详解FFMPEG API

2012-06-13 开花结果 阅 11882 转 91

转自：http://3xin2yi.info/wwwroot/tech/doku.php/tech:multimedia:ffmpeg

认识FFmpeg

FFMPEG堪称自由软件中最完备的一套多媒体支持库，它几乎实现了所有当下常见的数据封装格式、多媒体传输协议以及音视频编解码器。因此，对于从事多媒体技术开发的工程师来说，深入研究FFMPEG成为一门必不可少的工作，可以这样说，FFMPEG之于多媒体开发工程师的重要性正如kernel之于嵌入式系统工程师一般。

几个小知识：

FFMPEG项目是由法国人Fabrice Bellard发起的，此人也是著名的CPU模拟器项目QEMU的发起者，同时还是圆周率算法纪录的保持者。
FF是Fast Forward的意思，翻译成中文是“快进”。
FFMPEG的LOGO是一个”Z字扫描”示意图，Z字扫描用于将图像的二维频域数据一维化，同时保证了一维化的数据具备良好的统计特性，从而提高其后要进行的一维熵编码的效率。

关于耻辱柱（Hall of Shame）：FFMpeg大部分代码遵循LGPL许可证，如果使用者对FFMpeg进行了修改，要求公布修改的源代码；有少部分代码遵循GPL许可证，要求使用者同时公开使用FFMpeg的软件的源代码。实际上，除去部分大的系统软件开发商（Microsoft、Apple等）以及某些著名的音视频服务提供商（Divx、Real等）提供的自有播放器之外，绝大部分第三方开发的播放器都离不开FFMpeg的支持，像Linux桌面环境中的开源播放器VLC、MPlayer，Windows下的KMPlayer、暴风影音以及Android下几乎全部第三方播放器都是基于FFMpeg的。也有许多看似具备自主技术的播放器，其实也都不声不响地使用了FFMpeg，这种行为被称为“盗窃”，参与“盗窃”的公司的名字则被刻在耻辱柱上，国产播放器暴风影音、QQ影音于2009年上榜。

示例程序

```
[html]
<span style="font-size:18px;">解码：</span>

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include <sys/time.h>

#include "libavutil/avstring.h"
#include "libavformat/avformat.h"
#include "libavdevice/avdevice.h"
#include "libavutil/opt.h"
#include "libswscale/swscale.h"

#define DECODED_AUDIO_BUFFER_SIZE           192000

struct options
{
    int streamId;
    int frames;
    int nodec;
    int bplay;
    int thread_count;
    int64_t lstart;
    char finput[256];
    char foutput1[256];
    char foutput2[256];
};

int parse_options(struct options *opts, int argc, char** argv)
{
    int optidx;
    char *optstr;

    if (argc < 2) return -1;

```

```
    opts->streamId = -1;
    opts->lstart = -1;
    opts->frames = -1;
    opts->foutput1[0] = 0;
    opts->foutput2[0] = 0;
    opts->nodec = 0;
    opts->bplay = 0;
    opts->thread_count = 0;
    strcpy(opts->finput, argv[1]);

    optidx = 2;
    while (optidx < argc)
    {
        optstr = argv[optidx++];
        if (*optstr++ != '-') return -1;
        switch (*optstr++)
        {
        case 's':  //< stream id
            opts->streamId = atoi(optstr);
            break;
        case 'f':  //< frames
            opts->frames = atoi(optstr);
            break;
        case 'k':  //< skipped
            opts->lstart = atoll(optstr);
            break;
        case 'o':  //< output
            strcpy(opts->foutput1, optstr);
            strcat(opts->foutput1, ".mpg");
            strcpy(opts->foutput2, optstr);
            strcat(opts->foutput2, ".raw");
            break;
        case 'n': //decoding and output options
            if (strcmp("dec", optstr) == 0)
                opts->nodec = 1;
            break;
        case 'p':
            opts->bplay = 1;
            break;
        case 't':
            opts->thread_count = atoi(optstr);
            break;
        default:
            return -1;
        }
    }

    return 0;
}

void show_help(char* program)
{
    printf("Simple FFMPEG test program\n");
    printf("Usage: %s inputfile [-sstreamid [-fframes] [-kskipped] [-ooutput_filename(withou
t extension)] [-ndec] [-p] [-tthread_count]]\n",
           program);
    return;
}

static void log_callback(void* ptr, int level, const char* fmt, va_list vl)
{
    vfprintf(stdout, fmt, vl);
}

/*
* audio renderer code (oss)
*/
#include <sys/ioctl.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/soundcard.h>

#define OSS_DEVICE "/dev/dsp0"

struct audio_dsp
{
    int audio_fd;
    int channels;
    int format;
    int speed;
};
int map_formats(enum AVSampleFormat format)
{
    switch(format)
    {
        case AV_SAMPLE_FMT_U8:
            return AFMT_U8;
```

```
        case AV_SAMPLE_FMT_S16:
            return AFMT_S16_LE;
        default:
            return AFMT_U8;
    }
}
int set_audio(struct audio_dsp* dsp)
{
    if (dsp->audio_fd == -1)
    {
        printf("Invalid audio dsp id!\n");
        return -1;
    }

    if (-1 == ioctl(dsp->audio_fd, SNDCTL_DSP_SETFMT, &dsp->format))
    {
        printf("Failed to set dsp format!\n");
        return -1;
    }

    if (-1 == ioctl(dsp->audio_fd, SNDCTL_DSP_CHANNELS, &dsp->channels))
    {
        printf("Failed to set dsp format!\n");
        return -1;
    }

    if (-1 == ioctl(dsp->audio_fd, SNDCTL_DSP_SPEED, &dsp->speed))
    {
        printf("Failed to set dsp format!\n");
        return -1;
    }
    return 0;
}

int play_pcm(struct audio_dsp* dsp, unsigned char *buf, int size)
{
    if (dsp->audio_fd == -1)
    {
        printf("Invalid audio dsp id!\n");
        return -1;
    }

    if (-1 == write(dsp->audio_fd, buf, size))
    {
        printf("Failed to write audio dsp!\n");
        return -1;
    }

    return 0;
}
/* audio renderer code end */

/* video renderer code*/
#include <linux/fb.h>
#include <sys/mman.h>

#define FB_DEVICE "/dev/fb0"

enum pic_format
{
    eYUV_420_Planer,
};
struct video_fb
{
    int video_fd;
    struct fb_var_screeninfo vinfo;
    struct fb_fix_screeninfo finfo;
    unsigned char *fbp;
    AVFrame *frameRGB;
    struct
    {
        int x;
        int y;
    } video_pos;
};

int open_video(struct video_fb *fb, int x, int y)
{
    int screensize;
    fb->video_fd = open(FB_DEVICE, O_WRONLY);
    if (fb->video_fd == -1) return -1;

    if (ioctl(fb->video_fd, FBIOGET_FSCREENINFO, &fb->finfo)) return -2;
    if (ioctl(fb->video_fd, FBIOGET_VSCREENINFO, &fb->vinfo)) return -2;


    printf("video device: resolution %dx%d, %dbpp\n", fb->vinfo.xres, fb->vinfo.yres, fb->vinf
```

```
o.bits_per_pixel);
    screensize = fb->vinfo.xres * fb->vinfo.yres * fb->vinfo.bits_per_pixel / 8;
    fb->fbp = (unsigned char *) mmap(0, screensize, PROT_READ|PROT_WRITE, MAP_SHARED, fb->vide
o_fd, 0);
    if (fb->fbp == -1) return -3;

    if (x >= fb->vinfo.xres || y >= fb->vinfo.yres)
    {
        return -4;
    }
    else
    {
        fb->video_pos.x = x;
        fb->video_pos.y = y;
    }

    fb->frameRGB = avcodec_alloc_frame();
    if (!fb->frameRGB) return -5;

    return 0;
}

/* only 420P supported now */
int show_picture(struct video_fb *fb, AVFrame *frame, int width, int height, enum pic_format f
ormat)
{
    struct SwsContext *sws;
    int i;
    unsigned char *dest;
    unsigned char *src;

    if (fb->video_fd == -1) return -1;
    if ((fb->video_pos.x >= fb->vinfo.xres) || (fb->video_pos.y >= fb->vinfo.yres)) retur
n -2;

    if (fb->video_pos.x + width > fb->vinfo.xres)
    {
        width = fb->vinfo.xres - fb->video_pos.x;
    }
    if (fb->video_pos.y + height > fb->vinfo.yres)
    {
        height = fb->vinfo.yres - fb->video_pos.y;
    }

    if (format == PIX_FMT_YUV420P)
    {
        sws = sws_getContext(width, height, format, width, height, PIX_FMT_RGB32, SWS_FAST_BIL
INEAR, NULL, NULL, NULL);
        if (sws == 0)
        {
            return -3;
        }
        if (sws_scale(sws, frame->data, frame->linesize, 0, height, fb->frameRGB->data, fb->fr
ameRGB->linesize))
        {
            return -3;
        }

        dest = fb->fbp + (fb->video_pos.x+fb->vinfo.xoffset) * (fb->vinfo.bits_per_pixel/8) +
(fb->video_pos.y+fb->vinfo.yoffset) * fb->finfo.line_length;
        for (i = 0; i < height; i++)
        {
            memcpy(dest, src, width*4);
            src += fb->frameRGB->linesize[0];
            dest += fb->finfo.line_length;
        }
    }
    return 0;
}

void close_video(struct video_fb *fb)
{
    if (fb->video_fd != -1)
    {
        munmap(fb->fbp, fb->vinfo.xres * fb->vinfo.yres * fb->vinfo.bits_per_pixel / 8);
        close(fb->video_fd);
        fb->video_fd = -1;
    }
}
/* video renderer code end */

int main(int argc, char **argv)
{
    AVFormatContext* pCtx = 0;
    AVCodecContext *pCodecCtx = 0;
    AVCodec *pCodec = 0;
    AVPacket packet;
```

```
290.     AVFrame *pFrame = 0;
291.     FILE *fpo1 = NULL;
292.     FILE *fpo2 = NULL;
293.     int nframe;
294.     int err;
295.     int got_picture;
296.     int picwidth, picheight, linesize;
297.     unsigned char *pBuf;
298.     int i;
299.     int64_t timestamp;
300.     struct options opt;
301.     int usefo = 0;
302.     struct audio_dsp dsp;
303.     struct video_fb fb;
304.     int dusecs;
305.     float usecs1 = 0;
306.     float usecs2 = 0;
307.     struct timeval elapsed1, elapsed2;
308.     int decoded = 0;
309. 
310.     av_register_all();
311. 
312.     av_log_set_callback(log_callback);
313.     av_log_set_level(50);
314. 
315.     if (parse_options(&opt, argc, argv) < 0 || (strlen(opt.finput) == 0))
316.     {
317.         show_help(argv[0]);
318.         return 0;
319.     }
320. 
321. 
322.     err = avformat_open_input(&pCtx, opt.finput, 0, 0);
323.     if (err < 0)
324.     {
325.         printf("\n->(avformat_open_input)\tERROR:\t%d\n", err);
326.         goto fail;
327.     }
328.     err = avformat_find_stream_info(pCtx, 0);
329.     if (err < 0)
330.     {
331.         printf("\n->(avformat_find_stream_info)\tERROR:\t%d\n", err);
332.         goto fail;
333.     }
334.     if (opt.streamId < 0)
335.     {
336.         av_dump_format(pCtx, 0, pCtx->filename, 0);
337.         goto fail;
338.     }
339.     else
340.     {
341.         printf("\n extra data in Stream %d (%dB):", opt.streamId, pCtx->streams[opt.streamId]-
    >codec->extradata_size);
342.         for (i = 0; i < pCtx->streams[opt.streamId]->codec->extradata_size; i++)
343.         {
344.             if (i%16 == 0) printf("\n");
345.             printf("%2x  ", pCtx->streams[opt.streamId]->codec->extradata[i]);
346.         }
347.     }
348.     /* try opening output files */
349.     if (strlen(opt.foutput1) && strlen(opt.foutput2))
350.     {
351.         fpo1 = fopen(opt.foutput1, "wb");
352.         fpo2 = fopen(opt.foutput2, "wb");
353.         if (!fpo1 || !fpo2)
354.         {
355.             printf("\n->error opening output files\n");
356.             goto fail;
357.         }
358.         usefo = 1;
359.     }
360.     else
361.     {
362.         usefo = 0;
363.     }
364. 
365.     if (opt.streamId >= pCtx->nb_streams)
366.     {
367.         printf("\n->StreamId\tERROR\n");
368.         goto fail;
369.     }
370. 
371.     if (opt.lstart > 0)
372.     {
373.         err = av_seek_frame(pCtx, opt.streamId, opt.lstart, AVSEEK_FLAG_ANY);
374.         if (err < 0)
375.         {
```

```
            printf("\n->(av_seek_frame)\tERROR:\t%d\n", err);
            goto fail;
        }
    }

    /* for decoder configuration */
    if (!opt.nodec)
    {
        /* prepare codec */
        pCodecCtx = pCtx->streams[opt.streamId]->codec;

        if (opt.thread_count <= 16 && opt.thread_count > 0 )
        {
            pCodecCtx->thread_count = opt.thread_count;
            pCodecCtx->thread_type = FF_THREAD_FRAME;
        }
        pCodec = avcodec_find_decoder(pCodecCtx->codec_id);
        if (!pCodec)
        {
            printf("\n->can not find codec!\n");
            goto fail;
        }
        err = avcodec_open2(pCodecCtx, pCodec, 0);
        if (err < 0)
        {
            printf("\n->(avcodec_open)\tERROR:\t%d\n", err);
            goto fail;
        }
        pFrame = avcodec_alloc_frame();

        /* prepare device */
        if (opt.bplay)
        {
            /* audio devices */
            dsp.audio_fd = open(OSS_DEVICE, O_WRONLY);
            if (dsp.audio_fd == -1)
            {
                printf("\n-> can not open audio device\n");
                goto fail;
            }
            dsp.channels = pCodecCtx->channels;
            dsp.speed = pCodecCtx->sample_rate;
            dsp.format = map_formats(pCodecCtx->sample_fmt);
            if (set_audio(&dsp) < 0)
            {
                printf("\n-> can not set audio device\n");
                goto fail;
            }
            /* video devices */
            if (open_video(&fb, 0, 0) != 0)
            {
                printf("\n-> can not open video device\n");
                goto fail;
            }
        }
    }

    nframe = 0;
    while(nframe < opt.frames || opt.frames == -1)
    {
        gettimeofday(&elapsed1, NULL);
        err = av_read_frame(pCtx, &packet);
        if (err < 0)
        {
            printf("\n->(av_read_frame)\tERROR:\t%d\n", err);
            break;
        }
        gettimeofday(&elapsed2, NULL);
        dusecs = (elapsed2.tv_sec - elapsed1.tv_sec)*1000000 + (elapsed2.tv_usec - elapsed1.t
v_usec);
        usecs2 += dusecs;
        timestamp = av_rescale_q(packet.dts, pCtx->streams[packet.stream_index]->time_base, (A
VRational){1, AV_TIME_BASE});
        printf("\nFrame No %5d stream#%d\tsize %6dB, timestamp:%6lld, dts:%6lld, pts:%6ll
d, ", nframe++, packet.stream_index, packet.size,
               timestamp, packet.dts, packet.pts);

        if (packet.stream_index == opt.streamId)
        {
#if 0
            for (i = 0; i < 16; /*packet.size;*/ i++)
            {
                if (i%16 == 0) printf("\n pktdata: ");
                printf("%2x  ", packet.data[i]);
            }
            printf("\n");
#endif
```

```
460.            if (usefo)
461.            {
462.                fwrite(packet.data, packet.size, 1, fpo1);
463.                fflush(fpo1);
464.            }
465.
466.            if (pCtx->streams[opt.streamId]->codec->codec_type == AVMEDIA_TYPE_VIDEO && !opt.n
    odec)
467.            {
468.                picheight = pCtx->streams[opt.streamId]->codec->height;
469.                picwidth = pCtx->streams[opt.streamId]->codec->width;
470.
471.                gettimeofday(&elapsed1, NULL);
472.                avcodec_decode_video2(pCodecCtx, pFrame, &got_picture, &packet);
473.                decoded++;
474.                gettimeofday(&elapsed2, NULL);
475.                dusecs = (elapsed2.tv_sec - elapsed1.tv_sec)*1000000 + (elapsed2.tv_usec - ela
    psed1.tv_usec);
476.                usecs1 += dusecs;
477.
478.                if (got_picture)
479.                {
480.                    printf("[Video: type %d, ref %d, pts %lld, pkt_pts %lld, pkt_dts %ll
    d]",
481.                            pFrame->pict_type, pFrame->reference, pFrame->pts, pFrame->pkt_pt
    s, pFrame->pkt_dts);
482.
483.                    if (pCtx->streams[opt.streamId]->codec->pix_fmt == PIX_FMT_YUV420P)
484.                    {
485.                        if (usefo)
486.                        {
487.                            linesize = pFrame->linesize[0];
488.                            pBuf = pFrame->data[0];
489.                            for (i = 0; i < picheight; i++)
490.                            {
491.                                fwrite(pBuf, picwidth, 1, fpo2);
492.                                pBuf += linesize;
493.                            }
494.
495.                            linesize = pFrame->linesize[1];
496.                            pBuf = pFrame->data[1];
497.                            for (i = 0; i < picheight/2; i++)
498.                            {
499.                                fwrite(pBuf, picwidth/2, 1, fpo2);
500.                                pBuf += linesize;
501.                            }
502.
503.                            linesize = pFrame->linesize[2];
504.                            pBuf = pFrame->data[2];
505.                            for (i = 0; i < picheight/2; i++)
506.                            {
507.                                fwrite(pBuf, picwidth/2, 1, fpo2);
508.                                pBuf += linesize;
509.                            }
510.                            fflush(fpo2);
511.                        }
512.
513.                        if (opt.bplay)
514.                        {
515.                            /* show picture */
516.                            show_picture(&fb, pFrame, picheight, picwidth, PIX_FMT_YUV420P);
517.                        }
518.                    }
519.                }
520.                av_free_packet(&packet);
521.            }
522.            else if (pCtx->streams[opt.streamId]->codec->codec_type == AVMEDIA_TYPE_AUDI
    O && !opt.nodec)
523.            {
524.                int got;
525.
526.                gettimeofday(&elapsed1, NULL);
527.                avcodec_decode_audio4(pCodecCtx, pFrame, &got, &packet);
528.                decoded++;
529.                gettimeofday(&elapsed2, NULL);
530.                dusecs = (elapsed2.tv_sec - elapsed1.tv_sec)*1000000 + (elapsed2.tv_usec - ela
    psed1.tv_usec);
531.                usecs1 += dusecs;
532.
533.                if (got)
534.                {
535.                    printf("[Audio: %5dB raw data, decoding time: %d]", pFrame->linesize[0], d
    usecs);
536.                    if (usefo)
537.                    {
538.                        fwrite(pFrame->data[0],  pFrame->linesize[0], 1, fpo2);
539.                        fflush(fpo2);
```

```
                    }
                    if (opt.bplay)
                    {
                        play_pcm(&dsp, pFrame->data[0],  pFrame->linesize[0]);
                    }
                }
            }
        }

    if (!opt.nodec && pCodecCtx)
    {
        avcodec_close(pCodecCtx);
    }

    printf("\n%d frames parsed, average %.2f us per frame\n", nframe, usecs2/nframe);
    printf("%d frames decoded, average %.2f us per frame\n", decoded, usecs1/decoded);

fail:
    if (pCtx)
    {
        avformat_close_input(&pCtx);
    }
    if (fpo1)
    {
        fclose(fpo1);
    }
    if (fpo2)
    {
        fclose(fpo2);
    }
    if (!pFrame)
    {
        av_free(pFrame);
    }
    if (!usefo && (dsp.audio_fd != -1))
    {
        close(dsp.audio_fd);
    }
    if (!usefo && (fb.video_fd != -1))
    {
        close_video(&fb);
    }
    return 0;
}
```

这一小段代码可以实现的功能包括：

打开一个多媒体文件并获取基本的媒体信息。
获取编码器句柄。
根据给定的时间标签进行一个跳转。
读取数据帧。
解码音频帧或者视频帧。
关闭多媒体文件。

这些功能足以支持一个功能强大的多媒体播放器，因为最复杂的解复用、解码、数据分析过程已经在FFMpeg内部实现了，需要关注的仅剩同步问题。

用户接口

数据结构

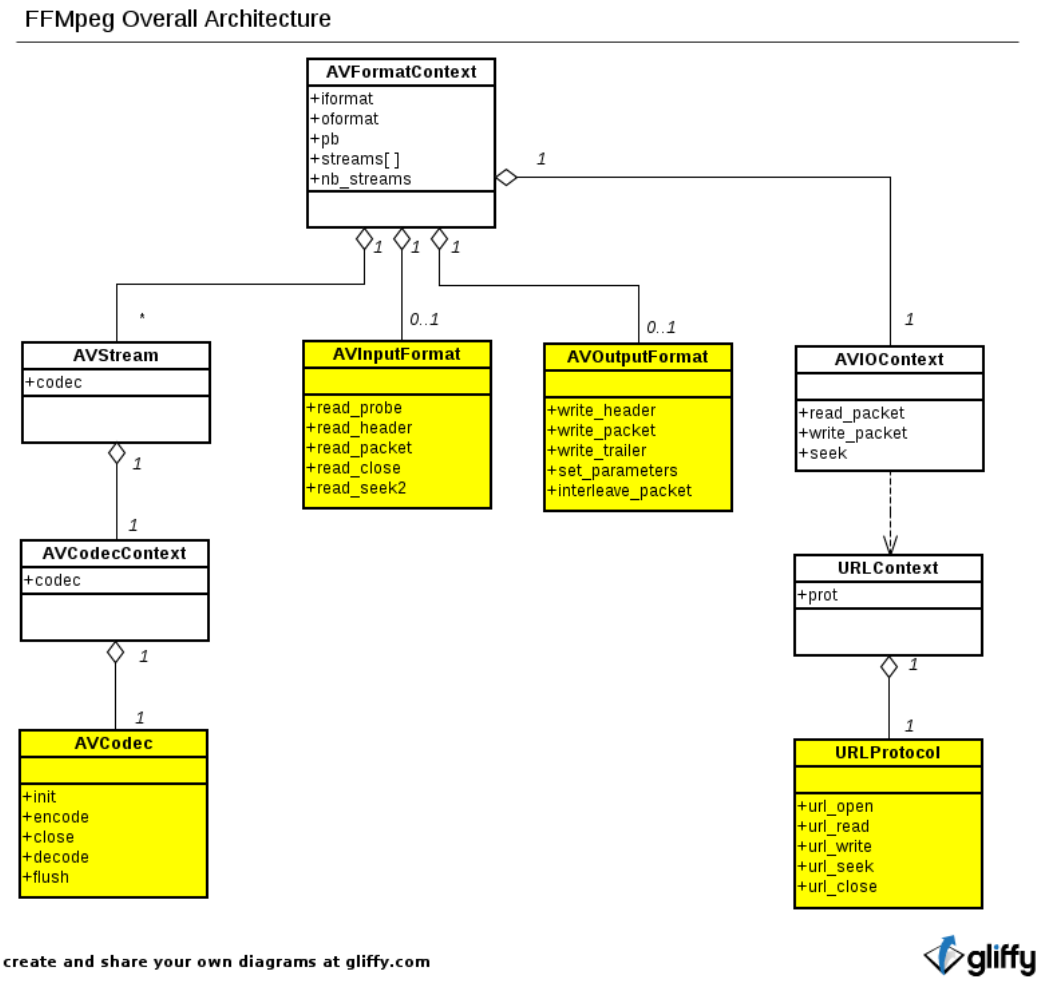
基本概念

编解码器、数据帧、媒体流和容器是数字媒体处理系统的四个基本概念。

首先需要统一术语：

容器 / 文件（Conainer/File）：即特定格式的多媒体文件。
媒体流（Stream）：指时间轴上的一段连续数据，如一段声音数据，一段视频数据或一段字幕数据，可以是压缩的，也可以是非压缩的，压缩的数据需要关联特定的编解码器。
数据帧 / 数据包（Frame/Packet）：通常，一个媒体流由大量的数据帧组成，对于压缩数据，帧对应着编解码器的最小处理单元。通常，分属于不同媒体流的数据帧交错复用于容器之中，参见交错。
编解码器：编解码器以帧为单位实现压缩数据和原始数据之间的相互转换。

在FFMPEG中，使用AVFormatContext、AVStream、AVCodecContext、AVCodec及AVPacket等结构来抽象这些基本要素，它们的关系如下图所示：



AVCodecContext

这是一个描述编解码器上下文的数据结构，包含了众多编解码器需要的参数信息，如下列出了部分比较重要的域：

```cpp
typedef struct AVCodecContext {

    ......

    /**
     * some codecs need / can use extradata like Huffman tables.
     * mjpeg: Huffman tables
     * rv10: additional flags
     * mpeg4: global headers (they can be in the bitstream or here)
     * The allocated memory should be FF_INPUT_BUFFER_PADDING_SIZE bytes larger
     * than extradata_size to avoid prolems if it is read with the bitstream reader.
     * The bytewise contents of extradata must not depend on the architecture or CPU endiannes
s.
     * - encoding: Set/allocated/freed by libavcodec.
     * - decoding: Set/allocated/freed by user.
     */
    uint8_t *extradata;
    int extradata_size;
    /**
     * This is the fundamental unit of time (in seconds) in terms
     * of which frame timestamps are represented. For fixed-fps content,
     * timebase should be 1/framerate and timestamp increments should be
     * identically 1.
     * - encoding: MUST be set by user.
     * - decoding: Set by libavcodec.
     */
    AVRational time_base;

    /* video only */
    /**
     * picture width / height.
     * - encoding: MUST be set by user.
     * - decoding: Set by libavcodec.
     * Note: For compatibility it is possible to set this instead of
     * coded_width/height before decoding.
     */
    int width, height;

```

```
    ......

    /* audio only */
    int sample_rate; ///< samples per second
    int channels;    ///< number of audio channels

    /**
     * audio sample format
     * - encoding: Set by user.
     * - decoding: Set by libavcodec.
     */
    enum SampleFormat sample_fmt;  ///< sample format

    /* The following data should not be initialized. */
    /**
     * Samples per packet, initialized when calling 'init'.
     */
    int frame_size;
    int frame_number;   ///< audio or video frame number

    ......

    char codec_name[32];
    enum AVMediaType codec_type; /* see AVMEDIA_TYPE_xxx */
    enum CodecID codec_id; /* see CODEC_ID_xxx */

    /**
     * fourcc (LSB first, so "ABCD" -> ('D'<<24) + ('C'<<16) + ('B'<<8) + 'A').
     * This is used to work around some encoder bugs.
     * A demuxer should set this to what is stored in the field used to identify the codec.
     * If there are multiple such fields in a container then the demuxer should choose the on
e
     * which maximizes the information about the used codec.
     * If the codec tag field in a container is larger then 32 bits then the demuxer should
     * remap the longer ID to 32 bits with a table or other structure. Alternatively a new
     * extra_codec_tag + size could be added but for this a clear advantage must be demonstrat
ed
     * first.
     * - encoding: Set by user, if not then the default based on codec_id will be used.
     * - decoding: Set by user, will be converted to uppercase by libavcodec during init.
     */
    unsigned int codec_tag;

    ......

    /**
     * Size of the frame reordering buffer in the decoder.
     * For MPEG-2 it is 1 IPB or 0 low delay IP.
     * - encoding: Set by libavcodec.
     * - decoding: Set by libavcodec.
     */
    int has_b_frames;

    /**
     * number of bytes per packet if constant and known or 0
     * Used by some WAV based audio codecs.
     */
    int block_align;

    ......

    /**
     * bits per sample/pixel from the demuxer (needed for huffyuv).
     * - encoding: Set by libavcodec.
     * - decoding: Set by user.
     */
     int bits_per_coded_sample;

     ......

} AVCodecContext;
```

如果是单纯使用libavcodec，这部分信息需要调用者进行初始化；如果是使用整个FFMPEG库，这部分信息在调用avformat_open_input和avformat_find_stream_info的过程中根据文件的头信息及媒体流内的头部信息完成初始化。其中几个主要域的释义如下：

1. extradata/extradata_size：这个buffer中存放了解码器可能会用到的额外信息，在av_read_frame中填充。一般来说，首先，某种具体格式的demuxer在读取格式头信息的时候会填充extradata，其次，如果demuxer没有做这个事情，比如可能在头部压根儿就没有相关的编解码信息，则相应的parser会继续从已经解复用出来的媒体流中继续寻找。在没有找到任何额外信息的情况下，这个buffer指针为空。
2. time_base：

3. width/height：视频的宽和高。
4. sample_rate/channels：音频的采样率和信道数目。
5. sample_fmt： 音频的原始采样格式。
6. codec_name/codec_type/codec_id/codec_tag：编解码器的信息。

AVStream

该结构体描述一个媒体流，定义如下：

```cpp
typedef struct AVStream {
    int index;    /**< stream index in AVFormatContext */
    int id;       /**< format-specific stream ID */
    AVCodecContext *codec; /**< codec context */
    /**
     * Real base framerate of the stream.
     * This is the lowest framerate with which all timestamps can be
     * represented accurately (it is the least common multiple of all
     * framerates in the stream). Note, this value is just a guess!
     * For example, if the time base is 1/90000 and all frames have either
     * approximately 3600 or 1800 timer ticks, then r_frame_rate will be 50/1.
     */
    AVRational r_frame_rate;

    ......

    /**
     * This is the fundamental unit of time (in seconds) in terms
     * of which frame timestamps are represented. For fixed-fps content,
     * time base should be 1/framerate and timestamp increments should be 1.
     */
    AVRational time_base;

    ......

    /**
     * Decoding: pts of the first frame of the stream, in stream time base.
     * Only set this if you are absolutely 100% sure that the value you set
     * it to really is the pts of the first frame.
     * This may be undefined (AV_NOPTS_VALUE).
     * @note The ASF header does NOT contain a correct start_time the ASF
     * demuxer must NOT set this.
     */
    int64_t start_time;
    /**
     * Decoding: duration of the stream, in stream time base.
     * If a source file does not specify a duration, but does specify
     * a bitrate, this value will be estimated from bitrate and file size.
     */
    int64_t duration;

#if LIBAVFORMAT_VERSION_INT < (53<<16)
    char language[4]; /** ISO 639-2/B 3-letter language code (empty string if undefined) */
#endif

    /* av_read_frame() support */
    enum AVStreamParseType need_parsing;
    struct AVCodecParserContext *parser;

    ......

    /* av_seek_frame() support */
    AVIndexEntry *index_entries; /**< Only used if the format does not
                                  support seeking natively. */
    int nb_index_entries;
    unsigned int index_entries_allocated_size;

    int64_t nb_frames;                 ///< number of frames in this stream if known or 0

    ......

    /**
     * Average framerate
     */
    AVRational avg_frame_rate;
    ......
} AVStream;
```

主要域的释义如下，其中大部分域的值可以由avformat_open_input根据文件头的信息确定，缺少的信息需要通过调用avforma
t_find_stream_info读帧及软解码进一步获取：

1. index/id：index对应流的索引，这个数字是自动生成的，根据index可以从AVFormatContext::streams表中索引到该流；而id则是流的标识，依赖于具体的容器格式。比如对于MPEG TS格式，id就是pid。
2. time_base：流的时间基准，是一个实数，该流中媒体数据的pts和dts都将以这个时间基准为粒度。通常，使用av_rescale/av_rescale_q可以实现不同时间基准的转换。
3. start_time：流的起始时间，以流的时间基准为单位，通常是该流中第一个帧的pts。
4. duration：流的总时间，以流的时间基准为单位。
5. need_parsing：对该流parsing过程的控制域。
6. nb_frames：流内的帧数目。
7. r_frame_rate/framerate/avg_frame_rate：帧率相关。
8. codec：指向该流对应的AVCodecContext结构，调用avformat_open_input时生成。
9. parser：指向该流对应的AVCodecParserContext结构，调用avformat_find_stream_info时生成。。

AVFormatContext

这个结构体描述了一个媒体文件或媒体流的构成和基本信息，定义如下：

```cpp
typedef struct AVFormatContext {
    const AVClass *av_class; /**< Set by avformat_alloc_context. */
    /* Can only be iformat or oformat, not both at the same time. */
    struct AVInputFormat *iformat;
    struct AVOutputFormat *oformat;
    void *priv_data;
    ByteIOContext *pb;
    unsigned int nb_streams;
    AVStream *streams[MAX_STREAMS];
    char filename[1024]; /**< input or output filename */
    /* stream info */
    int64_t timestamp;
#if LIBAVFORMAT_VERSION_INT < (53<<16)
    char title[512];
    char author[512];
    char copyright[512];
    char comment[512];
    char album[512];
    int year;  /**< ID3 year, 0 if none */
    int track; /**< track number, 0 if none */
    char genre[32]; /**< ID3 genre */
#endif

    int ctx_flags; /**< Format-specific flags, see AVFMTCTX_xx */
    /* private data for pts handling (do not modify directly). */
    /** This buffer is only needed when packets were already buffered but
       not decoded, for example to get the codec parameters in MPEG
       streams. */
    struct AVPacketList *packet_buffer;

    /** Decoding: position of the first frame of the component, in
       AV_TIME_BASE fractional seconds. NEVER set this value directly:
       It is deduced from the AVStream values.  */
    int64_t start_time;
    /** Decoding: duration of the stream, in AV_TIME_BASE fractional
       seconds. Only set this value if you know none of the individual stream
       durations and also dont set any of them. This is deduced from the
       AVStream values if not set.  */
    int64_t duration;
    /** decoding: total file size, 0 if unknown */
    int64_t file_size;
    /** Decoding: total stream bitrate in bit/s, 0 if not
       available. Never set it directly if the file_size and the
       duration are known as FFmpeg can compute it automatically. */
    int bit_rate;

    /* av_read_frame() support */
    AVStream *cur_st;
#if LIBAVFORMAT_VERSION_INT < (53<<16)
    const uint8_t *cur_ptr_deprecated;
    int cur_len_deprecated;
    AVPacket cur_pkt_deprecated;
#endif

    /* av_seek_frame() support */
    int64_t data_offset; /** offset of the first packet */
    int index_built;

    int mux_rate;
    unsigned int packet_size;
    int preload;
    int max_delay;

```

```
64. #define AVFMT_NOOUTPUTLOOP -1
65. #define AVFMT_INFINITEOUTPUTLOOP 0
66.     /** number of times to loop output in formats that support it */
67.     int loop_output;
68. 
69.     int flags;
70. #define AVFMT_FLAG_GENPTS       0x0001 ///< Generate missing pts even if it requires parsing f
    uture frames.
71. #define AVFMT_FLAG_IGNIDX       0x0002 ///< Ignore index.
72. #define AVFMT_FLAG_NONBLOCK     0x0004 ///< Do not block when reading packets from input.
73. #define AVFMT_FLAG_IGNDTS       0x0008 ///< Ignore DTS on frames that contain both DTS & PTS
74. #define AVFMT_FLAG_NOFILLIN     0x0010 ///< Do not infer any values from other values, just re
    turn what is stored in the container
75. #define AVFMT_FLAG_NOPARSE      0x0020 ///< Do not use AVParsers, you also must set AVFMT_FLA
    G_NOFILLIN as the fillin code works on frames and no parsing -> no frames. Also seeking to fra
    mes can not work if parsing to find frame boundaries has been disabled
76. #define AVFMT_FLAG_RTP_HINT     0x0040 ///< Add RTP hinting to the output file
77. 
78.     int loop_input;
79.     /** decoding: size of data to probe; encoding: unused. */
80.     unsigned int probesize;
81. 
82.     /**
83.      * Maximum time (in AV_TIME_BASE units) during which the input should
84.      * be analyzed in avformat_find_stream_info().
85.      */
86.     int max_analyze_duration;
87. 
88.     const uint8_t *key;
89.     int keylen;
90. 
91.     unsigned int nb_programs;
92.     AVProgram **programs;
93. 
94.     /**
95.      * Forced video codec_id.
96.      * Demuxing: Set by user.
97.      */
98.     enum CodecID video_codec_id;
99.     /**
100.      * Forced audio codec_id.
101.      * Demuxing: Set by user.
102.      */
103.     enum CodecID audio_codec_id;
104.     /**
105.      * Forced subtitle codec_id.
106.      * Demuxing: Set by user.
107.      */
108.     enum CodecID subtitle_codec_id;
109. 
110.     /**
111.      * Maximum amount of memory in bytes to use for the index of each stream.
112.      * If the index exceeds this size, entries will be discarded as
113.      * needed to maintain a smaller size. This can lead to slower or less
114.      * accurate seeking (depends on demuxer).
115.      * Demuxers for which a full in-memory index is mandatory will ignore
116.      * this.
117.      * muxing  : unused
118.      * demuxing: set by user
119.      */
120.     unsigned int max_index_size;
121. 
122.     /**
123.      * Maximum amount of memory in bytes to use for buffering frames
124.      * obtained from realtime capture devices.
125.      */
126.     unsigned int max_picture_buffer;
127. 
128.     unsigned int nb_chapters;
129.     AVChapter **chapters;
130. 
131.     /**
132.      * Flags to enable debugging.
133.      */
134.     int debug;
135. #define FF_FDEBUG_TS        0x0001
136. 
137.     /**
138.      * Raw packets from the demuxer, prior to parsing and decoding.
139.      * This buffer is used for buffering packets until the codec can
140.      * be identified, as parsing cannot be done without knowing the
141.      * codec.
142.      */
143.     struct AVPacketList *raw_packet_buffer;
144.     struct AVPacketList *raw_packet_buffer_end;
145. 
146.     struct AVPacketList *packet_buffer_end;
```

```
    AVMetadata *metadata;

    /**
     * Remaining size available for raw_packet_buffer, in bytes.
     * NOT PART OF PUBLIC API
     */
#define RAW_PACKET_BUFFER_SIZE 2500000
    int raw_packet_buffer_remaining_size;

    /**
     * Start time of the stream in real world time, in microseconds
     * since the unix epoch (00:00 1st January 1970). That is, pts=0
     * in the stream was captured at this real world time.
     * - encoding: Set by user.
     * - decoding: Unused.
     */
    int64_t start_time_realtime;
} AVFormatContext;
```

这是FFMpeg中最为基本的一个结构，是其他所有结构的根，是一个多媒体文件或流的根本抽象。其中:

nb_streams和streams所表示的AVStream结构指针数组包含了所有内嵌媒体流的描述；
iformat和oformat指向对应的demuxer和muxer指针；
pb则指向一个控制底层数据读写的ByteIOContext结构。
start_time和duration是从streams数组的各个AVStream中推断出的多媒体文件的起始时间和长度，以微妙为单位。

通常，这个结构由avformat_open_input在内部创建并以缺省值初始化部分成员。但是，如果调用者希望自己创建该结构，则需要显式为该结构的一些成员置缺省值——如果没有缺省值的话，会导致之后的动作产生异常。以下成员需要被关注：

probesize
mux_rate
packet_size
flags
max_analyze_duration
key
max_index_size
max_picture_buffer
max_delay

AVPacket

AVPacket定义在avcodec.h中，如下：

```
[cpp]
typedef struct AVPacket {
    /**
     * Presentation timestamp in AVStream->time_base units; the time at which
     * the decompressed packet will be presented to the user.
     * Can be AV_NOPTS_VALUE if it is not stored in the file.
     * pts MUST be larger or equal to dts as presentation cannot happen before
     * decompression, unless one wants to view hex dumps. Some formats misuse
     * the terms dts and pts/cts to mean something different. Such timestamps
     * must be converted to true pts/dts before they are stored in AVPacket.
     */
    int64_t pts;
    /**
     * Decompression timestamp in AVStream->time_base units; the time at which
     * the packet is decompressed.
     * Can be AV_NOPTS_VALUE if it is not stored in the file.
     */
    int64_t dts;
    uint8_t *data;
    int   size;
    int   stream_index;
    int   flags;
    /**
     * Duration of this packet in AVStream->time_base units, 0 if unknown.
     * Equals next_pts - this_pts in presentation order.
     */
    int   duration;
    void  (*destruct)(struct AVPacket *);
    void  *priv;
    int64_t pos;                            ///< byte position in stream, -1 if unknown

    /**
     * Time difference in AVStream->time_base units from the pts of this
```

```
     * packet to the point at which the output from the decoder has converged
     * independent from the availability of previous frames. That is, the
     * frames are virtually identical no matter if decoding started from
     * the very first frame or from this keyframe.
     * Is AV_NOPTS_VALUE if unknown.
     * This field is not the display duration of the current packet.
     *
     * The purpose of this field is to allow seeking in streams that have no
     * keyframes in the conventional sense. It corresponds to the
     * recovery point SEI in H.264 and match_time_delta in NUT. It is also
     * essential for some types of subtitle streams to ensure that all
     * subtitles are correctly displayed after seeking.
     */
    int64_t convergence_duration;
} AVPacket;
```

FFMPEG使用AVPacket来暂存解复用之后、解码之前的媒体数据（一个音/视频帧、一个字幕包等）及附加信息（解码时间戳、显示时间戳、时长等）。其中：

dts表示解码时间戳，pts表示显示时间戳，它们的单位是所属媒体流的时间基准。
stream_index给出所属媒体流的索引；
data为数据缓冲区指针，size为长度；
duration为数据的时长，也是以所属媒体流的时间基准为单位；
pos表示该数据在媒体流中的字节偏移量；
destruct为用于释放数据缓冲区的函数指针；
flags为标志域，其中，最低为置1表示该数据是一个关键帧。

AVPacket结构本身只是个容器，它使用data成员引用实际的数据缓冲区。这个缓冲区通常是由av_new_packet创建的，但也可能由FFMPEG的API创建（如av_read_frame）。当某个AVPacket结构的数据缓冲区不再被使用时，要需要通过调用av_free_packet释放。av_free_packet调用的是结构体本身的destruct函数，它的值有两种情况：1)av_destruct_packet_nofree或0；2)av_destruct_packet，其中，情况1)仅仅是将data和size的值清0而已，情况2)才会真正地释放缓冲区。

FFMPEG内部使用AVPacket结构建立缓冲区装载数据，同时提供destruct函数，如果FFMPEG打算自己维护缓冲区，则将destruct设为av_destruct_packet_nofree，用户调用av_free_packet清理缓冲区时并不能够将其释放；如果FFMPEG打算将该缓冲区彻底交给调用者，则将destruct设为av_destruct_packet，表示它能够被释放。安全起见，如果用户希望自由地使用一个FFMPEG内部创建的AVPacket结构，最好调用av_dup_packet进行缓冲区的克隆，将其转化为缓冲区能够被释放的AVPacket，以免对缓冲区的不当占用造成异常错误。av_dup_packet会为destruct指针为av_destruct_packet_nofree的AVPacket新建一个缓冲区，然后将原缓冲区的数据拷贝至新缓冲区，置data的值为新缓冲区的地址，同时设destruct指针为av_destruct_packet。

时间信息

时间信息用于实现多媒体同步。

同步的目的在于展示多媒体信息时，能够保持媒体对象之间固有的时间关系。同步有两类，一类是流内同步，其主要任务是保证单个媒体流内的时间关系，以满足感知要求，如按照规定的帧率播放一段视频；另一类是流间同步，主要任务是保证不同媒体流之间的时间关系，如音频和视频之间的关系（lipsync）。

对于固定速率的媒体，如固定帧率的视频或固定比特率的音频，可以将时间信息（帧率或比特率）置于文件首部（header），如AVI的hdrl List、MP4的moov box，还有一种相对复杂的方案是将时间信息嵌入媒体流的内部，如MPEG TS和Real video，这种方案可以处理变速率的媒体，亦可有效避免同步过程中的时间漂移。

FFMPEG会为每一个数据包打上时间标签，以更有效地支持上层应用的同步机制。时间标签有两种，一种是DTS，称为解码时间标签，另一种是PTS，称为显示时间标签。对于声音来说，这两个时间标签是相同的，但对于某些视频编码格式，由于采用了双向预测技术，会造成DTS和PTS的不一致。

无双向预测帧的情况：

```
图像类型: I   P   P   P   P   P   P ...  I   P   P
DTS:      0   1   2   3   4   5   6...   100 101 102
PTS:      0   1   2   3   4   5   6...   100 101 102
```

有双向预测帧的情况：

```
图像类型: I   P   B   B   P   B   B ...  I   P   B
DTS:      0   1   2   3   4   5   6 ...   100 101 102
PTS:      0   3   1   2   6   4   5 ...   100 104 102
```

对于存在双向预测帧的情况，通常要求解码器对图像重排序，以保证输出的图像顺序为显示顺序：

解码器输入：I P B B P B B

(DTS) 0 1 2 3 4 5 6
(PTS) 0 3 1 2 6 4 5
解码器输出：X I B B P B B P
(PTS) X 0 1 2 3 4 5 6

时间信息的获取：

通过调用avformat_find_stream_info，多媒体应用可以从AVFormatContext对象中拿到媒体文件的时间信息：主要是总时间长度和开始时间，此外还有与时间信息相关的比特率和文件大小。其中时间信息的单位是AV_TIME_BASE：微秒。

```cpp
typedef struct AVFormatContext {

    ......

    /** Decoding: position of the first frame of the component, in
       AV_TIME_BASE fractional seconds. NEVER set this value directly:
       It is deduced from the AVStream values.  */
    int64_t start_time;
    /** Decoding: duration of the stream, in AV_TIME_BASE fractional
       seconds. Only set this value if you know none of the individual stream
       durations and also dont set any of them. This is deduced from the
       AVStream values if not set.  */
    int64_t duration;
    /** decoding: total file size, 0 if unknown */
    int64_t file_size;
    /** Decoding: total stream bitrate in bit/s, 0 if not
       available. Never set it directly if the file_size and the
       duration are known as FFmpeg can compute it automatically. */
    int bit_rate;

    ......

} AVFormatContext;
```

以上4个成员变量都是只读的，基于FFMpeg的中间件需要将其封装到某个接口中，如：

```cpp
LONG GetDuratioin(IntfX*);
LONG GetStartTime(IntfX*);
LONG GetFileSize(IntfX*);
LONG GetBitRate(IntfX*);
```

APIs

avformat_open_input

int avformat_open_input(AVFormatContext **ic_ptr, const char *filename, AVInputFormat *fmt, AVDictionary **options);

avformat_open_input完成两个任务：

1. 打开一个文件或URL，基于字节流的底层输入模块得到初始化。
2. 解析多媒体文件或多媒体流的头信息，创建AVFormatContext结构并填充其中的关键字段，依次为各个原始流建立AVStream结构。

一个多媒体文件或多媒体流与其包含的原始流的关系如下：

多媒体文件/多媒体流 (movie.mkv)
原始流 1 (h.264 video)
原始流 2 (aac audio for Chinese)
原始流 3 (aac audio for english)
原始流 4 (Chinese Subtitle)
原始流 5 (English Subtitle)
...

关于输入参数：

ic_ptr，这是一个指向指针的指针，用于返回avformat_open_input内部构造的一个AVFormatContext结构体。
filename，指定文件名。
fmt，用于显式指定输入文件的格式，如果设为空则自动判断其输入格式。
options

这个函数通过解析多媒体文件或流的头信息及其他辅助数据，能够获取足够多的关于文件、流和编解码器的信息，但由于任何一种多媒体格式提供的信息都是有限的，而且不同的多媒体内容制作软件对头信息的设置不尽相同，此外这些软件在产生多媒体内容时难免会引入一些错误，因此这个函数并不保证能够获取所有需要的信息，在这种情况下，则需要考虑另一个函数：avformat_find_stream_info。

avformat_find_stream_info

```
int avformat_find_stream_info(AVFormatContext *ic, AVDictionary **options);
```

这个函数主要用于获取必要的编解码器参数，设置到ic→streams[i]→codec中。

首先必须得到各媒体流对应编解码器的类型和id，这是两个定义在avutils.h和avcodec.h中的枚举：

```cpp
enum AVMediaType {
    AVMEDIA_TYPE_UNKNOWN = -1,
    AVMEDIA_TYPE_VIDEO,
    AVMEDIA_TYPE_AUDIO,
    AVMEDIA_TYPE_DATA,
    AVMEDIA_TYPE_SUBTITLE,
    AVMEDIA_TYPE_ATTACHMENT,
    AVMEDIA_TYPE_NB
};
enum CodecID {
    CODEC_ID_NONE,

    /* video codecs */
    CODEC_ID_MPEG1VIDEO,
    CODEC_ID_MPEG2VIDEO, ///< preferred ID for MPEG-1/2 video decoding
    CODEC_ID_MPEG2VIDEO_XVMC,
    CODEC_ID_H261,
    CODEC_ID_H263,
    ...
};
```

通常，如果某种媒体格式具备完备而正确的头信息，调用avformat_open_input即可以得到这两个参数，但若是因某种原因avformat_open_input无法获取它们，这一任务将由avformat_find_stream_info完成。

其次还要获取各媒体流对应编解码器的时间基准。

此外，对于音频编解码器，还需要得到：

1. 采样率，
2. 声道数，
3. 位宽，
4. 帧长度（对于某些编解码器是必要的），

对于视频编解码器，则是：

1. 图像大小，
2. 色彩空间及格式，

av_read_frame

```
int av_read_frame(AVFormatContext *s, AVPacket *pkt);
```

这个函数用于从多媒体文件或多媒体流中读取媒体数据，获取的数据由AVPacket结构pkt来存放。对于音频数据，如果是固定比特率，则pkt中装载着一个或多个音频帧；如果是可变比特率，则pkt中装载有一个音频帧。对于视频数据，pkt中装载有一个视频帧。需要注意的是：再次调用本函数之前，必须使用av_free_packet释放pkt所占用的资源。

通过pkt→stream_index可以查到获取的媒体数据的类型，从而将数据送交相应的解码器进行后续处理。

av_seek_frame

```
int av_seek_frame(AVFormatContext *s, int stream_index, int64_t timestamp, int flags);
```

这个函数通过改变媒体文件的读写指针来实现对媒体文件的随机访问，支持以下三种方式：

基于时间的随机访问：具体而言就是将媒体文件读写指针定位到某个给定的时间点上，则之后调用av_read_frame时能够读到时间标签等于给定时间点的媒体数据，通常用于实现媒体播放器的快进、快退等功能。
基于文件偏移的随机访问：相当于普通文件的seek函数，timestamp也成为文件的偏移量。
基于帧号的随机访问：timestamp为要访问的媒体数据的帧号。

关于参数：

s：是个AVFormatContext指针，就是avformat_open_input返回的那个结构。
stream_index：指定媒体流，如果是基于时间的随机访问，则第三个参数timestamp将以此媒体流的时间基准为单位；如果设为负数，则相当于不指定具体的媒体流，FFMPEG会按照特定的算法寻找缺省的媒体流，此时，timestamp的单位为AV_TIME_BASE（微秒）。
timestamp：时间标签，单位取决于其他参数。
flags：定位方式，AVSEEK_FLAG_BYTE表示基于字节偏移，AVSEEK_FLAG_FRAME表示基于帧号，其它表示基于时间。

av_close_input_file

```
void av_close_input_file(AVFormatContext *s);
```

关闭一个媒体文件：释放资源，关闭物理IO。

avcodec_find_decoder

```
AVCodec *avcodec_find_decoder(enum CodecID id);
AVCodec *avcodec_find_decoder_by_name(const char *name);
```

根据给定的codec id或解码器名称从系统中搜寻并返回一个AVCodec结构的指针。

avcodec_open

```
int avcodec_open(AVCodecContext *avctx, AVCodec *codec);
```

此函数根据输入的AVCodec指针具体化AVCodecContext结构。在调用该函数之前，需要首先调用avcodec_alloc_context分配一个AVCodecContext结构，或调用avformat_open_input获取媒体文件中对应媒体流的AVCodecContext结构；此外还需要通过avcodec_find_decoder获取AVCodec结构。

这一函数还将初始化对应的解码器。

avcodec_decode_video2

解码一个视频帧。got_picture_ptr指示是否有解码数据输出。

输入数据在AVPacket结构中，输出数据在AVFrame结构中。AVFrame是定义在avcodec.h中的一个数据结构：

```
typedef struct AVFrame {
    FF_COMMON_FRAME
} AVFrame;
```

FF_COMMON_FRAME定义了诸多数据域，大部分由FFMpeg内部使用，对于用户来说，比较重要的主要包括：

```
#define FF_COMMON_FRAME ......
    uint8_t *data[4];    int linesize[4];    int key_frame;    int pict_type;    int64_t pts;\
    int reference;\
......
```

FFMpeg内部以planar的方式存储原始图像数据，即将图像像素分为多个平面（R/G/B或Y/U/V），data数组内的指针分别指向四个像素平面的起始位置，linesize数组则存放各个存贮各个平面的缓冲区的行宽：

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++data[0]->###################################++++++++++++
++++++++++++###########picture data##########++++++++++++
++++++++++++###################################++++++++++++
++++++++++++###################################++++++++++++
             ......................
++++++++++++###################################++++++++++++
|<------------------line_size[0]---------------------->|
```

此外，key_frame标识该图像是否是关键帧；pict_type表示该图像的编码类型：I(1)/P(2)/B(3)……；pts是以time_base为单位的时间标签，对于部分解码器如H.261、H.263和MPEG4，可以从头信息中获取；reference表示该图像是否被用作参考。

avcodec_decode_audio4

```
int avcodec_decode_audio4(AVCodecContext *avctx, AVFrame *frame, int *got_frame_ptr, AVPacket *avpkt);
```

解码一个音频帧。输入数据在AVPacket结构中，输出数据在frame中，got_frame_ptr表示是否有数据输出。

avcodec_close

```
int avcodec_close(AVCodecContext *avctx);
```

关闭解码器，释放avcodec_open中分配的资源。

上一篇：FFmpeg源码分析之数据流

下一篇：gstreamer launch config and open log macro