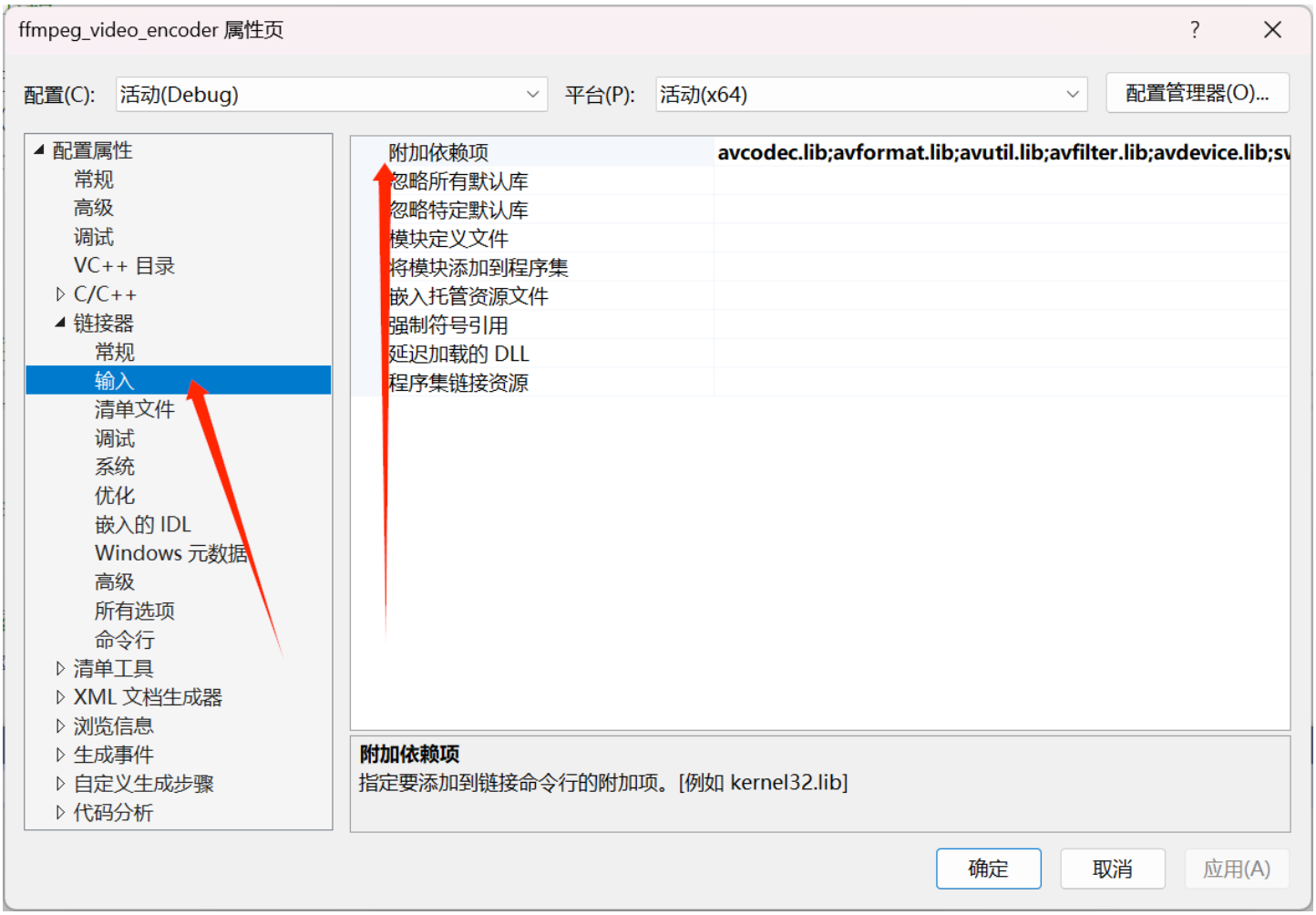


找到连接器这一栏

附加库目录


ffmpeg_video_encoder 属性页
配置(C): 活动(Debug)
平台(P): 活动(x64)
配置管理器(O)...
配置属性
常规
高级
调试
VC++ 目录
C/C++
链接器
常规
输入
清单文件
调试
系统
优化
嵌入的 IDL
Windows 元数据
高级
所有选项
命令行
清单工具
XML 文档生成器
浏览信息
生成事件
自定义生成步骤
代码分析
输出文件 $(OutDir)$(TargetName)$(TargetExt)
显示进度 未设置
版本
启用增量链接 是 (/INCREMENTAL)
增量链接数据库文件 $(IntDir)$(TargetName).ilk
取消显示启动版权标志 是 (/NOLOGO)
忽略导入库 否
注册输出 否
逐用户重定向 否
附加库目录 E:\LearnFFmpeg\ffmpeg_learning\ffmpeg_video_encoder
链接库依赖项 是
使用库依赖项输入 否
链接状态
阻止 DII 绑定
将链接器警告视为错误
强制文件输出
创建可热修补映像
指定节特性
输出文件
/OUT 选项可重写链接器创建的程序的默认名称和位置。
确定
取消
应用(A)

链接器 -> 输入 -> 附加依赖项

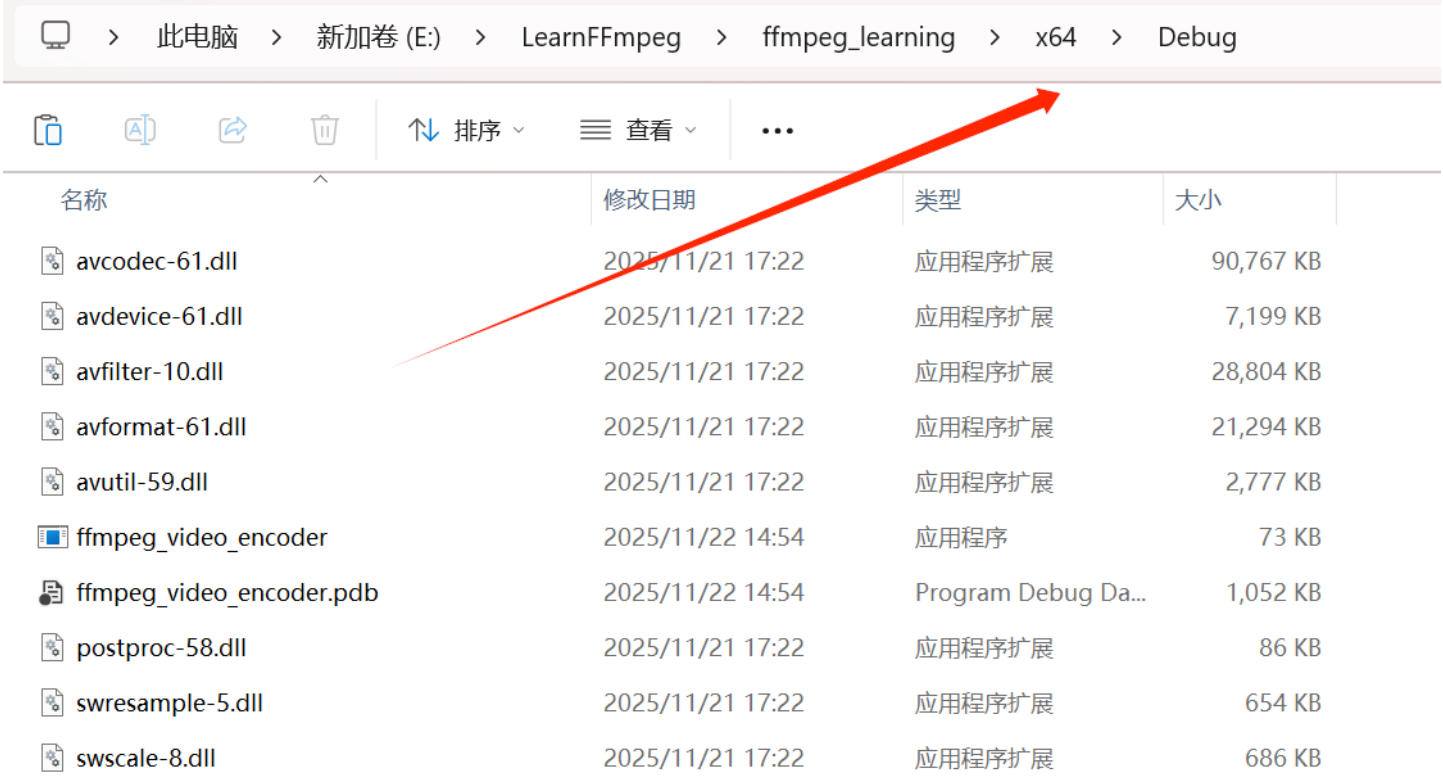


添加需要的库文件，例如

代码块

```
avcodec.lib
avformat.lib
avutil.lib
avfilter.lib
swscale.lib
swresample.lib
```

将FFmpeg bin下的dll文件复制到可执行文件同目录



FFmpeg源码剖析

av_register_all

该函数在所有基于ffmpeg的应用程序中几乎都是第一个被调用的，只有调用了该函数，才能使用复用器，编码器等

代码块

```
// 这里是一个宏定义
#define REGISTER_DEMUXER(X,x) { \
    extern AVInputFormat ff_##x##_demuxer; \
    if(CONFIG_##X##_DEMUXER) av_register_input_format(&ff_##x##_demuxer); }
/*
这是一个用于条件注册 FFmpeg 解复用器的宏，核心是通过宏拼接生成变量名和编译控制宏。
使用 ## 令牌粘贴操作符来动态生成特定格式的变量名和配置宏名。
利用 extern 声明外部变量，并通过 av_register_input_format 函数完成注册，实现了代码的复用和条件编译控制。
*/
```

代码块

```
// av_register_all源码如下
void av_register_all(void)
{
    static int initialized;

```

```
6       if (initialized)
7           return;
8       initialized = 1;
9       //注册所有的codec
10      avcodec_register_all();
11      //注册所有的MUXER（复用器和解复用器）
12      /* (de)muxers */
13      REGISTER_MUXER    (A64, a64);
14      REGISTER_DEMUXER  (AAC, aac);
15      REGISTER_MUXDEMUX (AC3, ac3);
16      REGISTER_DEMUXER  (ACT, act);
17      REGISTER_DEMUXER  (ADF, adf);
18      REGISTER_MUXER    (ADTS, adts);
19      REGISTER_MUXDEMUX (ADX, adx);
20      REGISTER_DEMUXER  (AEA, aea);
21      REGISTER_MUXDEMUX (AIFF, aiff);
22      REGISTER_MUXDEMUX (AMR, amr);
23      REGISTER_DEMUXER  (ANM, anm);
24      REGISTER_DEMUXER  (APC, apc);
25      REGISTER_DEMUXER  (APE, ape);
26      REGISTER_DEMUXER  (APPLEHTTP, applehttp);
27      REGISTER_MUXDEMUX (ASF, asf);
28      REGISTER_MUXDEMUX (ASS, ass);
29      REGISTER_MUXER    (ASF_STREAM, asf_stream);
30      REGISTER_MUXDEMUX (AU, au);
31      REGISTER_MUXDEMUX (AVI, avi);
32      REGISTER_DEMUXER  (AVISYNTH, avisynth);
33      REGISTER_MUXER    (AVM2, avm2);
34      REGISTER_DEMUXER  (AVS, avs);
35      REGISTER_DEMUXER  (BETHSOFTVID, bethsoftvid);
36      REGISTER_DEMUXER  (BFI, bfi);
37      REGISTER_DEMUXER  (BINTEXT, bintext);
38      REGISTER_DEMUXER  (BINK, bink);
39      REGISTER_MUXDEMUX (BIT, bit);
40      REGISTER_DEMUXER  (BMV, bmv);
41      REGISTER_DEMUXER  (C93, c93);
42      REGISTER_MUXDEMUX (CAF, caf);
43      REGISTER_MUXDEMUX (CAVSVIDEO, cavsvideo);
44      REGISTER_DEMUXER  (CDG, cdg);
45      REGISTER_MUXER    (CRC, crc);
46      REGISTER_MUXDEMUX (DAUD, daud);
47      REGISTER_DEMUXER  (DFA, dfa);
48      REGISTER_MUXDEMUX (DIRAC, dirac);
49      REGISTER_MUXDEMUX (DNXHD, dnxhd);
50      REGISTER_DEMUXER  (DSICIN, dsicin);
51      REGISTER_MUXDEMUX (DTS, dts);
52      REGISTER_MUXDEMUX (DV, dv);
```

```
53      REGISTER_DEMUXER  (DXA, dxa);
54      REGISTER_DEMUXER  (EA, ea);
55      REGISTER_DEMUXER  (EA_CDATA, ea_cdata);
56      REGISTER_MUXDEMUX (EAC3, eac3);
57      REGISTER_MUXDEMUX (FFM, ffm);
58      REGISTER_MUXDEMUX (FFMETADATA, ffmetadata);
59      REGISTER_MUXDEMUX (FILMSTRIP, filmstrip);
60      REGISTER_MUXDEMUX (FLAC, flac);
61      REGISTER_DEMUXER  (FLIC, flic);
62      REGISTER_MUXDEMUX (FLV, flv);
63      REGISTER_DEMUXER  (FOURXM, fourxm);
64      REGISTER_MUXER    (FRAMECRC, framecrc);
65      REGISTER_MUXER    (FRAMEMD5, framemd5);
66      REGISTER_MUXDEMUX (G722, g722);
67      REGISTER_MUXDEMUX (G723_1, g723_1);
68      REGISTER_DEMUXER  (G729, g729);
69      REGISTER_MUXER    (GIF, gif);
70      REGISTER_DEMUXER  (GSM, gsm);
71      REGISTER_MUXDEMUX (GXF, gxf);
72      REGISTER_MUXDEMUX (H261, h261);
73      REGISTER_MUXDEMUX (H263, h263);
74      REGISTER_MUXDEMUX (H264, h264);
75      REGISTER_DEMUXER  (ICO, ico);
76      REGISTER_DEMUXER  (IDCIN, idcin);
77      REGISTER_DEMUXER  (IDF, idf);
78      REGISTER_DEMUXER  (IFF, iff);
79      REGISTER_MUXDEMUX (IMAGE2, image2);
80      REGISTER_MUXDEMUX (IMAGE2PIPE, image2pipe);
81      REGISTER_DEMUXER  (INGENIENT, ingenient);
82      REGISTER_DEMUXER  (IPMOVIE, ipmovie);
83      REGISTER_MUXER    (IPOD, ipod);
84      REGISTER_MUXER    (ISMV, ismv);
85      REGISTER_DEMUXER  (ISS, iss);
86      REGISTER_DEMUXER  (IV8, iv8);
87      REGISTER_MUXDEMUX (IVF, ivf);
88      REGISTER_DEMUXER  (JV, jv);
89      REGISTER_MUXDEMUX (LATM, latm);
90      REGISTER_DEMUXER  (LMLM4, lmlm4);
91      REGISTER_DEMUXER  (LOAS, loas);
92      REGISTER_DEMUXER  (LXF, lxf);
93      REGISTER_MUXDEMUX (M4V, m4v);
94      REGISTER_MUXER    (MD5, md5);
95      REGISTER_MUXDEMUX (MATROSKA, matroska);
96      REGISTER_MUXER    (MATROSKA_AUDIO, matroska_audio);
97      REGISTER_MUXDEMUX (MICRODVD, microdvd);
98      REGISTER_MUXDEMUX (MJPEG, mjpeg);
99      REGISTER_MUXDEMUX (MLP, mlp);
```

```
100       REGISTER_DEMUXER  (MM, mm);
101       REGISTER_MUXDEMUX (MMF, mmf);
102       REGISTER_MUXDEMUX (MOV, mov);
103       REGISTER_MUXER    (MP2, mp2);
104       REGISTER_MUXDEMUX (MP3, mp3);
105       REGISTER_MUXER    (MP4, mp4);
106       REGISTER_DEMUXER  (MPC, mpc);
107       REGISTER_DEMUXER  (MPC8, mpc8);
108       REGISTER_MUXER    (MPEG1SYSTEM, mpeg1system);
109       REGISTER_MUXER    (MPEG1VCD, mpeg1vcd);
110       REGISTER_MUXER    (MPEG1VIDEO, mpeg1video);
111       REGISTER_MUXER    (MPEG2DVD, mpeg2dvd);
112       REGISTER_MUXER    (MPEG2SVCD, mpeg2svcd);
113       REGISTER_MUXER    (MPEG2VIDEO, mpeg2video);
114       REGISTER_MUXER    (MPEG2VOB, mpeg2vob);
115       REGISTER_DEMUXER  (MPEGPS, mpegps);
116       REGISTER_MUXDEMUX (MPEGTS, mpegts);
117       REGISTER_DEMUXER  (MPEGTSRAW, mpegtsraw);
118       REGISTER_DEMUXER  (MPEGVIDEO, mpegvideo);
119       REGISTER_MUXER    (MPJPEG, mpjpeg);
120       REGISTER_DEMUXER  (MSNWC_TCP, msnwc_tcp);
121       REGISTER_DEMUXER  (MTV, mtv);
122       REGISTER_DEMUXER  (MVI, mvi);
123       REGISTER_MUXDEMUX (MXF, mxf);
124       REGISTER_MUXER    (MXF_D10, mxf_d10);
125       REGISTER_DEMUXER  (MXG, mxg);
126       REGISTER_DEMUXER  (NC, nc);
127       REGISTER_DEMUXER  (NSV, nsv);
128       REGISTER_MUXER    (NULL, null);
129       REGISTER_MUXDEMUX (NUT, nut);
130       REGISTER_DEMUXER  (NUV, nuv);
131       REGISTER_MUXDEMUX (OGG, ogg);
132       REGISTER_MUXDEMUX (OMA, oma);
133       REGISTER_MUXDEMUX (PCM_ALAW,  pcm_alaw);
134       REGISTER_MUXDEMUX (PCM_MULAW, pcm_mulaw);
135       REGISTER_MUXDEMUX (PCM_F64BE, pcm_f64be);
136       REGISTER_MUXDEMUX (PCM_F64LE, pcm_f64le);
137       REGISTER_MUXDEMUX (PCM_F32BE, pcm_f32be);
138       REGISTER_MUXDEMUX (PCM_F32LE, pcm_f32le);
139       REGISTER_MUXDEMUX (PCM_S32BE, pcm_s32be);
140       REGISTER_MUXDEMUX (PCM_S32LE, pcm_s32le);
141       REGISTER_MUXDEMUX (PCM_S24BE, pcm_s24be);
142       REGISTER_MUXDEMUX (PCM_S24LE, pcm_s24le);
143       REGISTER_MUXDEMUX (PCM_S16BE, pcm_s16be);
144       REGISTER_MUXDEMUX (PCM_S16LE, pcm_s16le);
145       REGISTER_MUXDEMUX (PCM_S8,    pcm_s8);
146       REGISTER_MUXDEMUX (PCM_U32BE, pcm_u32be);
```

```
147       REGISTER_MUXDEMUX (PCM_U32LE, pcm_u32le);
148       REGISTER_MUXDEMUX (PCM_U24BE, pcm_u24be);
149       REGISTER_MUXDEMUX (PCM_U24LE, pcm_u24le);
150       REGISTER_MUXDEMUX (PCM_U16BE, pcm_u16be);
151       REGISTER_MUXDEMUX (PCM_U16LE, pcm_u16le);
152       REGISTER_MUXDEMUX (PCM_U8,    pcm_u8);
153       REGISTER_DEMUXER  (PMP, pmp);
154       REGISTER_MUXER    (PSP, psp);
155       REGISTER_DEMUXER  (PVA, pva);
156       REGISTER_DEMUXER  (QCP, qcp);
157       REGISTER_DEMUXER  (R3D, r3d);
158       REGISTER_MUXDEMUX (RAWVIDEO, rawvideo);
159       REGISTER_DEMUXER  (RL2, rl2);
160       REGISTER_MUXDEMUX (RM, rm);
161       REGISTER_MUXDEMUX (ROQ, roq);
162       REGISTER_DEMUXER  (RPL, rpl);
163       REGISTER_MUXDEMUX (RSO, rso);
164       REGISTER_MUXDEMUX (RTP, rtp);
165       REGISTER_MUXDEMUX (RTSP, rtsp);
166       REGISTER_MUXDEMUX (SAP, sap);
167       REGISTER_DEMUXER  (SBG, sbg);
168       REGISTER_DEMUXER  (SDP, sdp);
169   #if CONFIG_RTPDEC
170       av_register_rtp_dynamic_payload_handlers();
171       av_register_rdt_dynamic_payload_handlers();
172   #endif
173       REGISTER_DEMUXER  (SEGAFILM, segafilm);
174       REGISTER_MUXER    (SEGMENT, segment);
175       REGISTER_DEMUXER  (SHORTEN, shorten);
176       REGISTER_DEMUXER  (SIFF, siff);
177       REGISTER_DEMUXER  (SMACKER, smacker);
178       REGISTER_MUXDEMUX (SMJPEG, smjpeg);
179       REGISTER_DEMUXER  (SOL, sol);
180       REGISTER_MUXDEMUX (SOX, sox);
181       REGISTER_MUXDEMUX (SPDIF, spdif);
182       REGISTER_MUXDEMUX (SRT, srt);
183       REGISTER_DEMUXER  (STR, str);
184       REGISTER_MUXDEMUX (SWF, swf);
185       REGISTER_MUXER    (TG2, tg2);
186       REGISTER_MUXER    (TGP, tgp);
187       REGISTER_DEMUXER  (THP, thp);
188       REGISTER_DEMUXER  (TIERTEXSEQ, tiertexseq);
189       REGISTER_MUXER    (MKVTIMESTAMP_V2, mkvtimestamp_v2);
190       REGISTER_DEMUXER  (TMV, tmv);
191       REGISTER_MUXDEMUX (TRUEHD, truehd);
192       REGISTER_DEMUXER  (TTA, tta);
193       REGISTER_DEMUXER  (TXD, txd);
```

```
194         REGISTER_DEMUXER  (TTY, tty);
195         REGISTER_DEMUXER  (VC1, vc1);
196         REGISTER_MUXDEMUX (VC1T, vc1t);
197         REGISTER_DEMUXER  (VMD, vmd);
198         REGISTER_MUXDEMUX (VOC, voc);
199         REGISTER_DEMUXER  (VQF, vqf);
200         REGISTER_DEMUXER  (W64, w64);
201         REGISTER_MUXDEMUX (WAV, wav);
202         REGISTER_DEMUXER  (WC3, wc3);
203         REGISTER_MUXER    (WEBM, webm);
204         REGISTER_DEMUXER  (WSAUD, wsaud);
205         REGISTER_DEMUXER  (WSVQA, wsvqa);
206         REGISTER_MUXDEMUX (WTV, wtv);
207         REGISTER_DEMUXER  (WV, wv);
208         REGISTER_DEMUXER  (XA, xa);
209         REGISTER_DEMUXER  (XBIN, xbin);
210         REGISTER_DEMUXER  (XMV, xmv);
211         REGISTER_DEMUXER  (XWMA, xwma);
212         REGISTER_DEMUXER  (YOP, yop);
213         REGISTER_MUXDEMUX (YUV4MPEGPIPE, yuv4mpegpipe);
214
215         /* external libraries */
216     #if CONFIG_LIBMODPLUG
217         REGISTER_DEMUXER  (LIBMODPLUG, libmodplug);
218     #endif
219         REGISTER_MUXDEMUX (LIBNUT, libnut);
220         //注册所有的Protocol（位于DEMUXER之前（我的理解~~））
221         //文件也是一种Protocol
222         /* protocols */
223         REGISTER_PROTOCOL (APPLEHTTP, applehttp);
224         REGISTER_PROTOCOL (CACHE, cache);
225         REGISTER_PROTOCOL (CONCAT, concat);
226         REGISTER_PROTOCOL (CRYPTO, crypto);
227         REGISTER_PROTOCOL (FILE, file);
228         REGISTER_PROTOCOL (GOPHER, gopher);
229         REGISTER_PROTOCOL (HTTP, http);
230         REGISTER_PROTOCOL (HTTPPROXY, httpproxy);
231         REGISTER_PROTOCOL (HTTPS, https);
232         REGISTER_PROTOCOL (MMSH, mmsh);
233         REGISTER_PROTOCOL (MMST, mmst);
234         REGISTER_PROTOCOL (MD5,  md5);
235         REGISTER_PROTOCOL (PIPE, pipe);
236         REGISTER_PROTOCOL (RTMP, rtmp);
237     //如果包含了LibRTMP
238     #if CONFIG_LIBRTMP
239         REGISTER_PROTOCOL (RTMP, rtmpt);
240         REGISTER_PROTOCOL (RTMP, rtmpe);
```

```
241     REGISTER_PROTOCOL (RTMP, rtmpte);
242     REGISTER_PROTOCOL (RTMP, rtmps);
243 #endif
244     REGISTER_PROTOCOL (RTP, rtp);
245     REGISTER_PROTOCOL (TCP, tcp);
246     REGISTER_PROTOCOL (TLS, tls);
247     REGISTER_PROTOCOL (UDP, udp);
248 }
```

代码块

```
1  void av_register_output_format(AVOutputFormat *format)
2  {
3      AVOutputFormat **p;
4      p = &first_oformat;
5      while (*p != NULL) p = &(*p)->next;
6      *p = format;
7      format->next = NULL;
8  }
9
10 void av_register_input_format(AVInputFormat *format)
11 {
12     AVInputFormat **p;
13     p = &first_iformat;
14     while (*p != NULL) p = &(*p)->next;
15     *p = format;
16     format->next = NULL;
17 }
18
19 int ffurl_register_protocol(URLProtocol *protocol)
20 {
21     URLProtocol **p;
22     p = &first_protocol;
23     while (*p)
24         p = &(*p)->next;
25     *p             = protocol;
26     protocol->next = NULL;
27     return 0;
28 }
```

avcodec_register_all

代码块

```
1  #define REGISTER_ENCODER(X,x) { \
```

```
            extern AVCodec ff_##x##_encoder; \
            if(CONFIG_##X##_ENCODER)  avcodec_register(&ff_##x##_encoder); }
#define REGISTER_DECODER(X,x) { \
            extern AVCodec ff_##x##_decoder; \
            if(CONFIG_##X##_DECODER)  avcodec_register(&ff_##x##_decoder); }
#define REGISTER_ENCDEC(X,x)  REGISTER_ENCODER(X,x); REGISTER_DECODER(X,x)
```

代码块

```
//注册所有的AVCodec
void avcodec_register(AVCodec *codec)
{
    AVCodec **p;
    //初始化
    avcodec_init();
    //从第一个开始
    p = &first_avcodec;
    while (*p != NULL) p = &(*p)->next;
    *p = codec;
    codec->next = NULL;

    if (codec->init_static_data)
        codec->init_static_data(codec);
}

/* encoder management */
static AVCodec *first_avcodec = NULL;
```

常见结构体的初始化和销毁

代码块

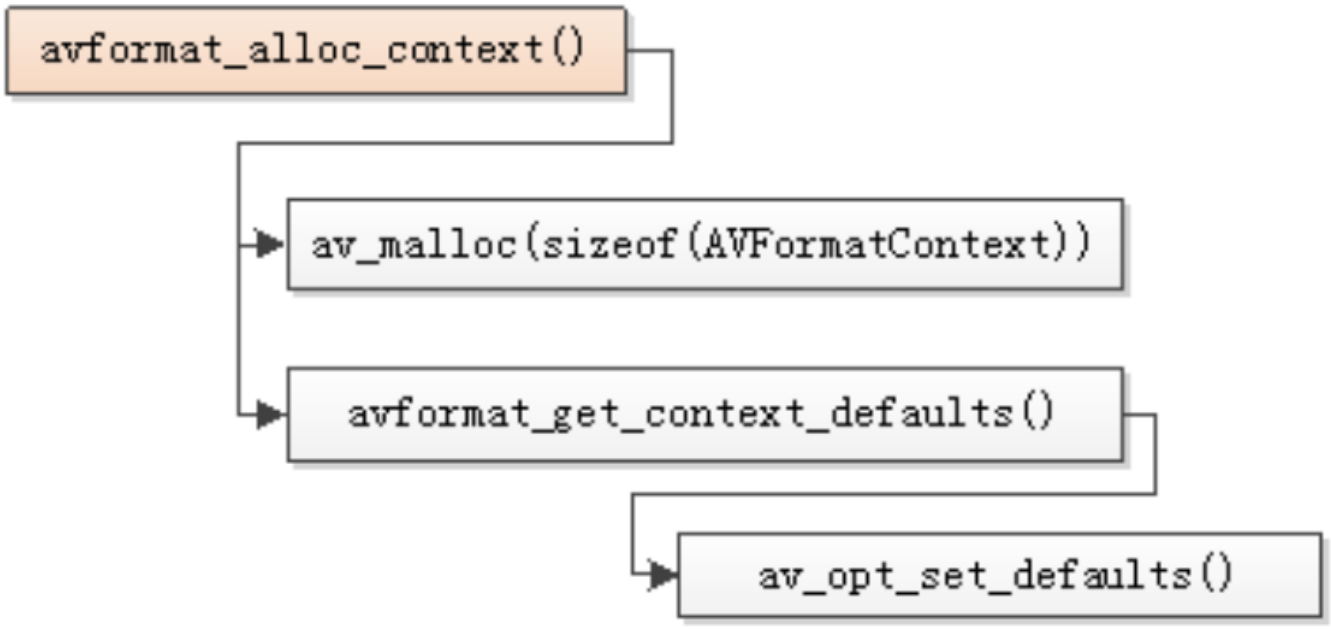
```
// 结构体              // 初始化                                  // 销毁
AVFormatContext       avformat_alloc_context()
avformat_free_context()
AVIOContext           avio_alloc_context()
AVStream              avformat_new_stream()
AVCodecContext        avcodec_alloc_context3()
AVFrame               av_frame_alloc();av_image_fill_arrays()    av_frame_free()
AVPacket              av_init_packet();av_new_packet()
av_free_packet()
```

avformat_alloc_context

代码块

```
AVFormatContext *avformat_alloc_context(void)
{
    AVFormatContext *ic;
    ic = av_malloc(sizeof(AVFormatContext));
    if (!ic) return ic;
    // 用于设置AVFormatContext字段的默认值
    avformat_get_context_defaults(ic);


    ic->internal = av_mallocz(sizeof(*ic->internal));
    if (!ic->internal) {
        avformat_free_context(ic);
        return NULL;
    }


    return ic;
}
/*
从代码中可以看出，avformat_alloc_context()调用av_malloc()为AVFormatContext结构体分配了内存，而且同时也给AVFormatContext中的internal字段分配内存（这个字段是FFmpeg内部使用的，先不分析）。此外调用了一个avformat_get_context_defaults()函数。该函数用于设置AVFormatContext的字段的默认值。它的定义也位于libavformat\options.c，确切的说就位于avformat_alloc_context()上面。我们看一下该函数的定义
*/

static void avformat_get_context_defaults(AVFormatContext *s)
{
    memset(s, 0, sizeof(AVFormatContext));


    s->av_class = &av_format_context_class;


    av_opt_set_defaults(s);
}
/*
从代码可以看出，avformat_get_context_defaults()首先调用memset()将AVFormatContext的所有字段置0。而后调用了一个函数av_opt_set_defaults()。av_opt_set_defaults()用于给字段设置默认值。
*/
```

avformat_alloc_context()代码的函数调用关系如下图所示



avformat_free_context

代码块

```
void avformat_free_context(AVFormatContext *s)
{
    int i;


    if (!s)
        return;


    av_opt_free(s);
    if (s->iformat && s->iformat->priv_class && s->priv_data)
        av_opt_free(s->priv_data);
    if (s->oformat && s->oformat->priv_class && s->priv_data)
        av_opt_free(s->priv_data);


    for (i = s->nb_streams - 1; i >= 0; i--) {
        ff_free_stream(s, s->streams[i]);
    }
    for (i = s->nb_programs - 1; i >= 0; i--) {
        av_dict_free(&s->programs[i]->metadata);
        av_freep(&s->programs[i]->stream_index);
        av_freep(&s->programs[i]);
    }
    av_freep(&s->programs);
    av_freep(&s->priv_data);
    while (s->nb_chapters--) {
```

```
28          av_dict_free(&s->chapters[s->nb_chapters]->metadata);
29          av_freep(&s->chapters[s->nb_chapters]);
30      }
31      av_freep(&s->chapters);
32      av_dict_free(&s->metadata);
33      av_freep(&s->streams);
34      av_freep(&s->internal);
35      flush_packet_queue(s);
36      av_free(s);
37  }
38  /*
39  从代码中可以看出，avformat_free_context()调用了各式各样的销毁函数：av_opt_free()，
    av_freep()，av_dict_free()。这些函数分别用于释放不同种类的变量，在这里不再详细讨论。在
    这里看一个释放AVStream的函数ff_free_stream()。该函数的定义位于libavformat\options.c
    （其实就在avformat_free_context()上方）
40  */
41  void ff_free_stream(AVFormatContext *s, AVStream *st) {
42      int j;
43      av_assert0(s->nb_streams>0);
44      av_assert0(s->streams[ s->nb_streams - 1 ] == st);
45
46
47      for (j = 0; j < st->nb_side_data; j++)
48          av_freep(&st->side_data[j].data);
49      av_freep(&st->side_data);
50      st->nb_side_data = 0;
51
52
53      if (st->parser) {
54          av_parser_close(st->parser);
55      }
56      if (st->attached_pic.data)
57          av_free_packet(&st->attached_pic);
58      av_dict_free(&st->metadata);
59      av_freep(&st->probe_data.buf);
60      av_freep(&st->index_entries);
61      av_freep(&st->codec->extradata);
62      av_freep(&st->codec->subtitle_header);
63      av_freep(&st->codec);
64      av_freep(&st->priv_data);
65      if (st->info)
66          av_freep(&st->info->duration_error);
67      av_freep(&st->info);
68      av_freep(&s->streams[ --s->nb_streams ]);
69  }
70  /*
```

```
从代码中可以看出，与释放AVFormatContext类似，释放AVStream的时候，也是调用了av_freep()，av_dict_free()这些函数释放有关的字段。如果使用了parser的话，会调用av_parser_close()关闭该parser
*/
```

avformat_open_input

代码块

```
int avformat_open_input(AVFormatContext **ps, const char *filename,AVInputFormat *fmt, AVDictionary **options);
/*
ps：函数调用成功之后处理过的AVFormatContext结构体
filename：打开的视音频流的URL
fmt：强制指定AVFormatContext中AVInputFormat，这个参数一般情况下可以设置为NULL，这样FFmpeg可以自动检测AVInputFormat
dictionay：附加的一些选项，一般情况下可以设置为NULL
*/
```

代码块

```
/*
其核心功能是打开一个媒体文件（或网络流），初始化 AVFormatContext 结构体，并读取文件的头部信息以获取流的基本信息，为后续的音视频解码做好准备
*/
int avformat_open_input(AVFormatContext **ps, const char *filename,
                        AVInputFormat *fmt, AVDictionary **options)
{
    // 定义一个指向 AVFormatContext 的指针，并初始化为传入的 *ps，这是函数操作的核心上下文对象
    AVFormatContext *s = *ps;
    // 用于存储函数调用的返回值，初始化为 0（表示成功）
    int ret = 0;
    // 用于临时存储用户传入的选项字典，避免直接修改原始字典
    AVDictionary *tmp = NULL;
    // 用于存储从媒体文件中解析出的 ID3v2 额外元数据（如封面图片）
    ID3v2ExtraMeta *id3v2_extra_meta = NULL;
    // 如果用户没有传入 AVFormatContext 对象（*ps 为 NULL），则调用 avformat_alloc_context()      // 分配一个新的。如果分配失败，返回内存不足错误
    if (!s && !(s = avformat_alloc_context()))
        return AVERROR(ENOMEM);
    // 检查 s->av_class 是否存在，这是 FFmpeg 中所有支持 AVOption 的结构体都应有的字段，用于校验     // 上下文是否合法
    if (!s->av_class) {
```

```
20          av_log(NULL, AV_LOG_ERROR, "Input context has not been properly
    allocated by avformat_alloc_context() and is not NULL either\n");
21          return AVERROR(EINVAL);
22      }
23      // 如果用户指定了输入格式（fmt 不为 NULL），则将其设置到上下文中
24      if (fmt)
25          s->iformat = fmt;
26
27      if (options)
28          av_dict_copy(&tmp, *options, 0);
29
30      if ((ret = av_opt_set_dict(s, &tmp)) < 0)
31          goto fail;
32      // 调用 init_input() 函数，该函数会根据文件名或 URL 打开输入流，并尝试探测文件格式
33      if ((ret = init_input(s, filename, &tmp)) < 0)
34          goto fail;
35      s->probe_score = ret;
36
37      if (s->format_whitelist && av_match_list(s->iformat->name, s-
    >format_whitelist, ',') <= 0) {
38          av_log(s, AV_LOG_ERROR, "Format not on whitelist\n");
39          ret = AVERROR(EINVAL);
40          goto fail;
41      }
42
43      avio_skip(s->pb, s->skip_initial_bytes);
44
45      /* Check filename in case an image number is expected. */
46      if (s->iformat->flags & AVFMT_NEEDNUMBER) {
47          if (!av_filename_number_test(filename)) {
48              ret = AVERROR(EINVAL);
49              goto fail;
50          }
51      }
52
53      s->duration = s->start_time = AV_NOPTS_VALUE;
54      av_strlcpy(s->filename, filename ? filename : "", sizeof(s->filename));
55
56      /* Allocate private data. */
57      if (s->iformat->priv_data_size > 0) {
58          if (!(s->priv_data = av_mallocz(s->iformat->priv_data_size))) {
59              ret = AVERROR(ENOMEM);
60              goto fail;
61          }
62          if (s->iformat->priv_class) {
63              *(const AVClass **) s->priv_data = s->iformat->priv_class;
64              av_opt_set_defaults(s->priv_data);
```

```
65              if ((ret = av_opt_set_dict(s->priv_data, &tmp)) < 0)
66                  goto fail;
67          }
68      }
69
70      /* e.g. AVFMT_NOFILE formats will not have a AVIOContext */
71      if (s->pb)
72          ff_id3v2_read(s, ID3v2_DEFAULT_MAGIC, &id3v2_extra_meta, 0);
73
74      if (!(s->flags&AVFMT_FLAG_PRIV_OPT) && s->iformat->read_header)
75          if ((ret = s->iformat->read_header(s)) < 0)
76              goto fail;
77
78      if (id3v2_extra_meta) {
79          if (!strcmp(s->iformat->name, "mp3") || !strcmp(s->iformat->name,
    "aac") ||
80              !strcmp(s->iformat->name, "tta")) {
81              if ((ret = ff_id3v2_parse_apic(s, &id3v2_extra_meta)) < 0)
82                  goto fail;
83          } else
84              av_log(s, AV_LOG_DEBUG, "demuxer does not support additional id3
    data, skipping\n");
85      }
86      ff_id3v2_free_extra_meta(&id3v2_extra_meta);
87
88      if ((ret = avformat_queue_attached_pictures(s)) < 0)
89          goto fail;
90
91      if (!(s->flags&AVFMT_FLAG_PRIV_OPT) && s->pb && !s->data_offset)
92          s->data_offset = avio_tell(s->pb);
93
94      s->raw_packet_buffer_remaining_size = RAW_PACKET_BUFFER_SIZE;
95
96      if (options) {
97          av_dict_free(options);
98          *options = tmp;
99      }
100     *ps = s;
101     return 0;
102
103 fail:
104     ff_id3v2_free_extra_meta(&id3v2_extra_meta);
105     av_dict_free(&tmp);
106     if (s->pb && !(s->flags & AVFMT_FLAG_CUSTOM_IO))
107         avio_close(s->pb);
108     avformat_free_context(s);
109     *ps = NULL;
```

```
    return ret;
}
/*
核心流程：avformat_open_input 函数的核心流程是：初始化上下文 → 打开输入 → 探测格式 → 读取文件头 → 解析元数据 → 返回结果。
错误处理：函数使用 goto fail 模式进行统一的错误处理和资源清理，确保发生错误时不会泄漏内存。
扩展性：通过 AVOption 机制支持灵活的参数配置，通过格式注册机制支持多种媒体格式，体现了 FFmpeg 框架的高度模块化和可扩展性
*/
```

init_input

init_input()作为一个内部函数，竟然包含了一行注释（一般内部函数都没有注释），足可以看出它的重要性。它的主要工作就是打开输入的视频数据并且探测视频的格式。该函数的定义位于libavformat\utils.c，如下所示

代码块

```
/* Open input file and probe the format if necessary. */
static int init_input(AVFormatContext *s, const char *filename,
                      AVDictionary **options)
{
    int ret;
    AVProbeData pd = { filename, NULL, 0 };
    int score = AVPROBE_SCORE_RETRY;

    if (s->pb) {
        s->flags |= AVFMT_FLAG_CUSTOM_IO;
        if (!s->iformat)
            return av_probe_input_buffer2(s->pb, &s->iformat, filename,
                                          s, 0, s->format_probesize);
        else if (s->iformat->flags & AVFMT_NOFILE)
            av_log(s, AV_LOG_WARNING, "Custom AVIOContext makes no sense and "
                                      "will be ignored with AVFMT_NOFILE
format.\n");
        return 0;
    }

    if ((s->iformat && s->iformat->flags & AVFMT_NOFILE) ||
        (!s->iformat && (s->iformat = av_probe_input_format2(&pd, 0, &score))))
        return score;

    if ((ret = avio_open2(&s->pb, filename, AVIO_FLAG_READ | s->avio_flags,
                          &s->interrupt_callback, options)) < 0)
        return ret;
```

```
    if (s->iformat)
        return 0;
    return av_probe_input_buffer2(s->pb, &s->iformat, filename,
                                  s, 0, s->format_probesize);
}
/*
这个函数在短短的几行代码中包含了好几个return，因此逻辑还是有点复杂的，我们可以梳理一下：
 在函数的开头的score变量是一个判决AVInputFormat的分数的门限值，如果最后得到的
AVInputFormat的分数低于该门限值，就认为没有找到合适的AVInputFormat。FFmpeg内部判断封装
格式的原理实际上是对每种AVInputFormat给出一个分数，满分是100分，越有可能正确的
AVInputFormat给出的分数就越高。最后选择分数最高的AVInputFormat作为推测结果。score的值
是一个宏定义AVPROBE_SCORE_RETRY，我们可以看一下它的定义：
*/
#define AVPROBE_SCORE_RETRY (AVPROBE_SCORE_MAX/4)
#define AVPROBE_SCORE_MAX       100 ///< maximum score
/*
由此我们可以得出score取值是25，即如果推测后得到的最佳AVInputFormat的分值低于25，就认为
没有找到合适的AVInputFormat
*/
/*
整个函数的逻辑大体如下：
（1）当使用了自定义的AVIOContext的时候（AVFormatContext中的AVIOContext不为空，即s-
>pb!=NULL），如果指定了AVInputFormat就直接返回，如果没有指定就调用
av_probe_input_buffer2()推测AVInputFormat。这一情况出现的不算很多，但是当我们从内存中
读取数据的时候（需要初始化自定义的AVIOContext），就会执行这一步骤。
（2）在更一般的情况下，如果已经指定了AVInputFormat，就直接返回；如果没有指定
AVInputFormat，就调用av_probe_input_format(NULL,…)根据文件路径判断文件格式。这里特意
把av_probe_input_format()的第1个参数写成“NULL”，是为了强调这个时候实际上并没有给函数提
供输入数据，此时仅仅通过文件路径推测AVInputFormat。
（3）如果发现通过文件路径判断不出来文件格式，那么就需要打开文件探测文件格式了，这个时候会
首先调用avio_open2()打开文件，然后调用av_probe_input_buffer2()推测AVInputFormat
*/
```

avformat_find_stream_info

该函数可以读取一部分视音频数据并且获得一些相关的信息。avformat_find_stream_info()的声明位于libavformat\avformat.h，如下所示

代码块

```

int avformat_find_stream_info(AVFormatContext *ic, AVDictionary **options);
/*
ic：输入的AVFormatContext
options：额外的选项，目前没有深入研究过
*/
```

```
// FFmpeg 中用于探测和解析媒体流详细信息的核心函数
/*
在打开媒体文件后，通过读取和解析一定数量的数据包，获取每个流的详细编码信息（如分辨率、帧率、采样率等），并计算流的时长、帧率等关键参数，为后续的解码操作提供必要的上下文信息
*/
int avformat_find_stream_info(AVFormatContext *ic, AVDictionary **options)
{
    // 声明循环变量、计数器、返回值等基本变量
    int i, count, ret = 0, j;
    int64_t read_size;
    AVStream *st;
    AVPacket pkt1, *pkt;
    // 记录当前IO指针位置，用于恢复或计算偏移
    int64_t old_offset  = avio_tell(ic->pb);
    // new streams might appear, no options for those
    // 记录初始的数据量（后续可能会发现新流）
    int orig_nb_streams = ic->nb_streams;
    int flush_codecs;
    // 设置最大分析时长，限制函数执行时间
    int64_t max_analyze_duration = ic->max_analyze_duration2;
    // 设置最大探测数据量，限制读取的数据大小
    int64_t probesize = ic->probesize2;


    if (!max_analyze_duration)
        max_analyze_duration = ic->max_analyze_duration;
    if (ic->probesize)
        probesize = ic->probesize;
    flush_codecs = probesize > 0;


    av_opt_set(ic, "skip_clear", "1", AV_OPT_SEARCH_CHILDREN);

    // 设置默认的分析时长和探测大小，FLV 格式默认分析 10 秒，其他格式默认 5 秒
    if (!max_analyze_duration) {
        if (!strcmp(ic->iformat->name, "flv") && !(ic->ctx_flags &
AVFMTCTX_NOHEADER)) {
            max_analyze_duration = 10*AV_TIME_BASE;
        } else
            max_analyze_duration = 5*AV_TIME_BASE;
    }


    if (ic->pb)
        av_log(ic, AV_LOG_DEBUG, "Before avformat_find_stream_info() pos:
%"PRId64" bytes read:%"PRId64" seeks:%d\n",
               avio_tell(ic->pb), ic->pb->bytes_read, ic->pb->seek_count);
```

```
51
52      // 初始化每个流的解析器和解码器
53      for (i = 0; i < ic->nb_streams; i++) {
54          // 用于存储找到的解码器
55          const AVCodec *codec;
56          // 用于存储线程相关的选项
57          AVDictionary *thread_opt = NULL;
58          // 获取当前代表的媒体流（音频 视频 或字幕）
59          st = ic->streams[i];
60
61          // 判断当前流是否为视频流或字幕流
62          if (st->codec->codec_type == AVMEDIA_TYPE_VIDEO ||
63              st->codec->codec_type == AVMEDIA_TYPE_SUBTITLE) {
64  /*              if (!st->time_base.num)
65                    st->time_base = */
66              // 时间戳
67              if (!st->codec->time_base.num)
68                  st->codec->time_base = st->time_base;
69          }
70          // only for the split stuff
71          // 当前流没有设置解析器&不禁用解析器
72          if (!st->parser && !(ic->flags & AVFMT_FLAG_NOPARSE)) {
73              // 根据解码器 ID 初始化一个解析器，解析器用于将原始数据包分割成完整的帧，并
    提取关键信息
74              st->parser = av_parser_init(st->codec->codec_id);
75              if (st->parser) {
76                  // 解析头部还是完全解析原始数据
77                  if (st->need_parsing == AVSTREAM_PARSE_HEADERS) {
78                      st->parser->flags |= PARSER_FLAG_COMPLETE_FRAMES;
79                  } else if (st->need_parsing == AVSTREAM_PARSE_FULL_RAW) {
80                      st->parser->flags |= PARSER_FLAG_USE_CODEC_TS;
81                  }
82              } else if (st->need_parsing) {
83                  av_log(ic, AV_LOG_VERBOSE, "parser not found for codec "
84                         "%s, packets or times may be invalid.\n",
85                         avcodec_get_name(st->codec->codec_id));
86              }
87          }
88          // 根据编码器ID查找对应的解码器
89          codec = find_decoder(ic, st, st->codec->codec_id);
90
91
92          /* Force thread count to 1 since the H.264 decoder will not extract
93           * SPS and PPS to extradata during multi-threaded decoding. */
94          // 强制设置线程数为 1，因为 H.264 解码器在多线程模式下无法正确提取 SPS/PPS 到
    额外数据中
95          av_dict_set(options ? &options[i] : &thread_opt, "threads", "1", 0);
```

```
96
97
98          if (ic->codec_whitelist)
99              av_dict_set(options ? &options[i] : &thread_opt,
    "codec_whitelist", ic->codec_whitelist, 0);
100
101
102         /* Ensure that subtitle_header is properly set. */
103         if (st->codec->codec_type == AVMEDIA_TYPE_SUBTITLE
104             && codec && !st->codec->codec) {
105             if (avcodec_open2(st->codec, codec, options ? &options[i] :
    &thread_opt) < 0)
106                 av_log(ic, AV_LOG_WARNING,
107                        "Failed to open codec in av_find_stream_info\n");
108         }
109
110
111         // Try to just open decoders, in case this is enough to get parameters.
112         // 获取解码器参数&打开解码器
113         if (!has_codec_parameters(st, NULL) && st->request_probe <= 0) {
114             if (codec && !st->codec->codec)
115                 if (avcodec_open2(st->codec, codec, options ? &options[i] :
    &thread_opt) < 0)
116                     av_log(ic, AV_LOG_WARNING,
117                            "Failed to open codec in av_find_stream_info\n");
118         }
119         if (!options)
120             av_dict_free(&thread_opt);
121     }
122
123     // 初始化用于计算帧率的时间戳信息
124     for (i = 0; i < ic->nb_streams; i++) {
125 #if FF_API_R_FRAME_RATE
126         ic->streams[i]->info->last_dts = AV_NOPTS_VALUE;
127 #endif
128         ic->streams[i]->info->fps_first_dts = AV_NOPTS_VALUE;
129         ic->streams[i]->info->fps_last_dts  = AV_NOPTS_VALUE;
130     }
131
132
133     count     = 0;
134     read_size = 0;
135     for (;;) {
136         // 检查是否有中段请求
137         if (ff_check_interrupt(&ic->interrupt_callback)) {
138             ret = AVERROR_EXIT;
139             av_log(ic, AV_LOG_DEBUG, "interrupted\n");
```

```
140             break;
141         }
142 
143 
144         /* check if one codec still needs to be handled */
145         // 遍历所有流，检查是否所有流都已获取到完整的参数信息
146         for (i = 0; i < ic->nb_streams; i++) {
147             // 设置默认的帧率分析帧数为 20 帧
148             int fps_analyze_framecount = 20;
149 
150 
151             st = ic->streams[i];
152             // 该流的编码器参数是否完整
153             if (!has_codec_parameters(st, NULL))
154                 break;
155             /* If the timebase is coarse (like the usual millisecond precision
156              * of mkv), we need to analyze more frames to reliably arrive at
157              * the correct fps. */
158             // 如果时间基精度较低（如 MKV 的毫秒精度），则将帧率分析帧数加倍（40 帧），以更准确地计算帧率
159             if (av_q2d(st->time_base) > 0.0005)
160                 fps_analyze_framecount *= 2;
161             // 如果解码器的时间基是可靠的，则不需要分析帧率，将帧数设为 0
162             if (!tb_unreliable(st->codec))
163                 fps_analyze_framecount = 0;
164             // 如果用户设置了 fps_probe_size，则使用该值覆盖默认的帧率分析帧数
165             if (ic->fps_probe_size >= 0)
166                 fps_analyze_framecount = ic->fps_probe_size;
167             // 如果该流是附加图片（如封面），则不需要分析帧率
168             if (st->disposition & AV_DISPOSITION_ATTACHED_PIC)
169                 fps_analyze_framecount = 0;
170             /* variable fps and no guess at the real fps */
171             // 如果视频流的帧率尚未确定，且分析的帧数不足，则跳出内层循环，继续读取数据包
172             if (!(st->r_frame_rate.num && st->avg_frame_rate.num) &&
173                 st->info->duration_count < fps_analyze_framecount &&
174                 st->codec->codec_type == AVMEDIA_TYPE_VIDEO)
175                 break;
176             // 如果流有解析器且需要分割数据，但解码器的额外数据（extradata）尚未设置，则跳出内层循环
177             if (st->parser && st->parser->parser->split &&
178                 !st->codec->extradata)
179                 break;
180             // 如果音视频流的第一个 DTS（解码时间戳）尚未获取，且分析的帧数未达到最大时间戳探测数，则跳出内层循环
181             if (st->first_dts == AV_NOPTS_VALUE &&
182                 !(ic->iformat->flags & AVFMT_NOTIMESTAMPS) &&
```

```
183                 st->codec_info_nb_frames < ic->max_ts_probe &&
184                 (st->codec->codec_type == AVMEDIA_TYPE_VIDEO ||
185                  st->codec->codec_type == AVMEDIA_TYPE_AUDIO))
186                 break;
187         }
188         if (i == ic->nb_streams) {
189             /* NOTE: If the format has no header, then we need to read some
190              * packets to get most of the streams, so we cannot stop here. */
191             if (!(ic->ctx_flags & AVFMTCTX_NOHEADER)) {
192                 /* If we found the info for all the codecs, we can stop. */
193                 ret = count;
194                 av_log(ic, AV_LOG_DEBUG, "All info found\n");
195                 flush_codecs = 0;
196                 break;
197             }
198         }
199         /* We did not get all the codec info, but we read too much data. */
200         if (read_size >= probesize) {
201             ret = count;
202             av_log(ic, AV_LOG_DEBUG,
203                    "Probe buffer size limit of %"PRId64" bytes reached\n",
    probesize);
204             for (i = 0; i < ic->nb_streams; i++)
205                 if (!ic->streams[i]->r_frame_rate.num &&
206                     ic->streams[i]->info->duration_count <= 1 &&
207                     ic->streams[i]->codec->codec_type == AVMEDIA_TYPE_VIDEO &&
208                     strcmp(ic->iformat->name, "image2"))
209                     av_log(ic, AV_LOG_WARNING,
210                            "Stream #%d: not enough frames to estimate rate; "
211                            "consider increasing probesize\n", i);
212             break;
213         }
214
215
216         /* NOTE: A new stream can be added there if no header in file
217          * (AVFMTCTX_NOHEADER). */
218         ret = read_frame_internal(ic, &pkt1);
219         if (ret == AVERROR(EAGAIN))
220             continue;
221
222
223         if (ret < 0) {
224             /* EOF or error*/
225             break;
226         }
227
228
```

```
229         if (ic->flags & AVFMT_FLAG_NOBUFFER)
230             free_packet_buffer(&ic->packet_buffer, &ic->packet_buffer_end);
231         {
232             pkt = add_to_pktbuf(&ic->packet_buffer, &pkt1,
233                                 &ic->packet_buffer_end);
234             if (!pkt) {
235                 ret = AVERROR(ENOMEM);
236                 goto find_stream_info_err;
237             }
238             if ((ret = av_dup_packet(pkt)) < 0)
239                 goto find_stream_info_err;
240         }
241
242
243         st = ic->streams[pkt->stream_index];
244         if (!(st->disposition & AV_DISPOSITION_ATTACHED_PIC))
245             read_size += pkt->size;
246
247
248         if (pkt->dts != AV_NOPTS_VALUE && st->codec_info_nb_frames > 1) {
249             /* check for non-increasing dts */
250             if (st->info->fps_last_dts != AV_NOPTS_VALUE &&
251                 st->info->fps_last_dts >= pkt->dts) {
252                 av_log(ic, AV_LOG_DEBUG,
253                        "Non-increasing DTS in stream %d: packet %d with DTS "
254                        "%"PRId64", packet %d with DTS %"PRId64"\n",
255                        st->index, st->info->fps_last_dts_idx,
256                        st->info->fps_last_dts, st->codec_info_nb_frames,
257                        pkt->dts);
258                 st->info->fps_first_dts =
259                 st->info->fps_last_dts  = AV_NOPTS_VALUE;
260             }
261             /* Check for a discontinuity in dts. If the difference in dts
262              * is more than 1000 times the average packet duration in the
263              * sequence, we treat it as a discontinuity. */
264             if (st->info->fps_last_dts != AV_NOPTS_VALUE &&
265                 st->info->fps_last_dts_idx > st->info->fps_first_dts_idx &&
266                 (pkt->dts - st->info->fps_last_dts) / 1000 >
267                 (st->info->fps_last_dts     - st->info->fps_first_dts) /
268                 (st->info->fps_last_dts_idx - st->info->fps_first_dts_idx)) {
269                 av_log(ic, AV_LOG_WARNING,
270                        "DTS discontinuity in stream %d: packet %d with DTS "
271                        "%"PRId64", packet %d with DTS %"PRId64"\n",
272                        st->index, st->info->fps_last_dts_idx,
273                        st->info->fps_last_dts, st->codec_info_nb_frames,
274                        pkt->dts);
275                 st->info->fps_first_dts =
```

```
276                 st->info->fps_last_dts  = AV_NOPTS_VALUE;
277             }
278 
279 
280             /* update stored dts values */
281             if (st->info->fps_first_dts == AV_NOPTS_VALUE) {
282                 st->info->fps_first_dts     = pkt->dts;
283                 st->info->fps_first_dts_idx = st->codec_info_nb_frames;
284             }
285             st->info->fps_last_dts     = pkt->dts;
286             st->info->fps_last_dts_idx = st->codec_info_nb_frames;
287         }
288         if (st->codec_info_nb_frames>1) {
289             int64_t t = 0;
290 
291 
292             if (st->time_base.den > 0)
293                 t = av_rescale_q(st->info->codec_info_duration, st->time_base,
    AV_TIME_BASE_Q);
294             if (st->avg_frame_rate.num > 0)
295                 t = FFMAX(t, av_rescale_q(st->codec_info_nb_frames,
    av_inv_q(st->avg_frame_rate), AV_TIME_BASE_Q));
296 
297 
298             if (   t == 0
299                 && st->codec_info_nb_frames>30
300                 && st->info->fps_first_dts != AV_NOPTS_VALUE
301                 && st->info->fps_last_dts  != AV_NOPTS_VALUE)
302                 t = FFMAX(t, av_rescale_q(st->info->fps_last_dts - st->info-
    >fps_first_dts, st->time_base, AV_TIME_BASE_Q));
303 
304 
305             if (t >= max_analyze_duration) {
306                 av_log(ic, AV_LOG_VERBOSE, "max_analyze_duration %"PRId64"
    reached at %"PRId64" microseconds\n",
307                        max_analyze_duration,
308                        t);
309                 if (ic->flags & AVFMT_FLAG_NOBUFFER)
310                     av_packet_unref(pkt);
311                 break;
312             }
313             if (pkt->duration) {
314                 st->info->codec_info_duration        += pkt->duration;
315                 st->info->codec_info_duration_fields += st->parser && st-
    >need_parsing && st->codec->ticks_per_frame ==2 ? st->parser->repeat_pict + 1
    : 2;
316             }
```

```
317         }
318 #if FF_API_R_FRAME_RATE
319         if (st->codec->codec_type == AVMEDIA_TYPE_VIDEO)
320             ff_rfps_add_frame(ic, st, pkt->dts);
321 #endif
322         if (st->parser && st->parser->parser->split && !st->codec->extradata) {
323             int i = st->parser->parser->split(st->codec, pkt->data, pkt->size);
324             if (i > 0 && i < FF_MAX_EXTRADATA_SIZE) {
325                 if (ff_alloc_extradata(st->codec, i))
326                     return AVERROR(ENOMEM);
327                 memcpy(st->codec->extradata, pkt->data,
328                        st->codec->extradata_size);
329             }
330         }
331
332
333         /* If still no information, we try to open the codec and to
334          * decompress the frame. We try to avoid that in most cases as
335          * it takes longer and uses more memory. For MPEG-4, we need to
336          * decompress for QuickTime.
337          *
338          * If CODEC_CAP_CHANNEL_CONF is set this will force decoding of at
339          * least one frame of codec data, this makes sure the codec initializes
340          * the channel configuration and does not only trust the values from
341          * the container. */
342         try_decode_frame(ic, st, pkt,
343                          (options && i < orig_nb_streams) ? &options[i] :
    NULL);
344
345
346         if (ic->flags & AVFMT_FLAG_NOBUFFER)
347             av_packet_unref(pkt);
348
349
350         st->codec_info_nb_frames++;
351         count++;
352     }
353
354
355     if (flush_codecs) {
356         AVPacket empty_pkt = { 0 };
357         int err = 0;
358         av_init_packet(&empty_pkt);
359
360
361         for (i = 0; i < ic->nb_streams; i++) {
362
```

```
363
364             st = ic->streams[i];
365
366
367             /* flush the decoders */
368             if (st->info->found_decoder == 1) {
369                 do {
370                     err = try_decode_frame(ic, st, &empty_pkt,
371                                            (options && i < orig_nb_streams)
372                                            ? &options[i] : NULL);
373                 } while (err > 0 && !has_codec_parameters(st, NULL));
374
375
376                 if (err < 0) {
377                     av_log(ic, AV_LOG_INFO,
378                         "decoding for stream %d failed\n", st->index);
379                 }
380             }
381         }
382     }
383
384
385     // close codecs which were opened in try_decode_frame()
386     for (i = 0; i < ic->nb_streams; i++) {
387         st = ic->streams[i];
388         avcodec_close(st->codec);
389     }
390
391
392     ff_rfps_calculate(ic);
393
394
395     for (i = 0; i < ic->nb_streams; i++) {
396         st = ic->streams[i];
397         if (st->codec->codec_type == AVMEDIA_TYPE_VIDEO) {
398             if (st->codec->codec_id == AV_CODEC_ID_RAWVIDEO && !st->codec-
    >codec_tag && !st->codec->bits_per_coded_sample) {
399                 uint32_t tag= avcodec_pix_fmt_to_codec_tag(st->codec->pix_fmt);
400                 if (avpriv_find_pix_fmt(avpriv_get_raw_pix_fmt_tags(), tag) ==
    st->codec->pix_fmt)
401                     st->codec->codec_tag= tag;
402             }
403
404
405             /* estimate average framerate if not set by demuxer */
406             if (st->info->codec_info_duration_fields &&
407                 !st->avg_frame_rate.num &&
```

```
408                 st->info->codec_info_duration) {
409                 int best_fps      = 0;
410                 double best_error = 0.01;
411 
412 
413                 if (st->info->codec_info_duration        >= INT64_MAX / st->time_base.num / 2||
414                     st->info->codec_info_duration_fields >= INT64_MAX / st->time_base.den ||
415                     st->info->codec_info_duration        < 0)
416                     continue;
417                 av_reduce(&st->avg_frame_rate.num, &st->avg_frame_rate.den,
418                           st->info->codec_info_duration_fields * (int64_t) st->time_base.den,
419                           st->info->codec_info_duration * 2 * (int64_t) st->time_base.num, 60000);
420 
421 
422                 /* Round guessed framerate to a "standard" framerate if it's
423                  * within 1% of the original estimate. */
424                 for (j = 0; j < MAX_STD_TIMEBASES; j++) {
425                     AVRational std_fps = { get_std_framerate(j), 12 * 1001 };
426                     double error       = fabs(av_q2d(st->avg_frame_rate) /
427                                               av_q2d(std_fps) - 1);
428 
429 
430                     if (error < best_error) {
431                         best_error = error;
432                         best_fps   = std_fps.num;
433                     }
434                 }
435                 if (best_fps)
436                     av_reduce(&st->avg_frame_rate.num, &st->avg_frame_rate.den,
437                               best_fps, 12 * 1001, INT_MAX);
438             }
439 
440 
441             if (!st->r_frame_rate.num) {
442                 if (    st->codec->time_base.den * (int64_t) st->time_base.num
443                     <= st->codec->time_base.num * st->codec->ticks_per_frame * (int64_t) st->time_base.den) {
444                     st->r_frame_rate.num = st->codec->time_base.den;
445                     st->r_frame_rate.den = st->codec->time_base.num * st->codec->ticks_per_frame;
446                 } else {
447                     st->r_frame_rate.num = st->time_base.den;
448                     st->r_frame_rate.den = st->time_base.num;
```

```
449                 }
450             }
451         } else if (st->codec->codec_type == AVMEDIA_TYPE_AUDIO) {
452             if (!st->codec->bits_per_coded_sample)
453                 st->codec->bits_per_coded_sample =
454                     av_get_bits_per_sample(st->codec->codec_id);
455             // set stream disposition based on audio service type
456             switch (st->codec->audio_service_type) {
457             case AV_AUDIO_SERVICE_TYPE_EFFECTS:
458                 st->disposition = AV_DISPOSITION_CLEAN_EFFECTS;
459                 break;
460             case AV_AUDIO_SERVICE_TYPE_VISUALLY_IMPAIRED:
461                 st->disposition = AV_DISPOSITION_VISUAL_IMPAIRED;
462                 break;
463             case AV_AUDIO_SERVICE_TYPE_HEARING_IMPAIRED:
464                 st->disposition = AV_DISPOSITION_HEARING_IMPAIRED;
465                 break;
466             case AV_AUDIO_SERVICE_TYPE_COMMENTARY:
467                 st->disposition = AV_DISPOSITION_COMMENT;
468                 break;
469             case AV_AUDIO_SERVICE_TYPE_KARAOKE:
470                 st->disposition = AV_DISPOSITION_KARAOKE;
471                 break;
472             }
473         }
474     }
475
476
477     if (probesize)
478     estimate_timings(ic, old_offset);
479
480
481     av_opt_set(ic, "skip_clear", "0", AV_OPT_SEARCH_CHILDREN);
482
483
484     if (ret >= 0 && ic->nb_streams)
485         /* We could not have all the codec parameters before EOF. */
486         ret = -1;
487     for (i = 0; i < ic->nb_streams; i++) {
488         const char *errmsg;
489         st = ic->streams[i];
490         if (!has_codec_parameters(st, &errmsg)) {
491             char buf[256];
492             avcodec_string(buf, sizeof(buf), st->codec, 0);
493             av_log(ic, AV_LOG_WARNING,
494                    "Could not find codec parameters for stream %d (%s): %s\n"
```

```
495                     "Consider increasing the value for the 'analyzeduration'
    and 'probesize' options\n",
496                     i, buf, errmsg);
497         } else {
498             ret = 0;
499         }
500     }
501 
502 
503     compute_chapters_end(ic);
504 
505 
506 find_stream_info_err:
507     for (i = 0; i < ic->nb_streams; i++) {
508         st = ic->streams[i];
509         if (ic->streams[i]->codec->codec_type != AVMEDIA_TYPE_AUDIO)
510             ic->streams[i]->codec->thread_count = 0;
511         if (st->info)
512             av_freep(&st->info->duration_error);
513         av_freep(&ic->streams[i]->info);
514     }
515     if (ic->pb)
516         av_log(ic, AV_LOG_DEBUG, "After avformat_find_stream_info() pos:
    %"PRId64" bytes read:%"PRId64" seeks:%d frames:%d\n",
517                avio_tell(ic->pb), ic->pb->bytes_read, ic->pb->seek_count,
    count);
518     return ret;
519 }
520 /*
521 由于avformat_find_stream_info()代码比较长，难以全部分析，在这里只能简单记录一下它的要
    点。该函数主要用于给每个媒体流（音频/视频）的AVStream结构体赋值。我们大致浏览一下这个函数
    的代码，会发现它其实已经实现了解码器的查找，解码器的打开，视音频帧的读取，视音频帧的解码等
    工作。换句话说，该函数实际上已经“走通”的解码的整个流程。
522 */
```

avformat_find_stream_info的核心作用：

1.补全信息：把avformat_open_input没拿到的详细编码参数补全

2.准备解码：为后续调用avcodec_open2和av_read_frame铺平道路

简单说，没有这个函数，你只知道"有视频有音频"，但不知道具体是什么规格的视频音频，有了它，你就掌握了所有解码播放需要的关键信息

avcodec_find_decoder 和 avcodec_find_encoder

avcodec_find_encoder()用于查找FFmpeg的编码器，avcodec_find_decoder()用于查找FFmpeg的解码器

代码块

```
/**
 * Find a registered encoder with a matching codec ID.
 *
 * @param id AVCodecID of the requested encoder
 * @return An encoder if one was found, NULL otherwise.
 */
AVCodec *avcodec_find_encoder(enum AVCodecID id);
/*
函数的参数是一个解码器的ID，返回查找到的解码器（没有找到就返回NULL）
*/
```

代码块

```
/**
 * Find a registered decoder with a matching codec ID.
 *
 * @param id AVCodecID of the requested decoder
 * @return A decoder if one was found, NULL otherwise.
 */
AVCodec *avcodec_find_decoder(enum AVCodecID id);
/*
函数的参数是一个解码器的ID，返回查找到的解码器（没有找到就返回NULL）
*/
```

函数调用关系图

avcodec_find_encoder()和avcodec_find_decoder()的函数调用关系图如下所示

avcodec_find_encoder()的源代码位于libavcodec\utils.c，如下所示

代码块

```
AVCodec *avcodec_find_encoder(enum AVCodecID id)
```

```
{
    return find_encdec(id, 1);
}

AVCodec *avcodec_find_decoder(enum AVCodecID id)
{
    return find_encdec(id, 0);
}
/*
从源代码可以看出avcodec_find_encoder()调用了一个find_encdec()，下面我们看一下find_encdec()的定义
可以看出avcodec_find_decoder()同样调用了find_encdec()，只是第2个参数设置为0，因此不再详细分析
*/
```

find_encdec

代码块

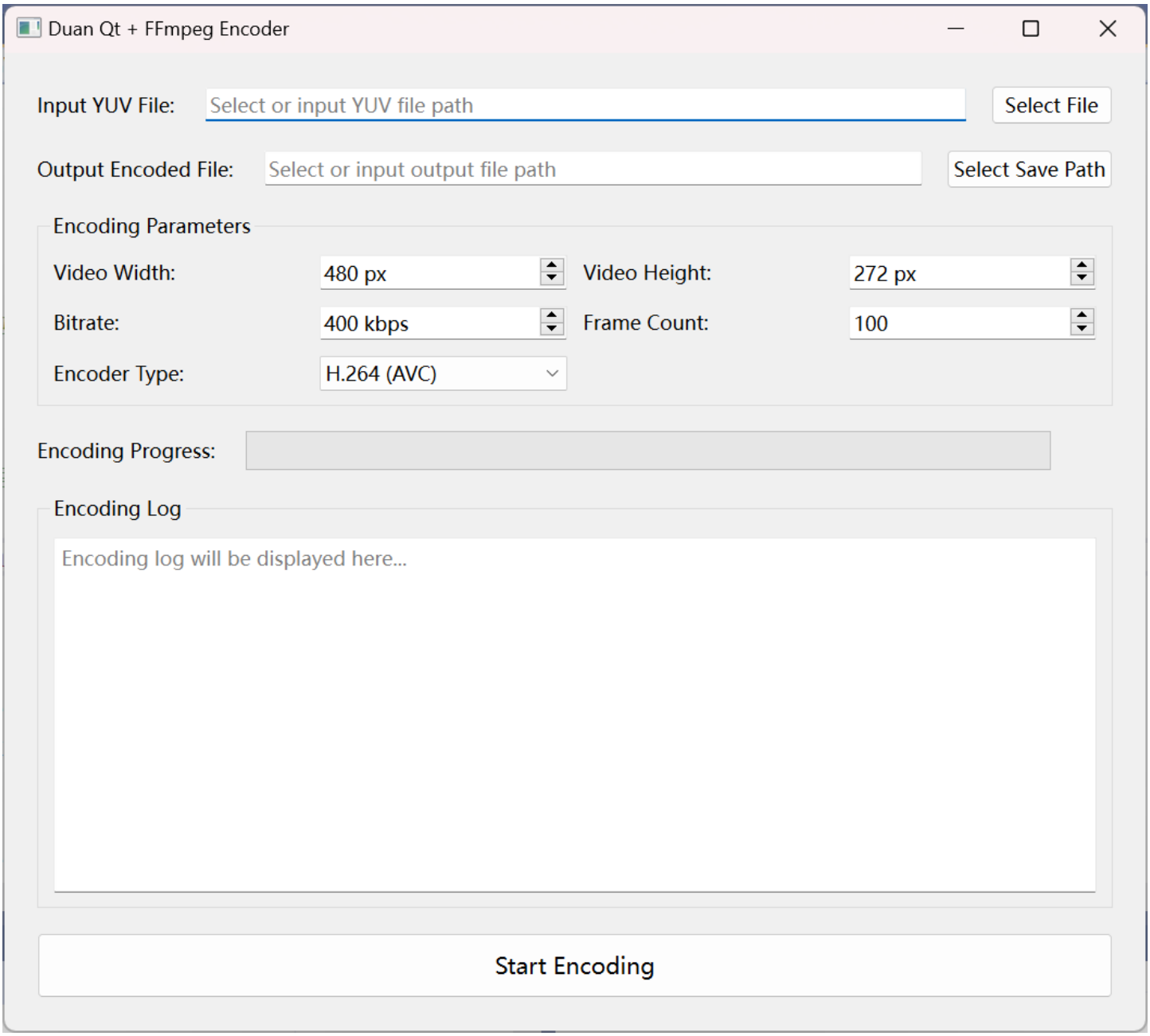
```
/*
声明一个静态的全局指针，指向编码器链表的第一个节点，FFmpeg在初始化时会将所有注册的编码器链接成一个单链表，这个指针就是链表的头
*/
static AVCodec *first_avcodec;

/*
参数id：要查找的编码器ID 如AV_CODEC_ID_H264
参数encoder：标志位，1表示查找编码器，0表示查找解码器
返回值：指向找到的AVCodec结构体的指针
*/
static AVCodec *find_encdec(enum AVCodecID id, int encoder)
{
    // p：用于遍历编解码器链表的临时指针
    // experimental：用于存储找到的实验性编解码器
    AVCodec *p, *experimental = NULL;
    // 将遍历指针p初始化为编解码器链表头节点
    p = first_avcodec;
    /*
    将传入的编解码器ID进行映射，处理已废弃的ID，确保使用最新的ID值，例如某些旧的编码ID可能被重命名或合并，这个函数会将其转换为当前有效的ID
    */
    id= remap_deprecated_codec_id(id);
    while (p) {
        // 当前编解码器是否符合要求
        if ((encoder ? av_codec_is_encoder(p) : av_codec_is_decoder(p)) &&
            p->id == id) {
```

```
            if (p->capabilities & CODEC_CAP_EXPERIMENTAL && !experimental) {
                experimental = p;
            } else
                return p;
        }
        p = p->next;
    }
    return experimental;
}
```

项目推荐

这里以自己做的一个小demo为例，基于QT + FFmpeg的编码器

https://github.com/reversible67/DuanEncoder

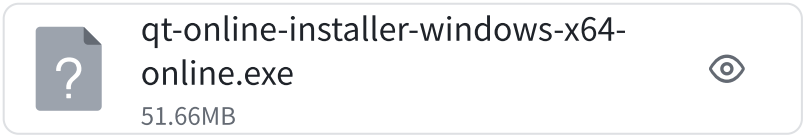


如何运行

1.下载FFmpeg预编译库（包含include lib dll）

上面已经提到过，这里不再赘述

2.离线安装QT



这里安装的是Qt6.5.3版本

在Visual Studio2019中安装插件 Qt Visual Studio Tools，然后手动配置一下include lib