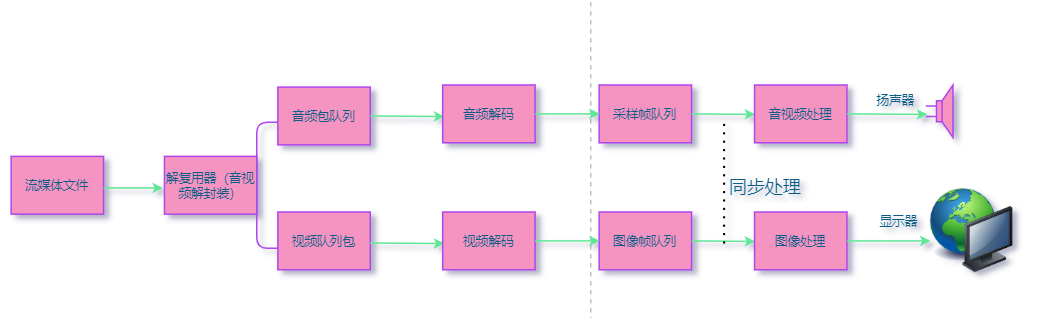
# ffmpeg实战实现音视频解封装！

## **一、前言**

今天我们要分享的主要音视频里面的解封装过程详细解析；在讲解 解封装之前，我们简单的来了解一下流媒体文件是如何被播放出来的，要实现播放，那这个过程到底要经历哪些技术处理呢？一般一个音视频流媒体文件播放实现流程图如下：

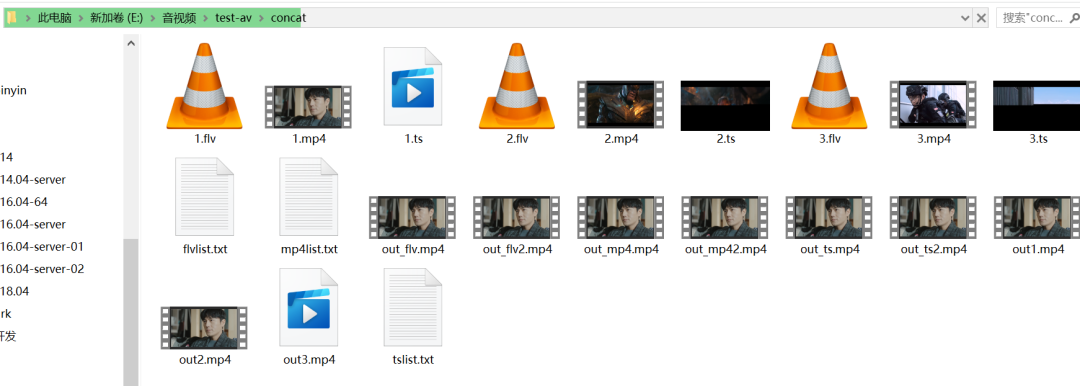
流媒体文件如何实现播放流程

从上面的流程图中，我们可以发现一个流媒体文件播放实现过程，看上去是不怎么复杂，但是其实里面有很多细小的技术点；今天暂时我们先来掌握解封装！

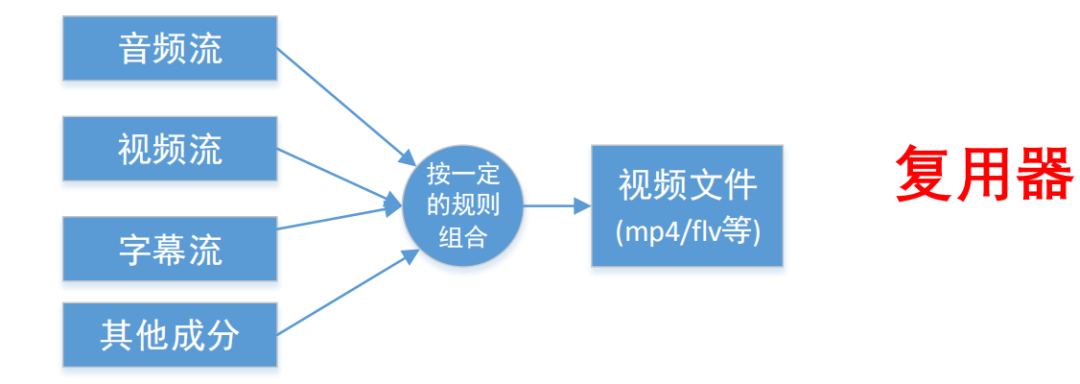
## **二、探索解封装的奥秘**

### **1、什么是解封装呢？**

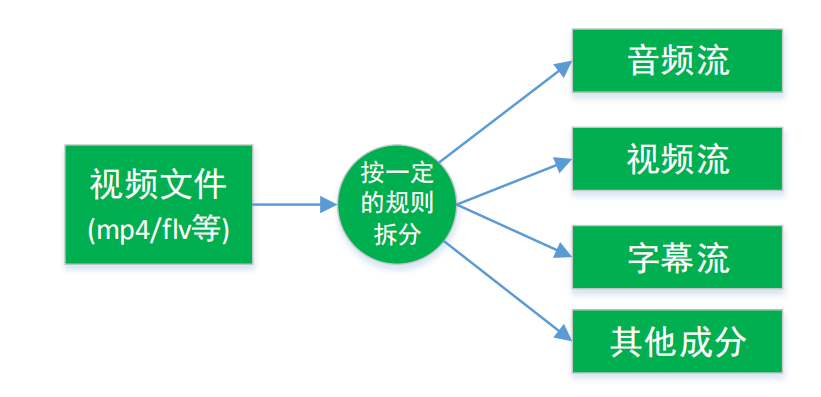
在了解什么是解封装之前，不知道大家平时在自己的电脑里面播放视频文件的时候，有没有注意视频文件的后缀格式呢，比如下面几种文件格式：

常用的几种封装格式

上面的mp4、flv、ts等 都是对 音视频数据 进行封装的一种 封装格式，通俗的讲，就把很多东西合成一个东西，只是合成的这个东西，表现形式不一样而已，用更加的专业术语来讲，这里的 合成 就是 复用器，我们可以用一张图来解释：

复用器

那么听了上面的解说，你自然而然的就会想到 解复用器 了，那么也就是解封装了，解封装的作用就跟上面的复用器起着相反的作用，就是把一个流媒体文件，拆解成音频数据和视频数据（专业的讲，一般被拆解成H.264编码的视频码流和AAC编码的音频码流），下面还是用一张图来解释：

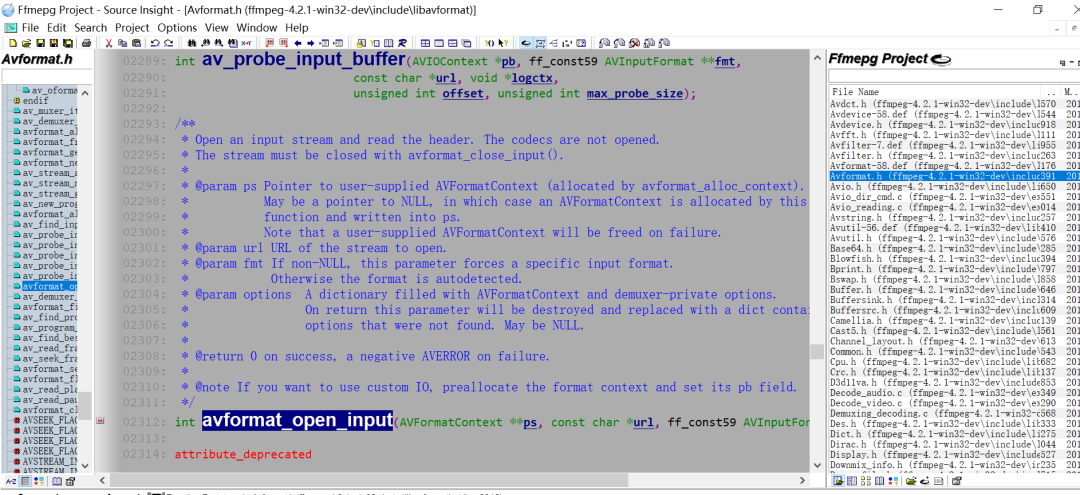
解封装（解复用器）

## **三、利用ffmpeg接口实战解封装实现**

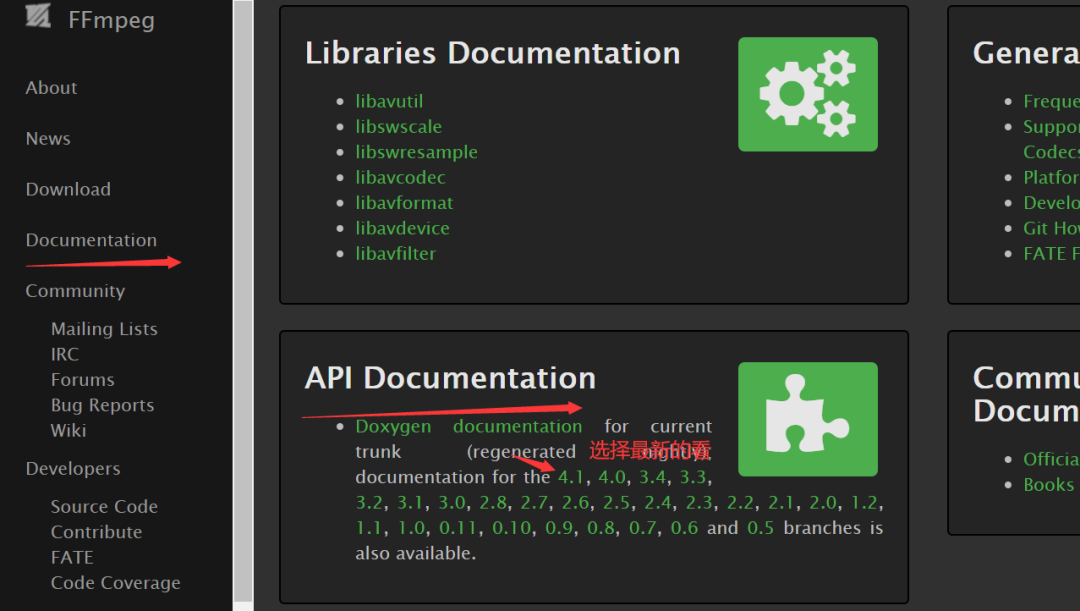
经过上面的讲解，想必大家对解封装的概念已经非常清楚了；那么接下来呢，我们就可以利用ffmpeg里面的libavformat库(它是一个包含用于多媒体容器格式的解复用器和复用器的库，里面有很多可供我们开发人员进行实战操作的api)调用相关api来实现解封装的具体操作。

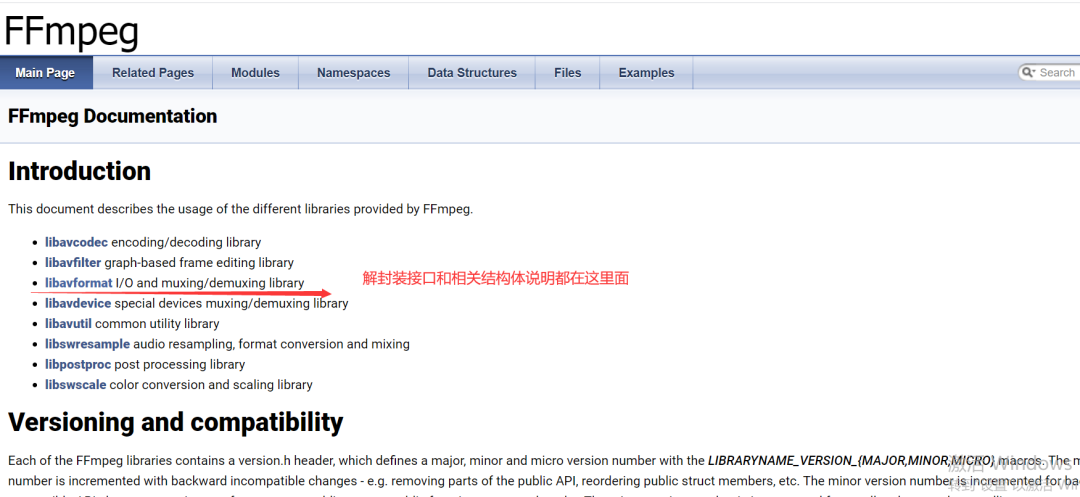
### **1、工欲善必先利其器：**

在开始写代码实现之前呢，我们还要了解一下 解封装的一个具体流程和相应的api。我们先把解封装实现相应的api接口得介绍一下，不然很多朋友直接看代码实现不知道什么意思，而且也不知道这些接口说明去哪里找（这个曾经在交流的时候，还真有人不知道api接口里面传的参数是什么意思，其实吧，ffmpeg官网手册api接口介绍里面有非常详细的介绍呢,或者ffmpeg源码里面也有api接口的详细说明使用！）；当然如果有时间，我觉得非常有必要去研究一下ffmpeg的源码阅读，千万不要停留在只会调用api的层次，更多的是 我们要了解背后深层次的东西：源码阅读，

 ffmpeg 4.2.1版本源码

好了，下面我们开始介绍解封装相关的接口和结构体说明；第一时间，大家可以去官网找到ffmpeg的api接口说明文档：

<https://www.ffmpeg.org/documentation.html>





解封装常用的api如下：

* avformat\_alloc\_context()：负责申请一个AVFormatContext结构体的内存，并进行初始化，它的函数原型如下：

AVFormatContext\* avformat\_alloc\_context (void)

* avformat\_free\_context()：释放AVFormatContext结构体里面的所有东西以及该结构体本身，函数原型如下：

void avformat\_free\_context(AVFormatContext \*s)

* avformat\_open\_input():从函数名称就知道是打开要输入的流媒体文件，函数原型如下：

int avformat\_open\_input ( AVFormatContext \*\*  ps,  
const char \*  url,  
AVInputFormat \*  fmt,  
AVDictionary \*\*  options   
)

参数说明：

ps:指向用户提供的AVFormatContext的指针（由avformat\_alloc\_context分配）。可能是指向NULL的指针，在这种情况下，此函数将分配AVFormatContext并将其写入ps。请注意，用户提供的AVFormatContext将在失败时释放。

url：要打开的流的url，也就是要打开的流媒体文件。

fmt:如果为非NULL，则此参数强制使用特定的输入格式。否则，将自动检测格式。

options:包含AVFormatContext和demuxer-private选项的字典。返回时，此参数将被销毁并替换为包含未找到的选项的dict。可能为NULL。

注意：返回值为0的时候表示成功，失败的时候返回AVERROR,跟linux里面的api接口机制类似。

avformat\_close\_input():关闭 打开的输入AVFormatContext，释放它及其所有内容，并将\*s设置为NULL；关闭后就不需要再调用avformat\_free\_context()进行释放了。它的函数原型如下：

void avformat\_close\_input (AVFormatContext \*\*s)

avformat\_find\_stream\_info():读取媒体文件的数据包以获取流信息，这对于没有标题的文件格式（例如MPEG）很有用。在MPEG-2重复帧模式的情况下，此功能还可以计算实际帧率。该功能不会更改逻辑文件的位置。被检查的分组可以被缓冲以用于以后的处理。函数原型如下：

int avformat\_find\_stream\_info ( AVFormatContext \*  ic, AVDictionary \*\*  options )

* 参数说明：

ic:媒体文件句柄

options:如果为非NULL，则是指向字典的ic.nb\_streams长指针数组，其中第i个成员包含与第i个流相对应的编解码器选项。返回时，每本词典将填充未找到的选项。

注意：此函数不能保证打开所有编解码器，因此选项在返回时为非空是完全正常的行为。

av\_read\_frame()：返回流的下一帧；此函数返回文件中存储的内容，并且不验证解码器是否存在有效的帧。它将文件中存储的内容拆分为多个帧，并为每个调用返回一个帧。它不会忽略有效帧之间的无效数据，从而为解码器提供可能的最大解码信息；如果pkt-> buf为NULL，则该数据包在下一个av\_read\_frame（）或avformat\_close\_input（）之前一直有效。否则，数据包将无限期有效。在这两种情况下，当不再需要该数据包时，都必须使用av\_packet\_unref释放它。对于视频，数据包恰好包含一帧。对于音频，如果每个帧具有已知的固定大小（例如PCM或ADPCM数据），则它包含整数个帧。如果音频帧具有可变大小（例如MPEG音频），则它包含一帧。始终将pkt-> pts，pkt-> dts和pkt-> duration设置为以AVStream.time\_base为单位的正确值（并猜测格式是否无法提供它们）。如果视频格式具有B帧，则pkt-> pts可以为AV\_NOPTS\_VALUE，因此，如果不对有效载荷进行解压缩，则最好依靠pkt-> dts。

函数原型如下：

int av\_read\_frame ( AVFormatContext \*  s,AVPacket \*  pkt )

注意：返回值为0时，表示成功，非0表示失败！

avformat\_seek\_file()：寻求时间戳记（或者说定位文件位置）；将进行搜索，以便可以成功呈现所有活动流的点将最接近ts，并且在min / max\_ts之内。活动流是所有具有AVStream.discard <AVDISCARD\_ALL的流。如果标志包含AVSEEK\_FLAG\_BYTE，则所有时间戳均以字节为单位，并且为文件位置（并非所有解复用器均支持）。如果标志包含AVSEEK\_FLAG\_FRAME，则所有时间戳都在具有stream\_index的流中的帧中（并非所有解复用器均支持）。否则，所有时间戳均以stream\_index选择的流为单位，或者如果stream\_index为-1，则以AV\_TIME\_BASE单位。如果标志包含AVSEEK\_FLAG\_ANY，则将非关键帧视为关键帧（并非所有解复用器均支持此关键帧）。如果标志包含AVSEEK\_FLAG\_BACKWARD，则将其忽略。

函数原型如下：

int avformat\_seek\_file ( AVFormatContext \*  s,  
int  stream\_index,  
int64\_t  min\_ts,  
int64\_t  ts,  
int64\_t  max\_ts,  
int  flags   
)

* 参数说明：

s：媒体文件句柄

stream\_index：流的索引，用作时基参考

min\_ts：最小可接受时间戳

ts：目标时间戳

max\_ts：最大可接受时间戳

flag:标志

注意：>=0表示返回成功，否则都是失败；同时要注意这是仍在构建中的新seek API的一部分。因此，请不要使用此功能。它可能随时更改，不要期望与ABI兼容

### **2、解封装相关结构体介绍：**

* AVFormatContext：从上面的api介绍中，我们可以经常看到这个结构体，它的重要性不言而喻了，它存储了音视频封装格式含有的信息，这里我不做具体介绍，列了几个出来，感兴趣的朋友可以去Avformat.h中查看：

typedef struct AVFormatContext {  
    /\*\*  
     \* A class **for** logging and @ref avoptions. Set by avformat\_alloc\_context().  
     \* Exports (de)muxer private options **if** they exist.  
     \*/  
    const AVClass \*av\_class;  
  
    /\*\*  
     \* The input container format.  
     \*  
     \* Demuxing only, set by avformat\_open\_input().  
     \*/  
    ff\_const59 struct AVInputFormat \*iformat;  
  
    /\*\*  
     \* The output container format.  
     \*  
     \* Muxing only, must be set by the caller before avformat\_write\_header().  
     \*/  
    ff\_const59 struct AVOutputFormat \*oformat;  
    /\*\*  
     \* Format private data. This is an AVOptions-enabled struct  
     \* **if** and only **if** iformat/oformat.priv\_class is not NULL.  
     \*  
     \* - muxing: set by avformat\_write\_header()  
     \* - demuxing: set by avformat\_open\_input()  
     \*/  
    void \*priv\_data;  
  
    /\*\*  
     \* I/O context.  
     \*  
     \* - demuxing: either set by the user before avformat\_open\_input() (**then**  
     \*             the user must close it manually) or set by avformat\_open\_input().  
     \* - muxing: set by the user before avformat\_write\_header(). The caller must  
     \*           take care of closing / freeing the IO context.  
     \*  
     \* Do NOT set this field **if** AVFMT\_NOFILE flag is set **in**  
     \* iformat/oformat.flags. In such a **case**, the (de)muxer will handle  
     \* I/O **in** some other way and this field will be NULL.  
     \*/  
    AVIOContext \*pb;  
  
    /\* stream info \*/  
    /\*\*  
     \* Flags signalling stream properties. A combination of AVFMTCTX\_\*.  
     \* Set by libavformat.  
     \*/  
    int ctx\_flags;  
  
    /\*\*  
     \* Number of elements **in** AVFormatContext.streams.  
     \*  
     \* Set by avformat\_new\_stream(), must not be modified by any other code.  
     \*/  
    unsigned int nb\_streams;  
    /\*\*  
     \* A list of all streams **in** the file. New streams are created with  
     \* avformat\_new\_stream().  
     \*  
     \* - demuxing: streams are created by libavformat **in** avformat\_open\_input().  
     \*             If AVFMTCTX\_NOHEADER is set **in** ctx\_flags, **then** new streams may also  
     \*             appear **in** av\_read\_frame().  
     \* - muxing: streams are created by the user before avformat\_write\_header().  
     \*  
     \* Freed by libavformat **in** avformat\_free\_context().  
     \*/  
    AVStream \*\*streams;  
#if FF\_API\_FORMAT\_FILENAME  
    /\*\*  
     \* input or output filename  
     \*  
     \* - demuxing: set by avformat\_open\_input()  
     \* - muxing: may be set by the caller before avformat\_write\_header()  
     \*  
     \* @deprecated Use url instead.  
     \*/  
    attribute\_deprecated  
    char filename[1024];  
#endif  
  
..............  
}；

大致简化为:

struct AVInputFormat \*iformat：输入数据的封装格式  
  
AVIOContext \*pb：输入数据的缓存  
  
unsigned int nb\_streams：视音频流的个数  
  
AVStream \*\*streams：视音频流  
  
char filename[1024]：文件名  
  
int64\_t duration：时长（单位：微秒us，转换为秒需要除以1000000）  
  
int bit\_rate：比特率（单位bps，转换为kbps需要除以1000）  
  
AVDictionary \*metadata：元数据

AVStream:表示存储每一个音频和视频流的信息。它也是在头文件AVformat.h里面查看：

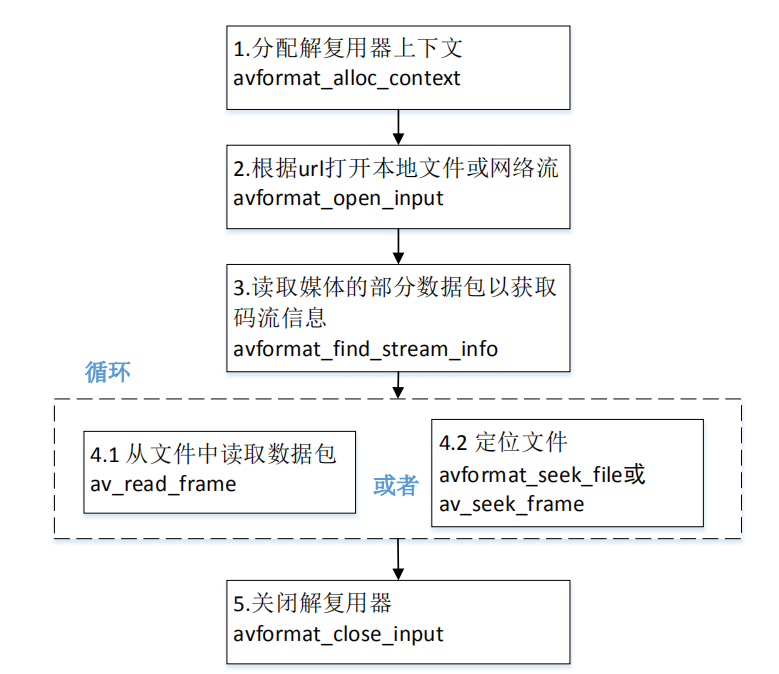
typedef struct AVStream {  
    int index;    /\*\*< stream index **in** AVFormatContext \*/  
    /\*\*  
     \* Format-specific stream ID.  
     \* decoding: set by libavformat  
     \* encoding: set by the user, replaced by libavformat **if** left unset  
     \*/  
    int id;  
#if FF\_API\_LAVF\_AVCTX  
    /\*\*  
     \* @deprecated use the codecpar struct instead  
     \*/  
    attribute\_deprecated  
    AVCodecContext \*codec;  
#endif  
    void \*priv\_data;  
  
    /\*\*  
     \* This is the fundamental unit of time (**in** seconds) **in** terms  
     \* of which frame timestamps are represented.  
     \*  
     \* decoding: set by libavformat  
     \* encoding: May be set by the caller before avformat\_write\_header() to  
     \*           provide a hint to the muxer about the desired timebase. In  
     \*           avformat\_write\_header(), the muxer will overwrite this field  
     \*           with the timebase that will actually be used **for** the timestamps  
     \*           written into the file (which may or may not be related to the  
     \*           user-provided one, depending on the format).  
     \*/  
    AVRational time\_base;  
  
................  
}

大致简化为：

int index：标识该视频/音频流  
  
AVCodecContext \*codec：指向该视频/音频流的AVCodecContext（它们一一对应）  
  
AVRational time\_base：时基。通过该值可以把PTS，DTS转化为真正的时间，只有AVStream中的time\_base是可用的。  
PTS\*time\_base=真正的时间  
  
int64\_t duration：该视频/音频流长度  
  
AVDictionary \*metadata：元数据信息  
  
AVRational avg\_frame\_rate：帧率  
  
AVPacket attached\_pic：附带的图片。比如说一些MP3，AAC音频文件附带的专辑封面。

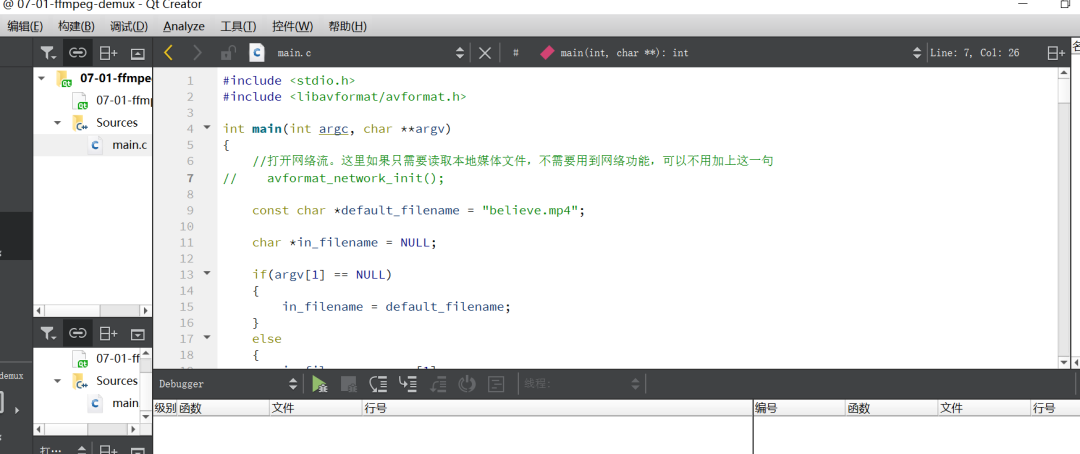
### **3、代码实现框架：解封装流程**

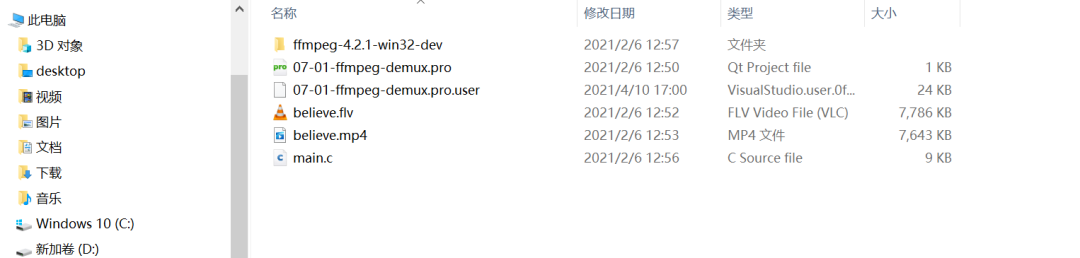
上面已经介绍了api和解封装结构体，剩下的就是我们该如何实现解封装的核心思想了，有了核心思想，我们就可以达到要实现的解封装效果了,具体流程图如下：



解封装实现流程图

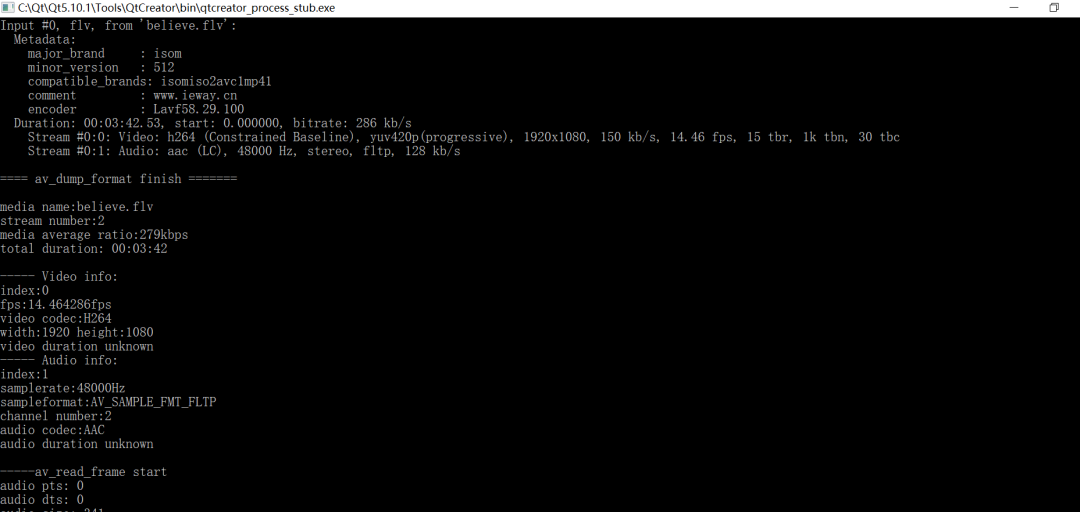
## **四、解封装具体实现代码：**

我这里开发环境是在qt下进行开发的，播放的是本地文件：



#include <stdio.h>  
#include <libavformat/avformat.h>  
  
int main(int argc, char \*\*argv)  
{  
    //打开网络流。这里如果只需要读取本地媒体文件，不需要用到网络功能，可以不用加上这一句  
//    avformat\_network\_init();  
  
    const char \*default\_filename = "believe.mp4";  
  
    char \*in\_filename = NULL;  
  
    **if**(argv[1] == NULL)  
    {  
        in\_filename = default\_filename;  
    }  
    **else**  
    {  
        in\_filename = argv[1];  
    }  
    printf("in\_filename = %s\n", in\_filename);  
  
    //AVFormatContext是描述一个媒体文件或媒体流的构成和基本信息的结构体  
    AVFormatContext \*ifmt\_ctx = NULL;           // 输入文件的demux  
  
    int videoindex = -1;        // 视频索引  
    int audioindex = -1;        // 音频索引  
  
  
    // 打开文件，主要是探测协议类型，如果是网络文件则创建网络链接  
    int ret = avformat\_open\_input(&ifmt\_ctx, in\_filename, NULL, NULL);  
    **if** (ret < 0)  //如果打开媒体文件失败，打印失败原因  
    {  
        char buf[1024] = { 0 };  
        av\_strerror(ret, buf, sizeof(buf) - 1);  
        printf("open %s failed:%s\n", in\_filename, buf);  
        goto failed;  
    }  
  
    ret = avformat\_find\_stream\_info(ifmt\_ctx, NULL);  
    **if** (ret < 0)  //如果打开媒体文件失败，打印失败原因  
    {  
        char buf[1024] = { 0 };  
        av\_strerror(ret, buf, sizeof(buf) - 1);  
        printf("avformat\_find\_stream\_info %s failed:%s\n", in\_filename, buf);  
        goto failed;  
    }  
  
    //打开媒体文件成功  
    printf\_s("\n==== av\_dump\_format in\_filename:%s ===\n", in\_filename);  
    av\_dump\_format(ifmt\_ctx, 0, in\_filename, 0);  
    printf\_s("\n==== av\_dump\_format finish =======\n\n");  
    // url: 调用avformat\_open\_input读取到的媒体文件的路径/名字  
    printf("media name:%s\n", ifmt\_ctx->url);  
    // nb\_streams: nb\_streams媒体流数量  
    printf("stream number:%d\n", ifmt\_ctx->nb\_streams);  
    // bit\_rate: 媒体文件的码率,单位为bps  
    printf("media average ratio:%lldkbps\n",(int64\_t)(ifmt\_ctx->bit\_rate/1024));  
    // 时间  
    int total\_seconds, hour, minute, second;  
    // duration: 媒体文件时长，单位微妙  
    total\_seconds = (ifmt\_ctx->duration) / AV\_TIME\_BASE;  // 1000us = 1ms, 1000ms = 1秒  
    hour = total\_seconds / 3600;  
    minute = (total\_seconds % 3600) / 60;  
    second = (total\_seconds % 60);  
    //通过上述运算，可以得到媒体文件的总时长  
    printf("total duration: %02d:%02d:%02d\n", hour, minute, second);  
    printf("\n");  
    /\*  
     \* 老版本通过遍历的方式读取媒体文件视频和音频的信息  
     \* 新版本的FFmpeg新增加了函数av\_find\_best\_stream，也可以取得同样的效果  
     \*/  
    **for** (uint32\_t i = 0; i < ifmt\_ctx->nb\_streams; i++)  
    {  
        AVStream \*in\_stream = ifmt\_ctx->streams[i];// 音频流、视频流、字幕流  
        //如果是音频流，则打印音频的信息  
        **if** (AVMEDIA\_TYPE\_AUDIO == in\_stream->codecpar->codec\_type)  
        {  
            printf("----- Audio info:\n");  
            // index: 每个流成分在ffmpeg解复用分析后都有唯一的index作为标识  
            printf("index:%d\n", in\_stream->index);  
            // sample\_rate: 音频编解码器的采样率，单位为Hz  
            printf("samplerate:%dHz\n", in\_stream->codecpar->sample\_rate);  
            // codecpar->format: 音频采样格式  
            **if** (AV\_SAMPLE\_FMT\_FLTP == in\_stream->codecpar->format)  
            {  
                printf("sampleformat:AV\_SAMPLE\_FMT\_FLTP\n");  
            }  
            **else** **if** (AV\_SAMPLE\_FMT\_S16P == in\_stream->codecpar->format)  
            {  
                printf("sampleformat:AV\_SAMPLE\_FMT\_S16P\n");  
            }  
            // channels: 音频信道数目  
            printf("channel number:%d\n", in\_stream->codecpar->channels);  
            // codec\_id: 音频压缩编码格式  
            **if** (AV\_CODEC\_ID\_AAC == in\_stream->codecpar->codec\_id)  
            {  
                printf("audio codec:AAC\n");  
            }  
            **else** **if** (AV\_CODEC\_ID\_MP3 == in\_stream->codecpar->codec\_id)  
            {  
                printf("audio codec:MP3\n");  
            }  
            **else**  
            {  
                printf("audio codec\_id:%d\n", in\_stream->codecpar->codec\_id);  
            }  
            // 音频总时长，单位为秒。注意如果把单位放大为毫秒或者微妙，音频总时长跟视频总时长不一定相等的  
            **if**(in\_stream->duration != AV\_NOPTS\_VALUE)  
            {  
                int duration\_audio = (in\_stream->duration) \* av\_q2d(in\_stream->time\_base);  
                //将音频总时长转换为时分秒的格式打印到控制台上  
                printf("audio duration: %02d:%02d:%02d\n",  
                       duration\_audio / 3600, (duration\_audio % 3600) / 60, (duration\_audio % 60));  
            }  
            **else**  
            {  
                printf("audio duration unknown");  
            }  
  
            printf("\n");  
  
            audioindex = i; // 获取音频的索引  
        }  
        **else** **if** (AVMEDIA\_TYPE\_VIDEO == in\_stream->codecpar->codec\_type)  //如果是视频流，则打印视频的信息  
        {  
            printf("----- Video info:\n");  
            printf("index:%d\n", in\_stream->index);  
            // avg\_frame\_rate: 视频帧率,单位为fps，表示每秒出现多少帧  
            printf("fps:%lffps\n", av\_q2d(in\_stream->avg\_frame\_rate));  
            **if** (AV\_CODEC\_ID\_MPEG4 == in\_stream->codecpar->codec\_id) //视频压缩编码格式  
            {  
                printf("video codec:MPEG4\n");  
            }  
            **else** **if** (AV\_CODEC\_ID\_H264 == in\_stream->codecpar->codec\_id) //视频压缩编码格式  
            {  
                printf("video codec:H264\n");  
            }  
            **else**  
            {  
                printf("video codec\_id:%d\n", in\_stream->codecpar->codec\_id);  
            }  
            // 视频帧宽度和帧高度  
            printf("width:%d height:%d\n", in\_stream->codecpar->width,  
                   in\_stream->codecpar->height);  
            //视频总时长，单位为秒。注意如果把单位放大为毫秒或者微妙，音频总时长跟视频总时长不一定相等的  
            **if**(in\_stream->duration != AV\_NOPTS\_VALUE)  
            {  
                int duration\_video = (in\_stream->duration) \* av\_q2d(in\_stream->time\_base);  
                printf("video duration: %02d:%02d:%02d\n",  
                       duration\_video / 3600,  
                       (duration\_video % 3600) / 60,  
                       (duration\_video % 60)); //将视频总时长转换为时分秒的格式打印到控制台上  
            }  
            **else**  
            {  
                printf("video duration unknown");  
            }  
  
            printf("\n");  
            videoindex = i;  
        }  
    }  
  
    AVPacket \*pkt = av\_packet\_alloc();  
  
    int pkt\_count = 0;  
    int print\_max\_count = 10;  
    printf("\n-----av\_read\_frame start\n");  
    **while** (1)  
    {  
        ret = av\_read\_frame(ifmt\_ctx, pkt);  
        **if** (ret < 0)  
        {  
            printf("av\_read\_frame end\n");  
            break;  
        }  
  
        **if**(pkt\_count++ < print\_max\_count)  
        {  
            **if** (pkt->stream\_index == audioindex)  
            {  
                printf("audio pts: %lld\n", pkt->pts);  
                printf("audio dts: %lld\n", pkt->dts);  
                printf("audio size: %d\n", pkt->size);  
                printf("audio pos: %lld\n", pkt->pos);  
                printf("audio duration: %lf\n\n",  
                       pkt->duration \* av\_q2d(ifmt\_ctx->streams[audioindex]->time\_base));  
            }  
            **else** **if** (pkt->stream\_index == videoindex)  
            {  
                printf("video pts: %lld\n", pkt->pts);  
                printf("video dts: %lld\n", pkt->dts);  
                printf("video size: %d\n", pkt->size);  
                printf("video pos: %lld\n", pkt->pos);  
                printf("video duration: %lf\n\n",  
                       pkt->duration \* av\_q2d(ifmt\_ctx->streams[videoindex]->time\_base));  
            }  
            **else**  
            {  
                printf("unknown stream\_index:\n", pkt->stream\_index);  
            }  
        }  
  
        av\_packet\_unref(pkt);  
    }  
  
    **if**(pkt)  
        av\_packet\_free(&pkt);  
failed:  
    **if**(ifmt\_ctx)  
        avformat\_close\_input(&ifmt\_ctx);  
  
  
    getchar(); //加上这一句，防止程序打印完信息马上退出  
    return 0;  
}

最终运行效果如下：

 运行结果

注意：不同封装格式的流媒体文件被解封装打印出来的信息是不同的，这点要注意!