

## 原 FFMpeg源代码简单分析：avformat\_find\_stream\_info()

2015年03月06日 11:15:37 阅读数：35964

=====

FFmpeg的库函数源代码分析文章列表：

### 【架构图】

[FFmpeg 源代码结构图 - 解码](#)

[FFmpeg 源代码结构图 - 编码](#)

### 【通用】

[FFmpeg 源代码简单分析：av\\_register\\_all\(\)](#)

[FFmpeg 源代码简单分析：avcodec\\_register\\_all\(\)](#)

[FFmpeg 源代码简单分析：内存的分配和释放（av\\_malloc\(\)、av\\_free\(\)等）](#)

[FFmpeg 源代码简单分析：常见结构体的初始化和销毁（AVFormatContext，AVFrame等）](#)

[FFmpeg 源代码简单分析：avio\\_open2\(\)](#)

[FFmpeg 源代码简单分析：av\\_find\\_decoder\(\)和av\\_find\\_encoder\(\)](#)

[FFmpeg 源代码简单分析：avcodec\\_open2\(\)](#)

[FFmpeg 源代码简单分析：avcodec\\_close\(\)](#)

### 【解码】

[图解 FFMPEG 打开媒体的函数 avformat\\_open\\_input](#)

[FFmpeg 源代码简单分析：avformat\\_open\\_input\(\)](#)

[FFmpeg 源代码简单分析：avformat\\_find\\_stream\\_info\(\)](#)

[FFmpeg 源代码简单分析：av\\_read\\_frame\(\)](#)

[FFmpeg 源代码简单分析：avcodec\\_decode\\_video2\(\)](#)

[FFmpeg 源代码简单分析：avformat\\_close\\_input\(\)](#)

### 【编码】

[FFmpeg 源代码简单分析：avformat\\_alloc\\_output\\_context2\(\)](#)

[FFmpeg 源代码简单分析：avformat\\_write\\_header\(\)](#)

[FFmpeg 源代码简单分析：avcodec\\_encode\\_video\(\)](#)

[FFmpeg 源代码简单分析：av\\_write\\_frame\(\)](#)

[FFmpeg 源代码简单分析：av\\_write\\_trailer\(\)](#)

### 【其它】

[FFmpeg 源代码简单分析：日志输出系统（av\\_log\(\)等）](#)

[FFmpeg 源代码简单分析：结构体成员管理系统 -AVClass](#)

[FFmpeg 源代码简单分析：结构体成员管理系统 -AVOption](#)

[FFmpeg 源代码简单分析：libswscale 的 sws\\_getContext\(\)](#)

[FFmpeg 源代码简单分析：libswscale 的 sws\\_scale\(\)](#)

[FFmpeg 源代码简单分析：libavdevice 的 avdevice\\_register\\_all\(\)](#)

[FFmpeg 源代码简单分析：libavdevice 的 gdigrab](#)

### 【脚本】

FFmpeg 源代码简单分析：makefile

FFmpeg 源代码简单分析：configure

【H.264】

FFmpeg 的 H.264 解码器源代码简单分析：概述

本文简单分析FFmpeg中一个常用的函数：avformat\_find\_stream\_info()。该函数可以读取一部分视音频数据并且获得一些相关的信息。avformat\_find\_stream\_info()的声明位于libavformat\avformat.h，如下所示。

```
[cpp]
1.  /**
2.   * Read packets of a media file to get stream information. This
3.   * is useful for file formats with no headers such as MPEG. This
4.   * function also computes the real framerate in case of MPEG-2 repeat
5.   * frame mode.
6.   * The logical file position is not changed by this function;
7.   * examined packets may be buffered for later processing.
8.   *
9.   * @param ic media file handle
10.  * @param options If non-NULL, an ic.nb_streams long array of pointers to
11.  *                dictionaries, where i-th member contains options for
12.  *                codec corresponding to i-th stream.
13.  *                On return each dictionary will be filled with options that were not found.
14.  * @return >=0 if OK, AVERROR_xxx on error
15.  *
16.  * @note this function isn't guaranteed to open all the codecs, so
17.  *        options being non-empty at return is a perfectly normal behavior.
18.  *
19.  * @todo Let the user decide somehow what information is needed so that
20.  *        we do not waste time getting stuff the user does not need.
21.  */
22. int avformat_find_stream_info(AVFormatContext *ic, AVDictionary **options);
```

简单解释一下它的参数的含义：

ic：输入的AVFormatContext。

options：额外的选项，目前没有深入研究过。

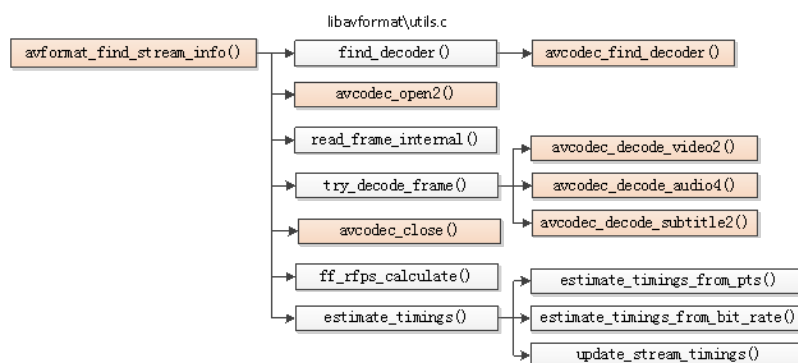
函数正常执行后返回值大于等于0。

该函数最典型的例子可以参考：[最简单的基于FFMPEG+SDL的视频播放器 ver2](#)（采用SDL2.0）

PS：由于该函数比较复杂，所以只看了一部分代码，以后有时间再进一步分析。

## 函数调用关系图

函数的调用关系如下图所示。



雷霄骅 (LeiXiaohua)  
leixiaohua1020@126.com  
<http://blog.csdn.net/leixiaohua1020>

## avformat\_find\_stream\_info()

avformat\_find\_stream\_info()的定义位于libavformatutils.c。它的代码比较长，如下所示。

```
[cpp]
1. int avformat_find_stream_info(AVFormatContext *ic, AVDictionary **options)
2. {
3.     int i, count, ret = 0;
4.     ...
```

```

3.     int i, count, ret = 0, j;
4.     int64_t read_size;
5.     AVStream *st;
6.     AVPacket pkt1, *pkt;
7.     int64_t old_offset = avio_tell(ic->pb);
8.     // new streams might appear, no options for those
9.     int orig_nb_streams = ic->nb_streams;
10.    int flush_codecs;
11.    int64_t max_analyze_duration = ic->max_analyze_duration2;
12.    int64_t probesize = ic->probesize2;
13.
14.
15.    if (!max_analyze_duration)
16.        max_analyze_duration = ic->max_analyze_duration;
17.    if (ic->probesize)
18.        probesize = ic->probesize;
19.    flush_codecs = probesize > 0;
20.
21.
22.    av_opt_set(ic, "skip_clear", "1", AV_OPT_SEARCH_CHILDREN);
23.
24.
25.    if (!max_analyze_duration) {
26.        if (!strcmp(ic->iformat->name, "flv") && !(ic->ctx_flags & AVFMTCTX_NOHEADER)) {
27.            max_analyze_duration = 10*AV_TIME_BASE;
28.        } else
29.            max_analyze_duration = 5*AV_TIME_BASE;
30.    }
31.
32.
33.    if (ic->pb)
34.        av_log(ic, AV_LOG_DEBUG, "Before avformat_find_stream_info() pos: %\"PRIu64\" bytes read:%\"PRIu64\" seeks:%d\\n",
35.            avio_tell(ic->pb), ic->pb->bytes_read, ic->pb->seek_count);
36.
37.
38.    for (i = 0; i < ic->nb_streams; i++) {
39.        const AVCodec *codec;
40.        AVDictionary *thread_opt = NULL;
41.        st = ic->streams[i];
42.
43.
44.        if (st->codec->codec_type == AVMEDIA_TYPE_VIDEO ||
45.            st->codec->codec_type == AVMEDIA_TYPE_SUBTITLE) {
46.            /* if (!st->time_base.num)
47.               st->time_base = */
48.            if (!st->codec->time_base.num)
49.                st->codec->time_base = st->time_base;
50.        }
51.        // only for the split stuff
52.        if (!st->parser && !(ic->flags & AVFMT_FLAG_NOPARSE)) {
53.            st->parser = av_parser_init(st->codec->codec_id);
54.            if (st->parser) {
55.                if (st->need_parsing == AVSTREAM_PARSE_HEADERS) {
56.                    st->parser->flags |= PARSE_FLAG_COMPLETE_FRAMES;
57.                } else if (st->need_parsing == AVSTREAM_PARSE_FULL_RAW) {
58.                    st->parser->flags |= PARSE_FLAG_USE_CODEC_TS;
59.                }
60.            } else if (st->need_parsing) {
61.                av_log(ic, AV_LOG_VERBOSE, "parser not found for codec "
62.                    "%s, packets or times may be invalid.\\n",
63.                    avcodec_get_name(st->codec->codec_id));
64.            }
65.        }
66.        codec = find_decoder(ic, st, st->codec->codec_id);
67.
68.
69.        /* Force thread count to 1 since the H.264 decoder will not extract
70.         * SPS and PPS to extradata during multi-threaded decoding. */
71.        av_dict_set(options ? &options[i] : &thread_opt, "threads", "1", 0);
72.
73.
74.        if (ic->codec_whitelist)
75.            av_dict_set(options ? &options[i] : &thread_opt, "codec_whitelist", ic->codec_whitelist, 0);
76.
77.
78.        /* Ensure that subtitle_header is properly set. */
79.        if (st->codec->codec_type == AVMEDIA_TYPE_SUBTITLE
80.            && codec && !st->codec->codec) {
81.            if (avcodec_open2(st->codec, codec, options ? &options[i] : &thread_opt) < 0)
82.                av_log(ic, AV_LOG_WARNING,
83.                    "Failed to open codec in av_find_stream_info\\n");
84.        }
85.
86.
87.        // Try to just open decoders, in case this is enough to get parameters.
88.        if (!has_codec_parameters(st, NULL) && st->request_probe <= 0) {
89.            if (codec && !st->codec->codec)
90.                if (avcodec_open2(st->codec, codec, options ? &options[i] : &thread_opt) < 0)
91.                    av_log(ic, AV_LOG_WARNING,
92.                        "Failed to open codec in av_find_stream_info\\n");
93.        }
94.        if (!options)

```

```

95.         av_dict_free(&thread_opt);
96.     }
97.
98.
99.     for (i = 0; i < ic->nb_streams; i++) {
100. #if FF_API_R_FRAME_RATE
101.         ic->streams[i]->info->last_dts = AV_NOPTS_VALUE;
102. #endif
103.         ic->streams[i]->info->fps_first_dts = AV_NOPTS_VALUE;
104.         ic->streams[i]->info->fps_last_dts = AV_NOPTS_VALUE;
105.     }
106.
107.
108.     count = 0;
109.     read_size = 0;
110.     for (;;) {
111.         if (ff_check_interrupt(&ic->interrupt_callback)) {
112.             ret = AVERROR_EXIT;
113.             av_log(ic, AV_LOG_DEBUG, "interrupted\n");
114.             break;
115.         }
116.
117.
118.         /* check if one codec still needs to be handled */
119.         for (i = 0; i < ic->nb_streams; i++) {
120.             int fps_analyze_framecount = 20;
121.
122.
123.             st = ic->streams[i];
124.             if (!has_codec_parameters(st, NULL))
125.                 break;
126.             /* If the timebase is coarse (like the usual millisecond precision
127.              * of mkv), we need to analyze more frames to reliably arrive at
128.              * the correct fps. */
129.             if (av_q2d(st->time_base) > 0.0005)
130.                 fps_analyze_framecount *= 2;
131.             if (!tb_unreliable(st->codec))
132.                 fps_analyze_framecount = 0;
133.             if (ic->fps_probe_size >= 0)
134.                 fps_analyze_framecount = ic->fps_probe_size;
135.             if (st->disposition & AV_DISPOSITION_ATTACHED_PIC)
136.                 fps_analyze_framecount = 0;
137.             /* variable fps and no guess at the real fps */
138.             if (!(st->r_frame_rate.num && st->avg_frame_rate.num) &&
139.                 st->info->duration_count < fps_analyze_framecount &&
140.                 st->codec->codec_type == AVMEDIA_TYPE_VIDEO)
141.                 break;
142.             if (st->parser && st->parser->parser->split &&
143.                 !st->codec->extradata)
144.                 break;
145.             if (st->first_dts == AV_NOPTS_VALUE &&
146.                 !(ic->iformat->flags & AVFMT_NOTIMESTAMPS) &&
147.                 st->codec_info_nb_frames < ic->max_ts_probe &&
148.                 (st->codec->codec_type == AVMEDIA_TYPE_VIDEO ||
149.                  st->codec->codec_type == AVMEDIA_TYPE_AUDIO))
150.                 break;
151.         }
152.         if (i == ic->nb_streams) {
153.             /* NOTE: If the format has no header, then we need to read some
154.              * packets to get most of the streams, so we cannot stop here. */
155.             if (!(ic->ctx_flags & AVFMTCTX_NOHEADER)) {
156.                 /* If we found the info for all the codecs, we can stop. */
157.                 ret = count;
158.                 av_log(ic, AV_LOG_DEBUG, "All info found\n");
159.                 flush_codecs = 0;
160.                 break;
161.             }
162.         }
163.         /* We did not get all the codec info, but we read too much data. */
164.         if (read_size >= probesize) {
165.             ret = count;
166.             av_log(ic, AV_LOG_DEBUG,
167.                 "Probe buffer size limit of %"PRIu64" bytes reached\n", probesize);
168.             for (i = 0; i < ic->nb_streams; i++)
169.                 if (!ic->streams[i]->r_frame_rate.num &&
170.                     ic->streams[i]->info->duration_count <= 1 &&
171.                     ic->streams[i]->codec->codec_type == AVMEDIA_TYPE_VIDEO &&
172.                     strcmp(ic->iformat->name, "image2"))
173.                         av_log(ic, AV_LOG_WARNING,
174.                             "Stream #%d: not enough frames to estimate rate; "
175.                             "consider increasing probesize\n", i);
176.             break;
177.         }
178.
179.
180.         /* NOTE: A new stream can be added there if no header in file
181.          * (AVFMTCTX_NOHEADER). */
182.         ret = read_frame_internal(ic, &pkt1);
183.         if (ret == AVERROR(EAGAIN))
184.             continue;
185.

```

```

186.
187.     if (ret < 0) {
188.         /* EOF or error*/
189.         break;
190.     }
191.
192.
193.     if (ic->flags & AVFMT_FLAG_NOBUFFER)
194.         free_packet_buffer(&ic->packet_buffer, &ic->packet_buffer_end);
195.     {
196.         pkt = add_to_pktbuf(&ic->packet_buffer, &pkt1,
197.                             &ic->packet_buffer_end);
198.         if (!pkt) {
199.             ret = AVERROR(ENOMEM);
200.             goto find_stream_info_err;
201.         }
202.         if ((ret = av_dup_packet(pkt)) < 0)
203.             goto find_stream_info_err;
204.     }
205.
206.
207.     st = ic->streams[pkt->stream_index];
208.     if (!(st->disposition & AV_DISPOSITION_ATTACHED_PIC))
209.         read_size += pkt->size;
210.
211.
212.     if (pkt->dts != AV_NOPTS_VALUE && st->codec_info_nb_frames > 1) {
213.         /* check for non-increasing dts */
214.         if (st->info->fps_last_dts != AV_NOPTS_VALUE &&
215.             st->info->fps_last_dts >= pkt->dts) {
216.             av_log(ic, AV_LOG_DEBUG,
217.                 "Non-increasing DTS in stream %d: packet %d with DTS "
218.                 "%PRIu64", packet %d with DTS %PRIu64"\n",
219.                 st->index, st->info->fps_last_dts_idx,
220.                 st->info->fps_last_dts, st->codec_info_nb_frames,
221.                 pkt->dts);
222.             st->info->fps_first_dts =
223.                 st->info->fps_last_dts = AV_NOPTS_VALUE;
224.         }
225.         /* Check for a discontinuity in dts. If the difference in dts
226.          * is more than 1000 times the average packet duration in the
227.          * sequence, we treat it as a discontinuity. */
228.         if (st->info->fps_last_dts != AV_NOPTS_VALUE &&
229.             st->info->fps_last_dts_idx > st->info->fps_first_dts_idx &&
230.             (pkt->dts - st->info->fps_last_dts) / 1000 >
231.             (st->info->fps_last_dts - st->info->fps_first_dts) /
232.             (st->info->fps_last_dts_idx - st->info->fps_first_dts_idx)) {
233.             av_log(ic, AV_LOG_WARNING,
234.                 "DTS discontinuity in stream %d: packet %d with DTS "
235.                 "%PRIu64", packet %d with DTS %PRIu64"\n",
236.                 st->index, st->info->fps_last_dts_idx,
237.                 st->info->fps_last_dts, st->codec_info_nb_frames,
238.                 pkt->dts);
239.             st->info->fps_first_dts =
240.                 st->info->fps_last_dts = AV_NOPTS_VALUE;
241.         }
242.
243.
244.         /* update stored dts values */
245.         if (st->info->fps_first_dts == AV_NOPTS_VALUE) {
246.             st->info->fps_first_dts = pkt->dts;
247.             st->info->fps_first_dts_idx = st->codec_info_nb_frames;
248.         }
249.         st->info->fps_last_dts = pkt->dts;
250.         st->info->fps_last_dts_idx = st->codec_info_nb_frames;
251.     }
252.     if (st->codec_info_nb_frames > 1) {
253.         int64_t t = 0;
254.
255.
256.         if (st->time_base.den > 0)
257.             t = av_rescale_q(st->info->codec_info_duration, st->time_base, AV_TIME_BASE_Q);
258.         if (st->avg_frame_rate.num > 0)
259.             t = FFMAX(t, av_rescale_q(st->codec_info_nb_frames, av_inv_q(st->avg_frame_rate), AV_TIME_BASE_Q));
260.
261.
262.         if (t == 0
263.             && st->codec_info_nb_frames > 30
264.             && st->info->fps_first_dts != AV_NOPTS_VALUE
265.             && st->info->fps_last_dts != AV_NOPTS_VALUE)
266.             t = FFMAX(t, av_rescale_q(st->info->fps_last_dts - st->info->fps_first_dts, st->time_base, AV_TIME_BASE_Q));
267.
268.
269.         if (t >= max_analyze_duration) {
270.             av_log(ic, AV_LOG_VERBOSE, "max_analyze_duration %PRIu64" reached at %PRIu64" microseconds\n",
271.                 max_analyze_duration,
272.                 t);
273.             if (ic->flags & AVFMT_FLAG_NOBUFFER)
274.                 av_packet_unref(pkt);
275.             break;
276.         }

```

```

277.         if (pkt->duration) {
278.             st->info->codec_info_duration += pkt->duration;
279.             st->info->codec_info_duration_fields += st->parser && st->need_parsing && st->codec->ticks_per_frame == 2 ? st->parse
r->repeat_pict + 1 : 2;
280.         }
281.     }
282. #if FF_API_R_FRAME_RATE
283.     if (st->codec->codec_type == AVMEDIA_TYPE_VIDEO)
284.         ff_rfps_add_frame(ic, st, pkt->dts);
285. #endif
286.     if (st->parser && st->parser->parser->split && !st->codec->extradata) {
287.         int i = st->parser->parser->split(st->codec, pkt->data, pkt->size);
288.         if (i > 0 && i < FF_MAX_EXTRADATA_SIZE) {
289.             if (ff_alloc_extradata(st->codec, i))
290.                 return AVERROR(ENOMEM);
291.             memcpy(st->codec->extradata, pkt->data,
292.                   st->codec->extradata_size);
293.         }
294.     }
295.
296.
297.     /* If still no information, we try to open the codec and to
298.      * decompress the frame. We try to avoid that in most cases as
299.      * it takes longer and uses more memory. For MPEG-4, we need to
300.      * decompress for QuickTime.
301.      *
302.      * If CODEC_CAP_CHANNEL_CONF is set this will force decoding of at
303.      * least one frame of codec data, this makes sure the codec initializes
304.      * the channel configuration and does not only trust the values from
305.      * the container. */
306.     try_decode_frame(ic, st, pkt,
307.                     (options && i < orig_nb_streams) ? &options[i] : NULL);
308.
309.
310.     if (ic->flags & AVFMT_FLAG_NOBUFFER)
311.         av_packet_unref(pkt);
312.
313.
314.     st->codec_info_nb_frames++;
315.     count++;
316. }
317.
318.
319. if (flush_codecs) {
320.     AVPacket empty_pkt = { 0 };
321.     int err = 0;
322.     av_init_packet(&empty_pkt);
323.
324.
325.     for (i = 0; i < ic->nb_streams; i++) {
326.
327.
328.         st = ic->streams[i];
329.
330.
331.         /* flush the decoders */
332.         if (st->info->found_decoder == 1) {
333.             do {
334.                 err = try_decode_frame(ic, st, &empty_pkt,
335.                                       (options && i < orig_nb_streams)
336.                                       ? &options[i] : NULL);
337.             } while (err > 0 && !has_codec_parameters(st, NULL));
338.
339.
340.             if (err < 0) {
341.                 av_log(ic, AV_LOG_INFO,
342.                       "decoding for stream %d failed\n", st->index);
343.             }
344.         }
345.     }
346. }
347.
348.
349. // close codecs which were opened in try_decode_frame()
350. for (i = 0; i < ic->nb_streams; i++) {
351.     st = ic->streams[i];
352.     avcodec_close(st->codec);
353. }
354.
355.
356. ff_rfps_calculate(ic);
357.
358.
359. for (i = 0; i < ic->nb_streams; i++) {
360.     st = ic->streams[i];
361.     if (st->codec->codec_type == AVMEDIA_TYPE_VIDEO) {
362.         if (st->codec->codec_id == AV_CODEC_ID_RAWVIDEO && !st->codec->codec_tag && !st->codec->bits_per_coded_sample) {
363.             uint32_t tag = avcodec_pix_fmt_to_codec_tag(st->codec->pix_fmt);
364.             if (avpriv_find_pix_fmt(avpriv_get_raw_pix_fmt_tags(), tag) == st->codec->pix_fmt)
365.                 st->codec->codec_tag = tag;
366.         }

```

```

367.
368.
369.     /* estimate average framerate if not set by demuxer */
370.     if (st->info->codec_info_duration_fields &&
371.         !st->avg_frame_rate.num &&
372.         st->info->codec_info_duration) {
373.         int best_fps = 0;
374.         double best_error = 0.01;
375.
376.
377.         if (st->info->codec_info_duration >= INT64_MAX / st->time_base.num / 2 ||
378.             st->info->codec_info_duration_fields >= INT64_MAX / st->time_base.den ||
379.             st->info->codec_info_duration < 0)
380.             continue;
381.         av_reduce(&st->avg_frame_rate.num, &st->avg_frame_rate.den,
382.                   st->info->codec_info_duration_fields * (int64_t) st->time_base.den,
383.                   st->info->codec_info_duration * 2 * (int64_t) st->time_base.num, 60000);
384.
385.
386.         /* Round guessed framerate to a "standard" framerate if it's
387.          * within 1% of the original estimate. */
388.         for (j = 0; j < MAX_STD_TIMEBASES; j++) {
389.             AVRational std_fps = { get_std_framerate(j), 12 * 1001 };
390.             double error = fabs(av_q2d(st->avg_frame_rate) /
391.                                 av_q2d(std_fps) - 1);
392.
393.
394.             if (error < best_error) {
395.                 best_error = error;
396.                 best_fps = std_fps.num;
397.             }
398.         }
399.         if (best_fps)
400.             av_reduce(&st->avg_frame_rate.num, &st->avg_frame_rate.den,
401.                       best_fps, 12 * 1001, INT_MAX);
402.     }
403.
404.
405.     if (!st->r_frame_rate.num) {
406.         if ( st->codec->time_base.den * (int64_t) st->time_base.num
407.             <= st->codec->time_base.num * st->codec->ticks_per_frame * (int64_t) st->time_base.den) {
408.             st->r_frame_rate.num = st->codec->time_base.den;
409.             st->r_frame_rate.den = st->codec->time_base.num * st->codec->ticks_per_frame;
410.         } else {
411.             st->r_frame_rate.num = st->time_base.den;
412.             st->r_frame_rate.den = st->time_base.num;
413.         }
414.     }
415. } else if (st->codec->codec_type == AVMEDIA_TYPE_AUDIO) {
416.     if (!st->codec->bits_per_coded_sample)
417.         st->codec->bits_per_coded_sample =
418.             av_get_bits_per_sample(st->codec->codec_id);
419.     /* set stream disposition based on audio service type
420.     switch (st->codec->audio_service_type) {
421.     case AV_AUDIO_SERVICE_TYPE_EFFECTS:
422.         st->disposition = AV_DISPOSITION_CLEAN_EFFECTS;
423.         break;
424.     case AV_AUDIO_SERVICE_TYPE_VISUALLY_IMPAIRED:
425.         st->disposition = AV_DISPOSITION_VISUAL_IMPAIRED;
426.         break;
427.     case AV_AUDIO_SERVICE_TYPE_HEARING_IMPAIRED:
428.         st->disposition = AV_DISPOSITION_HEARING_IMPAIRED;
429.         break;
430.     case AV_AUDIO_SERVICE_TYPE_COMMENTARY:
431.         st->disposition = AV_DISPOSITION_COMMENT;
432.         break;
433.     case AV_AUDIO_SERVICE_TYPE_KARAOKE:
434.         st->disposition = AV_DISPOSITION_KARAOKE;
435.         break;
436.     }
437. }
438. }
439.
440.
441. if (probesize)
442.     estimate_timings(ic, old_offset);
443.
444.
445. av_opt_set(ic, "skip_clear", "0", AV_OPT_SEARCH_CHILDREN);
446.
447.
448. if (ret >= 0 && ic->nb_streams)
449.     /* We could not have all the codec parameters before EOF. */
450.     ret = -1;
451. for (i = 0; i < ic->nb_streams; i++) {
452.     const char *errmsg;
453.     st = ic->streams[i];
454.     if (!has_codec_parameters(st, &errmsg)) {
455.         char buf[256];
456.         avcodec_string(buf, sizeof(buf), st->codec, 0);
457.         av_log(ic, AV_LOG_WARNING,

```

```

458.         "Could not find codec parameters for stream %d (%s): %s\n"
459.         "Consider increasing the value for the 'analyzeduration' and 'probesize' options\n",
460.         i, buf, errmsg);
461.     } else {
462.         ret = 0;
463.     }
464. }
465.
466.
467.     compute_chapters_end(ic);
468.
469.
470. find_stream_info_err:
471.     for (i = 0; i < ic->nb_streams; i++) {
472.         st = ic->streams[i];
473.         if (ic->streams[i]->codec->codec_type != AVMEDIA_TYPE_AUDIO)
474.             ic->streams[i]->codec->thread_count = 0;
475.         if (st->info)
476.             av_freep(&st->info->duration_error);
477.         av_freep(&ic->streams[i]->info);
478.     }
479.     if (ic->pb)
480.         av_log(ic, AV_LOG_DEBUG, "After avformat_find_stream_info() pos: %\"PRId64\" bytes read: \"%\"PRId64\" seeks: %d frames: %d\n",
481.             avio_tell(ic->pb), ic->pb->bytes_read, ic->pb->seek_count, count);
482.     return ret;
483. }

```

由于avformat\_find\_stream\_info()代码比较长，难以全部分析，在这里只能简单记录一下它的要点。该函数主要用于给每个媒体流（音频/视频）的AVStream结构体赋值。我们大致浏览一下这个函数的代码，会发现它其实已经实现了解码器的查找，解码器的打开，视音频帧的读取，视音频帧的解码等工作。换句话说，该函数实际上已经“走通”的解码的整个流程。下面看一下除了成员变量赋值之外，该函数的几个关键流程。

1. 查找解码器：find\_decoder()
2. 打开解码器：avcodec\_open2()
3. 读取完整的一帧压缩编码的数据：read\_frame\_internal()  
注：av\_read\_frame()内部实际上就是调用的read\_frame\_internal()。
4. 解码一些压缩编码数据：try\_decode\_frame()

下面选择上述流程中几个关键函数的代码简单看一下。

## find\_decoder()

find\_decoder()用于找到合适的解码器，它的定义如下所示。

```

1.  static const AVCodec *find_decoder(AVFormatContext *s, AVStream *st, enum AVCodecID codec_id)
2.  {
3.      if (st->codec->codec)
4.          return st->codec->codec;
5.
6.
7.      switch (st->codec->codec_type) {
8.      case AVMEDIA_TYPE_VIDEO:
9.          if (s->video_codec) return s->video_codec;
10.         break;
11.      case AVMEDIA_TYPE_AUDIO:
12.          if (s->audio_codec) return s->audio_codec;
13.          break;
14.      case AVMEDIA_TYPE_SUBTITLE:
15.          if (s->subtitle_codec) return s->subtitle_codec;
16.          break;
17.      }
18.
19.
20.      return avcodec_find_decoder(codec_id);
21. }

```

从代码中可以看出，如果指定的AVStream已经包含了解码器，则函数什么也不做直接返回。否则调用avcodec\_find\_decoder()获取解码器。avcodec\_find\_decoder()是一个FFmpeg的API函数，在这里不做详细分析。

## read\_frame\_internal()

read\_frame\_internal()的功能是读取一帧压缩码流数据。FFmpeg的API函数av\_read\_frame()内部调用的就是read\_frame\_internal()。有关这方面的知识可以参考文章：

[ffmpeg 源代码简单分析：av\\_read\\_frame\(\)](#)

因此，可以认为read\_frame\_internal()和av\_read\_frame()的功能基本上是等同的。

## try\_decode\_frame()

try\_decode\_frame()的功能可以从字面上的意思进行理解：“尝试解码一些帧”，它的定义如下所示。



[cpp]  

```
1.  /* returns 1 or 0 if or if not decoded data was returned, or a negative error */
2.  static int try_decode_frame(AVFormatContext *s, AVStream *st, AVPacket *avpkt,
3.                             AVDictionary **options)
4.  {
5.      const AVCodec *codec;
6.      int got_picture = 1, ret = 0;
7.      AVFrame *frame = av_frame_alloc();
8.      AVSubtitle subtitle;
9.      AVPacket pkt = *avpkt;
10.
11.
12.      if (!frame)
13.          return AVERROR(ENOMEM);
14.
15.
16.      if (!avcodec_is_open(st->codec) &&
17.          st->info->found_decoder <= 0 &&
18.          (st->codec->codec_id != -st->info->found_decoder || !st->codec->codec_id)) {
19.          AVDictionary *thread_opt = NULL;
20.
21.
22.          codec = find_decoder(s, st, st->codec->codec_id);
23.
24.
25.          if (!codec) {
26.              st->info->found_decoder = -st->codec->codec_id;
27.              ret = -1;
28.              goto fail;
29.          }
30.
31.
32.          /* Force thread count to 1 since the H.264 decoder will not extract
33.           * SPS and PPS to extradata during multi-threaded decoding. */
34.          av_dict_set(options ? options : &thread_opt, "threads", "1", 0);
35.          if (s->codec_whitelist)
36.              av_dict_set(options ? options : &thread_opt, "codec_whitelist", s->codec_whitelist, 0);
37.          ret = avcodec_open2(st->codec, codec, options ? options : &thread_opt);
38.          if (!options)
39.              av_dict_free(&thread_opt);
40.          if (ret < 0) {
41.              st->info->found_decoder = -st->codec->codec_id;
42.              goto fail;
43.          }
44.          st->info->found_decoder = 1;
45.      } else if (!st->info->found_decoder)
46.          st->info->found_decoder = 1;
47.
48.
49.      if (st->info->found_decoder < 0) {
50.          ret = -1;
51.          goto fail;
52.      }
53.
54.
55.      while ((pkt.size > 0 || (!pkt.data && got_picture)) &&
56.             ret >= 0 &&
57.             (!has_codec_parameters(st, NULL) || !has_decode_delay_been_guessed(st) ||
58.              (!st->codec_info_nb_frames &&
59.               st->codec->capabilities & CODEC_CAP_CHANNEL_CONF))) {
60.          got_picture = 0;
61.          switch (st->codec->codec_type) {
62.              case AVMEDIA_TYPE_VIDEO:
63.                  ret = avcodec_decode_video2(st->codec, frame,
64.                                              &got_picture, &pkt);
65.                  break;
66.              case AVMEDIA_TYPE_AUDIO:
67.                  ret = avcodec_decode_audio4(st->codec, frame, &got_picture, &pkt);
68.                  break;
69.              case AVMEDIA_TYPE_SUBTITLE:
70.                  ret = avcodec_decode_subtitle2(st->codec, &subtitle,
71.                                                &got_picture, &pkt);
72.                  ret = pkt.size;
73.                  break;
74.              default:
75.                  break;
76.          }
77.          if (ret >= 0) {
78.              if (got_picture)
79.                  st->nb_decoded_frames++;
80.              pkt.data += ret;
81.              pkt.size -= ret;
82.              ret = got_picture;
83.          }
84.      }
85.
86.
87.      if (!pkt.data && !got_picture)
88.          ret = -1;
89.
90.
```

```

90.
91.     fail:
92.         av_frame_free(&frame);
93.         return ret;
94.     }

```

从try\_decode\_frame()的定义可以看出，该函数首先判断视音频流的解码器是否已经打开，如果没有打开的话，先打开相应的解码器。接下来根据视音频流类型的不同，调用不同的解码函数进行解码：视频流调用avcodec\_decode\_video2()，音频流调用avcodec\_decode\_audio4()，字幕流调用avcodec\_decode\_subtitle2()。解码的循环会一直持续下去直到满足了while()的所有条件。

while()语句的条件中有一个has\_codec\_parameters()函数，用于判断AVStream中的成员变量是否都已经设置完毕。该函数在avformat\_find\_stream\_info()中的多个地方被使用过。下面简单看一下该函数。

## has\_codec\_parameters()

has\_codec\_parameters()用于检查AVStream中的成员变量是否都已经设置完毕。函数的定义如下。

```

[cpp]
1. static int has_codec_parameters(AVStream *st, const char **errmsg_ptr)
2. {
3.     AVCodecContext *avctx = st->codec;
4.
5.
6. #define FAIL(errmsg) do {                                \
7.     if (errmsg_ptr)                                       \
8.         *errmsg_ptr = errmsg;                            \
9.     return 0;                                           \
10. } while (0)
11.
12.
13. if ( avctx->codec_id == AV_CODEC_ID_NONE
14.     && avctx->codec_type != AVMEDIA_TYPE_DATA)
15.     FAIL("unknown codec");
16. switch (avctx->codec_type) {
17. case AVMEDIA_TYPE_AUDIO:
18.     if (!avctx->frame_size && determinable_frame_size(avctx))
19.         FAIL("unspecified frame size");
20.     if (st->info->found_decoder >= 0 &&
21.         avctx->sample_fmt == AV_SAMPLE_FMT_NONE)
22.         FAIL("unspecified sample format");
23.     if (!avctx->sample_rate)
24.         FAIL("unspecified sample rate");
25.     if (!avctx->channels)
26.         FAIL("unspecified number of channels");
27.     if (st->info->found_decoder >= 0 && !st->nb_decoded_frames && avctx->codec_id == AV_CODEC_ID_DTS)
28.         FAIL("no decodable DTS frames");
29.     break;
30. case AVMEDIA_TYPE_VIDEO:
31.     if (!avctx->width)
32.         FAIL("unspecified size");
33.     if (st->info->found_decoder >= 0 && avctx->pix_fmt == AV_PIX_FMT_NONE)
34.         FAIL("unspecified pixel format");
35.     if (st->codec->codec_id == AV_CODEC_ID_RV30 || st->codec->codec_id == AV_CODEC_ID_RV40)
36.         if (!st->sample_aspect_ratio.num && !st->codec->sample_aspect_ratio.num && !st->codec_info_nb_frames)
37.             FAIL("no frame in rv30/40 and no sar");
38.     break;
39. case AVMEDIA_TYPE_SUBTITLE:
40.     if (avctx->codec_id == AV_CODEC_ID_HDMV_PGS_SUBTITLE && !avctx->width)
41.         FAIL("unspecified size");
42.     break;
43. case AVMEDIA_TYPE_DATA:
44.     if (avctx->codec_id == AV_CODEC_ID_NONE) return 1;
45. }
46.
47.
48. return 1;
49. }

```

## estimate\_timings()

estimate\_timings()位于avformat\_find\_stream\_info()最后面，用于估算AVFormatContext以及AVStream的时长duration。它的代码如下所示。

```

1. static void estimate_timings(AVFormatContext *ic, int64_t old_offset)
2. {
3.     int64_t file_size;
4.
5.
6.     /* get the file size, if possible */
7.     if (ic->iformat->flags & AVFMT_NOFILE) {
8.         file_size = 0;
9.     } else {
10.         file_size = avio_size(ic->pb);
11.         file_size = FFMAX(0, file_size);
12.     }
13.
14.
15.     if ((!strcmp(ic->iformat->name, "mpeg") ||
16.         !strcmp(ic->iformat->name, "mpegts")) &&
17.         file_size && ic->pb->seekable) {
18.         /* get accurate estimate from the PTSes */
19.         estimate_timings_from_pts(ic, old_offset);
20.         ic->duration_estimation_method = AVFMT_DURATION_FROM_PTS;
21.     } else if (has_duration(ic)) {
22.         /* at least one component has timings - we use them for all
23.          * the components */
24.         fill_all_stream_timings(ic);
25.         ic->duration_estimation_method = AVFMT_DURATION_FROM_STREAM;
26.     } else {
27.         /* less precise: use bitrate info */
28.         estimate_timings_from_bit_rate(ic);
29.         ic->duration_estimation_method = AVFMT_DURATION_FROM_BITRATE;
30.     }
31.     update_stream_timings(ic);
32.
33.
34.     {
35.         int i;
36.         AVStream av_unused *st;
37.         for (i = 0; i < ic->nb_streams; i++) {
38.             st = ic->streams[i];
39.             av_dlog(ic, "%d: start_time: %0.3f duration: %0.3f\n", i,
40.                 (double) st->start_time / AV_TIME_BASE,
41.                 (double) st->duration / AV_TIME_BASE);
42.         }
43.         av_dlog(ic,
44.             "stream: start_time: %0.3f duration: %0.3f bitrate=%d kb/s\n",
45.             (double) ic->start_time / AV_TIME_BASE,
46.             (double) ic->duration / AV_TIME_BASE,
47.             ic->bit_rate / 1000);
48.     }
49. }

```

从estimate\_timings()的代码中可以看出，有3种估算方法：

- (1) 通过pts（显示时间戳）。该方法调用estimate\_timings\_from\_pts()。它的基本思想就是读取视音频流中的结束位置AVPacket的PTS和起始位置AVPacket的PTS，两者相减得到时长信息。
- (2) 通过已知流的时长。该方法调用fill\_all\_stream\_timings()。它的代码没有细看，但从函数的注释的意思来说，应该是当有些视音频流有时长信息的时候，直接赋值给其他视音频流。
- (3) 通过bitrate（码率）。该方法调用estimate\_timings\_from\_bit\_rate()。它的基本思想就是获得整个文件大小，以及整个文件的bitrate，两者相除之后得到时长信息。

## estimate\_timings\_from\_bit\_rate()

在这里附上上述几种方法中最简单的函数estimate\_timings\_from\_bit\_rate()的代码。

```

1. static void estimate_timings_from_bit_rate(AVFormatContext *ic)
2. {
3.     int64_t filesize, duration;
4.     int i, show_warning = 0;
5.     AVStream *st;
6.
7.
8.     /* if bit_rate is already set, we believe it */
9.     if (ic->bit_rate <= 0) {
10.        int bit_rate = 0;
11.        for (i = 0; i < ic->nb_streams; i++) {
12.            st = ic->streams[i];
13.            if (st->codec->bit_rate > 0) {
14.                if (INT_MAX - st->codec->bit_rate < bit_rate) {
15.                    bit_rate = 0;
16.                    break;
17.                }
18.                bit_rate += st->codec->bit_rate;
19.            }
20.        }
21.        ic->bit_rate = bit_rate;
22.    }
23.
24.
25.    /* if duration is already set, we believe it */
26.    if (ic->duration == AV_NOPTS_VALUE &&
27.        ic->bit_rate != 0) {
28.        filesize = ic->pb ? avio_size(ic->pb) : 0;
29.        if (filesize > ic->data_offset) {
30.            filesize -= ic->data_offset;
31.            for (i = 0; i < ic->nb_streams; i++) {
32.                st = ic->streams[i];
33.                if (st->time_base.num <= INT64_MAX / ic->bit_rate
34.                    && st->duration == AV_NOPTS_VALUE) {
35.                    duration = av_rescale(8 * filesize, st->time_base.den,
36.                                            ic->bit_rate *
37.                                            (int64_t) st->time_base.num);
38.                    st->duration = duration;
39.                    show_warning = 1;
40.                }
41.            }
42.        }
43.    }
44.    if (show_warning)
45.        av_log(ic, AV_LOG_WARNING,
46.               "Estimating duration from bitrate, this may be inaccurate\n");
47. }

```

从代码中可以看出，该函数做了两步工作：

- (1) 如果AVFormatContext中没有bit\_rate信息，就把所有AVStream的bit\_rate加起来作为AVFormatContext的bit\_rate信息。
- (2) 使用文件大小filesize除以bitrate得到时长信息。具体的方法是：

$$AVStream->duration = (filesize * 8 / bit\_rate) / time\_base$$

PS：

- 1) filesize乘以8是因为需要把Byte转换为Bit
- 2) 具体的实现函数是那个av\_rescale()函数。x=av\_rescale(a,b,c)的含义是x=a\*b/c。
- 3) 之所以要除以time\_base，是因为AVStream中的duration的单位是time\_base，注意这和AVFormatContext中的duration的单位（单位是AV\_TIME\_BASE，固定取值为1000000）是不一样的。

至此，avformat\_find\_stream\_info()主要的函数就分析完了。

## 雷霄骅

leixiaohua1020@126.com

<http://blog.csdn.net/leixiaohua1020>

版权声明：本文为博主原创文章，未经博主允许不得转载。 <https://blog.csdn.net/leixiaohua1020/article/details/44084321>

文章标签： FFMpeg 源代码 AVFormatContext 媒体信息

个人分类： FFMPEG

所属专栏： FFMpeg