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
/**
 * Open an input stream and read the header. The codecs are not opened.
 * The stream must be closed with avformat_close_input().
 *
 * @param ps Pointer to user-supplied AVFormatContext (allocated by avformat_alloc_context).
 *           May be a pointer to NULL, in which case an AVFormatContext is allocated by this
 *           function and written into ps.
 *           Note that a user-supplied AVFormatContext will be freed on failure.
 * @param filename Name of the stream to open.
 * @param fmt If non-NULL, this parameter forces a specific input format.
 *            Otherwise the format is autodetected.
 * @param options  A dictionary filled with AVFormatContext and demuxer-private options.
 *                 On return this parameter will be destroyed and replaced with a dict containing
 *                 options that were not found. May be NULL.
 *
 * @return 0 on success, a negative AVERROR on failure.
 *
 * @note If you want to use custom IO, preallocate the format context and set its pb field.
 */
int avformat_open_input(AVFormatContext **ps, const char *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。
dictionay：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
int avformat_open_input(AVFormatContext **ps, const char *filename,
                        AVInputFormat *fmt, AVDictionary **options)
{
    AVFormatContext *s = *ps;
    int ret = 0;
    AVDictionary *tmp = NULL;
    ID3v2ExtraMeta *id3v2_extra_meta = NULL;

    if (!s && !(s = avformat_alloc_context()))
        return AVERROR(ENOMEM);
    if (!s->av_class) {
        av_log(NULL, AV_LOG_ERROR, "Input context has not been properly allocated by avformat_alloc_context() and is not NULL either\
);
        return AVERROR(EINVAL);
    }
    if (fmt)
        s->iformat = fmt;

    if (options)
        av_dict_copy(&tmp, *options, 0);

    if ((ret = av_opt_set_dict(s, &tmp)) < 0)
        goto fail;

    if ((ret = init_input(s, filename, &tmp)) < 0)
        goto fail;
    s->probe_score = ret;

    if (s->format_whitelist && av_match_list(s->iformat->name, s->format_whitelist, ',') <= 0) {
        av_log(s, AV_LOG_ERROR, "Format not on whitelist\n");
        ret = AVERROR(EINVAL);
        goto fail;
    }

    avio_skip(s->pb, s->skip_initial_bytes);

    /* Check filename in case an image number is expected. */
    if (s->iformat->flags & AVFMT_NEEDNUMBER) {
        if (!av_filename_number_test(filename)) {
            ret = AVERROR(EINVAL);
            goto fail;
        }
    }

    s->duration = s->start_time = AV_NOPTS_VALUE;
    av_strlcpy(s->filename, filename ? filename : "", sizeof(s->filename));

    /* Allocate private data. */
    if (s->iformat->priv_data_size > 0) {
        if (!(s->priv_data = av_mallocz(s->iformat->priv_data_size))) {
            ret = AVERROR(ENOMEM);
            goto fail;
        }
        if (s->iformat->priv_class) {
            *(const AVClass **) s->priv_data = s->iformat->priv_class;
            av_opt_set_defaults(s->priv_data);
            if ((ret = av_opt_set_dict(s->priv_data, &tmp)) < 0)
                goto fail;
        }
    }

    /* e.g. AVFMT_NOFILE formats will not have a AVIOContext */
    if (s->pb)
        ff_id3v2_read(s, ID3v2_DEFAULT_MAGIC, &id3v2_extra_meta, 0);

    if (!(s->flags&AVFMT_FLAG_PRIV_OPT) && s->iformat->read_header)
        if ((ret = s->iformat->read_header(s)) < 0)
            goto fail;

    if (id3v2_extra_meta) {
        if (!strcmp(s->iformat->name, "mp3") || !strcmp(s->iformat->name, "aac") ||
            !strcmp(s->iformat->name, "tta")) {
            if ((ret = ff_id3v2_parse_apic(s, &id3v2_extra_meta)) < 0)
                goto fail;
        } else
            av_log(s, AV_LOG_DEBUG, "demuxer does not support additional id3 data, skipping\n");
    }
    ff_id3v2_free_extra_meta(&id3v2_extra_meta);

    if ((ret = avformat_queue_attached_pictures(s)) < 0)
        goto fail;

    if (!(s->flags&AVFMT_FLAG_PRIV_OPT) && s->pb && !s->data_offset)
        s->data_offset = avio_tell(s->pb);

    s->raw_packet_buffer_remaining_size = RAW_PACKET_BUFFER_SIZE;

    if (options) {
        av_dict_free(options);
        *options = tmp;
```

```
        *options = tmp;
    }
    *ps = s;
    return 0;

fail:
    ff_id3v2_free_extra_meta(&id3v2_extra_meta);
    av_dict_free(&tmp);
    if (s->pb && !(s->flags & AVFMT_FLAG_CUSTOM_IO))
        avio_close(s->pb);
    avformat_free_context(s);
    *ps = NULL;
    return ret;
}
```

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libavformat\utils.c，如下所示。

```cpp
/* Open input file and probe the format if necessary. */
static int init_input(AVFormatContext *s, const char *filename,
                      AVDictionary **options)
{
    int ret;
    AVProbeData pd = { filename, NULL, 0 };
    int score = AVPROBE_SCORE_RETRY;

    if (s->pb) {
        s->flags |= AVFMT_FLAG_CUSTOM_IO;
        if (!s->iformat)
            return av_probe_input_buffer2(s->pb, &s->iformat, filename,
                                          s, 0, s->format_probesize);
        else if (s->iformat->flags & AVFMT_NOFILE)
            av_log(s, AV_LOG_WARNING, "Custom AVIOContext makes no sense and "
                                      "will be ignored with AVFMT_NOFILE format.\n");
        return 0;
    }

    if ((s->iformat && s->iformat->flags & AVFMT_NOFILE) ||
        (!s->iformat && (s->iformat = av_probe_input_format2(&pd, 0, &score))))
        return score;

    if ((ret = avio_open2(&s->pb, filename, AVIO_FLAG_READ | s->avio_flags,
                          &s->interrupt_callback, options)) < 0)
        return ret;
    if (s->iformat)
        return 0;
    return av_probe_input_buffer2(s->pb, &s->iformat, filename,
                                  s, 0, s->format_probesize);
}
```

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：

```cpp
#define AVPROBE_SCORE_RETRY (AVPROBE_SCORE_MAX/4)
```

其中AVPROBE_SCORE_MAX是score的最大值，取值是100：

```cpp
#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。

```cpp
/**
 * Guess the file format.
 *
 * @param pd        data to be probed
 * @param is_opened Whether the file is already opened; determines whether
 *                  demuxers with or without AVFMT_NOFILE are probed.
 * @param score_max A probe score larger that this is required to accept a
 *                  detection, the variable is set to the actual detection
 *                  score afterwards.
 *                  If the score is <= AVPROBE_SCORE_MAX / 4 it is recommended
 *                  to retry with a larger probe buffer.
 */
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。

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

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

```cpp
AVInputFormat *av_probe_input_format2(AVProbeData *pd, int is_opened, int *score_max)
{
    int score_ret;
    AVInputFormat *fmt = av_probe_input_format3(pd, is_opened, &score_ret);
    if (score_ret > *score_max) {
        *score_max = score_ret;
        return fmt;
    } else
        return NULL;
}
```

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。

```cpp
/**
 * Guess the file format.
 *
 * @param is_opened Whether the file is already opened; determines whether
 *                  demuxers with or without AVFMT_NOFILE are probed.
 * @param score_ret The score of the best detection.
 */
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
#define AVPROBE_PADDING_SIZE 32             ///< extra allocated bytes at the end of the probe buffer
#define AVPROBE_SCORE_EXTENSION  50 ///< score for file extension
#define AVPROBE_SCORE_MIME       75 ///< score for file mime type
#define AVPROBE_SCORE_MAX       100 ///< maximum score

AVInputFormat *av_probe_input_format3(AVProbeData *pd, int is_opened,
                                      int *score_ret)
{
    AVProbeData lpd = *pd;
    AVInputFormat *fmt1 = NULL, *fmt;
    int score, nodat = 0, score_max = 0;
    const static uint8_t zerobuffer[AVPROBE_PADDING_SIZE];

    if (!lpd.buf)
        lpd.buf = zerobuffer;

    if (lpd.buf_size > 10 && ff_id3v2_match(lpd.buf, ID3v2_DEFAULT_MAGIC)) {
        int id3len = ff_id3v2_tag_len(lpd.buf);
        if (lpd.buf_size > id3len + 16) {
            lpd.buf      += id3len;
            lpd.buf_size -= id3len;
        } else if (id3len >= PROBE_BUF_MAX) {
            nodat = 2;
        } else
            nodat = 1;
    }

    fmt = NULL;
    while ((fmt1 = av_iformat_next(fmt1))) {
        if (!is_opened == !(fmt1->flags & AVFMT_NOFILE) && strcmp(fmt1->name, "image2"))
            continue;
        score = 0;
        if (fmt1->read_probe) {
            score = fmt1->read_probe(&lpd);
            if (fmt1->extensions && av_match_ext(lpd.filename, fmt1->extensions)) {
                if      (nodat == 0) score = FFMAX(score, 1);
                else if (nodat == 1) score = FFMAX(score, AVPROBE_SCORE_EXTENSION / 2 - 1);
                else                 score = FFMAX(score, AVPROBE_SCORE_EXTENSION);
            }
        } else if (fmt1->extensions) {
            if (av_match_ext(lpd.filename, fmt1->extensions))
                score = AVPROBE_SCORE_EXTENSION;
        }
        if (av_match_name(lpd.mime_type, fmt1->mime_type))
            score = FFMAX(score, AVPROBE_SCORE_MIME);
        if (score > score_max) {
            score_max = score;
            fmt       = fmt1;
        } else if (score == score_max)
            fmt = NULL;
    }
    if (nodat == 1)
        score_max = FFMIN(AVPROBE_SCORE_EXTENSION / 2 - 1, score_max);
    *score_ret = score_max;

    return fmt;
}
```

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

E_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
AVInputFormat ff_flv_demuxer = {
    .name           = "flv",
    .long_name      = NULL_IF_CONFIG_SMALL("FLV (Flash Video)"),
    .priv_data_size = sizeof(FLVContext),
    .read_probe     = flv_probe,
    .read_header    = flv_read_header,
    .read_packet    = flv_read_packet,
    .read_seek      = flv_read_seek,
    .read_close     = flv_read_close,
    .extensions     = "flv",
    .priv_class     = &flv_class,
};
```

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
static int flv_probe(AVProbeData *p)
{
    return probe(p, 0);
}
```

可见flv_probe()调用了一个probe()函数。probe()函数的定义如下。

```cpp
static int probe(AVProbeData *p, int live)
{
    const uint8_t *d = p->buf;
    unsigned offset = AV_RB32(d + 5);

    if (d[0] == 'F' &&
        d[1] == 'L' &&
        d[2] == 'V' &&
        d[3] < 5 && d[5] == 0 &&
        offset + 100 < p->buf_size &&
        offset > 8) {
        int is_live = !memcmp(d + offset + 40, "NGINX RTMP", 10);

        if (live == is_live)
            return AVPROBE_SCORE_MAX;
    }
    return 0;
}
```

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：

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
/**
 * Match instances of a name in a comma-separated list of names.
 * @param name  Name to look for.
 * @param names List of names.
 * @return 1 on match, 0 otherwise.
 */
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
int av_match_name(const char *name, const char *names)
{
    const char *p;
    int len, namelen;

    if (!name || !names)
        return 0;

    namelen = strlen(name);
    while ((p = strchr(names, ','))) {
        len = FFMAX(p - names, namelen);
        if (!av_strncasecmp(name, names, len))
            return 1;
        names = p + 1;
    }
    return !av_strcasecmp(name, names);
}
```

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
AVInputFormat ff_matroska_demuxer = {
    .name           = "matroska,webm",
    .long_name      = NULL_IF_CONFIG_SMALL("Matroska / WebM"),
    .extensions     = "mkv,mk3d,mka,mks",
    .priv_data_size = sizeof(MatroskaDemuxContext),
    .read_probe     = matroska_probe,
    .read_header    = matroska_read_header,
    .read_packet    = matroska_read_packet,
    .read_close     = matroska_read_close,
    .read_seek      = matroska_read_seek,
    .mime_type      = "audio/webm,audio/x-matroska,video/webm,video/x-matroska"
};
```

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
/**
 * Return a positive value if the given filename has one of the given
 * extensions, 0 otherwise.
 *
 * @param filename   file name to check against the given extensions
 * @param extensions a comma-separated list of filename extensions
 */
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
int av_match_ext(const char *filename, const char *extensions)
{
    const char *ext;

    if (!filename)
        return 0;

    ext = strrchr(filename, '.');
    if (ext)
        return av_match_name(ext + 1, extensions);
    return 0;
}
```

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
/**
 * Probe a bytestream to determine the input format. Each time a probe returns
 * with a score that is too low, the probe buffer size is increased and another
 * attempt is made. When the maximum probe size is reached, the input format
 * with the highest score is returned.
 *
 * @param pb the bytestream to probe
 * @param fmt the input format is put here
 * @param filename the filename of the stream
 * @param logctx the log context
 * @param offset the offset within the bytestream to probe from
 * @param max_probe_size the maximum probe buffer size (zero for default)
 * @return the score in case of success, a negative value corresponding to an
 *         the maximal score is AVPROBE_SCORE_MAX
 * AVERROR code otherwise
 */
int av_probe_input_buffer2(AVIOContext *pb, AVInputFormat **fmt,
                           const char *filename, void *logctx,
                           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。

返回推测后的得到的AVInputFormat的匹配分数。

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

```cpp
```

```c
int av_probe_input_buffer2(AVIOContext *pb, AVInputFormat **fmt,
                           const char *filename, void *logctx,
                           unsigned int offset, unsigned int max_probe_size)
{
    AVProbeData pd = { filename ? filename : "" };
    uint8_t *buf = NULL;
    int ret = 0, probe_size, buf_offset = 0;
    int score = 0;
    int ret2;

    if (!max_probe_size)
        max_probe_size = PROBE_BUF_MAX;
    else if (max_probe_size < PROBE_BUF_MIN) {
        av_log(logctx, AV_LOG_ERROR,
               "Specified probe size value %u cannot be < %u\n", max_probe_size, PROBE_BUF_MIN);
        return AVERROR(EINVAL);
    }

    if (offset >= max_probe_size)
        return AVERROR(EINVAL);

    if (pb->av_class) {
        uint8_t *mime_type_opt = NULL;
        av_opt_get(pb, "mime_type", AV_OPT_SEARCH_CHILDREN, &mime_type_opt);
        pd.mime_type = (const char *)mime_type_opt;
    }
#if 0
    if (!*fmt && pb->av_class && av_opt_get(pb, "mime_type", AV_OPT_SEARCH_CHILDREN, &mime_type) >= 0 && mime_type) {
        if (!av_strcasecmp(mime_type, "audio/aacp")) {
            *fmt = av_find_input_format("aac");
        }
        av_freep(&mime_type);
    }
#endif

    for (probe_size = PROBE_BUF_MIN; probe_size <= max_probe_size && !*fmt;
         probe_size = FFMIN(probe_size << 1,
                            FFMAX(max_probe_size, probe_size + 1))) {
        score = probe_size < max_probe_size ? AVPROBE_SCORE_RETRY : 0;

        /* Read probe data. */
        if ((ret = av_reallocp(&buf, probe_size + AVPROBE_PADDING_SIZE)) < 0)
            goto fail;
        if ((ret = avio_read(pb, buf + buf_offset,
                             probe_size - buf_offset)) < 0) {
            /* Fail if error was not end of file, otherwise, lower score. */
            if (ret != AVERROR_EOF)
                goto fail;

            score = 0;
            ret   = 0;          /* error was end of file, nothing read */
        }
        buf_offset += ret;
        if (buf_offset < offset)
            continue;
        pd.buf_size = buf_offset - offset;
        pd.buf = &buf[offset];

        memset(pd.buf + pd.buf_size, 0, AVPROBE_PADDING_SIZE);

        /* Guess file format. */
        *fmt = av_probe_input_format2(&pd, 1, &score);
        if (*fmt) {
            /* This can only be true in the last iteration. */
            if (score <= AVPROBE_SCORE_RETRY) {
                av_log(logctx, AV_LOG_WARNING,
                       "Format %s detected only with low score of %d, "
                       "misdetection possible!\n", (*fmt)->name, score);
            } else
                av_log(logctx, AV_LOG_DEBUG,
                       "Format %s probed with size=%d and score=%d\n",
                       (*fmt)->name, probe_size, score);
#if 0
            FILE *f = fopen("probestat.tmp", "ab");
            fprintf(f, "probe_size:%d format:%s score:%d filename:%s\n", probe_size, (*fmt)->name, score, filename);
            fclose(f);
#endif
        }
    }

    if (!*fmt)
        ret = AVERROR_INVALIDDATA;

fail:
    /* Rewind. Reuse probe buffer to avoid seeking. */
    ret2 = ffio_rewind_with_probe_data(pb, &buf, buf_offset);
    if (ret >= 0)
        ret = ret2;

    av_freep(&pd.mime_type);
    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。

```cpp
AVInputFormat ff_flv_demuxer = {
    .name           = "flv",
    .long_name      = NULL_IF_CONFIG_SMALL("FLV (Flash Video)"),
    .priv_data_size = sizeof(FLVContext),
    .read_probe     = flv_probe,
    .read_header    = flv_read_header,
    .read_packet    = flv_read_packet,
    .read_seek      = flv_read_seek,
    .read_close     = flv_read_close,
    .extensions     = "flv",
    .priv_class     = &flv_class,
};
```

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。

```cpp
static int flv_read_header(AVFormatContext *s)
{
    int offset, flags;

    avio_skip(s->pb, 4);
    flags = avio_r8(s->pb);

    s->ctx_flags |= AVFMTCTX_NOHEADER;

    if (flags & FLV_HEADER_FLAG_HASVIDEO)
        if (!create_stream(s, AVMEDIA_TYPE_VIDEO))
            return AVERROR(ENOMEM);
    if (flags & FLV_HEADER_FLAG_HASAUDIO)
        if (!create_stream(s, AVMEDIA_TYPE_AUDIO))
            return AVERROR(ENOMEM);
    // Flag doesn't indicate whether or not there is script-data present. Must
    // create that stream if it's encountered.

    offset = avio_rb32(s->pb);
    avio_seek(s->pb, offset, SEEK_SET);
    avio_skip(s->pb, 4);

    s->start_time = 0;

    return 0;
}
```

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。

```cpp
static AVStream *create_stream(AVFormatContext *s, int codec_type)
{
    AVStream *st = avformat_new_stream(s, NULL);
    if (!st)
        return NULL;
    st->codec->codec_type = codec_type;
    if (s->nb_streams>=3 ||(   s->nb_streams==2
                           && s->streams[0]->codec->codec_type != AVMEDIA_TYPE_DATA
                           && s->streams[1]->codec->codec_type != AVMEDIA_TYPE_DATA))
        s->ctx_flags &= ~AVFMTCTX_NOHEADER;

    avpriv_set_pts_info(st, 32, 1, 1000); /* 32 bit pts in ms */
    return st;
}
```

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。

经过上面的步骤AVInputFormat的read_header()完成了视音频流对应的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:liushidc@163.com