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

2015年03月03日 16:25:13　阅读数：41683

==========

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常见结构体的初始化和销毁函数的源代码。常见的结构体在文章：

《FFMPEG中最关键的结构体之间的关系》中已经有过叙述，包括：

AVFormatContext：统领全局的基本结构体。主要用于处理封装格式（FLV/MKV/RMVB等）。

AVIOContext：输入输出对应的结构体，用于输入输出（读写文件，RTMP协议等）。

AVStream，AVCodecContext：视音频流对应的结构体，用于视音频编解码。

AVFrame：存储非压缩的数据（视频对应RGB/YUV像素数据，音频对应PCM采样数据）

AVPacket：存储压缩数据（视频对应H.264等码流数据，音频对应AAC/MP3等码流数据）

他们之间的关系如下图所示（详细信息可以参考上文提到的文章）。

下文简单分析一下上述几个结构体的初始化和销毁函数。这些函数列表如下。

结构体	初始化	销毁
AVFormatContext	avformat_alloc_context()	avformat_free_context()
AVIOContext	avio_alloc_context()	
AVStream	avformat_new_stream()	
AVCodecContext	avcodec_alloc_context3()	
AVFrame	av_frame_alloc(); av_image_fill_arrays()	av_frame_free()
AVPacket	av_init_packet(); av_new_packet()	av_free_packet()

下面进入正文。

AVFormatContext

AVFormatContext的初始化函数是avformat_alloc_context()，销毁函数是avformat_free_context()。

avformat_alloc_context()

avformat_alloc_context()的声明位于libavformat\avformat.h，如下所示。

```cpp
/**
 * Allocate an AVFormatContext.
 * avformat_free_context() can be used to freethe context and everything
 * allocated by the framework within it.
 */
AVFormatContext*avformat_alloc_context(void);
```

avformat_alloc_context()的定义位于libavformat\options.c。代码如下所示。

```cpp
AVFormatContext *avformat_alloc_context(void)
{
    AVFormatContext *ic;
    ic = av_malloc(sizeof(AVFormatContext));
    if (!ic) return ic;
    avformat_get_context_defaults(ic);


    ic->internal = av_mallocz(sizeof(*ic->internal));
    if (!ic->internal) {
        avformat_free_context(ic);
        return NULL;
    }


    return ic;
}
```

从代码中可以看出，avformat_alloc_context()调用av_malloc()为AVFormatContext结构体分配了内存，而且同时也给AVFormatContext中的internal字段分配内存（这个字段是FFmpeg内部使用的，先不分析）。此外调用了一个avformat_get_context_defaults()函数。该函数用于设置AVFormatContext的字段的默认值。它的定义也位于libavformat\options.c，确切的说就位于avformat_alloc_context()上面。我们看一下该函数的定义。

```cpp
static void avformat_get_context_defaults(AVFormatContext *s)
{
    memset(s, 0, sizeof(AVFormatContext));


    s->av_class = &av_format_context_class;


    av_opt_set_defaults(s);
}
```

从代码可以看出，avformat_get_context_defaults()首先调用memset()将AVFormatContext的所有字段置0。而后调用了一个函数av_opt_set_defaults()。av_opt_set_defaults()用于给字段设置默认值。
avformat_alloc_context()代码的函数调用关系如下图所示。

avformat_free_context()

avformat_free_context()的声明位于libavformat\avformat.h，如下所示。

```cpp
/**
 * Free an AVFormatContext and all its streams.
 * @param s context to free
 */
void avformat_free_context(AVFormatContext *s);
```

avformat_free_context()的定义位于libavformat\options.c。代码如下所示。

```cpp
void avformat_free_context(AVFormatContext *s)
{
    int i;


    if (!s)
        return;


    av_opt_free(s);
    if (s->iformat && s->iformat->priv_class && s->priv_data)
        av_opt_free(s->priv_data);
    if (s->oformat && s->oformat->priv_class && s->priv_data)
        av_opt_free(s->priv_data);


    for (i = s->nb_streams - 1; i >= 0; i--) {
        ff_free_stream(s, s->streams[i]);
    }
    for (i = s->nb_programs - 1; i >= 0; i--) {
        av_dict_free(&s->programs[i]->metadata);
        av_freep(&s->programs[i]->stream_index);
        av_freep(&s->programs[i]);
    }
    av_freep(&s->programs);
    av_freep(&s->priv_data);
    while (s->nb_chapters--) {
        av_dict_free(&s->chapters[s->nb_chapters]->metadata);
        av_freep(&s->chapters[s->nb_chapters]);
    }
    av_freep(&s->chapters);
    av_dict_free(&s->metadata);
    av_freep(&s->streams);
    av_freep(&s->internal);
    flush_packet_queue(s);
    av_free(s);
}
```

从代码中可以看出，avformat_free_context()调用了各式各样的销毁函数：av_opt_free()，av_freep()，av_dict_free()。这些函数分别用于释放不同种类的变量，在这里不再详细讨论。在这里看一个释放AVStream的函数ff_free_stream()。该函数的定义位于libavformat\options.c（其实就在avformat_free_context()上方）。

```cpp
void ff_free_stream(AVFormatContext *s, AVStream *st) {
    int j;
    av_assert0(s->nb_streams>0);
    av_assert0(s->streams[ s->nb_streams - 1 ] == st);


    for (j = 0; j < st->nb_side_data; j++)
        av_freep(&st->side_data[j].data);
    av_freep(&st->side_data);
    st->nb_side_data = 0;


    if (st->parser) {
        av_parser_close(st->parser);
    }
    if (st->attached_pic.data)
        av_free_packet(&st->attached_pic);
    av_dict_free(&st->metadata);
    av_freep(&st->probe_data.buf);
    av_freep(&st->index_entries);
    av_freep(&st->codec->extradata);
    av_freep(&st->codec->subtitle_header);
    av_freep(&st->codec);
    av_freep(&st->priv_data);
    if (st->info)
        av_freep(&st->info->duration_error);
    av_freep(&st->info);
    av_freep(&s->streams[ --s->nb_streams ]);
}
```

从代码中可以看出，与释放AVFormatContext类似，释放AVStream的时候，也是调用了av_freep()，av_dict_free()这些函数释放有关的字段。如果使用了parser的话，会调用av_parser_close()关闭该parser。

AVIOContext

avio_alloc_context()

AVIOContext的初始化函数是avio_alloc_context()，销毁的时候使用av_free()释放掉其中的缓存即可。它的声明位于libavformat\avio.h中，如下所示。

```cpp
/**
 * Allocate and initialize an AVIOContext for buffered I/O. It must be later
 * freed with av_free().
 *
 * @param buffer Memory block for input/output operations via AVIOContext.
 *        The buffer must be allocated with av_malloc() and friends.
 * @param buffer_size The buffer size is very important for performance.
 *        For protocols with fixed blocksize it should be set to this blocksize.
 *        For others a typical size is a cache page, e.g. 4kb.
 * @param write_flag Set to 1 if the buffer should be writable, 0 otherwise.
 * @param opaque An opaque pointer to user-specific data.
 * @param read_packet  A function for refilling the buffer, may be NULL.
 * @param write_packet A function for writing the buffer contents, may be NULL.
 *        The function may not change the input buffers content.
 * @param seek A function for seeking to specified byte position, may be NULL.
 *
 * @return Allocated AVIOContext or NULL on failure.
 */
AVIOContext *avio_alloc_context(
                  unsigned char *buffer,
                  int buffer_size,
                  int write_flag,
                  void *opaque,
                  int (*read_packet)(void *opaque, uint8_t *buf, int buf_size),
                  int (*write_packet)(void *opaque, uint8_t *buf, int buf_size),
                  int64_t (*seek)(void *opaque, int64_t offset, int whence));
```

avio_alloc_context()定义位于libavformat\aviobuf.c中，如下所示。

```cpp
AVIOContext *avio_alloc_context(
                  unsigned char *buffer,
                  int buffer_size,
                  int write_flag,
                  void *opaque,
                  int (*read_packet)(void *opaque, uint8_t *buf, int buf_size),
                  int (*write_packet)(void *opaque, uint8_t *buf, int buf_size),
                  int64_t (*seek)(void *opaque, int64_t offset, int whence))
{
    AVIOContext *s = av_mallocz(sizeof(AVIOContext));
    if (!s)
        return NULL;
    ffio_init_context(s, buffer, buffer_size, write_flag, opaque,
                  read_packet, write_packet, seek);
    return s;
}
```

从代码中可以看出，avio_alloc_context()首先调用av_mallocz()为AVIOContext分配内存。而后调用了一个函数ffio_init_context()。该函数完成了真正的初始化工作。我们看一下ffio_init_context()函数的定义。

```cpp
int ffio_init_context(AVIOContext *s,
                      unsigned char *buffer,
                      int buffer_size,
                      int write_flag,
                      void *opaque,
                      int (*read_packet)(void *opaque, uint8_t *buf, int buf_size),
                      int (*write_packet)(void *opaque, uint8_t *buf, int buf_size),
                      int64_t (*seek)(void *opaque, int64_t offset, int whence))
{
    s->buffer      = buffer;
    s->orig_buffer_size =
    s->buffer_size = buffer_size;
    s->buf_ptr     = buffer;
    s->opaque      = opaque;
    s->direct      = 0;


    url_resetbuf(s, write_flag ? AVIO_FLAG_WRITE : AVIO_FLAG_READ);


    s->write_packet    = write_packet;
    s->read_packet     = read_packet;
    s->seek            = seek;
    s->pos             = 0;
    s->must_flush      = 0;
    s->eof_reached     = 0;
    s->error           = 0;
    s->seekable        = seek ? AVIO_SEEKABLE_NORMAL : 0;
    s->max_packet_size = 0;
    s->update_checksum = NULL;


    if (!read_packet && !write_flag) {
        s->pos     = buffer_size;
        s->buf_end = s->buffer + buffer_size;
    }
    s->read_pause = NULL;
    s->read_seek  = NULL;


    return 0;
}
```

从函数的代码可以看出，ffio_init_context()对AVIOContext中的缓存，函数指针等等进行了赋值。

AVStream，AVCodecContext

AVStream的初始化函数是avformat_new_stream()，销毁函数使用销毁AVFormatContext的avformat_free_context()就可以了。

avformat_new_stream()

avformat_new_stream()的声明位于libavformat\avformat.h中，如下所示。

```cpp
/**
 * Add a new stream to a media file.
 *
 * When demuxing, it is called by the demuxer in read_header(). If the
 * flag AVFMTCTX_NOHEADER is set in s.ctx_flags, then it may also
 * be called in read_packet().
 *
 * When muxing, should be called by the user before avformat_write_header().
 *
 * User is required to call avcodec_close() and avformat_free_context() to
 * clean up the allocation by avformat_new_stream().
 *
 * @param s media file handle
 * @param c If non-NULL, the AVCodecContext corresponding to the new stream
 * will be initialized to use this codec. This is needed for e.g. codec-specific
 * defaults to be set, so codec should be provided if it is known.
 *
 * @return newly created stream or NULL on error.
 */
AVStream *avformat_new_stream(AVFormatContext *s, const AVCodec *c);
```

avformat_new_stream()的定义位于libavformat\utils.c中，如下所示。

```cpp
AVStream *avformat_new_stream(AVFormatContext *s, const AVCodec *c)
{
    AVStream *st;
    int i;
    AVStream **streams;


    if (s->nb_streams >= INT_MAX/sizeof(*streams))
        return NULL;
    streams = av_realloc_array(s->streams, s->nb_streams + 1, sizeof(*streams));
    if (!streams)
        return NULL;
    s->streams = streams;


    st = av_mallocz(sizeof(AVStream));
    if (!st)
        return NULL;
    if (!(st->info = av_mallocz(sizeof(*st->info)))) {
        av_free(st);
        return NULL;
    }
    st->info->last_dts = AV_NOPTS_VALUE;


    st->codec = avcodec_alloc_context3(c);
    if (s->iformat) {
        /* no default bitrate if decoding */
        st->codec->bit_rate = 0;


        /* default pts setting is MPEG-like */
        avpriv_set_pts_info(st, 33, 1, 90000);
    }


    st->index      = s->nb_streams;
    st->start_time = AV_NOPTS_VALUE;
    st->duration   = AV_NOPTS_VALUE;
    /* we set the current DTS to 0 so that formats without any timestamps
     * but durations get some timestamps, formats with some unknown
     * timestamps have their first few packets buffered and the
     * timestamps corrected before they are returned to the user */
    st->cur_dts       = s->iformat ? RELATIVE_TS_BASE : 0;
    st->first_dts     = AV_NOPTS_VALUE;
    st->probe_packets = MAX_PROBE_PACKETS;
    st->pts_wrap_reference = AV_NOPTS_VALUE;
    st->pts_wrap_behavior = AV_PTS_WRAP_IGNORE;


    st->last_IP_pts = AV_NOPTS_VALUE;
    st->last_dts_for_order_check = AV_NOPTS_VALUE;
    for (i = 0; i < MAX_REORDER_DELAY + 1; i++)
        st->pts_buffer[i] = AV_NOPTS_VALUE;


    st->sample_aspect_ratio = (AVRational) { 0, 1 };


#if FF_API_R_FRAME_RATE
    st->info->last_dts      = AV_NOPTS_VALUE;
#endif
    st->info->fps_first_dts = AV_NOPTS_VALUE;
    st->info->fps_last_dts  = AV_NOPTS_VALUE;


    st->inject_global_side_data = s->internal->inject_global_side_data;


    s->streams[s->nb_streams++] = st;
    return st;
}
```

从代码中可以看出，avformat_new_stream()首先调用av_mallocz()为AVStream分配内存。接着给新分配的AVStream的各个字段赋上默认值。然后调用了另一个函数avcodec_alloc_context3()初始化AVStream中的AVCodecContext。

avcodec_alloc_context3()

avcodec_alloc_context3()的声明位于libavcodec\avcodec.h中，如下所示。

```cpp
/**
 * Allocate an AVCodecContext and set its fields to default values. The
 * resulting struct should be freed with avcodec_free_context().
 *
 * @param codec if non-NULL, allocate private data and initialize defaults
 *              for the given codec. It is illegal to then call avcodec_open2()
 *              with a different codec.
 *              If NULL, then the codec-specific defaults won't be initialized,
 *              which may result in suboptimal default settings (this is
 *              important mainly for encoders, e.g. libx264).
 *
 * @return An AVCodecContext filled with default values or NULL on failure.
 * @see avcodec_get_context_defaults
 */
AVCodecContext *avcodec_alloc_context3(const AVCodec *codec);
```

下面我们看一下avcodec_alloc_context3()的定义。下面我们看一下avcodec_alloc_context3()的定义。avcodec_alloc_context3()的定义位于libavcodec\options.c中。

```cpp
AVCodecContext *avcodec_alloc_context3(const AVCodec *codec)
{
    AVCodecContext *avctx= av_malloc(sizeof(AVCodecContext));


    if (!avctx)
        return NULL;


    if(avcodec_get_context_defaults3(avctx, codec) < 0){
        av_free(avctx);
        return NULL;
    }


    return avctx;
}
```

从代码中可以看出，avcodec_alloc_context3()首先调用av_malloc()为AVCodecContext分配存储空间，然后调用了一个函数avcodec_get_context_defaults3()用于设置该AVCodecContext的默认值。avcodec_get_context_defaults3()的定义如下。

```cpp
int avcodec_get_context_defaults3(AVCodecContext *s, const AVCodec *codec)
{
    int flags=0;
    memset(s, 0, sizeof(AVCodecContext));


    s->av_class = &av_codec_context_class;


    s->codec_type = codec ? codec->type : AVMEDIA_TYPE_UNKNOWN;
    if (codec)
        s->codec_id = codec->id;


    if(s->codec_type == AVMEDIA_TYPE_AUDIO)
        flags= AV_OPT_FLAG_AUDIO_PARAM;
    else if(s->codec_type == AVMEDIA_TYPE_VIDEO)
        flags= AV_OPT_FLAG_VIDEO_PARAM;
    else if(s->codec_type == AVMEDIA_TYPE_SUBTITLE)
        flags= AV_OPT_FLAG_SUBTITLE_PARAM;
    av_opt_set_defaults2(s, flags, flags);


    s->time_base           = (AVRational){0,1};
    s->get_buffer2         = avcodec_default_get_buffer2;
    s->get_format          = avcodec_default_get_format;
    s->execute             = avcodec_default_execute;
    s->execute2            = avcodec_default_execute2;
    s->sample_aspect_ratio = (AVRational){0,1};
    s->pix_fmt             = AV_PIX_FMT_NONE;
    s->sample_fmt          = AV_SAMPLE_FMT_NONE;
    s->timecode_frame_start = -1;


    s->reordered_opaque    = AV_NOPTS_VALUE;
    if(codec && codec->priv_data_size){
        if(!s->priv_data){
            s->priv_data= av_mallocz(codec->priv_data_size);
            if (!s->priv_data) {
                return AVERROR(ENOMEM);
            }
        }
        if(codec->priv_class){
            *(const AVClass**)s->priv_data = codec->priv_class;
            av_opt_set_defaults(s->priv_data);
        }
    }
    if (codec && codec->defaults) {
        int ret;
        const AVCodecDefault *d = codec->defaults;
        while (d->key) {
            ret = av_opt_set(s, d->key, d->value, 0);
            av_assert0(ret >= 0);
            d++;
        }
    }
    return 0;
}
```

avformat_new_stream()函数的调用结构如下所示。

AVFrame

AVFrame的初始化函数是av_frame_alloc()，销毁函数是av_frame_free()。在这里有一点需要注意，旧版的FFmpeg都是使用avcodec_alloc_frame()初始化AVFrame的，但是我在写这篇文章的时候，avcodec_alloc_frame()已经被标记为“过时的”了，为了保证与时俱进，决定分析新的API——av_frame_alloc()。

av_frame_alloc()

av_frame_alloc()的声明位于libavutil\frame.h，如下所示。

```cpp
/**
 * Allocate an AVFrame and set its fields to default values.  The resulting
 * struct must be freed using av_frame_free().
 *
 * @return An AVFrame filled with default values or NULL on failure.
 *
 * @note this only allocates the AVFrame itself, not the data buffers. Those
 * must be allocated through other means, e.g. with av_frame_get_buffer() or
 * manually.
 */
AVFrame *av_frame_alloc(void);
```

av_frame_alloc()的定义位于libavutil\frame.c。代码如下所示。

```cpp
AVFrame *av_frame_alloc(void)
{
    AVFrame *frame = av_mallocz(sizeof(*frame));


    if (!frame)
        return NULL;


    frame->extended_data = NULL;
    get_frame_defaults(frame);


    return frame;
}
```

从代码可以看出，av_frame_alloc()首先调用av_mallocz()为AVFrame结构体分配内存。而后调用了一个函数get_frame_defaults()用于设置一些默认参数。get_frame_defaults()定义如下。

```cpp
static void get_frame_defaults(AVFrame *frame)
{
    if (frame->extended_data != frame->data)
        av_freep(&frame->extended_data);


    memset(frame, 0, sizeof(*frame));


    frame->pts                   =
    frame->pkt_dts               =
    frame->pkt_pts               = AV_NOPTS_VALUE;
    av_frame_set_best_effort_timestamp(frame, AV_NOPTS_VALUE);
    av_frame_set_pkt_duration        (frame, 0);
    av_frame_set_pkt_pos             (frame, -1);
    av_frame_set_pkt_size            (frame, -1);
    frame->key_frame           = 1;
    frame->sample_aspect_ratio = (AVRational){ 0, 1 };
    frame->format              = -1; /* unknown */
    frame->extended_data       = frame->data;
    frame->color_primaries     = AVCOL_PRI_UNSPECIFIED;
    frame->color_trc           = AVCOL_TRC_UNSPECIFIED;
    frame->colorspace          = AVCOL_SPC_UNSPECIFIED;
    frame->color_range         = AVCOL_RANGE_UNSPECIFIED;
    frame->chroma_location     = AVCHROMA_LOC_UNSPECIFIED;
}
```

从av_frame_alloc()的代码我们可以看出，该函数并没有为AVFrame的像素数据分配空间。因此AVFrame中的像素数据的空间需要自行分配空间，例如使用avpicture_fill()，av_image_fill_arrays()等函数。

av_frame_alloc()函数的调用结构如下所示。

avpicture_fill()

avpicture_fill()的声明位于libavcodec\avcodec.h，如下所示。

```cpp
/**
 * Setup the picture fields based on the specified image parameters
 * and the provided image data buffer.
 *
 * The picture fields are filled in by using the image data buffer
 * pointed to by ptr.
 *
 * If ptr is NULL, the function will fill only the picture linesize
 * array and return the required size for the image buffer.
 *
 * To allocate an image buffer and fill the picture data in one call,
 * use avpicture_alloc().
 *
 * @param picture       the picture to be filled in
 * @param ptr           buffer where the image data is stored, or NULL
 * @param pix_fmt       the pixel format of the image
 * @param width         the width of the image in pixels
 * @param height        the height of the image in pixels
 * @return the size in bytes required for src, a negative error code
 * in case of failure
 *
 * @see av_image_fill_arrays()
 */
int avpicture_fill(AVPicture *picture, const uint8_t *ptr,
                   enum AVPixelFormat pix_fmt, int width, int height);
```

avpicture_fill()的定义位于libavcodec\avpicture.c，如下所示。

PS：目测这个函数未来也有可能成为“过时的”函数，因为通过观察这一年FFmpeg代码的变化，发现FFmpeg组织似乎想把AVFrame相关的函数（原先定义在AVCodec的头文件中）从AVCodec的代码中分离出来，形成一套单独的API。所以很多和AVFrame相关的名称为avcodec_XXX()的函数都被标记上了“过时的”标记。当然，上述推测也是我自己猜测的。

```cpp
int avpicture_fill(AVPicture *picture, const uint8_t *ptr,
                   enum AVPixelFormat pix_fmt, int width, int height)
{
    return av_image_fill_arrays(picture->data, picture->linesize,
                                ptr, pix_fmt, width, height, 1);
}
```

从代码中可以看出，avpicture_fill()仅仅是简单调用了一下av_image_fill_arrays()。也就是说这两个函数实际上是等同的。

av_image_fill_arrays()

av_image_fill_arrays()的声明位于libavutil\imgutils.h中，如下所示。

```cpp
/**
 * Setup the data pointers and linesizes based on the specified image
 * parameters and the provided array.
 *
 * The fields of the given image are filled in by using the src
 * address which points to the image data buffer. Depending on the
 * specified pixel format, one or multiple image data pointers and
 * line sizes will be set.  If a planar format is specified, several
 * pointers will be set pointing to the different picture planes and
 * the line sizes of the different planes will be stored in the
 * lines_sizes array. Call with !src to get the required
 * size for the src buffer.
 *
 * To allocate the buffer and fill in the dst_data and dst_linesize in
 * one call, use av_image_alloc().
 *
 * @param dst_data      data pointers to be filled in
 * @param dst_linesizes linesizes for the image in dst_data to be filled in
 * @param src           buffer which will contain or contains the actual image data, can be NULL
 * @param pix_fmt       the pixel format of the image
 * @param width         the width of the image in pixels
 * @param height        the height of the image in pixels
 * @param align         the value used in src for linesize alignment
 * @return the size in bytes required for src, a negative error code
 * in case of failure
 */
int av_image_fill_arrays(uint8_t *dst_data[4], int dst_linesize[4],
                         const uint8_t *src,
                         enum AVPixelFormat pix_fmt, int width, int height, int align);
```

av_image_fill_arrays()的定义位于libavutil\imgutils.c中。

```cpp
int av_image_fill_arrays(uint8_t *dst_data[4], int dst_linesize[4],
                         const uint8_t *src,
                         enum AVPixelFormat pix_fmt, int width, int height, int align)
{
    int ret, i;


    if ((ret = av_image_check_size(width, height, 0, NULL)) < 0)
        return ret;


    if ((ret = av_image_fill_linesizes(dst_linesize, pix_fmt, width)) < 0)
        return ret;


    for (i = 0; i < 4; i++)
        dst_linesize[i] = FFALIGN(dst_linesize[i], align);


    if ((ret = av_image_fill_pointers(dst_data, pix_fmt, width, NULL, dst_linesize)) < 0)
        return ret;


    return av_image_fill_pointers(dst_data, pix_fmt, height, (uint8_t *)src, dst_linesize);
}
```

av_image_fill_arrays()函数中包含3个函数：av_image_check_size()，av_image_fill_linesizes()，av_image_fill_pointers()。av_image_check_size()用于检查输入的宽高参数是否合理，即不能太大或者为负数。av_image_fill_linesizes()用于填充dst_linesize。av_image_fill_pointers()则用于填充dst_data。它们的定义相对比较简单，不再详细分析。
av_image_check_size()代码如下所示。

```cpp
int av_image_check_size(unsigned int w, unsigned int h, int log_offset, void *log_ctx)
{
    ImgUtils imgutils = { &imgutils_class, log_offset, log_ctx };


    if ((int)w>0 && (int)h>0 && (w+128)*(uint64_t)(h+128) < INT_MAX/8)
        return 0;


    av_log(&imgutils, AV_LOG_ERROR, "Picture size %ux%u is invalid\n", w, h);
    return AVERROR(EINVAL);
}
```

av_image_fill_linesizes()代码如下所示。

```cpp
int av_image_fill_linesizes(int linesizes[4], enum AVPixelFormat pix_fmt, int width)
{
    int i, ret;
    const AVPixFmtDescriptor *desc = av_pix_fmt_desc_get(pix_fmt);
    int max_step     [4];       /* max pixel step for each plane */
    int max_step_comp[4];       /* the component for each plane which has the max pixel step */


    memset(linesizes, 0, 4*sizeof(linesizes[0]));


    if (!desc || desc->flags & AV_PIX_FMT_FLAG_HWACCEL)
        return AVERROR(EINVAL);


    av_image_fill_max_pixsteps(max_step, max_step_comp, desc);
    for (i = 0; i < 4; i++) {
        if ((ret = image_get_linesize(width, i, max_step[i], max_step_comp[i], desc)) < 0)
            return ret;
        linesizes[i] = ret;
    }


    return 0;
}
```

av_image_fill_pointers()代码如下所示。

```cpp
int av_image_fill_pointers(uint8_t *data[4], enum AVPixelFormat pix_fmt, int height,
                           uint8_t *ptr, const int linesizes[4])
{
    int i, total_size, size[4] = { 0 }, has_plane[4] = { 0 };


    const AVPixFmtDescriptor *desc = av_pix_fmt_desc_get(pix_fmt);
    memset(data     , 0, sizeof(data[0])*4);


    if (!desc || desc->flags & AV_PIX_FMT_FLAG_HWACCEL)
        return AVERROR(EINVAL);


    data[0] = ptr;
    if (linesizes[0] > (INT_MAX - 1024) / height)
        return AVERROR(EINVAL);
    size[0] = linesizes[0] * height;


    if (desc->flags & AV_PIX_FMT_FLAG_PAL ||
        desc->flags & AV_PIX_FMT_FLAG_PSEUDOPAL) {
        size[0] = (size[0] + 3) & ~3;
        data[1] = ptr + size[0]; /* palette is stored here as 256 32 bits words */
        return size[0] + 256 * 4;
    }


    for (i = 0; i < 4; i++)
        has_plane[desc->comp[i].plane] = 1;


    total_size = size[0];
    for (i = 1; i < 4 && has_plane[i]; i++) {
        int h, s = (i == 1 || i == 2) ? desc->log2_chroma_h : 0;
        data[i] = data[i-1] + size[i-1];
        h = (height + (1 << s) - 1) >> s;
        if (linesizes[i] > INT_MAX / h)
            return AVERROR(EINVAL);
        size[i] = h * linesizes[i];
        if (total_size > INT_MAX - size[i])
            return AVERROR(EINVAL);
        total_size += size[i];
    }


    return total_size;
}
```

avpicture_fill()函数调用关系如下图所示。

av_frame_free()

av_frame_free ()的声明位于libavutil\frame.h，如下所示。

```cpp
/**
 * Free the frame and any dynamically allocated objects in it,
 * e.g. extended_data. If the frame is reference counted, it will be
 * unreferenced first.
 *
 * @param frame frame to be freed. The pointer will be set to NULL.
 */
void av_frame_free(AVFrame **frame);
```

av_frame_free ()的定义位于libavutil\frame.c。代码如下所示。

```cpp
void av_frame_free(AVFrame **frame)
{
    if (!frame || !*frame)
        return;


    av_frame_unref(*frame);
    av_freep(frame);
}
```

在释放AVFrame结构体之前，首先调用了一个函数av_frame_unref()。av_frame_unref()也是一个FFmpeg的API，它的作用是释放AVFrame中参考的缓存（还没完全弄懂），并且重置AVFrame中的字段。调用这个函数的目的应该是为了确保AVFrame可以被正常释放。代码如下。

```cpp
void av_frame_unref(AVFrame *frame)
{
    int i;


    for (i = 0; i < frame->nb_side_data; i++) {
        free_side_data(&frame->side_data[i]);
    }
    av_freep(&frame->side_data);


    for (i = 0; i < FF_ARRAY_ELEMS(frame->buf); i++)
        av_buffer_unref(&frame->buf[i]);
    for (i = 0; i < frame->nb_extended_buf; i++)
        av_buffer_unref(&frame->extended_buf[i]);
    av_freep(&frame->extended_buf);
    av_dict_free(&frame->metadata);
    av_buffer_unref(&frame->qp_table_buf);


    get_frame_defaults(frame);
}
```

AVPacket

AVPacket的初始化函数有两个：av_init_packet()，av_new_packet()。销毁函数是av_free_packet()。在初始化函数中av_init_packet()比较简单，初始化一些字段；而av_new_packet()相对“高级”一些，除了包含av_init_packet()的功能之外，还包含了AVPacket内部内存的分配。下面分别看看这些函数。

av_init_packet()

av_init_packet()的声明位于libavcodec\avcodec.h，如下所示。

```cpp
/**
 * Initialize optional fields of a packet with default values.
 *
 * Note, this does not touch the data and size members, which have to be
 * initialized separately.
 *
 * @param pkt packet
 */
void av_init_packet(AVPacket *pkt);
```

av_init_packet()的定义位于libavcodec\avpacket.c。如下所示。

```cpp
void av_init_packet(AVPacket *pkt)
{
    pkt->pts                   = AV_NOPTS_VALUE;
    pkt->dts                   = AV_NOPTS_VALUE;
    pkt->pos                   = -1;
    pkt->duration              = 0;
    pkt->convergence_duration = 0;
    pkt->flags                 = 0;
    pkt->stream_index          = 0;
#if FF_API_DESTRUCT_PACKET
FF_DISABLE_DEPRECATION_WARNINGS
    pkt->destruct              = NULL;
FF_ENABLE_DEPRECATION_WARNINGS
#endif
    pkt->buf                   = NULL;
    pkt->side_data             = NULL;
    pkt->side_data_elems       = 0;
}
```

av_new_packet()

av_new_packet()的声明位于libavcodec\avcodec.h。如下所示。

```cpp
/**
 * Allocate the payload of a packet and initialize its fields with
 * default values.
 *
 * @param pkt packet
 * @param size wanted payload size
 * @return 0 if OK, AVERROR_xxx otherwise
 */
int av_new_packet(AVPacket *pkt, int size);
```

av_new_packet()的定义位于libavcodec\avpacket.c。如下所示。

```cpp
int av_new_packet(AVPacket *pkt, int size)
{
    AVBufferRef *buf = NULL;
    int ret = packet_alloc(&buf, size);
    if (ret < 0)
        return ret;


    av_init_packet(pkt);
    pkt->buf      = buf;
    pkt->data     = buf->data;
    pkt->size     = size;
#if FF_API_DESTRUCT_PACKET
FF_DISABLE_DEPRECATION_WARNINGS
    pkt->destruct = dummy_destruct_packet;
FF_ENABLE_DEPRECATION_WARNINGS
#endif


    return 0;
}
```

从代码可以看出，av_new_packet()调用了av_init_packet(pkt)。此外还调用了一个函数packet_alloc()。packet_alloc()函数的定义如下。

```cpp
static int packet_alloc(AVBufferRef **buf, int size)
{
    int ret;
    if ((unsigned)size >= (unsigned)size + FF_INPUT_BUFFER_PADDING_SIZE)
        return AVERROR(EINVAL);


    ret = av_buffer_realloc(buf, size + FF_INPUT_BUFFER_PADDING_SIZE);
    if (ret < 0)
        return ret;


    memset((*buf)->data + size, 0, FF_INPUT_BUFFER_PADDING_SIZE);


    return 0;
}
```

packet_alloc()中调用av_buffer_realloc()为AVPacket分配内存。然后调用memset()将分配的内存置0。
PS：发现AVPacket的结构随着FFmpeg的发展越发复杂了。原先AVPacket中的数据仅仅存在一个uint8_t类型的数组里，而现在已经使用一个专门的结构体AVBufferRef存储数据。
av_new_packet()代码的函数调用关系如下图所示。

av_free_packet()

av_free_packet()的声明位于libavcodec\avcodec.h，如下所示。

```cpp
/**
 * Free a packet.
 *
 * @param pkt packet to free
 */
void av_free_packet(AVPacket *pkt);
```

av_free_packet()的定义位于libavcodec\avpacket.c。如下所示。

```cpp
void av_free_packet(AVPacket *pkt)
{
    if (pkt) {
FF_DISABLE_DEPRECATION_WARNINGS
        if (pkt->buf)
            av_buffer_unref(&pkt->buf);
#if FF_API_DESTRUCT_PACKET
        else if (pkt->destruct)
            pkt->destruct(pkt);
        pkt->destruct = NULL;
#endif
FF_ENABLE_DEPRECATION_WARNINGS
        pkt->data            = NULL;
        pkt->size            = 0;


        av_packet_free_side_data(pkt);
    }
}
```

从代码可以看出，av_free_packet()调用av_buffer_unref()释放AVPacket中的数据，而后还调用了av_packet_free_side_data()释放了side_data（存储封装格式可以提供的额外的数据）。

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

文章标签： ffmpeg avformatcontext avstream avcodeccontext avpacket

个人分类： FFMPEG

所属专栏： FFmpeg