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：35859

==========

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
/**
 * 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);
```

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。

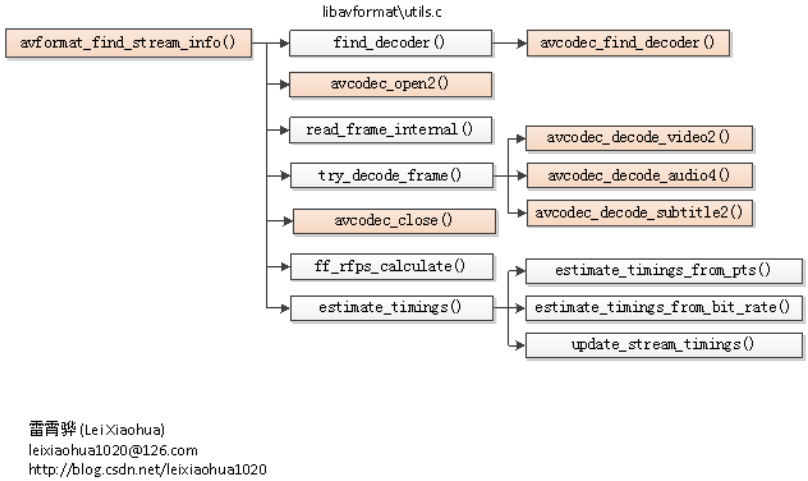
函数正常执行后返回值大于等于0。

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(Lei Xiaohua)
leixiaohua1020@126.com
http://blog.csdn.net/leixiaohua1020

avformat_find_stream_info()

avformat_find_stream_info()的定义位于libavformat\utils.c。它的代码比较长，如下所示。

```cpp
int avformat_find_stream_info(AVFormatContext *ic, AVDictionary **options)
{
    int i, count, ret = 0, j;
```

```
    int i, count, ret = 0, j;
    int64_t read_size;
    AVStream *st;
    AVPacket pkt1, *pkt;
    int64_t old_offset  = avio_tell(ic->pb);
    // new streams might appear, no options for those
    int orig_nb_streams = ic->nb_streams;
    int flush_codecs;
    int64_t max_analyze_duration = ic->max_analyze_duration2;
    int64_t probesize = ic->probesize2;


    if (!max_analyze_duration)
        max_analyze_duration = ic->max_analyze_duration;
    if (ic->probesize)
        probesize = ic->probesize;
    flush_codecs = probesize > 0;


    av_opt_set(ic, "skip_clear", "1", AV_OPT_SEARCH_CHILDREN);


    if (!max_analyze_duration) {
        if (!strcmp(ic->iformat->name, "flv") && !(ic->ctx_flags & AVFMTCTX_NOHEADER)) {
            max_analyze_duration = 10*AV_TIME_BASE;
        } else
            max_analyze_duration = 5*AV_TIME_BASE;
    }


    if (ic->pb)
        av_log(ic, AV_LOG_DEBUG, "Before avformat_find_stream_info() pos: %"PRId64" bytes read:%"PRId64" seeks:%d\n",
               avio_tell(ic->pb), ic->pb->bytes_read, ic->pb->seek_count);


    for (i = 0; i < ic->nb_streams; i++) {
        const AVCodec *codec;
        AVDictionary *thread_opt = NULL;
        st = ic->streams[i];


        if (st->codec->codec_type == AVMEDIA_TYPE_VIDEO ||
            st->codec->codec_type == AVMEDIA_TYPE_SUBTITLE) {
/*            if (!st->time_base.num)
              st->time_base = */
            if (!st->codec->time_base.num)
                st->codec->time_base = st->time_base;
        }
        // only for the split stuff
        if (!st->parser && !(ic->flags & AVFMT_FLAG_NOPARSE)) {
            st->parser = av_parser_init(st->codec->codec_id);
            if (st->parser) {
                if (st->need_parsing == AVSTREAM_PARSE_HEADERS) {
                    st->parser->flags |= PARSER_FLAG_COMPLETE_FRAMES;
                } else if (st->need_parsing == AVSTREAM_PARSE_FULL_RAW) {
                    st->parser->flags |= PARSER_FLAG_USE_CODEC_TS;
                }
            } else if (st->need_parsing) {
                av_log(ic, AV_LOG_VERBOSE, "parser not found for codec "
                       "%s, packets or times may be invalid.\n",
                       avcodec_get_name(st->codec->codec_id));
            }
        }
        codec = find_decoder(ic, st, st->codec->codec_id);


        /* Force thread count to 1 since the H.264 decoder will not extract
         * SPS and PPS to extradata during multi-threaded decoding. */
        av_dict_set(options ? &options[i] : &thread_opt, "threads", "1", 0);


        if (ic->codec_whitelist)
            av_dict_set(options ? &options[i] : &thread_opt, "codec_whitelist", ic->codec_whitelist, 0);


        /* Ensure that subtitle_header is properly set. */
        if (st->codec->codec_type == AVMEDIA_TYPE_SUBTITLE
            && codec && !st->codec->codec) {
            if (avcodec_open2(st->codec, codec, options ? &options[i] : &thread_opt) < 0)
                av_log(ic, AV_LOG_WARNING,
                       "Failed to open codec in av_find_stream_info\n");
        }


        // Try to just open decoders, in case this is enough to get parameters.
        if (!has_codec_parameters(st, NULL) && st->request_probe <= 0) {
            if (codec && !st->codec->codec)
                if (avcodec_open2(st->codec, codec, options ? &options[i] : &thread_opt) < 0)
                    av_log(ic, AV_LOG_WARNING,
                           "Failed to open codec in av_find_stream_info\n");
        }
        if (!options)
```

```
            av_dict_free(&thread_opt);
    }


    for (i = 0; i < ic->nb_streams; i++) {
#if FF_API_R_FRAME_RATE
        ic->streams[i]->info->last_dts = AV_NOPTS_VALUE;
#endif
        ic->streams[i]->info->fps_first_dts = AV_NOPTS_VALUE;
        ic->streams[i]->info->fps_last_dts  = AV_NOPTS_VALUE;
    }


    count     = 0;
    read_size = 0;
    for (;;) {
        if (ff_check_interrupt(&ic->interrupt_callback)) {
            ret = AVERROR_EXIT;
            av_log(ic, AV_LOG_DEBUG, "interrupted\n");
            break;
        }


        /* check if one codec still needs to be handled */
        for (i = 0; i < ic->nb_streams; i++) {
            int fps_analyze_framecount = 20;


            st = ic->streams[i];
            if (!has_codec_parameters(st, NULL))
                break;
            /* If the timebase is coarse (like the usual millisecond precision
             * of mkv), we need to analyze more frames to reliably arrive at
             * the correct fps. */
            if (av_q2d(st->time_base) > 0.0005)
                fps_analyze_framecount *= 2;
            if (!tb_unreliable(st->codec))
                fps_analyze_framecount = 0;
            if (ic->fps_probe_size >= 0)
                fps_analyze_framecount = ic->fps_probe_size;
            if (st->disposition & AV_DISPOSITION_ATTACHED_PIC)
                fps_analyze_framecount = 0;
            /* variable fps and no guess at the real fps */
            if (!(st->r_frame_rate.num && st->avg_frame_rate.num) &&
                st->info->duration_count < fps_analyze_framecount &&
                st->codec->codec_type == AVMEDIA_TYPE_VIDEO)
                break;
            if (st->parser && st->parser->parser->split &&
                !st->codec->extradata)
                break;
            if (st->first_dts == AV_NOPTS_VALUE &&
                !(ic->iformat->flags & AVFMT_NOTIMESTAMPS) &&
                st->codec_info_nb_frames < ic->max_ts_probe &&
                (st->codec->codec_type == AVMEDIA_TYPE_VIDEO ||
                 st->codec->codec_type == AVMEDIA_TYPE_AUDIO))
                break;
        }
        if (i == ic->nb_streams) {
            /* NOTE: If the format has no header, then we need to read some
             * packets to get most of the streams, so we cannot stop here. */
            if (!(ic->ctx_flags & AVFMTCTX_NOHEADER)) {
                /* If we found the info for all the codecs, we can stop. */
                ret = count;
                av_log(ic, AV_LOG_DEBUG, "All info found\n");
                flush_codecs = 0;
                break;
            }
        }
        /* We did not get all the codec info, but we read too much data. */
        if (read_size >= probesize) {
            ret = count;
            av_log(ic, AV_LOG_DEBUG,
                   "Probe buffer size limit of %"PRId64" bytes reached\n", probesize);
            for (i = 0; i < ic->nb_streams; i++)
                if (!ic->streams[i]->r_frame_rate.num &&
                    ic->streams[i]->info->duration_count <= 1 &&
                    ic->streams[i]->codec->codec_type == AVMEDIA_TYPE_VIDEO &&
                    strcmp(ic->iformat->name, "image2"))
                    av_log(ic, AV_LOG_WARNING,
                           "Stream #%d: not enough frames to estimate rate; "
                           "consider increasing probesize\n", i);
            break;
        }


        /* NOTE: A new stream can be added there if no header in file
         * (AVFMTCTX_NOHEADER). */
        ret = read_frame_internal(ic, &pkt1);
        if (ret == AVERROR(EAGAIN))
            continue;
```

```
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.                        "%"PRId64", packet %d with DTS %"PRId64"\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.                        "%"PRId64", packet %d with DTS %"PRId64"\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 %"PRId64" reached at %"PRId64" microseconds\n",
271.                        max_analyze_duration,
272.                        t);
273.                 if (ic->flags & AVFMT_FLAG_NOBUFFER)
274.                     av_packet_unref(pkt);
275.                 break;
276.             }
```

```
            if (pkt->duration) {
                st->info->codec_info_duration        += pkt->duration;
                st->info->codec_info_duration_fields += st->parser && st->need_parsing && st->codec->ticks_per_frame ==2 ? st->parse
r->repeat_pict + 1 : 2;
            }
        }
#if FF_API_R_FRAME_RATE
        if (st->codec->codec_type == AVMEDIA_TYPE_VIDEO)
            ff_rfps_add_frame(ic, st, pkt->dts);
#endif
        if (st->parser && st->parser->parser->split && !st->codec->extradata) {
            int i = st->parser->parser->split(st->codec, pkt->data, pkt->size);
            if (i > 0 && i < FF_MAX_EXTRADATA_SIZE) {
                if (ff_alloc_extradata(st->codec, i))
                    return AVERROR(ENOMEM);
                memcpy(st->codec->extradata, pkt->data,
                       st->codec->extradata_size);
            }
        }


        /* If still no information, we try to open the codec and to
         * decompress the frame. We try to avoid that in most cases as
         * it takes longer and uses more memory. For MPEG-4, we need to
         * decompress for QuickTime.
         *
         * If CODEC_CAP_CHANNEL_CONF is set this will force decoding of at
         * least one frame of codec data, this makes sure the codec initializes
         * the channel configuration and does not only trust the values from
         * the container. */
        try_decode_frame(ic, st, pkt,
                         (options && i < orig_nb_streams) ? &options[i] : NULL);


        if (ic->flags & AVFMT_FLAG_NOBUFFER)
            av_packet_unref(pkt);


        st->codec_info_nb_frames++;
        count++;
    }


    if (flush_codecs) {
        AVPacket empty_pkt = { 0 };
        int err = 0;
        av_init_packet(&empty_pkt);


        for (i = 0; i < ic->nb_streams; i++) {


            st = ic->streams[i];


            /* flush the decoders */
            if (st->info->found_decoder == 1) {
                do {
                    err = try_decode_frame(ic, st, &empty_pkt,
                                           (options && i < orig_nb_streams)
                                           ? &options[i] : NULL);
                } while (err > 0 && !has_codec_parameters(st, NULL));


                if (err < 0) {
                    av_log(ic, AV_LOG_INFO,
                        "decoding for stream %d failed\n", st->index);
                }
            }
        }
    }


    // close codecs which were opened in try_decode_frame()
    for (i = 0; i < ic->nb_streams; i++) {
        st = ic->streams[i];
        avcodec_close(st->codec);
    }


    ff_rfps_calculate(ic);


    for (i = 0; i < ic->nb_streams; i++) {
        st = ic->streams[i];
        if (st->codec->codec_type == AVMEDIA_TYPE_VIDEO) {
            if (st->codec->codec_id == AV_CODEC_ID_RAWVIDEO && !st->codec->codec_tag && !st->codec->bits_per_coded_sample) {
                uint32_t tag= avcodec_pix_fmt_to_codec_tag(st->codec->pix_fmt);
                if (avpriv_find_pix_fmt(avpriv_get_raw_pix_fmt_tags(), tag) == st->codec->pix_fmt)
                    st->codec->codec_tag= tag;
            }
```

```
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,
```

```
                    "Could not find codec parameters for stream %d (%s): %s\n"
                    "Consider increasing the value for the 'analyzeduration' and 'probesize' options\n",
                    i, buf, errmsg);
        } else {
            ret = 0;
        }
    }


    compute_chapters_end(ic);


find_stream_info_err:
    for (i = 0; i < ic->nb_streams; i++) {
        st = ic->streams[i];
        if (ic->streams[i]->codec->codec_type != AVMEDIA_TYPE_AUDIO)
            ic->streams[i]->codec->thread_count = 0;
        if (st->info)
            av_freep(&st->info->duration_error);
        av_freep(&ic->streams[i]->info);
    }
    if (ic->pb)
        av_log(ic, AV_LOG_DEBUG, "After avformat_find_stream_info() pos: %"PRId64" bytes read:%"PRId64" seeks:%d frames:%d\n",
               avio_tell(ic->pb), ic->pb->bytes_read, ic->pb->seek_count, count);
    return ret;
}
```

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。

```cpp
static const AVCodec *find_decoder(AVFormatContext *s, AVStream *st, enum AVCodecID codec_id)
{
    if (st->codec->codec)
        return st->codec->codec;


    switch (st->codec->codec_type) {
    case AVMEDIA_TYPE_VIDEO:
        if (s->video_codec)    return s->video_codec;
        break;
    case AVMEDIA_TYPE_AUDIO:
        if (s->audio_codec)    return s->audio_codec;
        break;
    case AVMEDIA_TYPE_SUBTITLE:
        if (s->subtitle_codec) return s->subtitle_codec;
        break;
    }


    return avcodec_find_decoder(codec_id);
}
```

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
/* returns 1 or 0 if or if not decoded data was returned, or a negative error */
static int try_decode_frame(AVFormatContext *s, AVStream *st, AVPacket *avpkt,
                            AVDictionary **options)
{
    const AVCodec *codec;
    int got_picture = 1, ret = 0;
    AVFrame *frame = av_frame_alloc();
    AVSubtitle subtitle;
    AVPacket pkt = *avpkt;


    if (!frame)
        return AVERROR(ENOMEM);


    if (!avcodec_is_open(st->codec) &&
        st->info->found_decoder <= 0 &&
        (st->codec->codec_id != -st->info->found_decoder || !st->codec->codec_id)) {
        AVDictionary *thread_opt = NULL;


        codec = find_decoder(s, st, st->codec->codec_id);


        if (!codec) {
            st->info->found_decoder = -st->codec->codec_id;
            ret                     = -1;
            goto fail;
        }


        /* Force thread count to 1 since the H.264 decoder will not extract
         * SPS and PPS to extradata during multi-threaded decoding. */
        av_dict_set(options ? options : &thread_opt, "threads", "1", 0);
        if (s->codec_whitelist)
            av_dict_set(options ? options : &thread_opt, "codec_whitelist", s->codec_whitelist, 0);
        ret = avcodec_open2(st->codec, codec, options ? options : &thread_opt);
        if (!options)
            av_dict_free(&thread_opt);
        if (ret < 0) {
            st->info->found_decoder = -st->codec->codec_id;
            goto fail;
        }
        st->info->found_decoder = 1;
    } else if (!st->info->found_decoder)
        st->info->found_decoder = 1;


    if (st->info->found_decoder < 0) {
        ret = -1;
        goto fail;
    }


    while ((pkt.size > 0 || (!pkt.data && got_picture)) &&
           ret >= 0 &&
           (!has_codec_parameters(st, NULL) || !has_decode_delay_been_guessed(st) ||
            (!st->codec_info_nb_frames &&
             st->codec->codec->capabilities & CODEC_CAP_CHANNEL_CONF))) {
        got_picture = 0;
        switch (st->codec->codec_type) {
        case AVMEDIA_TYPE_VIDEO:
            ret = avcodec_decode_video2(st->codec, frame,
                                        &got_picture, &pkt);
            break;
        case AVMEDIA_TYPE_AUDIO:
            ret = avcodec_decode_audio4(st->codec, frame, &got_picture, &pkt);
            break;
        case AVMEDIA_TYPE_SUBTITLE:
            ret = avcodec_decode_subtitle2(st->codec, &subtitle,
                                           &got_picture, &pkt);
            ret = pkt.size;
            break;
        default:
            break;
        }
        if (ret >= 0) {
            if (got_picture)
                st->nb_decoded_frames++;
            pkt.data += ret;
            pkt.size -= ret;
            ret       = got_picture;
        }
    }


    if (!pkt.data && !got_picture)
        ret = -1;

```

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

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
static int has_codec_parameters(AVStream *st, const char **errmsg_ptr)
{
    AVCodecContext *avctx = st->codec;


#define FAIL(errmsg) do {                                       \
        if (errmsg_ptr)                                         \
            *errmsg_ptr = errmsg;                               \
        return 0;                                               \
    } while (0)


    if (   avctx->codec_id == AV_CODEC_ID_NONE
        && avctx->codec_type != AVMEDIA_TYPE_DATA)
        FAIL("unknown codec");
    switch (avctx->codec_type) {
    case AVMEDIA_TYPE_AUDIO:
        if (!avctx->frame_size && determinable_frame_size(avctx))
            FAIL("unspecified frame size");
        if (st->info->found_decoder >= 0 &&
            avctx->sample_fmt == AV_SAMPLE_FMT_NONE)
            FAIL("unspecified sample format");
        if (!avctx->sample_rate)
            FAIL("unspecified sample rate");
        if (!avctx->channels)
            FAIL("unspecified number of channels");
        if (st->info->found_decoder >= 0 && !st->nb_decoded_frames && avctx->codec_id == AV_CODEC_ID_DTS)
            FAIL("no decodable DTS frames");
        break;
    case AVMEDIA_TYPE_VIDEO:
        if (!avctx->width)
            FAIL("unspecified size");
        if (st->info->found_decoder >= 0 && avctx->pix_fmt == AV_PIX_FMT_NONE)
            FAIL("unspecified pixel format");
        if (st->codec->codec_id == AV_CODEC_ID_RV30 || st->codec->codec_id == AV_CODEC_ID_RV40)
            if (!st->sample_aspect_ratio.num && !st->codec->sample_aspect_ratio.num && !st->codec_info_nb_frames)
                FAIL("no frame in rv30/40 and no sar");
        break;
    case AVMEDIA_TYPE_SUBTITLE:
        if (avctx->codec_id == AV_CODEC_ID_HDMV_PGS_SUBTITLE && !avctx->width)
            FAIL("unspecified size");
        break;
    case AVMEDIA_TYPE_DATA:
        if (avctx->codec_id == AV_CODEC_ID_NONE) return 1;
    }


    return 1;
}
```

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。

```cpp
static void estimate_timings(AVFormatContext *ic, int64_t old_offset)
{
    int64_t file_size;


    /* get the file size, if possible */
    if (ic->iformat->flags & AVFMT_NOFILE) {
        file_size = 0;
    } else {
        file_size = avio_size(ic->pb);
        file_size = FFMAX(0, file_size);
    }


    if ((!strcmp(ic->iformat->name, "mpeg") ||
         !strcmp(ic->iformat->name, "mpegts")) &&
        file_size && ic->pb->seekable) {
        /* get accurate estimate from the PTSes */
        estimate_timings_from_pts(ic, old_offset);
        ic->duration_estimation_method = AVFMT_DURATION_FROM_PTS;
    } else if (has_duration(ic)) {
        /* at least one component has timings - we use them for all
         * the components */
        fill_all_stream_timings(ic);
        ic->duration_estimation_method = AVFMT_DURATION_FROM_STREAM;
    } else {
        /* less precise: use bitrate info */
        estimate_timings_from_bit_rate(ic);
        ic->duration_estimation_method = AVFMT_DURATION_FROM_BITRATE;
    }
    update_stream_timings(ic);


    {
        int i;
        AVStream av_unused *st;
        for (i = 0; i < ic->nb_streams; i++) {
            st = ic->streams[i];
            av_dlog(ic, "%d: start_time: %0.3f duration: %0.3f\n", i,
                    (double) st->start_time / AV_TIME_BASE,
                    (double) st->duration   / AV_TIME_BASE);
        }
        av_dlog(ic,
                "stream: start_time: %0.3f duration: %0.3f bitrate=%d kb/s\n",
                (double) ic->start_time / AV_TIME_BASE,
                (double) ic->duration   / AV_TIME_BASE,
                ic->bit_rate / 1000);
    }
}
```

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。

```cpp
static void estimate_timings_from_bit_rate(AVFormatContext *ic)
{
    int64_t filesize, duration;
    int i, show_warning = 0;
    AVStream *st;


    /* if bit_rate is already set, we believe it */
    if (ic->bit_rate <= 0) {
        int bit_rate = 0;
        for (i = 0; i < ic->nb_streams; i++) {
            st = ic->streams[i];
            if (st->codec->bit_rate > 0) {
                if (INT_MAX - st->codec->bit_rate < bit_rate) {
                    bit_rate = 0;
                    break;
                }
                bit_rate += st->codec->bit_rate;
            }
        }
        ic->bit_rate = bit_rate;
    }


    /* if duration is already set, we believe it */
    if (ic->duration == AV_NOPTS_VALUE &&
        ic->bit_rate != 0) {
        filesize = ic->pb ? avio_size(ic->pb) : 0;
        if (filesize > ic->data_offset) {
            filesize -= ic->data_offset;
            for (i = 0; i < ic->nb_streams; i++) {
                st      = ic->streams[i];
                if (   st->time_base.num <= INT64_MAX / ic->bit_rate
                    && st->duration == AV_NOPTS_VALUE) {
                    duration = av_rescale(8 * filesize, st->time_base.den,
                                          ic->bit_rate *
                                          (int64_t) st->time_base.num);
                    st->duration = duration;
                    show_warning = 1;
                }
            }
        }
    }
    if (show_warning)
        av_log(ic, AV_LOG_WARNING,
               "Estimating duration from bitrate, this may be inaccurate\n");
}
```

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

文章标签： FFmpeg 源代码 AVFormatContext 媒体信息

个人分类： FFMPEG

所属专栏： FFmpeg