

原 FFMpeg源代码简单分析：avformat_open_input()

2015年03月05日 00:13:10 阅读数：32507

=====

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_open_input()。该函数用于打开多媒体数据并且获得一些相关的信息。它的声明位于libavformat\avformat.h，如下所示。

```
[cpp]
1.  /**
2.   * Open an input stream and read the header. The codecs are not opened.
3.   * The stream must be closed with avformat_close_input().
4.   *
5.   * @param ps Pointer to user-supplied AVFormatContext (allocated by avformat_alloc_context).
6.   *           May be a pointer to NULL, in which case an AVFormatContext is allocated by this
7.   *           function and written into ps.
8.   *           Note that a user-supplied AVFormatContext will be freed on failure.
9.   * @param filename Name of the stream to open.
10.  * @param fmt If non-NULL, this parameter forces a specific input format.
11.  *            Otherwise the format is autodetected.
12.  * @param options A dictionary filled with AVFormatContext and demuxer-private options.
13.  *               On return this parameter will be destroyed and replaced with a dict containing
14.  *               options that were not found. May be NULL.
15.  *
16.  * @return 0 on success, a negative AVERRO on failure.
17.  *
18.  * @note If you want to use custom IO, preallocate the format context and set its pb field.
19.  */
20.  int avformat_open_input(AVFormatContext **ps, const char *filename, AVInputFormat *fmt, AVDictionary **options);
```

代码中的英文注释写的已经比较详细了，在这里拿中文简单叙述一下。

ps：函数调用成功之后处理过的AVFormatContext结构体。

file：打开的视音频流的URL。

fmt：强制指定AVFormatContext中AVInputFormat的。这个参数一般情况下可以设置为NULL，这样FFmpeg可以自动检测AVInputFormat。

dictionary：附加的一些选项，一般情况下可以设置为NULL。

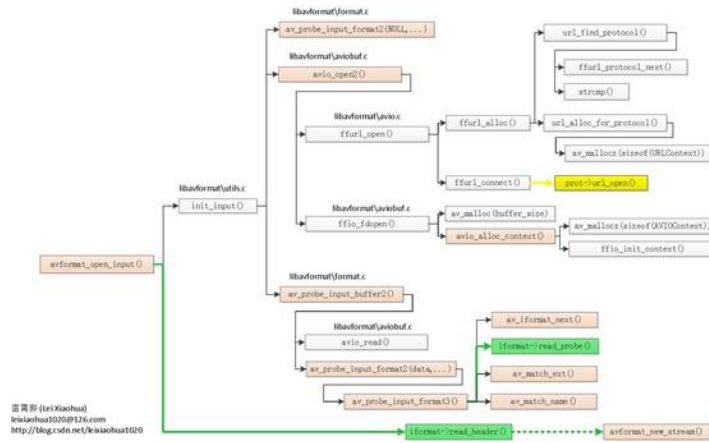
函数执行成功的话，其返回值大于等于0。

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）](#)

此前已经粗略写了1篇关于avformat_open_input()的文章《[图解FFMPEG打开媒体的函数avformat_open_input](#)》，还转载了一篇注释比较详细的文章《[FFMPEG源码分析:avformat_open_input\(\) \(媒体打开函数\)](#)》。但是个人感觉这个函数确实太重要了，可以算作FFmpeg的“灵魂”，所以打算再写一篇文章分析一下它的结构。

函数调用关系图

函数调用结构图如下所示。



单击查看更清晰的图片

avformat_open_input()

下面看一下的定义，位于libavformat\utils.c中，如下所示。

```
[cpp]
1. int avformat_open_input(AVFormatContext **ps, const char *filename,
2.                        AVInputFormat *fmt, AVDictionary **options)
3. {
4.     AVFormatContext *s = *ps;
5.     int ret = 0;
6.     AVDictionary *tmp = NULL;
7.     ID3v2ExtraMeta *id3v2_extra_meta = NULL;
8.
9.     if (!s && !(s = avformat_alloc_context()))
10.         return AVERROR(ENOMEM);
11.     if (!s->av_class) {
12.         av_log(NULL, AV_LOG_ERROR, "Input context has not been properly allocated by avformat_alloc_context() and is not NULL either\
13. );
14.         return AVERROR(EINVAL);
15.     }
16.     if (fmt)
17.         s->iformat = fmt;
18.
19.     if (options)
20.         av_dict_copy(&tmp, *options, 0);
21.
22.     if ((ret = av_opt_set_dict(s, &tmp)) < 0)
23.         goto fail;
24.
25.     if ((ret = init_input(s, filename, &tmp)) < 0)
26.         goto fail;
27.     s->probe_score = ret;
28.
29.     if (s->format_whitelist && av_match_list(s->iformat->name, s->format_whitelist, ',') <= 0) {
30.         av_log(s, AV_LOG_ERROR, "Format not on whitelist\n");
31.         ret = AVERROR(EINVAL);
32.         goto fail;
33.     }
34.
35.     avio_skip(s->pb, s->skip_initial_bytes);
36.
37.     /* Check filename in case an image number is expected. */
38.     if (s->iformat->flags & AVFMT_NEEDNUMBER) {
39.         if (!av_filename_number_test(filename)) {
40.             ret = AVERROR(EINVAL);
41.             goto fail;
42.         }
43.     }
44.
45.     s->duration = s->start_time = AV_NOPTS_VALUE;
46.     av_strlcpy(s->filename, filename ? filename : "", sizeof(s->filename));
47.
48.     /* Allocate private data. */
49.     if (s->iformat->priv_data_size > 0) {
50.         if (!(s->priv_data = av_mallocz(s->iformat->priv_data_size))) {
51.             ret = AVERROR(ENOMEM);
52.             goto fail;
53.         }
54.         if (s->iformat->priv_class) {
55.             *(const AVClass **) s->priv_data = s->iformat->priv_class;
56.             av_opt_set_defaults(s->priv_data);
57.             if ((ret = av_opt_set_dict(s->priv_data, &tmp)) < 0)
58.                 goto fail;
59.         }
60.     }
```

```

59.     }
60.
61.     /* e.g. AVFMT_NOFILE formats will not have a AVIOContext */
62.     if (s->pb)
63.         ff_id3v2_read(s, ID3v2_DEFAULT_MAGIC, &id3v2_extra_meta, 0);
64.
65.     if (!(s->flags&AVFMT_FLAG_PRIV_OPT) && s->iformat->read_header)
66.         if ((ret = s->iformat->read_header(s)) < 0)
67.             goto fail;
68.
69.     if (id3v2_extra_meta) {
70.         if (!strcmp(s->iformat->name, "mp3") || !strcmp(s->iformat->name, "aac") ||
71.             !strcmp(s->iformat->name, "tta")) {
72.             if ((ret = ff_id3v2_parse_apic(s, &id3v2_extra_meta)) < 0)
73.                 goto fail;
74.         } else
75.             av_log(s, AV_LOG_DEBUG, "demuxer does not support additional id3 data, skipping\n");
76.     }
77.     ff_id3v2_free_extra_meta(&id3v2_extra_meta);
78.
79.     if ((ret = avformat_queue_attached_pictures(s)) < 0)
80.         goto fail;
81.
82.     if (!(s->flags&AVFMT_FLAG_PRIV_OPT) && s->pb && !s->data_offset)
83.         s->data_offset = avio_tell(s->pb);
84.
85.     s->raw_packet_buffer_remaining_size = RAW_PACKET_BUFFER_SIZE;
86.
87.     if (options) {
88.         av_dict_free(options);
89.         *options = tmp;
90.     }
91.     *ps = s;
92.     return 0;
93.
94. fail:
95.     ff_id3v2_free_extra_meta(&id3v2_extra_meta);
96.     av_dict_free(&tmp);
97.     if (s->pb && !(s->flags & AVFMT_FLAG_CUSTOM_IO))
98.         avio_close(s->pb);
99.     avformat_free_context(s);
100.    *ps = NULL;
101.    return ret;
102. }

```

avformat_open_input()源代码比较长，一部分是一些容错代码，比如说如果发现传入的AVFormatContext指针没有初始化过，就调用avformat_alloc_context()初始化该结构体；还有一部分是针对一些格式做的特殊处理，比如id3v2信息的处理等等。有关上述两种信息不再详细分析，在这里只选择它关键的两个函数进行分析：

init_input()：绝大部分初始化工作都是在这里做的。

s->iformat->read_header()：读取多媒体数据文件头，根据视音频流创建相应的AVStream。

下面我们逐一看看上述函数。

init_input()

init_input()作为一个内部函数，竟然包含了一行注释（一般内部函数都没有注释），足可以看出它的重要性。它的主要工作就是打开输入的视频数据并且探测视频的格式。该函数的定义位于libavformatutils.c，如下所示。

```

1.  /* Open input file and probe the format if necessary. */
2.  static int init_input(AVFormatContext *s, const char *filename,
3.                      AVDictionary **options)
4.  {
5.      int ret;
6.      AVProbeData pd = { filename, NULL, 0 };
7.      int score = AVPROBE_SCORE_RETRY;
8.
9.      if (s->pb) {
10.         s->flags |= AVFMT_FLAG_CUSTOM_IO;
11.         if (!s->iformat)
12.             return av_probe_input_buffer2(s->pb, &s->iformat, filename,
13.                                           s, 0, s->format_probesize);
14.         else if (s->iformat->flags & AVFMT_NOFILE)
15.             av_log(s, AV_LOG_WARNING, "Custom AVIOContext makes no sense and "
16.                    "will be ignored with AVFMT_NOFILE format.\n");
17.         return 0;
18.     }
19.
20.     if ((s->iformat && s->iformat->flags & AVFMT_NOFILE) ||
21.         (!s->iformat && (s->iformat = av_probe_input_format2(&pd, 0, &score))))
22.         return score;
23.
24.     if ((ret = avio_open2(&s->pb, filename, AVIO_FLAG_READ | s->avio_flags,
25.                          &s->interrupt_callback, options)) < 0)
26.         return ret;
27.     if (s->iformat)
28.         return 0;
29.     return av_probe_input_buffer2(s->pb, &s->iformat, filename,
30.                                  s, 0, s->format_probesize);
31. }

```

这个函数在短短的几行代码中包含了好几个return，因此逻辑还是有点复杂的，我们可以梳理一下：

在函数的开头的score变量是一个判决AVInputFormat的分数的门限值，如果最后得到的AVInputFormat的分数低于该门限值，就认为没有找到合适的AVInputFormat。FFmpeg内部判断封装格式的原理实际上是对每种AVInputFormat给出一个分数，满分是100分，越有可能正确的AVInputFormat给出的分数就越高。最后选择分数最高的AVInputFormat作为推测结果。score的值是一个宏定义AVPROBE_SCORE_RETRY，我们可以看一下它的定义：

```

1.  #define AVPROBE_SCORE_RETRY (AVPROBE_SCORE_MAX/4)

```

其中AVPROBE_SCORE_MAX是score的最大值，取值是100：

```

1.  #define AVPROBE_SCORE_MAX 100 ///< maximum score

```

由此我们可以得出score取值是25，即如果推测后得到的最佳AVInputFormat的分值低于25，就认为没有找到合适的AVInputFormat。

整个函数的逻辑大体如下：

- (1) 当使用了自定义的AVIOContext的时候（AVFormatContext中的AVIOContext不为空，即s->pb!=NULL），如果指定了AVInputFormat就直接返回，如果没有指定就调用av_probe_input_buffer2()推测AVInputFormat。这一情况出现的不算很多，但是当我们从内存中读取数据的时候（需要初始化自定义的AVIOContext），就会执行这一步骤。
- (2) 在更一般的情况下，如果已经指定了AVInputFormat，就直接返回；如果没有指定AVInputFormat，就调用av_probe_input_format(NULL,...)根据文件路径判断文件格式。这里特意把av_probe_input_format()的第1个参数写成“NULL”，是为了强调这个时候实际上并没有给函数提供输入数据，此时仅仅通过文件路径推测AVInputFormat。
- (3) 如果发现通过文件路径判断不出来文件格式，那么就需要打开文件探测文件格式了，这个时候会首先调用avio_open2()打开文件，然后调用av_probe_input_buffer2()推测AVInputFormat。

下面分析一下av_probe_input_format(), avio_open2(), av_probe_input_buffer2()这几个函数。

av_probe_input_format2()

av_probe_input_format2()是一个API函数，声明位于libavformat\avformat.h，如下所示。

```

1.  /**
2.   * Guess the file format.
3.   *
4.   * @param pd      data to be probed
5.   * @param is_opened Whether the file is already opened; determines whether
6.   *                 demuxers with or without AVFMT_NOFILE are probed.
7.   * @param score_max A probe score larger than this is required to accept a
8.   *                 detection, the variable is set to the actual detection
9.   *                 score afterwards.
10.  *                 If the score is <= AVPROBE_SCORE_MAX / 4 it is recommended
11.  *                 to retry with a larger probe buffer.
12.  */
13. AVInputFormat *av_probe_input_format2(AVProbeData *pd, int is_opened, int *score_max);

```

该函数用于根据输入数据查找合适的AVInputFormat。参数含义如下所示：

pd：存储输入数据信息的AVProbeData结构体。

is_opened：文件是否打开。

score_max：判决AVInputFormat的门限值。只有某格式判决分数大于该门限值的时候，函数才会返回该封装格式，否则返回NULL。

该函数中涉及到一个结构体AVProbeData，用于存储输入文件的一些信息，它的定义如下所示。

```

1.  /**
2.   * This structure contains the data a format has to probe a file.
3.   */
4.  typedef struct AVProbeData {
5.      const char *filename;
6.      unsigned char *buf; /**< Buffer must have AVPROBE_PADDING_SIZE of extra allocated bytes filled with zero. */
7.      int buf_size;      /**< Size of buf except extra allocated bytes */
8.      const char *mime_type; /**< mime_type, when known. */
9.  } AVProbeData;

```

av_probe_input_format2()函数的定义位于libavformat/format.c，如下所示。

```

1.  AVInputFormat *av_probe_input_format2(AVProbeData *pd, int is_opened, int *score_max)
2.  {
3.      int score_ret;
4.      AVInputFormat *fmt = av_probe_input_format3(pd, is_opened, &score_ret);
5.      if (score_ret > *score_max) {
6.          *score_max = score_ret;
7.          return fmt;
8.      } else
9.          return NULL;
10. }

```

从函数中可以看出，av_probe_input_format2()调用了av_probe_input_format3()，并且增加了一个判断，当av_probe_input_format3()返回的分数大于score_max的时候，才会返回AVInputFormat，否则返回NULL。

下面我们看一下av_probe_input_format3()。

av_probe_input_format3()

av_probe_input_format3()是一个API函数，声明位于libavformat/avformat.h，如下所示。

```

1.  /**
2.   * Guess the file format.
3.   *
4.   * @param is_opened Whether the file is already opened; determines whether
5.   *                 demuxers with or without AVFMT_NOFILE are probed.
6.   * @param score_ret The score of the best detection.
7.   */
8.  AVInputFormat *av_probe_input_format3(AVProbeData *pd, int is_opened, int *score_ret);

```

从函数声明中可以看出，av_probe_input_format3()和av_probe_input_format2()的区别是函数的第3个参数不同：av_probe_input_format2()是

一个分数的门限值，而av_probe_input_format3()是一个探测后的最匹配的格式的分数值。

av_probe_input_format3()的定义位于libavformat/format.c，如下所示。

```
[cpp]
1. #define AVPROBE_PADDING_SIZE 32          ///< extra allocated bytes at the end of the probe buffer
2. #define AVPROBE_SCORE_EXTENSION 50 ///< score for file extension
3. #define AVPROBE_SCORE_MIME 75 ///< score for file mime type
4. #define AVPROBE_SCORE_MAX 100 ///< maximum score
5.
6. AVInputFormat *av_probe_input_format3(AVProbeData *pd, int is_opened,
7.                                       int *score_ret)
8. {
9.     AVProbeData lpd = *pd;
10.    AVInputFormat *fmt1 = NULL, *fmt;
11.    int score, nodat = 0, score_max = 0;
12.    const static uint8_t zerobuffer[AVPROBE_PADDING_SIZE];
13.
14.    if (!lpd.buf)
15.        lpd.buf = zerobuffer;
16.
17.    if (lpd.buf_size > 10 && ff_id3v2_match(lpd.buf, ID3v2_DEFAULT_MAGIC)) {
18.        int id3len = ff_id3v2_tag_len(lpd.buf);
19.        if (lpd.buf_size > id3len + 16) {
20.            lpd.buf += id3len;
21.            lpd.buf_size -= id3len;
22.        } else if (id3len >= PROBE_BUF_MAX) {
23.            nodat = 2;
24.        } else
25.            nodat = 1;
26.    }
27.
28.    fmt = NULL;
29.    while ((fmt1 = av_iformat_next(fmt1))) {
30.        if (!is_opened == !(fmt1->flags & AVFMT_NOFILE) && strcmp(fmt1->name, "image2"))
31.            continue;
32.        score = 0;
33.        if (fmt1->read_probe) {
34.            score = fmt1->read_probe(&lpd);
35.            if (fmt1->extensions && av_match_ext(lpd.filename, fmt1->extensions)) {
36.                if (nodat == 0) score = FFMAX(score, 1);
37.                else if (nodat == 1) score = FFMAX(score, AVPROBE_SCORE_EXTENSION / 2 - 1);
38.                else
39.                    score = FFMAX(score, AVPROBE_SCORE_EXTENSION);
40.            } else if (fmt1->extensions) {
41.                if (av_match_ext(lpd.filename, fmt1->extensions))
42.                    score = AVPROBE_SCORE_EXTENSION;
43.            }
44.            if (av_match_name(lpd.mime_type, fmt1->mime_type))
45.                score = FFMAX(score, AVPROBE_SCORE_MIME);
46.            if (score > score_max) {
47.                score_max = score;
48.                fmt = fmt1;
49.            } else if (score == score_max)
50.                fmt = NULL;
51.        }
52.        if (nodat == 1)
53.            score_max = FFMIN(AVPROBE_SCORE_EXTENSION / 2 - 1, score_max);
54.        *score_ret = score_max;
55.
56.        return fmt;
57.    }
```

av_probe_input_format3()根据输入数据查找合适的AVInputFormat。输入的数据位于AVProbeData中。前文已经提到过，AVProbeData定义如下。

```
[cpp]
1. /**
2.  * This structure contains the data a format has to probe a file.
3.  */
4. typedef struct AVProbeData {
5.     const char *filename;
6.     unsigned char *buf; ///< Buffer must have AVPROBE_PADDING_SIZE of extra allocated bytes filled with zero. */
7.     int buf_size;       ///< Size of buf except extra allocated bytes */
8.     const char *mime_type; ///< mime_type, when known. */
9. } AVProbeData;
```

其中filename是文件路径，buf存储用于推测AVInputFormat的媒体数据，最后还有个mime_type保存媒体的类型。其中buf可以为空，但是其后面无论如何都需要填充AVPROBE_PADDING_SIZE个0（AVPROBE_PADDING_SIZE取值为32，即32个0）。

该函数最主要的部分是一个循环。该循环调用av_iformat_next()遍历FFmpeg中所有的AVInputFormat，并根据以下规则确定AVInputFormat和输入媒体数据的匹配分数（score，反应匹配程度）：

- (1) 如果AVInputFormat中包含read_probe(), 就调用read_probe()函数获取匹配分数（这一方法如果结果匹配的话，一般会获得AVPROBE_SCORE_MAX的分值，即100分）。如果不包含该函数，就使用av_match_ext()函数比较输入媒体的扩展名和AVInputFormat的扩展名是否匹配，如果匹配的话，设定匹配分数为AVPROBE_SCORE_EXTENSION（AVPROBE_SCORE_EXTENSION取值为50，即50分）。
- (2) 使用av_match_name()比较输入媒体的mime_type和AVInputFormat的mime_type，如果匹配的话，设定匹配分数为AVPROBE_SCORE_MIME（AVPROBE_SCORE_MIME取值为75，即75分）。
- (3) 如果该AVInputFormat的匹配分数大于此前的最大匹配分数，则记录当前的匹配分数为最大匹配分数，并且记录当前的AVInputFormat为最佳匹配的AVInputFormat。

上述过程中涉及到以下几个知识点：

AVInputFormat->read_probe()

AVInputFormat中包含read_probe()是用于获得匹配函数的函数指针，不同的封装格式包含不同的实现函数。例如，FLV封装格式的AVInputFormat模块定义（位于libavformat/flvdec.c）如下所示。

```
[cpp]
1. AVInputFormat ff_flv_demuxer = {
2.     .name      = "flv",
3.     .long_name  = NULL_IF_CONFIG_SMALL("FLV (Flash Video)"),
4.     .priv_data_size = sizeof(FLVContext),
5.     .read_probe  = flv_probe,
6.     .read_header = flv_read_header,
7.     .read_packet = flv_read_packet,
8.     .read_seek   = flv_read_seek,
9.     .read_close  = flv_read_close,
10.    .extensions  = "flv",
11.    .priv_class   = &flv_class,
12. };
```

其中，read_probe()函数对应的是flv_probe()函数。我们可以看一下flv_probe()函数的定义：

```
[cpp]
1. static int flv_probe(AVProbeData *p)
2. {
3.     return probe(p, 0);
4. }
```

可见flv_probe()调用了probe()函数。probe()函数的定义如下。

```
[cpp]
1. static int probe(AVProbeData *p, int live)
2. {
3.     const uint8_t *d = p->buf;
4.     unsigned offset = AV_RB32(d + 5);
5.
6.     if (d[0] == 'F' &&
7.         d[1] == 'L' &&
8.         d[2] == 'V' &&
9.         d[3] < 5 && d[5] == 0 &&
10.         offset + 100 < p->buf_size &&
11.         offset > 8) {
12.         int is_live = !memcmp(d + offset + 40, "NGINX RTMP", 10);
13.
14.         if (live == is_live)
15.             return AVPROBE_SCORE_MAX;
16.     }
17.     return 0;
18. }
```

从probe()函数我们可以看出，该函数做了如下工作：

- (1) 获得第6至第9字节的数据（对应Headersize字段）并且做大小端转换，然后存入offset变量。之所以要进行大小端转换是因为FLV是以“大端”方式存储数据，而操作系统是以“小端”方式存储数据，这一转换主要通过AV_RB32()函数实现。AV_RB32()是一个宏定义，其对应的函数是av_bswap32()。
- (2) 检查开头3个字符（Signature）是否为“FLV”。
- (3) 第4个字节（Version）小于5。

(4) 第6个字节 (Headersize的第1个字节?) 为0。

(5) offset取值大于8。

参照FLV文件头的格式可以对上述判断有一个更清晰的认识：

Flv Header	Signature (3 字节) 为文件标识, 总为"FLV", (0x46, 0x4c, 0x66)
	Version (1 字节) 为版本, 目前为 0x01
	Flags (1 字节) 前 5 位保留, 必须为 0。第 6 位表示是否存在音频 Tag。第 7 位保留, 必须为 0。第 8 位表示是否存在视频 Tag。
	Headersize (4 字节) 为从 File Header 开始到 File Body 开始的字节数, 版本 1 中总为 9。

此外代码中还包含了有关live方式的FLV格式的判断, 在这里我们不加探讨。对于我们打开FLV文件来说, live和is_live两个变量取值都为0。也就是说满足上述5个条件的话, 就可以认为输入媒体数据是FLV封装格式了。满足上述条件, probe()函数返回AVPROBE_SCORE_MAX (AVPROBE_SCORE_MAX取值为100, 即100分), 否则返回0 (0分)。

av_match_name()

av_match_name()是一个API函数, 声明位于libavutil\avstring.h, 如下所示。

```
[cpp]
1.  /**
2.   * Match instances of a name in a comma-separated list of names.
3.   * @param name Name to look for.
4.   * @param names List of names.
5.   * @return 1 on match, 0 otherwise.
6.   */
7.  int av_match_name(const char *name, const char *names);
```

av_match_name()用于比较两个格式的名称。简单地说就是比较字符串。注意该函数的字符串是不区分大小写的: 字符都转换为小写进行比较。

```
[cpp]
1.  int av_match_name(const char *name, const char *names)
2.  {
3.      const char *p;
4.      int len, namelen;
5.
6.      if (!name || !names)
7.          return 0;
8.
9.      namelen = strlen(name);
10.     while ((p = strchr(names, ',')) {
11.         len = FFMX(p - names, namelen);
12.         if (!av_strcasecmp(name, names, len))
13.             return 1;
14.         names = p + 1;
15.     }
16.     return !av_strcasecmp(name, names);
17. }
```

上述函数还有一点需要注意, 其中使用了一个while()循环, 用于搜索","。这是因为FFmpeg中有些格式是对应多种格式名称的, 例如MKV格式的解复用器 (Demuxer) 的定义如下。

```
[cpp]
1.  AVInputFormat ff_matroska_demuxer = {
2.      .name = "matroska,webm",
3.      .long_name = NULL_IF_CONFIG_SMALL("Matroska / WebM"),
4.      .extensions = "mkv,mk3d,mka,mks",
5.      .priv_data_size = sizeof(MatroskaDemuxContext),
6.      .read_probe = matroska_probe,
7.      .read_header = matroska_read_header,
8.      .read_packet = matroska_read_packet,
9.      .read_close = matroska_read_close,
10.     .read_seek = matroska_read_seek,
11.     .mime_type = "audio/webm,audio/x-matroska,video/webm,video/x-matroska"
12. };
```

从代码可以看出, ff_matroska_demuxer中的name字段对应"matroska,webm", mime_type字段对应"audio/webm,audio/x-matroska,video/webm,video/x-matroska"。av_match_name()函数对于这样的字符串, 会把它按照","截断成一个个的名称, 然后一一进行比较。

av_match_ext()

av_match_ext()是一个API函数，声明位于libavformat\avformat.h（注意位置和av_match_name()不一样），如下所示。

```
[cpp]
1.  /**
2.   * Return a positive value if the given filename has one of the given
3.   * extensions, 0 otherwise.
4.   *
5.   * @param filename  file name to check against the given extensions
6.   * @param extensions a comma-separated list of filename extensions
7.   */
8.  int av_match_ext(const char *filename, const char *extensions);
```

av_match_ext()用于比较文件的后缀。该函数首先通过反向查找的方式找到输入文件名中的“.”，就可以通过获取“.”后面的字符串来得到该文件的后缀。然后调用av_match_name()，采用和比较格式名称的方法比较两个后缀。

```
[cpp]
1.  int av_match_ext(const char *filename, const char *extensions)
2.  {
3.      const char *ext;
4.
5.      if (!filename)
6.          return 0;
7.
8.      ext = strrchr(filename, '.');
9.      if (ext)
10.         return av_match_name(ext + 1, extensions);
11.      return 0;
12.  }
```

avio_open2()

有关avio_open2()的分析可以参考文章：[FFmpeg源代码简单分析：avio_open2\(\)](#)

av_probe_input_buffer2()

av_probe_input_buffer2()是一个API函数，它根据输入的媒体数据推测该媒体数据的AVInputFormat，声明位于libavformat\avformat.h，如下所示。

```
[cpp]
1.  /**
2.   * Probe a bytestream to determine the input format. Each time a probe returns
3.   * with a score that is too low, the probe buffer size is increased and another
4.   * attempt is made. When the maximum probe size is reached, the input format
5.   * with the highest score is returned.
6.   *
7.   * @param pb the bytestream to probe
8.   * @param fmt the input format is put here
9.   * @param filename the filename of the stream
10.  * @param logctx the log context
11.  * @param offset the offset within the bytestream to probe from
12.  * @param max_probe_size the maximum probe buffer size (zero for default)
13.  * @return the score in case of success, a negative value corresponding to an
14.  *         the maximal score is AVPROBE_SCORE_MAX
15.  * AVERROR code otherwise
16.  */
17.  int av_probe_input_buffer2(AVIOContext *pb, AVInputFormat **fmt,
18.                             const char *filename, void *logctx,
19.                             unsigned int offset, unsigned int max_probe_size);
```

av_probe_input_buffer2()参数的含义如下所示：

- pb：用于读取数据的AVIOContext。
- fmt：输出推测出来的AVInputFormat。
- filename：输入媒体的路径。
- logctx：日志（没有研究过）。
- offset：开始推测AVInputFormat的偏移量。
- max_probe_size：用于推测格式的媒体数据的最大值。

返回推测后得到的AVInputFormat的匹配分数。

av_probe_input_buffer2()的定义位于libavformat/format.c, 如下所示。

```
[cpp]
1. int av_probe_input_buffer2(AVIOContext *pb, AVInputFormat **fmt,
2.                           const char *filename, void *logctx,
3.                           unsigned int offset, unsigned int max_probe_size)
4. {
5.     AVProbeData pd = { filename ? filename : "" };
6.     uint8_t *buf = NULL;
7.     int ret = 0, probe_size, buf_offset = 0;
8.     int score = 0;
9.     int ret2;
10.
11.     if (!max_probe_size)
12.         max_probe_size = PROBE_BUF_MAX;
13.     else if (max_probe_size < PROBE_BUF_MIN) {
14.         av_log(logctx, AV_LOG_ERROR,
15.              "Specified probe size value %u cannot be < %u\n", max_probe_size, PROBE_BUF_MIN);
16.         return AVERROR(EINVAL);
17.     }
18.
19.     if (offset >= max_probe_size)
20.         return AVERROR(EINVAL);
21.
22.     if (pb->av_class) {
23.         uint8_t *mime_type_opt = NULL;
24.         av_opt_get(pb, "mime_type", AV_OPT_SEARCH_CHILDREN, &mime_type_opt);
25.         pd.mime_type = (const char *)mime_type_opt;
26.     }
27.     #if 0
28.     if (!*fmt && pb->av_class && av_opt_get(pb, "mime_type", AV_OPT_SEARCH_CHILDREN, &mime_type) >= 0 && mime_type) {
29.         if (!av_strcasecmp(mime_type, "audio/aacp")) {
30.             *fmt = av_find_input_format("aac");
31.         }
32.         av_freep(&mime_type);
33.     }
34.     #endif
35.
36.     for (probe_size = PROBE_BUF_MIN; probe_size <= max_probe_size && !*fmt;
37.          probe_size = FFMIN(probe_size << 1,
38.                             FFMAX(max_probe_size, probe_size + 1))) {
39.         score = probe_size < max_probe_size ? AVPROBE_SCORE_RETRY : 0;
40.
41.         /* Read probe data. */
42.         if ((ret = av_reallocp(&buf, probe_size + AVPROBE_PADDING_SIZE)) < 0)
43.             goto fail;
44.         if ((ret = avio_read(pb, buf + buf_offset,
45.                             probe_size - buf_offset)) < 0) {
46.             /* Fail if error was not end of file, otherwise, lower score. */
47.             if (ret != AVERROR_EOF)
48.                 goto fail;
49.
50.             score = 0;
51.             ret = 0;          /* error was end of file, nothing read */
52.         }
53.         buf_offset += ret;
54.         if (buf_offset < offset)
55.             continue;
56.         pd.buf_size = buf_offset - offset;
57.         pd.buf = &buf[offset];
58.
59.         memset(pd.buf + pd.buf_size, 0, AVPROBE_PADDING_SIZE);
60.
61.         /* Guess file format. */
62.         *fmt = av_probe_input_format2(&pd, 1, &score);
63.         if (*fmt) {
64.             /* This can only be true in the last iteration. */
65.             if (score <= AVPROBE_SCORE_RETRY) {
66.                 av_log(logctx, AV_LOG_WARNING,
67.                      "Format %s detected only with low score of %d, "
68.                      "misdetecction possible!\n", (*fmt)->name, score);
69.             } else
70.                 av_log(logctx, AV_LOG_DEBUG,
71.                      "Format %s probed with size=%d and score=%d\n",
72.                      (*fmt)->name, probe_size, score);
73.             #if 0
74.             FILE *f = fopen("probestat.tmp", "ab");
75.             fprintf(f, "probe_size:%d format:%s score:%d filename:%s\n", probe_size, (*fmt)->name, score, filename);
76.             fclose(f);
77.             #endif
78.         }
79.     }
80.
81.     if (!*fmt)
```

```

82.         ret = AVERROR_INVALIDDATA;
83.
84.     fail:
85.         /* Rewind. Reuse probe buffer to avoid seeking. */
86.         ret2 = ffio_rewind_with_probe_data(pb, &buf, buf_offset);
87.         if (ret >= 0)
88.             ret = ret2;
89.
90.         av_freep(&pd.mime_type);
91.         return ret < 0 ? ret : score;
92.     }

```

av_probe_input_buffer2()首先需要确定用于推测格式的媒体数据的最大值max_probe_size。max_probe_size默认为PROBE_BUF_MAX（PROBE_BUF_MAX取值为1 << 20，即1048576Byte，大约1MB）。

在确定了max_probe_size之后，函数就会进入到一个循环中，调用avio_read()读取数据并且使用av_probe_input_format2()（该函数前文已经记录过）推测文件格式。

肯定有人会奇怪这里为什么要使用一个循环，而不是只运行一次？其实这个循环是一个逐渐增加输入媒体数据量的过程。av_probe_input_buffer2()并不是一次性读取max_probe_size字节的媒体数据，我个人感觉可能是因为这样做不是很经济，因为推测大部分媒体格式根本用不到1MB这么多的媒体数据。因此函数中使用一个probe_size存储需要读取的字节数，并且随着循环次数的增加逐渐增加这个值。函数首先从PROBE_BUF_MIN（取值为2048）个字节开始读取，如果通过这些数据已经可以推测出AVInputFormat，那么就可以直接退出循环了（参考for循环的判断条件“!fmt”）；如果没有推测出来，就增加probe_size的量为过去的2倍（参考for循环的表达式“probe_size << 1”），继续推测AVInputFormat；如果一直读取到max_probe_size字节的数据依然没能确定AVInputFormat，则会退出循环并且返回错误信息。

AVInputFormat-> read_header()

在调用完init_input()完成基本的初始化并且推测得到相应的AVInputFormat之后，avformat_open_input()会调用AVInputFormat的read_header()方法读取媒体文件的文件头并且完成相关的初始化工作。read_header()是一个位于AVInputFormat结构体中的一个函数指针，对于不同的封装格式，会调用不同的read_header()的实现函数。举个例子，当输入视频的封装格式为FLV的时候，会调用FLV的AVInputFormat中的read_header()。FLV的AVInputFormat定义位于libavformat/flvdec.c文件中，如下所示。

```

1.  AVInputFormat ff_flv_demuxer = {
2.      .name           = "flv",
3.      .long_name      = NULL_IF_CONFIG_SMALL("FLV (Flash Video)"),
4.      .priv_data_size = sizeof(FLVContext),
5.      .read_probe      = flv_probe,
6.      .read_header     = flv_read_header,
7.      .read_packet     = flv_read_packet,
8.      .read_seek       = flv_read_seek,
9.      .read_close      = flv_read_close,
10.     .extensions      = "flv",
11.     .priv_class       = &flv_class,
12. };

```

可以看出read_header()指向了flv_read_header()函数。flv_read_header()的实现同样位于libavformat/flvdec.c文件中，如下所示。

```

1. static int flv_read_header(AVFormatContext *s)
2. {
3.     int offset, flags;
4.
5.     avio_skip(s->pb, 4);
6.     flags = avio_r8(s->pb);
7.
8.     s->ctx_flags |= AVFMTCTX_NOHEADER;
9.
10.    if (flags & FLV_HEADER_FLAG_HASVIDEO)
11.        if (!create_stream(s, AVMEDIA_TYPE_VIDEO))
12.            return AVERROR(ENOMEM);
13.    if (flags & FLV_HEADER_FLAG_HASAUDIO)
14.        if (!create_stream(s, AVMEDIA_TYPE_AUDIO))
15.            return AVERROR(ENOMEM);
16.    // Flag doesn't indicate whether or not there is script-data present. Must
17.    // create that stream if it's encountered.
18.
19.    offset = avio_rb32(s->pb);
20.    avio_seek(s->pb, offset, SEEK_SET);
21.    avio_skip(s->pb, 4);
22.
23.    s->start_time = 0;
24.
25.    return 0;
26. }

```

可以看出，函数读取了FLV的文件头并且判断其中是否包含视频流和音频流。如果包含视频流或者音频流，就会调用create_stream()函数。

create_stream()函数定义也位于libavformat/flvdec.c中，如下所示。

```

1. static AVStream *create_stream(AVFormatContext *s, int codec_type)
2. {
3.     AVStream *st = avformat_new_stream(s, NULL);
4.     if (!st)
5.         return NULL;
6.     st->codec->codec_type = codec_type;
7.     if (s->nb_streams>=3 || (s->nb_streams==2
8.         && s->streams[0]->codec->codec_type != AVMEDIA_TYPE_DATA
9.         && s->streams[1]->codec->codec_type != AVMEDIA_TYPE_DATA))
10.        s->ctx_flags &= ~AVFMTCTX_NOHEADER;
11.
12.    avpriv_set_pts_info(st, 32, 1, 1000); /* 32 bit pts in ms */
13.    return st;
14. }

```

从代码中可以看出，create_stream()调用了API函数avformat_new_stream()创建相应的视频流和音频流。

上面这段解析FLV头的代码可以参考一下FLV封装格式的文件头格式，如下图所示。

Flv Header	Signature (3 字节) 为文件标识，总为“FLV”，(0x46, 0x4c, 0x66)
	Version (1 字节) 为版本，目前为 0x01
	Flags (1 字节) 前 5 位保留，必须为 0。第 6 位表示是否存在音频 Tag。第 7 位保留，必须为 0。第 8 位表示是否存在视频 Tag。
	Headersize (4 字节) 为从 File Header 开始到 File Body 开始的字节数，版本 1 中总为 9。

经过上面的步骤AVInputFormat的read_header()完成了音视频流对应的AVStream的创建。至此，avformat_open_input()中的主要代码分析完毕。

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/44064715>

文章标签：[AVFormatContext](#)

[打开媒体](#)

[AVInputFormat](#)

[源代码](#)

[FFmpeg](#)

个人分类：[FFMPEG](#)

所属专栏：[FFmpeg](#)

此PDF由spygg生成,请尊重原作者版权!!!

我的邮箱:liushide@163.com