ffplay播放器-7-8 音频输出和音频重采样

《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

7 音频输出模块

ffplay的音频输出通过SDL实现。
音频输出的主要流程：

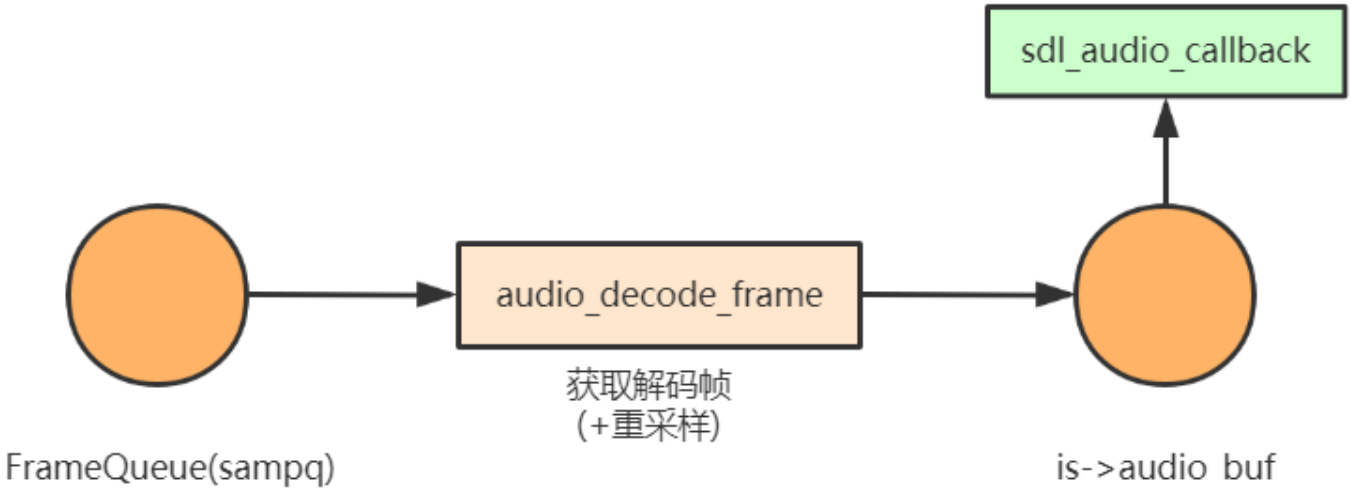
- 打开SDL音频设备，设置参数
- 启动SDL音频设备播放
- SDL音频回调函数读取数据，这个时候我们就要从FrameQueue读取frame填充回调函数提供的buffer空间。

audio的输出在SDL下是被动的，即在开启SDL音频后，当SDL需要数据输出时则通过回调函数的方式告诉应用者需要传入多少数据，但这里存在一些问题：

- ffmpeg解码一个AVPacket的音频到AVFrame后，**在AVFrame中存储的音频数据大小与SDL回调所需要的数据不一定相等 （回调函数每次要获取的数据量都是固定）**；
- 特别是如果要实现声音变速播放功能，那每帧AVFrame做变速后的数据大小大概率和SDL回调锁需要的数据大小不一致。

这就需要再增加一级缓冲区解决问题，即是从FrameQueue队列读取到Frame的数据后，先缓存到一个buffer里，然后再从该buffer读取数据给到SDL回调函数。

在audio输出时，主要模型如下图：



在这个模型中，sdl通过sdl_audio_callback函数向ffplay要音频数据，ffplay将sampq中的数据通过`audio_decode_frame`函数取出，放入`is->audio_buf`，然后送出给sdl。在后续回调时先找`audio_buf`要数据，数据不足的情况下，再调用`audio_decode_frame`补充`audio_buf`

> 注意`audio_decode_frame`这个函数名很具有迷惑性，实际上，这个函数是没有解码功能的！这个函数主要是处理sampq到audio_buf的过程，最多只是执行了重采样（数据源和输出参数不一致时则做重采样）。

打开SDL音频设备

SDL音频输出的参数是一开始就设置好的，当码流的解出来的音频参数和预设的输出参数不一致时，则需要重采样成
预设参数一致数据，这样才能正常播放。

音频设备的打开实际是在解复用线程中实现的。解复用线程中先打开音频设备(设定音频回调函数供SDL音频播放线程回调)，然后再创建音频解码线程。调用链如下：

```
1 main() -->
2 stream_open() -->
3 read_thread() -->
4 stream_component_open() -->
5     audio_open(is, channel_layout, nb_channels, sample_rate, &is->aud
  io_tgt);
```

先看打开sdl音频输出的代码（stream_component_open函数）：

```
//代码在stream_component_open
//先不看filter相关代码，即默认CONFIG_AVFILTER宏为0
//从avctx(即AVCodecContext)中获取音频格式参数
sample_rate    = avctx->sample_rate;
nb_channels    = avctx->channels;
channel_layout = avctx->channel_layout;
//调用audio_open打开sdl音频输出，实际打开的设备参数保存在audio_tgt，返回值表示输出设备的缓冲区大小
if ((ret = audio_open(is, channel_layout, nb_channels, sample_rate, &is->audio_tgt)) < 0)
    goto fail;
is->audio_hw_buf_size = ret;
is->audio_src = is->audio_tgt; //暂且将数据源参数等同于目标输出参数
//初始化audio_buf相关参数
is->audio_buf_size  = 0;
is->audio_buf_index = 0;
```

由于不同的音频输出设备支持的参数不同，音轨的参数不一定能被输出设备支持（此时就需要重采样了），`audio_tgt`就保存了输出设备参数。

> audio_open是ffplay封装的函数，会优先尝试请求参数能否打开输出设备，尝试失败后会自动查找最佳的参数重新尝试。不再具体分析。

`audio_src`一开始与`audio_tgt`是一样的，如果输出设备支持音轨参数，那么`audio_src`可以一直保持与`audio_tgt`一致，否则将在后面代码中自动修正为音轨参数，并引入重采样机制。

最后初始化了几个audio_buf相关的参数。这里介绍下audio_buf相关的几个变量：

- audio_buf: 从要输出的AVFrame中取出的音频数据（PCM），如果有必要，则对该数据重采样。
- audio_buf_size: audio_buf的总大小
- audio_buf_index: 下一次可读的audio_buf的index位置。
- **audio_write_buf_size**：audio_buf剩余的buffer长度，即audio_buf_size – audio_buf_index

在`audio_open`函数内，通过通过`SDL_OpenAudioDevice`注册`sdl_audio_callback`函数为音频输出的回调函数。那么，主要的音频输出的逻辑就在`sdl_audio_callback`函数内了。

打开音频设备audio_open

audio_open()函数填入期望的音频参数，打开音频设备后，将实际的音频参数存入输出参数is->audio_tgt中，后面音频播放线程用会用到此参数，使用此参数将原始音频数据重采样，转换为音

频设备支持的格式。

```
static int audio_open(void *opaque, int64_t wanted_channel_layout, int wanted_nb_channels, int wanted_sample_rate, struct AudioParams *audio_hw_params)
{
    SDL_AudioSpec wanted_spec, spec;
    const char *env;
    static const int next_nb_channels[] = {0, 0, 1, 6, 2, 6, 4, 6};
    static const int next_sample_rates[] = {0, 44100, 48000, 96000, 192000};
    int next_sample_rate_idx = FF_ARRAY_ELEMS(next_sample_rates) - 1;

    env = SDL_getenv("SDL_AUDIO_CHANNELS");
    if (env) {  // 若环境变量有设置，优先从环境变量取得声道数和声道布局
        wanted_nb_channels = atoi(env);
        wanted_channel_layout = av_get_default_channel_layout(wanted_nb_channels);
    }
    if (!wanted_channel_layout || wanted_nb_channels != av_get_channel_layout_nb_channels(wanted_channel_layout)) {
        wanted_channel_layout = av_get_default_channel_layout(wanted_nb_channels);
        wanted_channel_layout &= ~AV_CH_LAYOUT_STEREO_DOWNMIX;
    }
    // 根据channel_layout获取nb_channels，当传入参数wanted_nb_channels不匹配时，此处会作修正
    wanted_nb_channels = av_get_channel_layout_nb_channels(wanted_channel_layout);
    wanted_spec.channels = wanted_nb_channels;
    wanted_spec.freq = wanted_sample_rate;
    if (wanted_spec.freq <= 0 || wanted_spec.channels <= 0) {
        av_log(NULL, AV_LOG_ERROR, "Invalid sample rate or channel count!\n");
        return -1;
    }
    while (next_sample_rate_idx && next_sample_rates[next_sample_rate_idx] >= wanted_spec.freq)
```

```
27          next_sample_rate_idx--;  // 从采样率数组中找到第一个不大于传入参数wanted_sample_rate的值
28      // 音频采样格式有两大类型：planar和packed，假设一个双声道音频文件，一个左声道采样点记作L，一个右声道采样点记作R，则：
29      // planar存储格式：(plane1)LLLLLLLL...LLLL (plane2)RRRRRRRR...RRRR
30      // packed存储格式：(plane1)LRLRLRLR.....................LRLR
31      // 在这两种采样类型下，又细分多种采样格式，如AV_SAMPLE_FMT_S16、AV_SAMPLE_FMT_S16P等，
32      // 注意SDL2.0目前不支持planar格式
33      // channel_layout是int64_t类型，表示音频声道布局，每bit代表一个特定的声道，参考channel_layout.h中的定义，一目了然
34      // 数据量(bits/秒) = 采样率(Hz) * 采样深度(bit) * 声道数
35      wanted_spec.format = AUDIO_S16SYS;
36      wanted_spec.silence = 0;
37      /*
38       * 一次读取多长的数据
39       * SDL_AUDIO_MAX_CALLBACKS_PER_SEC一秒最多回调次数，避免频繁的回调
40       *  Audio buffer size in samples (power of 2)
41       */
42      wanted_spec.samples = FFMAX(SDL_AUDIO_MIN_BUFFER_SIZE, 2 << av_log2(wanted_spec.freq / SDL_AUDIO_MAX_CALLBACKS_PER_SEC));
43      wanted_spec.callback = sdl_audio_callback;
44      wanted_spec.userdata = opaque;
45      // 打开音频设备并创建音频处理线程。期望的参数是wanted_spec，实际得到的硬件参数是spec
46      // 1) SDL提供两种使音频设备取得音频数据方法：
47      //    a. push，SDL以特定的频率调用回调函数，在回调函数中取得音频数据
48      //    b. pull，用户程序以特定的频率调用SDL_QueueAudio()，向音频设备提供数据。此种情况wanted_spec.callback=NULL
49      // 2) 音频设备打开后播放静音，不启动回调，调用SDL_PauseAudio(0)后启动回调，开始正常播放音频
50      // SDL_OpenAudioDevice()第一个参数为NULL时，等价于SDL_OpenAudio()
51      while (!(audio_dev = SDL_OpenAudioDevice(NULL, 0, &wanted_spec, &spec, SDL_AUDIO_ALLOW_FREQUENCY_CHANGE | SDL_AUDIO_ALLOW_CHANNELS_CHANGE))) {
52          av_log(NULL, AV_LOG_WARNING, "SDL_OpenAudio (%d channels, %d Hz): %s\n",
53                 wanted_spec.channels, wanted_spec.freq, SDL_GetError
```

```
());
54          wanted_spec.channels = next_nb_channels[FFMIN(7, wanted_spe
    c.channels)];
55          if (!wanted_spec.channels) {
56              wanted_spec.freq = next_sample_rates[next_sample_rate_i
    dx--];
57              wanted_spec.channels = wanted_nb_channels;
58              if (!wanted_spec.freq) {
59                  av_log(NULL, AV_LOG_ERROR,
60                         "No more combinations to try, audio open fai
    led\n");
61                  return -1;
62              }
63          }
64          wanted_channel_layout = av_get_default_channel_layout(wante
    d_spec.channels);
65      }
66      // 检查打开音频设备的实际参数：采样格式
67      if (spec.format != AUDIO_S16SYS) {
68          av_log(NULL, AV_LOG_ERROR,
69                 "SDL advised audio format %d is not supported!\n", s
    pec.format);
70          return -1;
71      }
72      // 检查打开音频设备的实际参数：声道数
73      if (spec.channels != wanted_spec.channels) {
74          wanted_channel_layout = av_get_default_channel_layout(spec.
    channels);
75          if (!wanted_channel_layout) {
76              av_log(NULL, AV_LOG_ERROR,
77                     "SDL advised channel count %d is not supported!
    \n", spec.channels);
78              return -1;
79          }
80      }
81      // wanted_spec是期望的参数，spec是实际的参数，wanted_spec和spec都是SDL
    中的结构。
82      // 此处audio_hw_params是FFmpeg中的参数，输出参数供上级函数使用
83      // audio_hw_params保存的参数，就是在做重采样的时候要转成的格式。
84      audio_hw_params->fmt = AV_SAMPLE_FMT_S16;
```

```
    audio_hw_params->freq = spec.freq;
    audio_hw_params->channel_layout = wanted_channel_layout;
    audio_hw_params->channels =  spec.channels;
    /* audio_hw_params->frame_size这里只是计算一个采样点占用的字节数 */
    audio_hw_params->frame_size = av_samples_get_buffer_size(NULL, audio_hw_params->channels,
                                                             1, audio_hw_params->fmt, 1);
    audio_hw_params->bytes_per_sec = av_samples_get_buffer_size(NULL, audio_hw_params->channels,
                                                                audio_hw_params->freq,
                                                                audio_hw_params->fmt, 1);
    if (audio_hw_params->bytes_per_sec <= 0 || audio_hw_params->frame_size <= 0) {
        av_log(NULL, AV_LOG_ERROR, "av_samples_get_buffer_size failed\n");
        return -1;
    }
    // 比如2帧数据，一帧就是1024个采样点， 1024*2*2 * 2 = 8096字节
    return spec.size;   /* 硬件内部缓存的数据字节, samples * channels *byte_per_sample */
}
```

回调函数逻辑`sdl_audio_callback`

再来看`sdl_audio_callback`

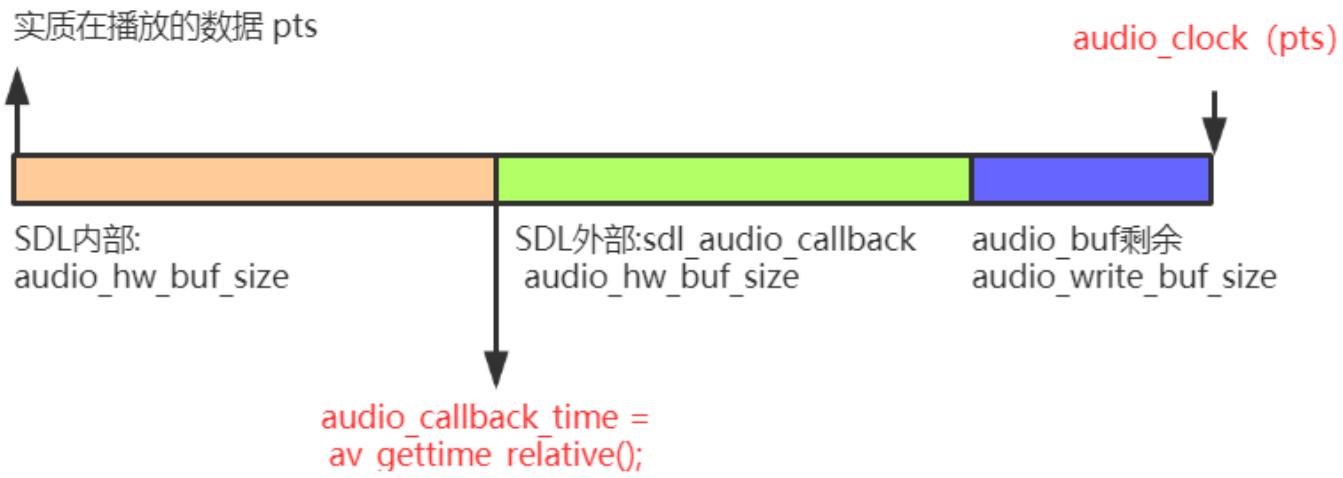
```
static void sdl_audio_callback(void *opaque, Uint8 *stream, int len)
{
    VideoState *is = opaque;
    int audio_size, len1;

    audio_callback_time = av_gettime_relative();

    while (len > 0) {   // 循环读取，直到读取到足够的数据
```

```
        /* (1)如果is->audio_buf_index < is->audio_buf_size则说明上次拷贝还剩余一些数据,
         * 先拷贝到stream再调用audio_decode_frame
         * (2)如果audio_buf消耗完了，则调用audio_decode_frame重新填充audio_buf
         */
        if (is->audio_buf_index >= is->audio_buf_size) {
            audio_size = audio_decode_frame(is);
            if (audio_size < 0) {
                /* if error, just output silence */
                is->audio_buf = NULL;
                is->audio_buf_size = SDL_AUDIO_MIN_BUFFER_SIZE / is->audio_tgt.frame_size
                        * is->audio_tgt.frame_size;
            } else {
                if (is->show_mode != SHOW_MODE_VIDEO)
                    update_sample_display(is, (int16_t *)is->audio_buf, audio_size);
                is->audio_buf_size = audio_size;
            }
            is->audio_buf_index = 0;
        }
        //根据缓冲区剩余大小量力而行
        len1 = is->audio_buf_size - is->audio_buf_index;
        if (len1 > len)
            len1 = len;
        //根据audio_volume决定如何输出audio_buf
        /* 判断是否为静音，以及当前音量的大小，如果音量为最大则直接拷贝数据 */
        if (!is->muted && is->audio_buf && is->audio_volume == SDL_MIX_MAXVOLUME)
            memcpy(stream, (uint8_t *)is->audio_buf + is->audio_buf_index, len1);
        else {
            memset(stream, 0, len1);
            // 调整音量
            /* 如果处于mute状态则直接使用stream填0数据，暂停时is->audio_buf = NULL */
            if (!is->muted && is->audio_buf)
                SDL_MixAudioFormat(stream, (uint8_t *)is->audio_buf + is->audio_buf_index,
```

```
                                      AUDIO_S16SYS, len1, is->audio_volume);
        }
        len -= len1;
        stream += len1;
        /* 更新is->audio_buf_index，指向audio_buf中未被拷贝到stream的数据（剩余数据）的起始位置 */
        is->audio_buf_index += len1;
    }
    is->audio_write_buf_size = is->audio_buf_size - is->audio_buf_index;
    /* Let's assume the audio driver that is used by SDL has two periods. */
    if (!isnan(is->audio_clock)) {
        set_clock_at(&is->audclk,
                     is->audio_clock - (double)(2 * is->audio_hw_buf_size + is->audio_write_buf_size) / is->audio_tgt.bytes_per_sec,
                     is->audio_clock_serial,
                     audio_callback_time / 1000000.0);
        sync_clock_to_slave(&is->extclk, &is->audclk);
    }
}
```

`sdl_audio_callback`函数是一个典型的缓冲区输出过程，看代码和注释应该可以理解。具体看3个细节：

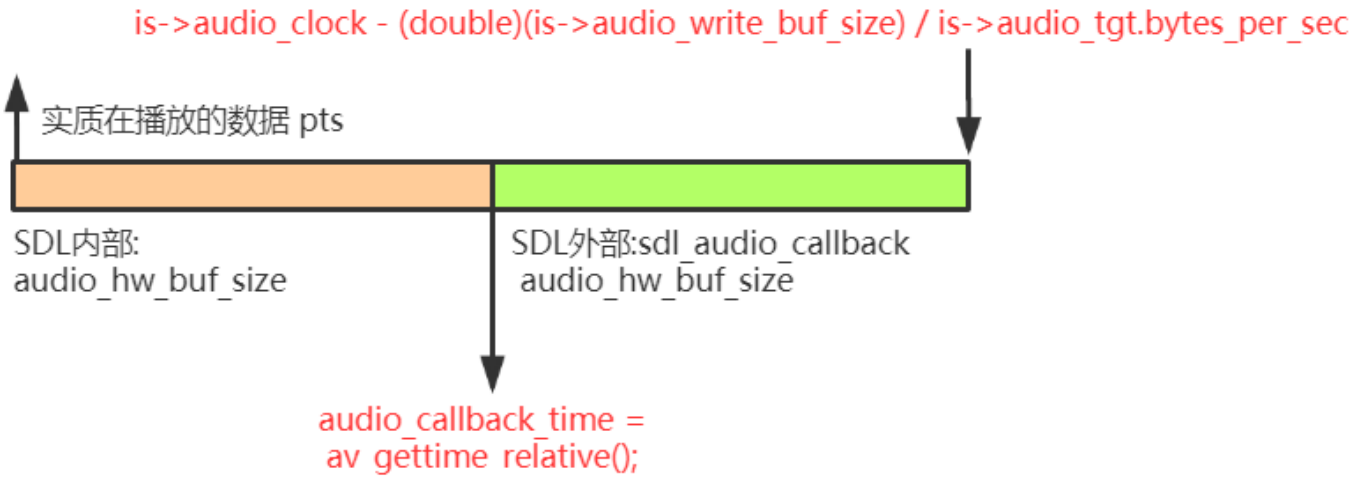
- 输出audio_buf到stream，如果audio_volume为最大音量，则只需memcpy复制给stream即可。否则，可以利用SDL_MixAudioFormat进行音量调整和混音
- 如果audio_buf消耗完了，就调用`audio_decode_frame`重新填充audio_buf。接下来会继续分析audio_decode_frame函数
- set_clock_at更新audclk时，audio_clock是当前audio_buf的显示结束时间(pts+duration)，由于audio driver本身会持有一小块缓冲区，典型地会是两块交替使用，所以有`2 * is->audio_hw_buf_size`，至于为什么还要audio_write_buf_size，一图胜千言。



我们先来is->audio_clock是在audio_decode_frame赋值：

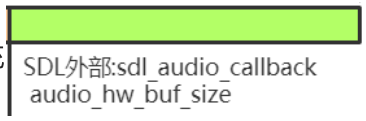
```
is->audio_clock = af->pts + (double) af->frame->nb_samples / af->frame->sample_rate;
```

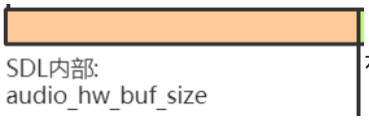
从这里可以看出来，这里的时间戳是audio_buf结束位置的时间戳，而不是audio_buf起始位置的时间戳，所以当audio_buf有剩余时（），那实际数据的pts就变成is->audio_clock - (double)(is->audio_write_buf_size) / is->audio_tgt.bytes_per_sec，即是

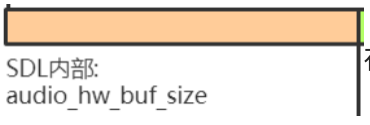


再考虑到，实质上audio_hw_buf_size*2这些数据实际都没有播放出去，所以就有

```
is->audio_clock - (double)(2 * is->audio_hw_buf_size + is->audio_write_buf_size) / is->audio_tgt.bytes_per_sec。
```

再加上我们在SDL回调进行填充 

时，实际上 

是有开始被播放，所以我们这里采用的相对时间是，刚回调产生的，就是内部 

在播放

的时候，那相对时间实际也在走。

```
static void sdl_audio_callback(void *opaque, Uint8 *stream, int len)
{
    VideoState *is = opaque;
    int audio_size, len1;

    audio_callback_time = av_gettime_relative();
```

最终
set_clock_at(&is->audclk, is->audio_clock - (double)(2 * is->audio_hw_buf_size + is->audio_write_buf_size) / is->audio_tgt.bytes_per_sec, is->audio_clock_serial, audio_callback_time / 1000000.0);

回调函数读取数据

接下来看下`audio_decode_frame`（省略重采样代码）：

```
1  static int audio_decode_frame(VideoState *is)
2  {
3      int data_size, resampled_data_size;
4      int64_t dec_channel_layout;
5      av_unused double audio_clock0;
6      int wanted_nb_samples;
7      Frame *af;
8      if (is->paused)//暂停状态，返回-1，sdl_audio_callback会处理为输出静音
9          return -1;
10     do {//1. 从sampq取一帧，必要时丢帧
11         if (!(af = frame_queue_peek_readable(&is->sampq)))
12             return -1;
13         frame_queue_next(&is->sampq);
14     } while (af->serial != is->audioq.serial);
15     //2. 计算这一帧的字节数
16     data_size = av_samples_get_buffer_size(NULL, af->frame->channels
   ,
17                                            af->frame->nb_samples,
18                                            af->frame->format, 1);
19     //[]计算dec_channel_layout，用于确认是否需要重新初始化重采样（难道af->cha
   nnel_layout不可靠？不理解）
```

```
    dec_channel_layout =
        (af->frame->channel_layout && af->frame->channels == av_get_channel_layout_nb_channels(af->frame->channel_layout)) ?
        af->frame->channel_layout : av_get_default_channel_layout(af->frame->channels);
    wanted_nb_samples = synchronize_audio(is, af->frame->nb_samples);
    //[]判断是否需要重新初始化重采样
    if (af->frame->format        != is->audio_src.fmt            ||
        dec_channel_layout       != is->audio_src.channel_layout ||
        af->frame->sample_rate   != is->audio_src.freq           ||
        (wanted_nb_samples       != af->frame->nb_samples && !is->swr_ctx)) {
        //……
    }
    //3. 获取这一帧的数据
    if (is->swr_ctx) {//[]如果初始化了重采样，则对这一帧数据重采样输出
    }else {
        is->audio_buf = af->frame->data[0];
        resampled_data_size = data_size;
    }
    audio_clock0 = is->audio_clock;//audio_clock0用于打印调试信息
    //4. 更新audio_clock，audio_clock_serial
    /* update the audio clock with the pts */
    if (!isnan(af->pts))
        is->audio_clock = af->pts + (double) af->frame->nb_samples / af->frame->sample_rate;
    else
        is->audio_clock = NAN;
    is->audio_clock_serial = af->serial;
    return resampled_data_size;//返回audio_buf的数据大小
}
```

`audio_decode_frame`并没有真正意义上的`decode`代码，最多是进行了重采样。主流程有以下步骤：

1. 从sampq取一帧，必要时丢帧。如发生了seek，此时serial会不连续，就需要丢帧处理
2. 计算这一帧的字节数。通过av_samples_get_buffer_size可以方便计算出结果
3. 获取这一帧的数据。对于frame格式和输出设备不同的，需要重采样；如果格式相同，则直接拷贝指针输出即可。总之，需要在audio_buf中保存与输出设备格式相同的音频数据
4. 更新audio_clock，audio_clock_serial。用于设置audclk.

在省略了重采样代码后看，相对容易理解。

至此，音频输出的主要代码就分析完了。中间我们省略了filter和resample相关的代码，有研究后再补充。

8 音频重采样

FFmpeg解码得到的音频帧的格式未必能被SDL支持，在这种情况下，需要进行音频重采样，即将音频帧格式转换为SDL支持的音频格式，否则是无法正常播放的。
音频重采样涉及两个步骤：

1. 打开音频设备时进行的准备工作：确定SDL支持的音频格式，作为后期音频重采样的目标格式。这一部分内容参考《7 音频输出模块》
2. 音频播放线程中，取出音频帧后，若有需要(音频帧格式与SDL支持音频格式不匹配)则进行重采样，否则直接输出

重采样逻辑

音频重采样在`audio_decode_frame()`中实现，`audio_decode_frame()`就是从音频frame队列中取出一个frame，按指定格式经过重采样后输出（解码不是在该函数进行）。
重采样的细节很琐碎，直接看注释：

```
static int audio_decode_frame(VideoState *is)
{
    int data_size, resampled_data_size;
    int64_t dec_channel_layout;
    av_unused double audio_clock0;
    int wanted_nb_samples;
    Frame *af;

    if (is->paused)
        return -1;

    do {
#if defined(_WIN32)
        while (frame_queue_nb_remaining(&is->sampq) == 0) {
            if ((av_gettime_relative() - audio_callback_time) > 1000000LL * is->audio_hw_buf_size / is->audio_tgt.bytes_per_sec / 2)
                return -1;
            av_usleep (1000);
        }
#endif
```

```
20          // 若队列头部可读，则由af指向可读帧
21          if (!(af = frame_queue_peek_readable(&is->sampq)))
22              return -1;
23          frame_queue_next(&is->sampq);
24      } while (af->serial != is->audioq.serial);
25
26      // 根据frame中指定的音频参数获取缓冲区的大小
27      data_size = av_samples_get_buffer_size(NULL,
28                                             af->frame->channels,
29                                             af->frame->nb_samples,
30                                             af->frame->format, 1);
31      // 获取声道布局
32      dec_channel_layout =
33              (af->frame->channel_layout &&
34               af->frame->channels == av_get_channel_layout_nb_channe
   ls(af->frame->channel_layout)) ?
35                  af->frame->channel_layout : av_get_default_channel_
   layout(af->frame->channels);
36      // 获取样本数校正值：若同步时钟是音频，则不调整样本数；否则根据同步需要调整样
   本数
37      wanted_nb_samples = synchronize_audio(is, af->frame->nb_samples
   );
38      // is->audio_tgt是SDL可接受的音频帧数，是audio_open()中取得的参数
39      // 在audio_open()函数中又有"is->audio_src = is->audio_tgt""
40      // 此处表示：如果frame中的音频参数 == is->audio_src == is->audio_tg
   t,
41      // 那音频重采样的过程就免了(因此时is->swr_ctr是NULL)
42      // 否则使用frame(源)和is->audio_tgt(目标)中的音频参数来设置is->swr_ct
   x,
43      // 并使用frame中的音频参数来赋值is->audio_src
44      if (af->frame->format        != is->audio_src.fmt            ||
45              dec_channel_layout       != is->audio_src.channel_layou
   t ||
46              af->frame->sample_rate   != is->audio_src.freq
   ||
47              (wanted_nb_samples       != af->frame->nb_samples && !i
   s->swr_ctx)) {
48          swr_free(&is->swr_ctx);
49          is->swr_ctx = swr_alloc_set_opts(NULL,
50                                           is->audio_tgt.channel_layo
```

```
ut,  // 目标输出
                                            is->audio_tgt.fmt,
                                            is->audio_tgt.freq,
                                            dec_channel_layout,
// 数据源
                                            af->frame->format,
                                            af->frame->sample_rate,
                                            0, NULL);
        if (!is->swr_ctx || swr_init(is->swr_ctx) < 0) {
            av_log(NULL, AV_LOG_ERROR,
                   "Cannot create sample rate converter for convers
ion of %d Hz %s %d channels to %d Hz %s %d channels!\n",
                   af->frame->sample_rate, av_get_sample_fmt_name(a
f->frame->format), af->frame->channels,
                   is->audio_tgt.freq, av_get_sample_fmt_name(is->a
udio_tgt.fmt), is->audio_tgt.channels);
            swr_free(&is->swr_ctx);
            return -1;
        }
        is->audio_src.channel_layout = dec_channel_layout;
        is->audio_src.channels       = af->frame->channels;
        is->audio_src.freq = af->frame->sample_rate;
        is->audio_src.fmt = af->frame->format;
    }

    if (is->swr_ctx) {
        // 重采样输入参数1：输入音频样本数是af->frame->nb_samples
        const uint8_t **in = (const uint8_t **)af->frame->extended_
data;
        // 重采样输入参数2：输入音频缓冲区
        uint8_t **out = &is->audio_buf1;
        // 重采样输出参数1：输出音频缓冲区尺寸
        int out_count = (int64_t)wanted_nb_samples *
                            is->audio_tgt.freq / af->frame->sample_
rate + 256;
        // 重采样输出参数2：输出音频缓冲区
        int out_size  = av_samples_get_buffer_size(NULL, is->audio_
tgt.channels,
                                                   out_count, is->a
udio_tgt.fmt, 0);
```

```
82          int len2;
83          if (out_size < 0) {
84              av_log(NULL, AV_LOG_ERROR, "av_samples_get_buffer_size
    () failed\n");
85              return -1;
86          }
87          // 如果frame中的样本数经过校正，则条件成立
88          if (wanted_nb_samples != af->frame->nb_samples) {
89              if (swr_set_compensation(is->swr_ctx,
90                                       (wanted_nb_samples - af->frame
    ->nb_samples) *
91                                          is->audio_tgt.freq / af->fr
    ame->sample_rate,
92                                       wanted_nb_samples * is->audio_
    tgt.freq /
93                                          af->frame->sample_rate) < 0
    ) {
94                  av_log(NULL, AV_LOG_ERROR, "swr_set_compensation()
     failed\n");
95                  return -1;
96              }
97          }
98          av_fast_malloc(&is->audio_buf1, &is->audio_buf1_size, out_s
    ize);
99          if (!is->audio_buf1)
100             return AVERROR(ENOMEM);
101         // 音频重采样：返回值是重采样后得到的音频数据中单个声道的样本数
102         len2 = swr_convert(is->swr_ctx, out, out_count, in, af->fra
    me->nb_samples);
103         if (len2 < 0) {
104             av_log(NULL, AV_LOG_ERROR, "swr_convert() failed\n");
105             return -1;
106         }
107         if (len2 == out_count) {
108             av_log(NULL, AV_LOG_WARNING, "audio buffer is probably
     too small\n");
109             if (swr_init(is->swr_ctx) < 0)
110                 swr_free(&is->swr_ctx);
111         }
112         // 重采样返回的一帧音频数据大小(以字节为单位)
```

```
113         is->audio_buf = is->audio_buf1;
114         resampled_data_size = len2 * is->audio_tgt.channels * av_ge
    t_bytes_per_sample(is->audio_tgt.fmt);
115     } else {
116         // 未经重采样，则将指针指向frame中的音频数据
117         is->audio_buf = af->frame->data[0];
118         resampled_data_size = data_size;
119     }
120 
121     audio_clock0 = is->audio_clock;
122     /* update the audio clock with the pts */
123     if (!isnan(af->pts))
124         is->audio_clock = af->pts + (double) af->frame->nb_samples
    / af->frame->sample_rate;
125     else
126         is->audio_clock = NAN;
127     is->audio_clock_serial = af->serial;
128 #ifdef DEBUG
129     {
130         static double last_clock;
131         printf("audio: delay=%0.3f clock=%0.3f clock0=%0.3f\n",
132                is->audio_clock - last_clock,
133                is->audio_clock, audio_clock0);
134         last_clock = is->audio_clock;
135     }
136 #endif
137     return resampled_data_size;
138 }
```

样本补偿

swr_set_compensation说明

```
/**
* Activate resampling compensation ("soft" compensation). This function is
* internally called when needed in swr_next_pts().
*
* @param[in,out] s             allocated Swr context. If it is not initialized,
*                              or SWR_FLAG_RESAMPLE is not set, swr_init() is
*                              called with the flag set.
```

```
* @param[in]    sample_delta  delta in PTS per sample（每个样本的增量，单位pts）
* @param[in]    compensation_distance number of samples to compensate for（要补偿的样本数）
* @return    >= 0 on success, AVERROR error codes if:
*           @li @c s is NULL,
*           @li @c compensation_distance is less than 0,
*           @li @c compensation_distance is 0 but sample_delta is not,
*           @li compensation unsupported by resampler, or
*           @li swr_init() fails when called.
*/
int swr_set_compensation(struct SwrContext *s, int sample_delta, int compensation_distance);
```

sample_delta： (wanted_nb_samples - af->frame->nb_samples) * is->audio_tgt.freq / af->frame->sample_rate；
compensation_distance： wanted_nb_samples * is->audio_tgt.freq / af->frame->sample_rate)

注意是谁是除数，谁是被除数。