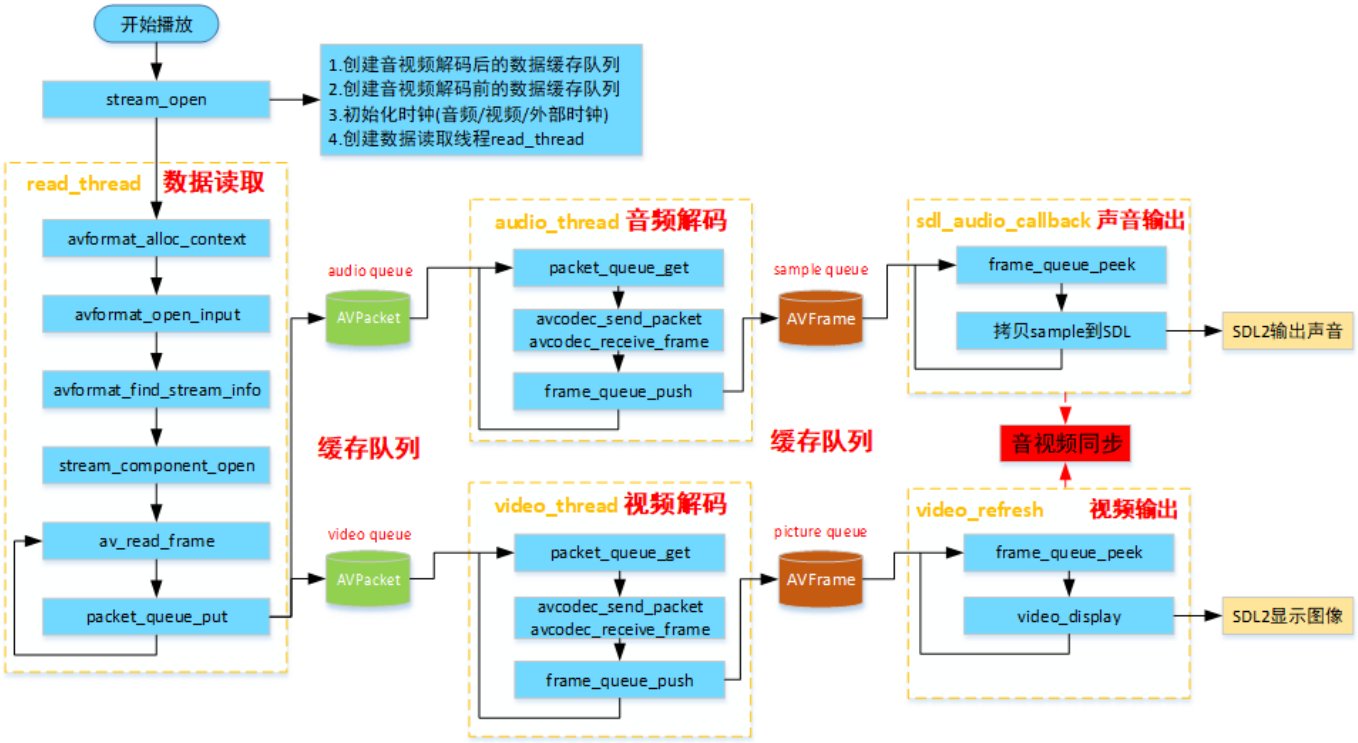
ffplay播放器-4数据读取线程

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

4 数据读取线程



从ffplay框架分析我们可以看到，ffplay有专门的线程read_thread()读取数据，且在调用av_read_frame读取数据包之前需要做例如打开文件，查找配置解码器，初始化音视频输出等准备阶段，主要包括三大步骤：

- 准备工作
- For循环读取数据
- 退出线程处理

一 准备工作

1. avformat_alloc_context 创建上下文
2. ic->interrupt_callback.callback = decode_interrupt_cb;
3. avformat_open_input打开媒体文件
4. avformat_find_stream_info 读取媒体文件的包获取更多的stream信息
5. 检测是否指定播放起始时间，如果指定时间则seek到指定位置avformat_seek_file
6. 查找查找AVStream，讲对应的index值记录到st_index[AVMEDIA_TYPE_NB];
   a. 根据用户指定来查找流avformat_match_stream_specifier
   b. 使用av_find_best_stream查找流
7. 从待处理流中获取相关参数，设置显示窗口的宽度、高度及宽高比
8. stream_component_open打开音频、视频、字幕解码器，并创建相应的解码线程以及进行对应输出参数的初始化。

二 For循环读取数据

1. 检测是否退出

2. 检测是否暂停/继续
3. 检测是否需要seek
4. 检测video是否为attached_pic
5. 检测队列是否已经有足够数据
6. 检测码流是否已经播放结束
    a. 是否循环播放
    b. 是否自动退出
7. 使用av_read_frame读取数据包
8. 检测数据是否读取完毕
9. 检测是否在播放范围内
10. 到这步才将数据插入对应的队列

三 退出线程处理

1. 如果解复用器有打开则关闭avformat_close_input
2. 调用SDL_PushEvent发送退出事件FF_QUIT_EVENT
3. 消耗互斥量wait_mutex

4.1 准备工作

准备工作主要包括以下步骤：

1. avformat_alloc_context 创建上下文
2. ic->interrupt_callback.callback = decode_interrupt_cb;
3. avformat_open_input打开媒体文件
4. avformat_find_stream_info 读取媒体文件的包获取更多的stream信息
5. 检测是否指定播放起始时间，如果指定时间则seek到指定位置avformat_seek_file
6. 查找查找AVStream，讲对应的index值记录到st_index[AVMEDIA_TYPE_NB];
    a. 根据用户指定来查找流avformat_match_stream_specifier
    b. 使用av_find_best_stream查找流
7. 通过AVCodecParameters和av_guess_sample_aspect_ratio计算出显示窗口的宽、高
8. stream_component_open打开音频、视频、字幕解码器，并创建相应的解码线程以及进行对应输出参数的初始化。

1 avformat_alloc_context 创建上下文

调用avformat_alloc_context创建解复用器上下文

```
// 1. 创建上下文结构体，这个结构体是最上层的结构体，表示输入上下文
ic = avformat_alloc_context();
```

最终该ic 赋值给VideoState的ic变量

```
is->ic = ic; // videoState的ic指向分配的ic
```

2 ic->interrupt_callback

```
/* 2.设置中断回调函数，如果出错或者退出，就根据目前程序设置的状态选择继续check或者直接退出 */
/* 当执行耗时操作时（一般是在执行while或者for循环的数据读取时），会调用interrupt_callback.callback
 * 回调函数中返回1则代表ffmpeg结束耗时操作退出当前函数的调用
* 回调函数中返回0则代表ffmpeg内部继续执行耗时操作，直到完成既定的任务(比如读取到既定的数据包)
*/
ic->interrupt_callback.callback = decode_interrupt_cb;
ic->interrupt_callback.opaque = is;
```

interrupt_callback用于ffmpeg内部在执行耗时操作时检查调用者是否有退出请求，避免用户退出请求没有及时响应。
怎么去测试在哪里触发?
在ubuntu使用gdb进行调试：我们之前讲的在ubuntu下编译ffmpeg，在lqf@ubuntu:~/ffmpeg_sources/**ffmpeg-4.2.1**目录下有**ffplay_g**，我们可以通过 gdb ./ffplay_g来播放视频，然后在decode_interrupt_cb打断点。

avformat_open_input的触发

```
#0  decode_interrupt_cb (ctx=0x7ffff7e36040) at fftools/ffplay.c:2715
#1  0x00000000007d99b7 in ff_check_interrupt (cb=0x7fffd00014b0)
    at libavformat/avio.c:667
#2  retry_transfer_wrapper (transfer_func=0x7dd950 <file_read>, size_min=1,
    size=32768, buf=0x7fffd0001700 "", h=0x7fffd0001480)
    at libavformat/avio.c:374
#3  ffurl_read (h=0x7fffd0001480, buf=0x7fffd0001700 "", size=32768)
    at libavformat/avio.c:411
```

```
#4  0x000000000068cd9c in read_packet_wrapper (size=<optimized out>,
    buf=<optimized out>, s=0x7fffd00011c0) at libavformat/aviobuf.c:
535
#5  fill_buffer (s=0x7fffd00011c0) at libavformat/aviobuf.c:584
#6  avio_read (s=s@entry=0x7fffd00011c0, buf=0x7fffd0009710 "",
    size=size@entry=2048) at libavformat/aviobuf.c:677
#7  0x00000000006b7780 in av_probe_input_buffer2 (pb=0x7fffd00011c0,
    fmt=0x7fffd0000948,
    filename=filename@entry=0x31d50e0 "source.200kbps.768x320.flv",
    logctx=logctx@entry=0x7fffd0000940, offset=offset@entry=0,
    max_probe_size=1048576) at libavformat/format.c:262
#8  0x00000000007b631d in init_input (options=0x7fffdd9bcb50,
    filename=0x31d50e0 "source.200kbps.768x320.flv", s=0x7fffd000094
0)
    at libavformat/utils.c:443
#9  avformat_open_input (ps=ps@entry=0x7fffdd9bcbf8,
    filename=0x31d50e0 "source.200kbps.768x320.flv", fmt=<optimized
out>,
```

可以看到是在libavformat/avio.c:374行有触发到

```
362: static inline int retry_transfer_wrapper(URLContext *h, uint8_t *buf,
363:                                          int size, int size_min,
364:                                          int (*transfer_func)(URLContext *h,
365:                                                               uint8_t *buf,
366:                                                               int size))
367: {
368:     int ret, len;
369:     int fast_retries = 5;
370:     int64_t wait_since = 0;
371:
372:     len = 0;
373:     while (len < size_min) {
374:         if (ff_check_interrupt(&h->interrupt_callback))
375:             return AVERROR_EXIT;
```

avformat_find_stream_info的触发

```
#0  decode_interrupt_cb (ctx=0x7ffff7e36040) at fftools/ffplay.c:2715
#1  0x00000000007b25bc in avformat_find_stream_info (ic=0x7fffd000094
0,
    options=0x0) at libavformat/utils.c:3693
#2  0x00000000004a6ea9 in read_thread (arg=0x7ffff7e36040)
```

从该调用栈可以看出来 avformat_find_stream_info也会触发ic->interrupt_callback的调用，具体可以看代码（libavformat/utils.c:3693行）

```
3691:     for (;;) {
3692:         int analyzed_all_streams;
3693:         if (ff_check_interrupt(&ic->interrupt_callback))
3694:             ret = AVERROR_EXIT;
3695:             av_log(ic, AV_LOG_DEBUG, "interrupted\n");
3696:             break;
3697:         }
```

av_read_frame的触发

```
#0  decode_interrupt_cb (ctx=0x7ffff7e36040) at fftools/ffplay.c:2715
#1  0x00000000007d99b7 in ff_check_interrupt (cb=0x7fffd00014b0)
    at libavformat/avio.c:667
#2  retry_transfer_wrapper (transfer_func=0x7dd950 <file_read>, size_min=1,
    size=32768, buf=0x7fffd0009710 "FLV\001\005", h=0x7fffd0001480)
    at libavformat/avio.c:374
#3  ffurl_read (h=0x7fffd0001480, buf=0x7fffd0009710 "FLV\001\005", size=32768)
    at libavformat/avio.c:411
#4  0x000000000068cd9c in read_packet_wrapper (size=<optimized out>,
    buf=<optimized out>, s=0x7fffd00011c0) at libavformat/aviobuf.c:535
#5  fill_buffer (s=0x7fffd00011c0) at libavformat/aviobuf.c:584
#6  avio_read (s=s@entry=0x7fffd00011c0, buf=0x7fffd00dbf6d "\177", size=45,
    size@entry=90) at libavformat/aviobuf.c:677
#7  0x00000000007a99d5 in append_packet_chunked (s=0x7fffd00011c0,
    pkt=pkt@entry=0x7fffdd9bca00, size=size@entry=90)
    at libavformat/utils.c:293
#8  0x00000000007aa969 in av_get_packet (s=<optimized out>,
    pkt=pkt@entry=0x7fffdd9bca00, size=size@entry=90)
    at libavformat/utils.c:317
#9  0x00000000006b350a in flv_read_packet (s=0x7fffd0000940,
    pkt=0x7fffdd9bca00) at libavformat/flvdec.c:1295
#10 0x00000000007aad6d in ff_read_packet (s=s@entry=0x7fffd0000940,
    pkt=pkt@entry=0x7fffdd9bca00) at libavformat/utils.c:856
---Type <return> to continue, or q <return> to quit---
```

```
#11 0x00000000007ae291 in read_frame_internal (s=0x7fffd0000940,
    pkt=0x7fffdd9bcc00) at libavformat/utils.c:1582
#12 0x00000000007af422 in av_read_frame (s=0x7fffd0000940,
    pkt=pkt@entry=0x7fffdd9bcc00) at libavformat/utils.c:1779
#13 0x00000000004a68b1 in read_thread (arg=0x7ffff7e36040)
    at fftools/ffplay.c:3008
```

这里的触发和**avformat_open_input一致，大家可以自行跟踪调用栈。**

3 avformat_open_input()打开媒体文件

函数原型：

```
/**
* Open an input stream and read the header. The codecs are not opened.
* The stream must be closed with avformat_close_input().
*
* @param ps Pointer to user-supplied AVFormatContext (allocated by avformat_alloc_context).
* May be a pointer to NULL, in which case an AVFormatContext is allocated by this
* function and written into ps.
* Note that a user-supplied AVFormatContext will be freed on failure.
* @param url URL of the stream to open.
* @param fmt If non-NULL, this parameter forces a specific input format.
* Otherwise the format is autodetected.
* @param options A dictionary filled with AVFormatContext and demuxer-private options.
* On return this parameter will be destroyed and replaced with a dict containing
* options that were not found. May be NULL.
*
* @return 0 on success, a negative AVERROR on failure.
*
* @note If you want to use custom IO, preallocate the format context and set its pb field.
*/
int avformat_open_input(AVFormatContext **ps, const char *url, ff_const59 AVInputFormat
*fmt, AVDictionary **options);
```

avformat_open_input用于打开输入文件（对于RTMP/RTSP/HTTP网络流也是一样，在ffmpeg内部都抽象为URLProtocol，这里描述为文件是为了方便与后续提到的AVStream的流作区分），读取视频文件的基本信息。

需要提到的两个参数是fmt和options。通过fmt可以强制指定视频文件的封装，options可以传递额外参数给封装(AVInputFormat)。
主要代码：

```
//特定选项处理
if (!av_dict_get(format_opts, "scan_all_pmts", NULL, AV_DICT_MATCH_CASE)) {
    av_dict_set(&format_opts, "scan_all_pmts", "1", AV_DICT_DONT_OVERWRITE);
    scan_all_pmts_set = 1;
}
/* 3.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等 */
err = avformat_open_input(&ic, is->filename, is->iformat, &format_opts);
if (err < 0) {
    print_error(is->filename, err);
    ret = -1;
    goto fail;
}
if (scan_all_pmts_set)
    av_dict_set(&format_opts, "scan_all_pmts", NULL, AV_DICT_MATCH_CASE);

if ((t = av_dict_get(format_opts, "", NULL, AV_DICT_IGNORE_SUFFIX))) {
    av_log(NULL, AV_LOG_ERROR, "Option %s not found.\n", t->key);
    ret = AVERROR_OPTION_NOT_FOUND;
    goto fail;
}
```

scan_all_pmts是mpegts的一个选项，表示扫描全部的ts流的"Program Map Table"表。这里在没有设定该选项的时候，强制设为1。最后执行avformat_open_input。

使用gdb跟踪options的设置，在av_opt_set打断点
(gdb) b av_opt_set
(gdb) r
#0 av_opt_set_dict2 (obj=obj@entry=0x7fffd0000940,
options=options@entry=0x7fffdd9bcb50, search_flags=search_flags@entry=0)

at libavutil/opt.c:1588
#1 0x00000000011c6837 in av_opt_set_dict (obj=obj@entry=0x7fffd0000940,
options=options@entry=0x7fffdd9bcb50) at libavutil/opt.c:1605
#2 0x00000000007b5f8b in avformat_open_input (ps=ps@entry=0x7fffdd9bcbf8,
filename=0x31d23d0 "source.200kbps.768x320.flv", fmt=<optimized out>,
options=0x2e2d450 <format_opts>) at libavformat/utils.c:560
#3 0x00000000004a70ae in read_thread (arg=0x7ffff7e36040)
at fftools/ffplay.c:2780
.....
(gdb) l
1583
1584 if (!options)
1585 return 0;
1586
1587 while ((t = av_dict_get(*options, "", t, AV_DICT_IGNORE_SUFFIX))) {
1588 ret = av_opt_set(obj, t->key, t->value, search_flags);
1589 if (ret == AVERROR_OPTION_NOT_FOUND)
1590 ret = av_dict_set(&tmp, t->key, t->value, 0);
1591 if (ret < 0) {
1592 av_log(obj, AV_LOG_ERROR, "Error setting option %s to value %s.\n", t->key, t->value);

(gdb) print **options
$3 = {count = 1, elems = 0x7fffd0001200}
(gdb) print (*options)->elems
$4 = (AVDictionaryEntry *) 0x7fffd0001200
(gdb) print *((*options)->elems)
$5 = {key = 0x7fffd0001130 "scan_all_pmts", value = 0x7fffd0001150 "1"}
(gdb)

参数的设置最终都是设置到对应的解复用器，比如：

- mpegts.c

```
176: static const AVOption options[] = {
177:     MPEGTS_OPTIONS,
178:     {"fix_teletext_pts", "try to fix pts values of dvb teletext streams", offsetof
179:      {.i64 = 1}, 0, 1, AV_OPT_FLAG_DECODING_PARAM },
180:     {"ts_packetsize", "output option carrying the raw packet size", offsetof(MpegT
181:      {.i64 = 0}, 0, 0, AV_OPT_FLAG_DECODING_PARAM | AV_OPT_FLAG_EXPORT | AV_OPT_FL
182:     {"scan_all_pmts", "scan and combine all PMTs", offsetof(MpegTSContext, scan_al
183:      {.i64 = -1}, -1, 1, AV_OPT_FLAG_DECODING_PARAM },
184:     {"skip_unknown_pmt", "skip PMTs for programs not advertised in the PAT", offse
185:      {.i64 = 0}, 0, 1, AV_OPT_FLAG_DECODING_PARAM },
186:     {"merge_pmt_versions", "re-use streams when PMT's version/pids change", offset
187:      {.i64 = 0}, 0, 1,  AV_OPT_FLAG_DECODING_PARAM },
188:     {"skip_changes", "skip changing / adding streams / programs", offsetof(MpegTSC
189:      {.i64 = 0}, 0, 1, 0 },
190:     {"skip_clear", "skip clearing programs", offsetof(MpegTSContext, skip_clear),
191:      {.i64 = 0}, 0, 1, 0 },
192:     { NULL },
193: };
```

- flvdec.c

```
1357: #define OFFSET(x) offsetof(FLVContext, x)
1358: #define VD AV_OPT_FLAG_VIDEO_PARAM | AV_OPT_FLAG_DECODING_PARAM
1359: static const AVOption options[] = {
1360:     { "flv_metadata", "Allocate streams according to the onMetaData array", OFFSET(trust_metadata), AV_OPT_TYPE
1361:     { "flv_full_metadata", "Dump full metadata of the onMetadata", OFFSET(dump_full_metadata), AV_OPT_TYPE_BOOL
1362:     { "flv_ignore_prevtag", "Ignore the Size of previous tag", OFFSET(trust_datasize), AV_OPT_TYPE_BOOL, { .i64
1363:     { "missing_streams", "", OFFSET(missing_streams), AV_OPT_TYPE_INT, { .i64 = 0 }, 0, 0xFF, VD | AV_OPT_FLAG_
1364:     { NULL }
1365: };
```

4 avformat_find_stream_info()

在打开了文件后，就可以从AVFormatContext中读取流信息了。一般调用avformat_find_stream_info获取完整的流信息。为什么在调用了avformat_open_input后，仍然需要调用avformat_find_stream_info才能获取正确的流信息呢？看下注释：

```
/**
* Read packets of a media file to get stream information. This
* is useful for file formats with no headers such as MPEG. This
* function also computes the real framerate in case of MPEG-2 repeat
* frame mode.
* The logical file position is not changed by this function;
* examined packets may be buffered for later processing.
*
* @param ic media file handle
* @param options If non-NULL, an ic.nb_streams long array of pointers to
* dictionaries, where i-th member contains options for
* codec corresponding to i-th stream.
* On return each dictionary will be filled with options that were not found.
* @return >=0 if OK, AVERROR_xxx on error
*
* @note this function isn't guaranteed to open all the codecs, so
* options being non-empty at return is a perfectly normal behavior.
*
* @todo Let the user decide somehow what information is needed so that
* we do not waste time getting stuff the user does not need.
*/
int avformat_find_stream_info(AVFormatContext *ic, AVDictionary **options);
```

该函数是通过读取媒体文件的部分数据来分析流信息。在一些缺少头信息的封装下特别有用，比如说MPEG（里应该说ts更准确）(FLV文件也是需要读取packet 分析流信息)。而被读取用以分析流信息的数据可能被缓存，供av_read_frame时使用，在播放时并不会跳过这部分packet的读取。

5 检测是否指定播放起始时间

如果指定时间则seek到指定位置avformat_seek_file。
可以通过 ffplay -ss 设置起始时间，时间格式hh:mm:ss，比如
ffplay -ss 00:00:30 test.flv 则是从30秒的起始位置开始播放。

具体调用流程，可以在opt_seek 函数打断点进行测试

```
{ "ss", HAS_ARG, { .func_arg = opt_seek }, "seek to a given position in seconds", "pos" },
{ "t", HAS_ARG, { .func_arg = opt_duration }, "play  \"duration\" seconds of audio/video", "duration" },
```

```
 /* if seeking requested, we execute it */
/* 5. 检测是否指定播放起始时间 */
if (start_time != AV_NOPTS_VALUE) {
    int64_t timestamp;

    timestamp = start_time;
    /* add the stream start time */
    if (ic->start_time != AV_NOPTS_VALUE)
        timestamp += ic->start_time;
    // seek的指定的位置开始播放
    ret = avformat_seek_file(ic, -1, INT64_MIN, timestamp, INT64_MAX, 0);
    if (ret < 0) {
        av_log(NULL, AV_LOG_WARNING, "%s: could not seek to position %0.3f\n",
                is->filename, (double)timestamp / AV_TIME_BASE);
    }
}
```

6 查找查找AVStream

一个媒体文件，对应有0~n个音频流、0~n个视频流、0~n个字幕流，比如这里我们用了2_audio.mp4是有2个音频流，1个视频流

具体现在那个流进行播放我们有两种策略：

1. 在播放起始指定对应的流
2. 使用缺省的流进行播放

1 在播放起始指定对应的流

ffplay是通过通过命令可以指定流

```
{ "ast", OPT_STRING | HAS_ARG | OPT_EXPERT, {
&wanted_stream_spec[AVMEDIA_TYPE_AUDIO] }, "select desired audio stream",
"stream_specifier" },
{ "vst", OPT_STRING | HAS_ARG | OPT_EXPERT, {
&wanted_stream_spec[AVMEDIA_TYPE_VIDEO] }, "select desired video stream",
"stream_specifier" },
{ "sst", OPT_STRING | HAS_ARG | OPT_EXPERT, {
&wanted_stream_spec[AVMEDIA_TYPE_SUBTITLE] }, "select desired subtitle stream",
"stream_specifier" },
```

可以通过

- -ast n 指定音频流（比如我们在看电影时，有些电影可以支持普通话和英文切换，此时可以用该命令进行选择）
- -vst n 指定视频流
- -vst n 指定字幕流

讲对应的index值记录到st_index[AVMEDIA_TYPE_NB];

2 使用缺省的流进行播放

如果我们没有指定，则ffplay主要是通过`av_find_best_stream`来选择，其原型为：

```
/**
 * Find the "best" stream in the file.
 * The best stream is determined according to various heuristics as the most
 * likely to be what the user expects.
 * If the decoder parameter is non-NULL, av_find_best_stream will find the
 * default decoder for the stream's codec; streams for which no decoder can
 * be found are ignored.
 *
 * @param ic                media file handle
 * @param type              stream type: video, audio, subtitles, etc.
 * @param wanted_stream_nb  user-requested stream number,
 *                          or -1 for automatic selection
 * @param related_stream    try to find a stream related (eg. in the same
 *                          program) to this one, or -1 if none
 * @param decoder_ret       if non-NULL, returns the decoder for the
 *                          selected stream
 * @param flags             flags; none are currently defined
 * @return  the non-negative stream number in case of success,
 *          AVERROR_STREAM_NOT_FOUND if no stream with the requested type
 *          could be found,
 *          AVERROR_DECODER_NOT_FOUND if streams were found but no decoder
 * @note  If av_find_best_stream returns successfully and decoder_ret is not
 *        NULL, then *decoder_ret is guaranteed to be set to a valid AVCodec.
 */
int av_find_best_stream(AVFormatContext *ic,
                        enum AVMediaType type,  //要选择的流类型
                        int wanted_stream_nb,   //目标流索引
                        int related_stream,     //相关流索引
```

```
                        AVCodec **decoder_ret,
                        int flags);
```

具体代码流程

```
//根据用户指定来查找流
for (i = 0; i < ic->nb_streams; i++) {
    AVStream *st = ic->streams[i];
    enum AVMediaType type = st->codecpar->codec_type;
    st->discard = AVDISCARD_ALL;
    if (type >= 0 && wanted_stream_spec[type] && st_index[type] == -1)
        if (avformat_match_stream_specifier(ic, st, wanted_stream_spec[type]) > 0)
            st_index[type] = i;
}
for (i = 0; i < AVMEDIA_TYPE_NB; i++) {
    if (wanted_stream_spec[i] && st_index[i] == -1) {
        av_log(NULL, AV_LOG_ERROR, "Stream specifier %s does not match any %s stream\n", wanted_stream_spec[i], av_get_media_type_string(i));
        st_index[i] = INT_MAX;
    }
}
//利用av_find_best_stream选择流,
if (!video_disable)
    st_index[AVMEDIA_TYPE_VIDEO] =
    av_find_best_stream(ic, AVMEDIA_TYPE_VIDEO,
                        st_index[AVMEDIA_TYPE_VIDEO], -1, NULL, 0);
if (!audio_disable)
    st_index[AVMEDIA_TYPE_AUDIO] =
    av_find_best_stream(ic, AVMEDIA_TYPE_AUDIO,
                        st_index[AVMEDIA_TYPE_AUDIO],
                        st_index[AVMEDIA_TYPE_VIDEO],
                        NULL, 0);
if (!video_disable && !subtitle_disable)
    st_index[AVMEDIA_TYPE_SUBTITLE] =
    av_find_best_stream(ic, AVMEDIA_TYPE_SUBTITLE,
```

```
                        st_index[AVMEDIA_TYPE_SUBTITLE],
                        (st_index[AVMEDIA_TYPE_AUDIO] >= 0 ?
                         st_index[AVMEDIA_TYPE_AUDIO] :
                         st_index[AVMEDIA_TYPE_VIDEO]),
                        NULL, 0);
```

- 如果用户没有指定流，或指定部分流，或指定流不存在，则主要由av_find_best_stream发挥作用。
- 如果指定了**正确的**wanted_stream_nb，一般情况都是直接返回该指定流，即用户选择的流。

- 如果指定了相关流，且未指定目标流的情况，会在相关流的同一个节目中查找所需类型的流，但一般结果，都是返回该类型第1个流。

7 通过AVCodecParameters和av_guess_sample_aspect_ratio计算出显示窗口的宽、高

```
//7 从待处理流中获取相关参数，设置显示窗口的宽度、高度及宽高比
    if (st_index[AVMEDIA_TYPE_VIDEO] >= 0) {
        AVStream *st = ic->streams[st_index[AVMEDIA_TYPE_VIDEO]];
        AVCodecParameters *codecpar = st->codecpar;
        /*根据流和帧宽高比猜测帧的样本宽高比。
         * 由于帧宽高比由解码器设置，但流宽高比由解复用器设置，因此这两者可能不相等。
         * 此函数会尝试返回待显示帧应当使用的宽高比值。
         * 基本逻辑是优先使用流宽高比(前提是值是合理的)，其次使用帧宽高比。
         * 这样，流宽高比(容器设置，易于修改)可以覆盖帧宽高比。
        */
        AVRational sar = av_guess_sample_aspect_ratio(ic, st, NULL);
        if (codecpar->width) {
            // 设置显示窗口的大小和宽高比
            set_default_window_size(codecpar->width, codecpar->height, sar);
        }
    }
```

具体流程如上所示，这里实质只是设置了default_width、default_height变量的大小，没有真正改变窗口的大小。真正调整窗口大小是在视频显示调用video_open()函数进行设置。

8 stream_component_open()

经过以上步骤，文件打开成功，且获取了流的基本信息，并选择音频流、视频流、字幕流。接下来就可以所选流对应的解码器了。

```
/* open the streams */
    /* 5.打开视频、音频解码器。在此会打开相应解码器，并创建相应的解码线程。 */
    if (st_index[AVMEDIA_TYPE_AUDIO] >= 0) {
        stream_component_open(is, st_index[AVMEDIA_TYPE_AUDIO]);
    }

    ret = -1;
    if (st_index[AVMEDIA_TYPE_VIDEO] >= 0) {
        ret = stream_component_open(is, st_index[AVMEDIA_TYPE_VIDEO]
);
    }
    if (is->show_mode == SHOW_MODE_NONE) {
        //选择怎么显示，如果视频打开成功，就显示视频画面，否则，显示音频对应的频谱
图
        is->show_mode = ret >= 0 ? SHOW_MODE_VIDEO : SHOW_MODE_RDFT;
    }

    if (st_index[AVMEDIA_TYPE_SUBTITLE] >= 0) {
        stream_component_open(is, st_index[AVMEDIA_TYPE_SUBTITLE]);
    }
```

音频、视频、字幕等流都要调用stream_component_open，他们直接有共同的流程，也有差异化的流程，差异化流程使用switch进行区分。具体原型

int stream_component_open(VideoState *is, int stream_index);

stream_index

看下`stream_component_open`.函数也比较长，逐步分析：

```
/*  为解码器分配一个编解码器上下文结构体 */
    avctx = avcodec_alloc_context3(NULL);
    if (!avctx)
        return AVERROR(ENOMEM);
```

```
    /* 将码流中的编解码器信息拷贝到新分配的编解码器上下文结构体 */
    ret = avcodec_parameters_to_context(avctx, ic->streams[stream_index]->codecpar);
    if (ret < 0)
        goto fail;
    // 设置pkt_timebase
    avctx->pkt_timebase = ic->streams[stream_index]->time_base;
```

先是通过`avcodec_alloc_context3`分配了解码器上下文`AVCodecContex`，然后通过`avcodec_parameters_to_context`把所选流的解码参数赋给`avctx`，最后设了`time_base`.

补充：avcodec_parameters_to_context 解码时用，avcodec_parameters_from_context则用于编码。

```
 /* 根据codec_id查找解码器 */
    codec = avcodec_find_decoder(avctx->codec_id);

    switch(avctx->codec_type){// 获取指定的解码器名字，如果没有设置则为NULL
        case AVMEDIA_TYPE_AUDIO   : is->last_audio_stream    = stream_index;
            forced_codec_name =    audio_codec_name; break; // 获取指定的解码器名字
        case AVMEDIA_TYPE_SUBTITLE: is->last_subtitle_stream = stream_index;
            forced_codec_name = subtitle_codec_name; break; // 获取指定的解码器名字
        case AVMEDIA_TYPE_VIDEO   : is->last_video_stream    = stream_index;
            forced_codec_name =    video_codec_name; break; // 获取指定的解码器名字
    }
    }
    if (forced_codec_name)
        codec = avcodec_find_decoder_by_name(forced_codec_name);
    if (!codec) {
        if (forced_codec_name) av_log(NULL, AV_LOG_WARNING,
                                      "No codec could be found with name '%s'\n", forced_codec_name);
        else                   av_log(NULL, AV_LOG_WARNING,
```

```
                                       "No decoder could be found for
codec %s\n", avcodec_get_name(avctx->codec_id));
        ret = AVERROR(EINVAL);
        goto fail;
    }
```

这段主要是通过`avcodec_find_decoder`找到所需解码器（AVCodec）。如果用户有指定解码器，则设置`forced_codec_name`，并通过`avcodec_find_decoder_by_name`查找解码器。找到解码器后，就可以通过`avcodec_open2`打开解码器了。（forced_codec_name对应到音频、视频、字幕不同的传入的解码器名字，如果有设置，比如ffplay -acodec aac xx.flv, 此时audio_codec_name被设置为"aac"，则相应的forced_codec_name为“aac”）

最后，是一个大的switch-case:

```
switch (avctx->codec_type) {
    case AVMEDIA_TYPE_AUDIO:
        sample_rate    = avctx->sample_rate;
        nb_channels    = avctx->channels;
        channel_layout = avctx->channel_layout;

        /* prepare audio output 准备音频输出*/
        if ((ret = audio_open(is, channel_layout, nb_channels, sampl
e_rate, &is->audio_tgt)) < 0)
            goto fail;
        is->audio_hw_buf_size = ret;
        is->audio_src = is->audio_tgt;
        is->audio_buf_size  = 0;
        is->audio_buf_index = 0;

        /* init averaging filter 初始化averaging滤镜, 非audio master时
使用 */
        is->audio_diff_avg_coef  = exp(log(0.01) / AUDIO_DIFF_AVG_NB
);
        is->audio_diff_avg_count = 0;
        /* 由于我们没有精确的音频数据填充FIFO,故只有在大于该阈值时才进行校正音频
同步*/
        is->audio_diff_threshold = (double)(is->audio_hw_buf_size) /
is->audio_tgt.bytes_per_sec;

```

```
21          is->audio_stream = stream_index;    // 获取audio的stream索引
22          is->audio_st = ic->streams[stream_index];  // 获取audio的stre
    am指针
23          // 初始化ffplay封装的音频解码器
24          decoder_init(&is->auddec, avctx, &is->audioq, is->continue_r
    ead_thread);
25          if ((is->ic->iformat->flags & (AVFMT_NOBINSEARCH | AVFMT_NOG
    ENSEARCH | AVFMT_NO_BYTE_SEEK)) && !is->ic->iformat->read_seek) {
26              is->auddec.start_pts = is->audio_st->start_time;
27              is->auddec.start_pts_tb = is->audio_st->time_base;
28          }
29          // 启动音频解码线程
30          if ((ret = decoder_start(&is->auddec, audio_thread, "audio_d
    ecoder", is)) < 0)
31              goto out;
32          SDL_PauseAudioDevice(audio_dev, 0);
33          break;
34      case AVMEDIA_TYPE_VIDEO:
35          is->video_stream = stream_index;    // 获取video的stream索引
36          is->video_st = ic->streams[stream_index];// 获取video的stream
    指针
37          // 初始化ffplay封装的视频解码器
38          decoder_init(&is->viddec, avctx, &is->videoq, is->continue_r
    ead_thread);
39          // 启动视频频解码线程
40          if ((ret = decoder_start(&is->viddec, video_thread, "video_d
    ecoder", is)) < 0)
41              goto out;
42          is->queue_attachments_req = 1; // 使能请求mp3、aac等音频文件的封
    面
43          break;
44      case AVMEDIA_TYPE_SUBTITLE: // 视频是类似逻辑处理
45          is->subtitle_stream = stream_index;
46          is->subtitle_st = ic->streams[stream_index];
47
48          decoder_init(&is->subdec, avctx, &is->subtitleq, is->continu
    e_read_thread);
49          if ((ret = decoder_start(&is->subdec, subtitle_thread, "subt
    itle_decoder", is)) < 0)
50              goto out;
```

```
51          break;
52      default:
53          break;
54      }
```

即根据具体的流类型，作特定的初始化。但不论哪种流，基本步骤都包括了ffplay封装的解码器的初始化和启动解码器线程：

- **decoder_init 初始化解码器**
  - d->avctx = avctx; 绑定对应的解码器上下文
  - d->queue = queue; 绑定对应的packet队列
  - d->empty_queue_cond = empty_queue_cond; 绑定VideoState的continue_read_thread，当解码线程没有packet可读时唤醒read_thread赶紧读取数据
  - d->start_pts = AV_NOPTS_VALUE; 初始化start_pts
  - d->pkt_serial = -1; 初始化pkt_serial

- **decoder_start启动解码器**
  - packet_queue_start 启用对应的packet 队列
  - SDL_CreateThread 创建对应的解码线程

需要注意的是，对应音频而言，这里还初始化了输出参数，这块在讲音频输出的时候再重点展开。

以上是准备的工作，我们再来看for循环。

4.2 For循环读取数据

主要包括以下步骤：

1. 检测是否退出
2. 检测是否暂停/继续
3. 检测是否需要seek
4. 检测video是否为attached_pic
5. 检测队列是否已经有足够数据
6. 检测码流是否已经播放结束
   a. 是否循环播放
   b. 是否自动退出
7. 使用av_read_frame读取数据包
8. 检测数据是否读取完毕
9. 检测是否在播放范围内
10. 到这步才将数据插入对应的队列

1. 检测是否退出

```
// 1 检测是否退出
if (is->abort_request)
    break;
```

当退出事件发生时，调用do_exit() -> stream_close() -> 将is->abort_request置为1。退出该for循环，并最终退出该线程。

2. 检测是否暂停/继续

这里的暂停、继续只是对网络流有意义

```
 // 2 检测是否暂停/继续
if (is->paused != is->last_paused) {
    is->last_paused = is->paused;
    if (is->paused)
        is->read_pause_return = av_read_pause(ic); // 网络流的时候有用
    else
        av_read_play(ic);
}
```

比如rtsp

av_read_pause

```
/* pause the stream */
static int rtsp_read_pause(AVFormatContext *s)
{
    RTSPState *rt = s->priv_data;
    RTSPMessageHeader reply1, *reply = &reply1;

    if (rt->state != RTSP_STATE_STREAMING)
        return 0;
```

```
    else if (!(rt->server_type == RTSP_SERVER_REAL && rt->need_subscription)) {
        ff_rtsp_send_cmd(s, "PAUSE", rt->control_uri, NULL, reply, NULL);
        if (reply->status_code != RTSP_STATUS_OK) {
            return ff_rtsp_averror(reply->status_code, -1);
        }
    }
    rt->state = RTSP_STATE_PAUSED;
    return 0;
}
```

av_read_play

```
static int rtsp_read_play(AVFormatContext *s)
{
    RTSPState *rt = s->priv_data;
    RTSPMessageHeader reply1, *reply = &reply1;
    ......
    ff_rtsp_send_cmd(s, "PLAY", rt->control_uri, cmd, reply, NULL);
    ....
    rt->state = RTSP_STATE_STREAMING;
    return 0;
}
```

3. 检测是否需要seek

```
//  3 检测是否seek
if (is->seek_req) { // 是否有seek请求
    int64_t seek_target = is->seek_pos;
    int64_t seek_min    = is->seek_rel > 0 ? seek_target - is->seek_rel + 2: INT64_MIN;
    int64_t seek_max    = is->seek_rel < 0 ? seek_target - is->seek_rel - 2: INT64_MAX;
    // FIXME the +-2 is due to rounding being not done in the correct direction in generation
```

```
    //      of the seek_pos/seek_rel variables
    // 修复由于四舍五入，没有再seek_pos/seek_rel变量的正确方向上进行
    ret = avformat_seek_file(is->ic, -1, seek_min, seek_target, seek_max, is->seek_flags);
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR,
               "%s: error while seeking\n", is->ic->url);
    } else {
        /* seek的时候，要把原先的数据情况，并重启解码器，put flush_pkt的目的是告知解码线程需要
                 * reset decoder
                 */
        if (is->audio_stream >= 0) { // 如果有音频流
            packet_queue_flush(&is->audioq);    // 清空packet队列数据
            // 放入flush pkt, 用来开起新的一个播放序列, 解码器读取到flush_pkt也清空解码器
            packet_queue_put(&is->audioq, &flush_pkt);
        }
        if (is->subtitle_stream >= 0) { // 如果有字幕流
            packet_queue_flush(&is->subtitleq); // 和上同理
            packet_queue_put(&is->subtitleq, &flush_pkt);
        }
        if (is->video_stream >= 0) {    // 如果有视频流
            packet_queue_flush(&is->videoq);    // 和上同理
            packet_queue_put(&is->videoq, &flush_pkt);
        }
        if (is->seek_flags & AVSEEK_FLAG_BYTE) {
            set_clock(&is->extclk, NAN, 0);
        } else {
            set_clock(&is->extclk, seek_target / (double)AV_TIME_BASE, 0);
        }
    }
    is->seek_req = 0;
    is->queue_attachments_req = 1;
    is->eof = 0;
    if (is->paused)
        step_to_next_frame(is); // 如果本身是pause状态的则显示一帧继续暂停
}
```

主要的seek操作通过avformat_seek_file完成（该函数的具体使用在播放控制seek时做详解）。根据avformat_seek_file的返回值，如果seek成功，需要：

1. 清除PacketQueue的缓存，并放入一个flush_pkt。放入的flush_pkt可以让PacketQueue的serial增1，以区分seek前后的数据（PacketQueue函数的分析 ），该flush_pkt也会触发解码器重新刷新解码器缓存avcodec_flush_buffers()，以避免解码时使用了原来的buffer作为参考而出现马赛克。
2. 同步外部时钟。在后续音视频同步的课程中再具体分析。

这里还要注意：如果播放器本身是pause的状态，则
if (is->paused)
step_to_next_frame(is); // 如果本身是pause状态的则显示一帧继续暂停

4. 检测video是否为attached_pic

```
// 4 检测video是否为attached_pic
if (is->queue_attachments_req) {
    // 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，所以需要注意的是音频文件不一定只存在音频流本身
    if (is->video_st && is->video_st->disposition & AV_DISPOSITION_ATTACHED_PIC) {
        AVPacket copy = { 0 };
        if ((ret = av_packet_ref(&copy, &is->video_st->attached_pic)) < 0)
            goto fail;
        packet_queue_put(&is->videoq, &copy);
        packet_queue_put_nullpacket(&is->videoq, is->video_stream);
    }
    is->queue_attachments_req = 0;
}
```

`AV_DISPOSITION_ATTACHED_PIC` 是一个标志。如果一个流中含有这个标志的话，那么就是说这个流是 ***.mp3等** 文件中的一个 `Video Stream` 。并且该流只有一个 `AVPacket` ，也就是 `attached_pic` 。这个 `AVPacket` 中所存储的内容就是这个 ***.mp3等** 文件的封面图片。
因此，也可以很好的解释了文章开头提到的为什么 `st->disposition & AV_DISPOSITION_ATTACHED_PIC` 这个操作可以决定是否可以继续向缓冲区中添加 `AVPacket` 。

5. 检测队列是否已经有足够数据

音频、视频、字幕队列都不是无限大的，如果不加以限制一直往队列放入packet，那将导致队列占用大量的内存空间，影响系统的性能，所以必须对队列的缓存大小进行控制。
PacketQueue默认情况下会有大小限制，达到这个大小后，就需要等待10ms，以让消费者——解码线程能有时间消耗。

```
// 5 检测队列是否已经有足够数据
/* if the queue are full, no need to read more */
/* 缓存队列有足够的包，不需要继续读取数据 */
if (infinite_buffer<1 && // 缓冲区不是无限大
    (is->audioq.size + is->videoq.size + is->subtitleq.size > MAX_QUEUE_SIZE
            || (stream_has_enough_packets(is->audio_st, is->audio_stream, &is->audioq) &&
                stream_has_enough_packets(is->video_st, is->video_stream, &is->videoq) &&
                stream_has_enough_packets(is->subtitle_st, is->subtitle_stream, &is->subtitleq)))) {
    /* wait 10 ms */
    SDL_LockMutex(wait_mutex);
    // 如果没有唤醒则超时10ms退出，比如在seek操作时这里会被唤醒
    SDL_CondWaitTimeout(is->continue_read_thread, wait_mutex, 10);
    SDL_UnlockMutex(wait_mutex);
    continue;
}
```

缓冲区满有两种可能：

1. audioq，videoq，subtitleq三个PacketQueue的总字节数达到了MAX_QUEUE_SIZE（15M，为什么是15M？这里只是一个经验计算值，比如4K视频的码率以50Mbps计算，则15MB可以缓存2.4秒，从这么计算实际上如果我们真的是播放4K片源，15MB是偏小的数值，有些片源比较坑 同一个文件位置附近的pts差值超过5秒，此时如果视频要缓存5秒才能做同步，那15MB的缓存大小就不够了）
2. 音频、视频、字幕流都已有够用的包（stream_has_enough_packets），**注意：3者要同时成立**

第一种好理解，看下第二种中的stream_has_enough_packets：

```
static int stream_has_enough_packets(AVStream *st, int stream_id, PacketQueue *queue) {
    return stream_id < 0 || // 没有该流
           queue->abort_request || // 请求退出
           (st->disposition & AV_DISPOSITION_ATTACHED_PIC) || // 是ATT
```

```
  ACHED_PIC
5             queue->nb_packets > MIN_FRAMES // packet数>25
6                 && (!queue->duration ||     // 满足PacketQueue总时长为0
7                     av_q2d(st->time_base) * queue->duration > 1.0);
  //或总时长超过1s
8 }
```

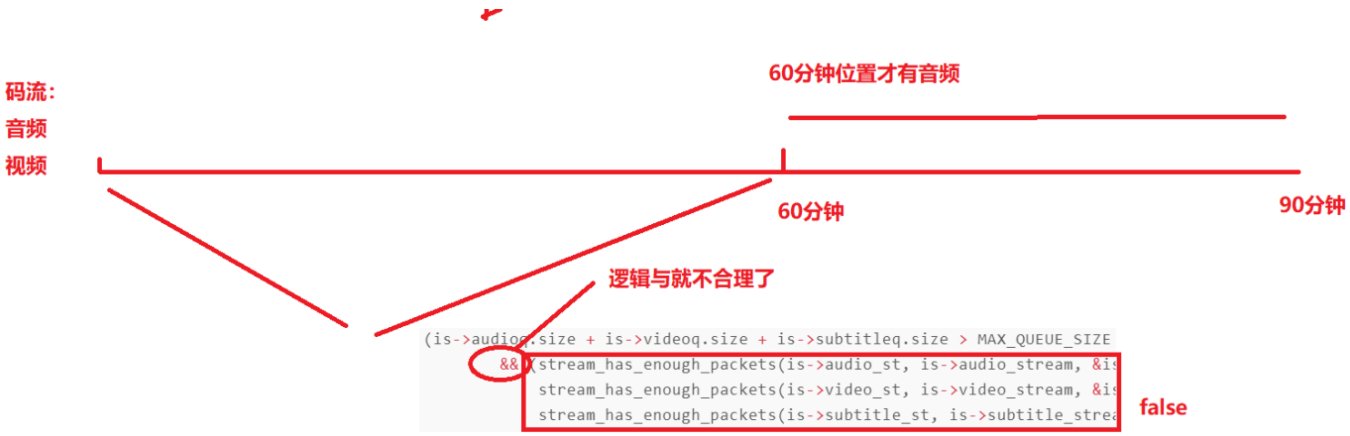
:

有这么几种情况包是够用的：

1. 流没有打开（stream_id < 0），没有相应的流返回逻辑true
2. 有退出请求（queue->abort_request）
3. 配置了AV_DISPOSITION_ATTACHED_PIC
4. packet队列内包个数大于MIN_FRAMES（>25），并满足PacketQueue总时长为0或总时长超过1s

思路：

- 总数据大小
- 每个packet队列的情况。



6. 检测码流是否已经播放结束

非暂停状态才进一步检测码流是否已经播放完毕（注意：**数据播放完毕**和码流**数据读取完毕**是两个概念。）

PacketQueue和FrameQueue都消耗完毕，才是真正的播放完毕

```
1 // 6 检测码流是否已经播放结束
2 if (!is->paused // 非暂停
3     && // 这里的执行是因为码流读取完毕后 插入空包所致
4     (!is->audio_st // 没有音频流
```

```
5       || (is->auddec.finished == is->audioq.serial // 或者音频播放完毕
6           && frame_queue_nb_remaining(&is->sampq) == 0))
7     && (!is->video_st // 没有视频流
8         || (is->viddec.finished == is->videoq.serial // 或者视频播放完毕
9             && frame_queue_nb_remaining(&is->pictq) == 0))) {
10    if (loop != 1           // a 是否循环播放
11        && (!loop || --loop)) {
12        stream_seek(is, start_time != AV_NOPTS_VALUE ? start_time : 0, 0, 0);
13    } else if (autoexit) {  // b 是否自动退出
14        ret = AVERROR_EOF;
15        goto fail;
16    }
17 }
```

这里判断播放已完成的条件需要同时满足满足：

1. 不在暂停状态
2. 音频未打开；或者打开了，但是解码已解完所有packet，自定义的解码器（decoder）serial等于PacketQueue的serial，并且FrameQueue中没有数据帧
   PacketQueue.serial -> packet.serail -> decoder.pkt_serial
   decoder.finished = decoder.pkt_serial

   is->auddec.finished == is->audioq.serial 最新的播放序列的packet都解码完毕
   frame_queue_nb_remaining(&is->sampq) == 0 对应解码后的数据也播放完毕
3. 视频未打开；或者打开了，但是解码已解完所有packet，自定义的解码器（decoder）serial等于PacketQueue的serial，并且FrameQueue中没有数据帧。

在确认目前码流已播放结束的情况下，用户有两个变量可以控制播放器行为：

1. loop: 控制播放次数（当前这次也算在内，也就是最小就是1次了），0表示无限次
2. autoexit：自动退出，也就是播放完成后自动退出。

loop条件简化的非常不友好，其意思是：如果loop==1，那么已经播了1次了，无需再seek重新播放；如果loop不是1，==0，随意，无限次循环；减1后还大于0（--loop），也允许循环

a. 是否循环播放

如果循环播放，即是将文件seek到起始位置 stream_seek(is, start_time != AV_NOPTS_VALUE ? start_time : 0, 0, 0);，这里讲的的起始位置不一定是从头开始，具体也要看用户是否指定了起始播放位置

b. 是否自动退出

如果播放完毕自动退出

7. 使用av_read_frame读取数据包

读取数据包很简单，但要注意传入的packet，av_read_frame不会释放其数据，而是每次都重新申请数据。

```
// 7.读取媒体数据，得到的是音视频分离后、解码前的数据
ret = av_read_frame(ic, pkt); // 调用不会释放pkt的数据，都是要自己去释放
```

8. 检测数据是否读取完毕

```
// 8 检测数据是否读取完毕
if (ret < 0) {
    if ((ret == AVERROR_EOF || avio_feof(ic->pb))
        && !is->eof)
    {
        // 插入空包说明码流数据读取完毕了，之前讲解码的时候说过刷空包是为了从解码器把所有帧都读出来
        if (is->video_stream >= 0)
            packet_queue_put_nullpacket(&is->videoq, is->video_stream);
        if (is->audio_stream >= 0)
            packet_queue_put_nullpacket(&is->audioq, is->audio_stream);
        if (is->subtitle_stream >= 0)
            packet_queue_put_nullpacket(&is->subtitleq, is->subtitle_stream);
        is->eof = 1;        // 文件读取完毕
    }
    if (ic->pb && ic->pb->error)
        break;
    SDL_LockMutex(wait_mutex);
    SDL_CondWaitTimeout(is->continue_read_thread, wait_mutex, 10);
    SDL_UnlockMutex(wait_mutex);
    continue;       // 继续循环 保证线程的运行，比如要seek到某个位置播放可以继续响应
```

```
} else {
    is->eof = 0;
}
```

数据读取完毕后，放对应音频、视频、字幕队列插入“空包”，以通知解码器冲刷buffer，将缓存的所有数据都解出来frame并去出来。
然后继续在for{}循环，直到收到退出命令，或者loop播放，或者seek等操作。

9. 检测是否在播放范围内

播放器可以设置：-ss 起始位置，以及 -t 播放时长

```
// 9 检测是否在播放范围内
/* check if packet is in play range specified by user, then queue, otherwise discard */
stream_start_time = ic->streams[pkt->stream_index]->start_time;// 获取流的起始时间
pkt_ts = pkt->pts == AV_NOPTS_VALUE ? pkt->dts : pkt->pts;// 获取packet的时间戳
// 这里的duration是在命令行时用来指定播放长度
pkt_in_play_range = duration == AV_NOPTS_VALUE ||
    (pkt_ts - (stream_start_time != AV_NOPTS_VALUE ? stream_start_time : 0)) *
                av_q2d(ic->streams[pkt->stream_index]->time_base) -
    (double)(start_time != AV_NOPTS_VALUE ? start_time : 0) / 1000000
    <= ((double)duration / 1000000);
```

从流获取的参数

- **stream_start_time**：是从当前流AVStream->start_time获取到的时间，如果没有定义具体的值则默认为AV_NOPTS_VALUE，即该值是无效的；那stream_start_time有意义的就是0值；
- **pkt_ts**：当前packet的时间戳，pts有效就用pts的，pts无效就用dts的；

ffplay播放的参数

duration： 使用"-t value"指定的播放时长，默认值AV_NOPTS_VALUE，即该值无效不用参考
start_time：使用“-ss value”指定播放的起始位置，默认AV_NOPTS_VALUE，即该值无效不用参考

pkt_in_play_range的值为0或1。

- 当没有指定duration播放时长时，很显然duration == AV_NOPTS_VALUE的逻辑值为1，所以pkt_in_play_range为1；
- 当duration被指定（-t value）且有效时，主要判断

```
(pkt_ts - (stream_start_time != AV_NOPTS_VALUE ? stream_start_time : 0)) *
                av_q2d(ic->streams[pkt->stream_index]->time_base) -
    (double)(start_time != AV_NOPTS_VALUE ? start_time : 0) / 1000000
    <= ((double)duration / 1000000);
```

实质就是当前时间戳 pkt_ts - start_time 是否 < duration，这里分为：
stream_start_time是否有效：有效就用实际值，无效就是从0开始
start_time 是否有效，有效就用实际值，无效就是从0开始
即是**pkt_ts - stream_start_time** - start_time < duration （为了简单，这里没有考虑时间单位）

10. 到这步才将数据插入对应的队列

```
// 10 将音视频数据分别送入相应的queue中
if (pkt->stream_index == is->audio_stream && pkt_in_play_range) {
    packet_queue_put(&is->audioq, pkt);
} else if (pkt->stream_index == is->video_stream && pkt_in_play_range
           && !(is->video_st->disposition & AV_DISPOSITION_ATTACHED_PIC)) {
           //printf("pkt pts:%ld, dts:%ld\n", pkt->pts, pkt->dts);
           packet_queue_put(&is->videoq, pkt);
} else if (pkt->stream_index == is->subtitle_stream && pkt_in_play_range) {
    packet_queue_put(&is->subtitleq, pkt);
} else {
    av_packet_unref(pkt);// 不入队列则直接释放数据
}
```

这里的代码就很直白了，将packet放入到对应的PacketQueue

4.3 退出线程处理

主要包括以下步骤：

1. 如果解复用器有打开则关闭avformat_close_input
2. 调用SDL_PushEvent发送退出事件FF_QUIT_EVENT
    a. 发送的FF_QUIT_EVENT退出播放事件由event_loop()函数相应，收到FF_QUIT_EVENT后调用do_exit()做退出操作。
3. 消耗互斥量wait_mutex

课后作业

- seek怎么做
- 数据播放完毕和码流数据读取完毕
- 循环播放
- 指定播放位置