ffplay播放器-5-6音视频解码线程

《FFmpeg/WebRTC/RTMP音视频流媒体高级开发教程》 – Darren老师：QQ326873713
课程链接：https://ke.qq.com/course/468797?tuin=137bb271

5 6解码线程

ffplay的解码线程独立于数据读线程，并且每种类型的流(AVStream)都有其各自的解码线程，如：

- video_thread用于解码video stream；
- audio_thread用于解码audio stream；
- subtitle_thread用于解码subtitle stream。

为方便阅读，先列一张表格，梳理各个变量、函数名称。

类型	PacketQueue	FrameQueue	vidck	解码线程
视频	videoq	pictq	vidcllk	video_thread
音频	audioq	sampq	audclk	audio_thread
字幕	subtitleq	subpq	无	subtitle_thread

其中PacketQueue用于存放从read_thread取到的各自播放时间内的AVPacket。FrameQueue用于存放各自解码后的AVFrame。Clock用于同步音视频。解码线程负责将PacketQueue数据解码为AVFrame，并存入FrameQueue。
对于不同流，其解码过程大同小异。

```
/**
* 解码器封装
*/
typedef struct Decoder {
  AVPacket pkt;
  PacketQueue *queue;         // 数据包队列
  AVCodecContext *avctx;      // 解码器上下文
  int  pkt_serial;          // 包序列
  int  finished;            // =0，解码器处于工作状态；=非0，解码器处于空闲状态
  int  packet_pending;      // =0，解码器处于异常状态，需要考虑重置解码器；=1，解码器处于正常状态
  SDL_cond *empty_queue_cond;  // 检查到packet队列空时发送 signal缓存read_thread读取数据
  int64_t  start_pts;         // 初始化时是stream的start time
  AVRational start_pts_tb;       // 初始化时是stream的time_base
  int64_t  next_pts;          // 记录最近一次解码后的frame的pts，当解出来的部分帧没有有效的pts时则使用next_pts进行推算
  AVRational next_pts_tb;        // next_pts的单位
  SDL_Thread *decoder_tid;       // 线程句柄
} Decoder;
```

解码器相关的函数（decoder我们ffplay自定义，重新封装的。 avcodec才是ffmpeg的提供的）

- 初始化解码器

```
void decoder_init(Decoder *d, AVCodecContext *avctx, PacketQueue *queue,
                  SDL_cond *empty_queue_cond);
```

- 启动解码器

```
int decoder_start(Decoder *d, int (*fn)(void *), const char *thread_name, void* arg)
```

- 解帧

```
int decoder_decode_frame(Decoder *d, AVFrame *frame, AVSubtitle *sub);
```

- 终止解码器

```
void decoder_abort(Decoder *d, FrameQueue *fq);
```

- 销毁解码器

```
void decoder_destroy(Decoder *d);
```

使用方法

- 启动解码线程
  - decoder_init()
  - decoder_start()
- 解码线程具体流程

    - decoder_decode_frame()
- 退出解码线程
    - decoder_abort()
    - decoder_destroy()

5 视频解码线程

数据来源：从read_thread线程而来
数据处理：在video_thread进行解码，具体调用get_video_frame
数据出口：在video_refresh读取frame进行显示

video_thread()

我们先看video_thead，对于滤镜部分（CONFIG_AVFILTER定义部分），这里不做分析，简化后的代码如下：

```
// 视频解码线程
static int video_thread(void *arg)
{
    VideoState *is = arg;
    AVFrame *frame = av_frame_alloc();  // 分配解码帧
    double pts;                 // pts
    double duration;            // 帧持续时间
    int ret;
    // 1 获取stream timebase
    AVRational tb = is->video_st->time_base; // 获取stream timebase
    // 2 获取帧率，以便计算每帧picture的duration
    AVRational frame_rate = av_guess_frame_rate(is->ic, is->video_st
, NULL);

    if (!frame)
        return AVERROR(ENOMEM);

    for (;;) {  // 循环取出视频解码的帧数据
        // 3 解码获取一帧视频画面
        ret = get_video_frame(is, frame);
        if (ret < 0)
            goto the_end;   //解码结束，什么时候会结束
        if (!ret)           //没有解码得到画面，什么情况下会得不到解后的帧
            continue;
```

```
        // 4 计算帧持续时间和换算pts值为秒
        // 1/帧率 = duration 单位秒, 没有帧率时则设置为0, 有帧率帧计算出帧间隔
        duration = (frame_rate.num && frame_rate.den ? av_q2d((AVRational){frame_rate.den,
                                                               frame_rate.num}) : 0);
        // 根据AVStream timebase计算出pts值, 单位为秒
        pts = (frame->pts == AV_NOPTS_VALUE) ? NAN : frame->pts * av_q2d(tb);
        // 5 将解码后的视频帧插入队列
        ret = queue_picture(is, frame, pts, duration, frame->pkt_pos, is->viddec.pkt_serial);
        // 6 释放frame对应的数据
        av_frame_unref(frame); // 正常情况下frame对应的buf以被av_frame_move_ref

        if (ret < 0) // 返回值小于0则退出线程
            goto the_end;
    }
 the_end:
    av_frame_free(&frame);  // 释放frame
    return 0;
}
```

在该流程中，当调用函数返回值小于<0时则退出线程。

线程的总体流程很清晰：

1. 获取stream timebase，以便将frame的pts转成秒为单位
2. 获取帧率，以便计算每帧picture的duration
3. 获取解码后的视频帧，具体调用get_video_frame()实现
4. 计算帧持续时间和换算pts值为秒
5. 将解码后的视频帧插入队列，具体调用queue_picture()实现
6. 释放frame对应的数据

我们重点讲解get_video_frame()和queue_picture()

get_video_frame()

`get_video_frame`简化如下：

```
static int get_video_frame(VideoState *is, AVFrame *frame)
{
```

```
    int got_picture;
    // 1. 获取解码后的视频帧
    if ((got_picture = decoder_decode_frame(&is->viddec, frame, NULL
)) < 0) {
        return -1; // 返回-1意味着要退出解码线程，所以要分析decoder_decode_
frame什么情况下返回-1
    }

    if (got_picture) {
        // 2. 分析获取到的该帧是否要drop掉
       .....
    }

    return got_picture;
}
```

主要流程：

1. 调用`decoder_decode_frame`解码并获取解码后的视频帧；
2. 分析如果获取到帧是否需要drop掉（逻辑就是如果刚解出来就落后主时钟，那就没有必要放入Frame队列，再拿去播放，但是也是有一定的条件的，见下面分析）

被简化的部分主要是针对丢帧的一个处理：

```
if (got_picture) {
    // 2. 分析获取到的该帧是否要drop掉，该机制的目的是在放入帧队列前先drop掉过时
的视频帧
    double dpts = NAN;

    if (frame->pts != AV_NOPTS_VALUE)
        dpts = av_q2d(is->video_st->time_base) * frame->pts;    //计
算出秒为单位的pts

    frame->sample_aspect_ratio = av_guess_sample_aspect_ratio(is->ic
, is->video_st, frame);

    if (framedrop>0 || // 允许drop帧
        (framedrop && get_master_sync_type(is) != AV_SYNC_VIDEO_MAST
ER))//非视频同步模式
    {
        if (frame->pts != AV_NOPTS_VALUE) { // pts值有效
```

```
14              double diff = dpts - get_master_clock(is);
15              if (!isnan(diff) &&     // 差值有效
16                  fabs(diff) < AV_NOSYNC_THRESHOLD && // 差值在可同步范围
    呢
17                  diff - is->frame_last_filter_delay < 0 && // 和过滤器
    有关系
18                  is->viddec.pkt_serial == is->vidclk.serial && // 同一
    序列的包
19                  is->videoq.nb_packets) { // packet队列至少有1帧数据
20                  is->frame_drops_early++;
21                  printf("%s(%d) diff:%lfs, drop frame, drops:%d\n",
22                         __FUNCTION__, __LINE__, diff, is->frame_drops
    _early);
23                  av_frame_unref(frame);
24                  got_picture = 0;
25              }
26          }
27      }
28  }
```

先确定进入丢帧检测流程，控制是否进入丢帧检测有3种情况

1. 控制是否丢帧的开关变量是`framedrop`，为1，则始终判断是否丢帧；
2. `framedrop`为0，则始终不丢帧；
3. `framedrop`为-1（默认值），则在主时钟不是video的时候，判断是否丢帧。

如果进入丢帧检测流程，drop帧需要下列因素都成立：

1. !isnan(diff)：当前pts和主时钟的差值是有效值；
2. fabs(diff) < AV_NOSYNC_THRESHOLD：差值在可同步范围内，这里设置的是10秒，意思是如果差值太大这里就不管了了，可能流本身录制的时候就有问题，这里不能随便把帧都drop掉；
3. diff - is->frame_last_filter_delay < 0：和过滤器有关系，不设置过滤器时简化为 diff < 0;
4. is->viddec.pkt_serial == is->vidclk.serial：解码器的serial和时钟的serial相同，即是至少显示了一帧图像，因为只有显示的时候才调用update_video_pts()设置到video clk的serial；
5. is->videoq.nb_packets：至少packetqueue有1个包。

接下来看下真正解码的过程——`decoder_decode_frame`，这个函数也包含了对audio和subtitle的解码，其返回值：

- -1：请求退出解码器线程
- 0：解码器已经完全冲刷，没有帧可读，这里也说明对应码流播放结束
- 1：正常解码获取到帧

先看简化后的主干代码（注意for(;;)这个大循环）：

```
static int decoder_decode_frame(Decoder *d, AVFrame *frame, AVSubtitle *sub) {
    for (;;) { // 大循环
        //1. 流连续情况下获取解码后的帧
        if (d->queue->serial == d->pkt_serial) {
            do {
                if (d->queue->abort_request)
                    return -1;  // 是否请求退出
                ret = avcodec_receive_frame(d->avctx, frame);
                if (ret == AVERROR_EOF) {
                    return 0;   // 解码器已完全冲刷，没有帧可读了
                }
                if (ret >= 0)
                    return 1;   // 读取到解码帧
            } while (ret != AVERROR(EAGAIN));
        }
        //2. 获取一个packet，如果播放序列不一致(数据不连续)则过滤掉“过时”的packet
        do {
            if (d->queue->nb_packets == 0)//如果没有数据可读则唤醒read_thread
                SDL_CondSignal(d->empty_queue_cond);
            if (packet_queue_get(d->queue, &pkt, 1, &d->pkt_serial) < 0)// 阻塞方式读packet
                return -1;
        } while (d->queue->serial != d->pkt_serial); // 播放序列的判断
        //3. 将packet送入解码器
        avcodec_send_packet(d->avctx, &pkt);
    }
}
```

`decoder_decode_frame`的主干代码是一个循环，要拿到一帧解码数据，或解码出错、文件结束，才会返回。

循环内可以分解为3个步骤：

1. **同一播放序列流连续的情况下，不断调用avcodec_receive_frame获取解码后的frame。**
    a. `d->queue`就是video PacketQueue(videoq)
    b. `d->pkt_serial`是最近一次取的packet的序列号。在判断完`d->queue->serial == d->pkt_serial`确保流连续后，循环调用`avcodec_receive_frame`，有取到帧就返回。（即

使还没送入新的Packet，这是为了兼容一个Packet可以解出多个Frame的情况）

2. **获取一个packet，如果播放序列不一致(数据不连续)则过滤掉“过时”的packet。**主要阻塞调用`packet_queue_get`（）。另外，会在PacketQueue为空时，发送`empty_queue_cond`条件信号，通知读线程继续读数据。（`empty_queue_cond`就是`continue_read_thread`，可以参考read线程的分析，查看读线程何时会等待该条件量。
3. **将packet送入解码器。**

1 同一播放序列流连续的情况下，不断调用avcodec_receive_frame获取解码后的frame。

我们先看avcodec_receive_frame的具体流程，这里先省略Audio的case：

```
// 1. 流连续情况下获取解码后的帧
if (d->queue->serial == d->pkt_serial) { // 1.1 先判断是否是同一播放序列的数据
    do {
        if (d->queue->abort_request)
            return -1;  // 是否请求退出
        // 1.2. 获取解码帧
        switch (d->avctx->codec_type) {
            case AVMEDIA_TYPE_VIDEO:
                ret = avcodec_receive_frame(d->avctx, frame);
                //printf("frame pts:%ld, dts:%ld\n", frame->pts, frame->pkt_dts);
                if (ret >= 0) {
                    if (decoder_reorder_pts == -1) {
                        frame->pts = frame->best_effort_timestamp;
                    } else if (!decoder_reorder_pts) {
                        frame->pts = frame->pkt_dts;
                    }
                }
                break;
            case AVMEDIA_TYPE_AUDIO:
                ret = avcodec_receive_frame(d->avctx, frame);
                ....
                break;
        }

        // 1.3. 检查解码是否已经结束，解码结束返回0
```

```
        if (ret == AVERROR_EOF) {
            d->finished = d->pkt_serial;
            printf("avcodec_flush_buffers %s(%d)\n", __FUNCTION__, __LINE__);
            avcodec_flush_buffers(d->avctx); // 调用该函数后可以再次解码，只要有数据packet进入
            return 0;
        }
        // 1.4. 正常解码返回1
        if (ret >= 0)
            return 1;
    } while (ret != AVERROR(EAGAIN));   // 1.5 没帧可读时ret返回EAGIN，需要继续送packet
}
```

注意返回值：

- -1：请求退出解码器线程
- 0：解码器已经完全冲刷，没有帧可读，这里也说明对应码流播放结束
- 1：正常解码获取到帧

这里重点分析

（1）decoder_reorder_pts

```
ret = avcodec_receive_frame(d->avctx, frame);

if (ret >= 0) {
    if (decoder_reorder_pts == -1) {
        frame->pts = frame->best_effort_timestamp;
    } else if (!decoder_reorder_pts) {
        frame->pts = frame->pkt_dts;
    }
}
```

decoder_reorder_pts：让ffmpeg排序pts 0=off 1=on -1=auto，默认为-1 （ffplay配置 -drp value 进行设置）

- 0：frame的pts使用pkt_dts，这种情况基本不会出现
- 1：frame保留自己的pts

- -1：frame的pts使用frame->best_effort_timestamp，best_effort_timestamp是经过算法计算出来的值，主要是“尝试为可能有错误的时间戳猜测出适当单调的时间戳”，大部分情况下还是frame->pts，或者就是frame->pkt_dts。

（2）avcodec_flush_buffers

使用“空包”冲刷解码器后，如果要再次解码则需要调用avcodec_flush_buffers()，之所以在这个节点调用avcodec_flush_buffers()，主要是让我们在循环播放码流的时候可以继续正常解码。

2 获取一个packet，如果播放序列不一致(数据不连续)则过滤掉“过时”的packet

```
// 2 获取一个packet，如果播放序列不一致(数据不连续)则过滤掉“过时”的packet
do {
    // 2.1 如果没有数据可读则唤醒read_thread，实际是continue_read_thread SDL_cond
    if (d->queue->nb_packets == 0)  // 没有数据可读
        SDL_CondSignal(d->empty_queue_cond);// 通知read_thread放入packet
    // 2.2 如果还有pending的packet则使用它
    if (d->packet_pending) {
        av_packet_move_ref(&pkt, &d->pkt);
        d->packet_pending = 0;
    } else {
        // 2.3 阻塞式读取packet，这里好理解，就是读packet并获取serial
        if (packet_queue_get(d->queue, &pkt, 1, &d->pkt_serial) < 0)
            return -1;
    }
    if(d->queue->serial != d->pkt_serial) {
        // darren自己的代码
        printf("%s(%d) discontinue:queue->serial:%d,pkt_serial:%d\n",
               __FUNCTION__, __LINE__, d->queue->serial, d->pkt_serial);
        av_packet_unref(&pkt); // fixed me? 释放要过滤的packet
    }
} while (d->queue->serial != d->pkt_serial);// 如果不是同一播放序列(流不
```

```
    连续)则继续读取
```

重点：

（1）如果还有pending的packet则使用它

```
// 2.2 如果还有pending的packet则使用它
if (d->packet_pending) {
    av_packet_move_ref(&pkt, &d->pkt);
    d->packet_pending = 0;
}
```

pending包packet和`packet_pending`的概念的来源，来自send失败时重新发送：

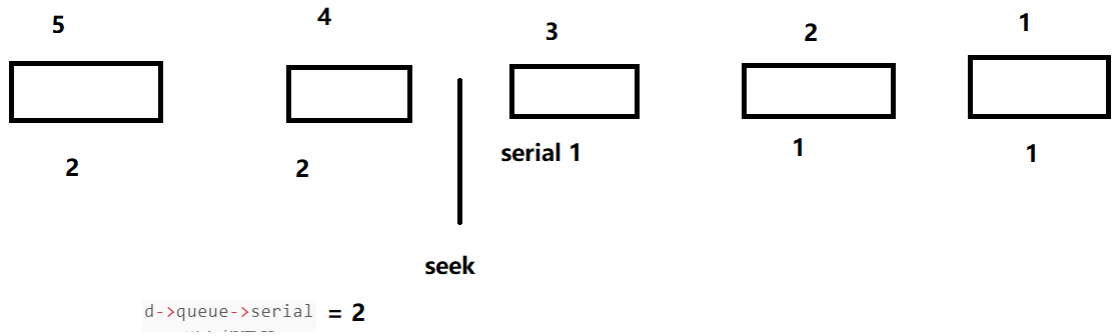
```
if (avcodec_send_packet(d->avctx, &pkt) == AVERROR(EAGAIN)) {
    av_log(d->avctx, AV_LOG_ERROR, "Receive_frame and send_packet both returned EAGAIN, which is an API violation.\n");
    d->packet_pending = 1;
    av_packet_move_ref(&d->pkt, &pkt);
}
```

如果`avcodec_send_packet`返回`EAGAIN`，则把当前`pkt`存入`d->pkt`，然后置标志位`packet_pending`为1。

（2）do {} while (d->queue->serial != d->pkt_serial);// 如果不是同一播放序列(流不连续)则继续读取

- d->queue->serial是最新的播放序列，当读取出来的packet的serial和最新的serial不同时则过滤掉，继续读取packet，但检测到不是同一serial，是不是应该释放掉packet的数据？比如下列代码：

```
if(d->queue->serial != d->pkt_serial) {
    // darren自己的代码
    printf("%s(%d) discontinue:queue->serial:%d,pkt_serial:%d\n",
           __FUNCTION__, __LINE__, d->queue->serial, d->pkt_serial);
    av_packet_unref(&pkt); // fixed me? 释放要过滤的packet
}
```



3 将packet送入解码器

```
// 3 将packet送入解码器
if (pkt.data == flush_pkt.data) {//
    // when seeking or when switching to a different stream
    avcodec_flush_buffers(d->avctx); //清空里面的缓存帧
    d->finished = 0;        // 重置为0
    d->next_pts = d->start_pts;     // 主要用在了audio
    d->next_pts_tb = d->start_pts_tb;// 主要用在了audio
} else {
    if (d->avctx->codec_type == AVMEDIA_TYPE_SUBTITLE) {
        int got_frame = 0;
        ret = avcodec_decode_subtitle2(d->avctx, sub, &got_frame, &p
kt);
        if (ret < 0) {
            ret = AVERROR(EAGAIN);
        } else {
            if (got_frame && !pkt.data) {
                d->packet_pending = 1;
                av_packet_move_ref(&d->pkt, &pkt);
            }
            ret = got_frame ? 0 : (pkt.data ? AVERROR(EAGAIN) : AVER
ROR_EOF);
        }
    } else {
        if (avcodec_send_packet(d->avctx, &pkt) == AVERROR(EAGAIN))
{
            av_log(d->avctx, AV_LOG_ERROR, "Receive_frame and send_p
acket both returned EAGAIN, which is an API violation.\n");
```

```
            d->packet_pending = 1;
            av_packet_move_ref(&d->pkt, &pkt);
        }
    }
    av_packet_unref(&pkt);  // 一定要自己去释放音视频 字幕数据
}
```

重点：

（1）有针对`flush_pkt`的处理

```
if (pkt.data == flush_pkt.data) {//
    // when seeking or when switching to a different stream
    avcodec_flush_buffers(d->avctx); //清空里面的缓存帧
    d->finished = 0;        // 重置为0
    d->next_pts = d->start_pts;     // 主要用在了audio
    d->next_pts_tb = d->start_pts_tb;// 主要用在了audio
}
```

了解过PacketQueue的代码，我们知道在往PacketQueue送入一个flush_pkt后，PacketQueue的serial值会加1，而送入的flush_pkt和PacketQueue的**新serial值保持一致**。所以如果有“过时（旧serial）”Packet，过滤后，取到新的播放序列第一个pkt将是flush_pkt。
根据api要求，此时需要调用`avcodec_flush_buffers`。

也要注意d->finished = 0; 的重置。

（2）avcodec_send_packet后出现AVERROR(EAGAIN)，则说明我们要继续调用avcodec_receive_frame()将frame读取，再调用avcodec_send_packet发packet。由于出现AVERROR(EAGAIN)返回值解码器内部没有接收传入的packet，但又没法放回PacketQueue，所以我们就缓存到了自封装的Decoder的pkt（即是d->pkt），并将 d->packet_pending = 1，以备下次继续使用该packet

```
if (avcodec_send_packet(d->avctx, &pkt) == AVERROR(EAGAIN)) {
    av_log(d->avctx, AV_LOG_ERROR, "Receive_frame and send_packet both returned EAGAIN, which is an API violation.\n");
    d->packet_pending = 1;
    av_packet_move_ref(&d->pkt, &pkt);
}
```

queue_picture()

上面，我们就分析完video_thread中关键的`get_video_frame`函数，根据所分析的代码，已经可以取到正确解码后的一帧数据。接下来就要把这一帧放入FrameQueue：

```
// 4 计算帧持续时间和换算pts值为秒
// 1/帧率 = duration 单位秒, 没有帧率时则设置为0, 有帧率帧计算出帧间隔
duration = (frame_rate.num && frame_rate.den ? av_q2d((AVRational){frame_rate.den, frame_rate.num}) : 0);
// 根据AVStream timebase计算出pts值, 单位为秒
pts = (frame->pts == AV_NOPTS_VALUE) ? NAN : frame->pts * av_q2d(tb);
// 5 将解码后的视频帧插入队列
ret = queue_picture(is, frame, pts, duration, frame->pkt_pos, is->viddec.pkt_serial);
// 6 释放frame对应的数据
av_frame_unref(frame);
```

主要调用`queue_picture`：

```
static int queue_picture(VideoState *is, AVFrame *src_frame, double pts,
                         double duration, int64_t pos, int serial)
{
    Frame *vp;

    if (!(vp = frame_queue_peek_writable(&is->pictq))) // 检测队列是否有可写空间
        return -1;      // Frame队列满了则返回-1
    // 执行到这步说已经获取到了可写入的Frame
    vp->sar = src_frame->sample_aspect_ratio;
    vp->uploaded = 0;

    vp->width = src_frame->width;
    vp->height = src_frame->height;
    vp->format = src_frame->format;

    vp->pts = pts;
    vp->duration = duration;
    vp->pos = pos;
```

```
    vp->serial = serial;

    set_default_window_size(vp->width, vp->height, vp->sar);

    av_frame_move_ref(vp->frame, src_frame); // 将src中所有数据拷贝到dst中，并复位src。
    frame_queue_push(&is->pictq);    // 更新写索引位置
    return 0;
}
```

`queue_picture`的代码很直观：

- 首先`frame_queue_peek_writable`取FrameQueue的当前写节点；
- 然后把该拷贝的拷贝给节点(struct Frame)保存
- 再`frame_queue_push`，“push”节点到队列中。唯一需要关注的是，AVFrame的拷贝是通过`av_frame_move_ref`实现的，所以拷贝后`src_frame`就是无效的了。

作业思考

思考下面流程

- flush_pkt的作用
- Decoder的packet_pending和pkt的作用
- 解码流程：avcodec_receive_frame-> packet_queue_get-> avcodec_send_packet

6 音频解码线程

数据来源：从read_thread()线程而来
数据处理：在audio_thread()进行解码，具体调用decoder_decode_frame()
数据出口：在sdl_audio_callback()->audio_decode_frame()读取frame进行播放

audio_thread()

我们先看audio_thraed()，对于滤镜部分（CONFIG_AVFILTER定义部分），这里不做分析，简化后的代码如下：

```
// 音频解码线程
static int audio_thread(void *arg)
{
    VideoState *is = arg;
    AVFrame *frame = av_frame_alloc();  // 分配解码帧
```

```
    Frame *af;
    int got_frame = 0;  // 是否读取到帧
    AVRational tb;      // timebase
    int ret = 0;

    if (!frame)
        return AVERROR(ENOMEM);

    do {
        // 1. 读取解码帧
        if ((got_frame = decoder_decode_frame(&is->auddec, frame, NULL)) < 0)
            goto the_end;

        if (got_frame) {
                tb = (AVRational){1, frame->sample_rate};   // 设置为sample_rate为timebase
                // 2. 获取可写Frame
                if (!(af = frame_queue_peek_writable(&is->sampq))) // 获取可写帧
                    goto the_end;
                // 3. 设置Frame并放入FrameQueue
                af->pts = (frame->pts == AV_NOPTS_VALUE) ? NAN : frame->pts * av_q2d(tb);
                af->pos = frame->pkt_pos;
                af->serial = is->auddec.pkt_serial;
                af->duration = av_q2d((AVRational){frame->nb_samples, frame->sample_rate});

                av_frame_move_ref(af->frame, frame); //转移
                frame_queue_push(&is->sampq); // 更新写索引
        }
    } while (ret >= 0 || ret == AVERROR(EAGAIN) || ret == AVERROR_EOF);

the_end:
    av_frame_free(&frame);
    return ret;
}
```

从简化后的代码来看，逻辑和video_thread()基本是类似的且更简单，这里主要重点讲解

```
tb = (AVRational){1, frame->sample_rate};   // 设置为sample_rate为timebase
```

为什么video_thread()是tb是采用了stream->base_base，这里却不是，这个时候就要回到decoder_decode_frame()函数，我们主要是重点看audio部分，其余都已经在《视频解码线程》节讲解过

```
static int decoder_decode_frame(Decoder *d, AVFrame *frame, AVSubtitle *sub) {
    ...
    for (;;) {
        AVPacket pkt;
        // 1. 流连续情况下获取解码后的帧
        if (d->queue->serial == d->pkt_serial) { // 1.1 先判断是否是同一播放序列的数据
            do {
                .........
                switch (d->avctx->codec_type) {
                case AVMEDIA_TYPE_VIDEO:
                    ....
                    break;
                case AVMEDIA_TYPE_AUDIO:
                    ret = avcodec_receive_frame(d->avctx, frame);
                    if (ret >= 0) {
                        AVRational tb = (AVRational){1, frame->sample_rate};    //
                        if (frame->pts != AV_NOPTS_VALUE) {
                            // 如果frame->pts正常则先将其从pkt_timebase转成{1, frame->sample_rate}
                            // pkt_timebase实质就是stream->time_base
                            frame->pts = av_rescale_q(frame->pts, d->avctx->pkt_timebase, tb);
                        }
                        else if (d->next_pts != AV_NOPTS_VALUE) {
                            // 如果frame->pts不正常则使用上一帧更新的next_pts和next_pts_tb
                            // 转成{1, frame->sample_rate}
                            frame->pts = av_rescale_q(d->next_pts, d->next_pts_tb, tb);
                        }
                        if (frame->pts != AV_NOPTS_VALUE) {
```

```
28                              // 根据当前帧的pts和nb_samples预估下一帧的pts
29                              d->next_pts = frame->pts + frame->nb_sam
   ples;
30                              d->next_pts_tb = tb; // 设置timebase
31                          }
32                      }
33                      break;
34                  }
35
36                  ....
37              } while (ret != AVERROR(EAGAIN));   // 1.5 没帧可读时ret返
   回EAGIN，需要继续送packet
38          }
39          ....
40          }
```

从上可以看出来，将audio frame从decoder_decode_frame取出来后，已由stream->time_base转成了{1, frame->sample_rate}作为time_base。

音频解码线程主要是讲解了和视频解码线程差异化部分，其他共同部分参考视频解码线程的讲解。