# 原 FFmpeg源代码简单分析：avcodec_open2()

2015年03月08日 00:14:29　阅读数：33254

=====================================================

FFmpeg的库函数源代码分析文章列表：

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的avcodec_open2()函数。该函数用于初始化一个视音频编解码器的AVCodecContext。avcodec_open2()的声明位于libavcodec\avcodec.h，如下所示。

[cpp]

```cpp
/**
 * Initialize the AVCodecContext to use the given AVCodec. Prior to using this
 * function the context has to be allocated with avcodec_alloc_context3().
 *
 * The functions avcodec_find_decoder_by_name(), avcodec_find_encoder_by_name(),
 * avcodec_find_decoder() and avcodec_find_encoder() provide an easy way for
 * retrieving a codec.
 *
 * @warning This function is not thread safe!
 *
 * @code
 * avcodec_register_all();
 * av_dict_set(&opts, "b", "2.5M", 0);
 * codec = avcodec_find_decoder(AV_CODEC_ID_H264);
 * if (!codec)
 *     exit(1);
 *
 * context = avcodec_alloc_context3(codec);
 *
 * if (avcodec_open2(context, codec, opts) < 0)
 *     exit(1);
 * @endcode
 *
 * @param avctx The context to initialize.
 * @param codec The codec to open this context for. If a non-NULL codec has been
 *              previously passed to avcodec_alloc_context3() or
 *              avcodec_get_context_defaults3() for this context, then this
 *              parameter MUST be either NULL or equal to the previously passed
 *              codec.
 * @param options A dictionary filled with AVCodecContext and codec-private options.
 *                On return this object will be filled with options that were not found.
 *
 * @return zero on success, a negative value on error
 * @see avcodec_alloc_context3(), avcodec_find_decoder(), avcodec_find_encoder(),
 *      av_dict_set(), av_opt_find().
 */
int avcodec_open2(AVCodecContext *avctx, const AVCodec *codec, AVDictionary **options);
```

用中文简单转述一下avcodec_open2()各个参数的含义：

avctx：需要初始化的AVCodecContext。

codec：输入的AVCodec

options：一些选项。例如使用libx264编码的时候，“preset”，“tune”等都可以通过该参数设置。

该函数最典型的例子可以参考：

最简单的基于FFMPEG的视频编码器（YUV编码为H.264）

## 函数调用关系图

avcodec_open2()函数调用关系非常简单，如下图所示。

## avcodec_open2()

avcodec_open2()的定义位于libavcodec\utils.c，如下所示。

[cpp]

```cpp
int avcodec_open2(AVCodecContext *avctx, const AVCodec *codec, AVDictionary **options)
{
    int ret = 0;
    AVDictionary *tmp = NULL;
    //如果已经打开，直接返回
    if (avcodec_is_open(avctx))
        return 0;

```

```c
    if ((!codec && !avctx->codec)) {
        av_log(avctx, AV_LOG_ERROR, "No codec provided to avcodec_open2()\n");
        return AVERROR(EINVAL);
    }
    if ((codec && avctx->codec && codec != avctx->codec)) {
        av_log(avctx, AV_LOG_ERROR, "This AVCodecContext was allocated for %s, "
                                    "but %s passed to avcodec_open2()\n", avctx->codec->name, codec->name);
        return AVERROR(EINVAL);
    }
    if (!codec)
        codec = avctx->codec;

    if (avctx->extradata_size < 0 || avctx->extradata_size >= FF_MAX_EXTRADATA_SIZE)
        return AVERROR(EINVAL);

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

    ret = ff_lock_avcodec(avctx);
    if (ret < 0)
        return ret;
    //各种Malloc
    avctx->internal = av_mallocz(sizeof(AVCodecInternal));
    if (!avctx->internal) {
        ret = AVERROR(ENOMEM);
        goto end;
    }

    avctx->internal->pool = av_mallocz(sizeof(*avctx->internal->pool));
    if (!avctx->internal->pool) {
        ret = AVERROR(ENOMEM);
        goto free_and_end;
    }

    avctx->internal->to_free = av_frame_alloc();
    if (!avctx->internal->to_free) {
        ret = AVERROR(ENOMEM);
        goto free_and_end;
    }

    if (codec->priv_data_size > 0) {
        if (!avctx->priv_data) {
            avctx->priv_data = av_mallocz(codec->priv_data_size);
            if (!avctx->priv_data) {
                ret = AVERROR(ENOMEM);
                goto end;
            }
            if (codec->priv_class) {
                *(const AVClass **)avctx->priv_data = codec->priv_class;
                av_opt_set_defaults(avctx->priv_data);
            }
        }
        if (codec->priv_class && (ret = av_opt_set_dict(avctx->priv_data, &tmp)) < 0)
            goto free_and_end;
    } else {
        avctx->priv_data = NULL;
    }
    //将输入的AVDictionary形式的选项设置到AVCodecContext
    if ((ret = av_opt_set_dict(avctx, &tmp)) < 0)
        goto free_and_end;

    if (avctx->codec_whitelist && av_match_list(codec->name, avctx->codec_whitelist, ',') <= 0) {
        av_log(avctx, AV_LOG_ERROR, "Codec (%s) not on whitelist\n", codec->name);
        ret = AVERROR(EINVAL);
        goto free_and_end;
    }

    // only call ff_set_dimensions() for non H.264/VP6F codecs so as not to overwrite previously setup dimensions
    if (!(avctx->coded_width && avctx->coded_height && avctx->width && avctx->height &&
          (avctx->codec_id == AV_CODEC_ID_H264 || avctx->codec_id == AV_CODEC_ID_VP6F))) {
    if (avctx->coded_width && avctx->coded_height)
        ret = ff_set_dimensions(avctx, avctx->coded_width, avctx->coded_height);
    else if (avctx->width && avctx->height)
        ret = ff_set_dimensions(avctx, avctx->width, avctx->height);
    if (ret < 0)
        goto free_and_end;
    }
    //检查宽和高
    if ((avctx->coded_width || avctx->coded_height || avctx->width || avctx->height)
        && (  av_image_check_size(avctx->coded_width, avctx->coded_height, 0, avctx) < 0
           || av_image_check_size(avctx->width,       avctx->height,       0, avctx) < 0)) {
        av_log(avctx, AV_LOG_WARNING, "Ignoring invalid width/height values\n");
        ff_set_dimensions(avctx, 0, 0);
    }
    //检查宽高比
    if (avctx->width > 0 && avctx->height > 0) {
        if (av_image_check_sar(avctx->width, avctx->height,
                               avctx->sample_aspect_ratio) < 0) {
            av_log(avctx, AV_LOG_WARNING, "ignoring invalid SAR: %u/%u\n",
                   avctx->sample_aspect_ratio.num,
                   avctx->sample_aspect_ratio.den);
            avctx->sample_aspect_ratio = (AVRational){ 0, 1 };
        }
    }

    /* if the decoder init function was already called previously,
     * free the already allocated subtitle_header before overwriting it */
    if (av_codec_is_decoder(codec))
        av_freep(&avctx->subtitle_header);

    if (avctx->channels > FF_SANE_NB_CHANNELS) {
        ret = AVERROR(EINVAL);
        goto free_and_end;
    }

    avctx->codec = codec;
    if ((avctx->codec_type == AVMEDIA_TYPE_UNKNOWN || avctx->codec_type == codec->type) &&
        avctx->codec_id == AV_CODEC_ID_NONE) {
        avctx->codec_type = codec->type;
        avctx->codec_id   = codec->id;
    }
    if (avctx->codec_id != codec->id || (avctx->codec_type != codec->type
                                         && avctx->codec_type != AVMEDIA_TYPE_ATTACHMENT)) {
        av_log(avctx, AV_LOG_ERROR, "Codec type or id mismatches\n");
        ret = AVERROR(EINVAL);
        goto free_and_end;
    }
    avctx->frame_number = 0;
    avctx->codec_descriptor = avcodec_descriptor_get(avctx->codec_id);
    //检查编码器是否出于“实验”阶段
    if (avctx->codec->capabilities & CODEC_CAP_EXPERIMENTAL &&
        avctx->strict_std_compliance > FF_COMPLIANCE_EXPERIMENTAL) {
        const char *codec_string = av_codec_is_encoder(codec) ? "encoder" : "decoder";
        AVCodec *codec2;
        av_log(avctx, AV_LOG_ERROR,
               "The %s '%s' is experimental but experimental codecs are not enabled, "
               "add '-strict %d' if you want to use it.\n",
               codec_string, codec->name, FF_COMPLIANCE_EXPERIMENTAL);
        codec2 = av_codec_is_encoder(codec) ? avcodec_find_encoder(codec->id) : avcodec_find_decoder(codec->id);
        if (!(codec2->capabilities & CODEC_CAP_EXPERIMENTAL))
            av_log(avctx, AV_LOG_ERROR, "Alternatively use the non experimental %s '%s'.\n",
                codec_string, codec2->name);
        ret = AVERROR_EXPERIMENTAL;
        goto free_and_end;
    }

    if (avctx->codec_type == AVMEDIA_TYPE_AUDIO &&
        (!avctx->time_base.num || !avctx->time_base.den)) {
        avctx->time_base.num = 1;
        avctx->time_base.den = avctx->sample_rate;
    }

    if (!HAVE_THREADS)
        av_log(avctx, AV_LOG_WARNING, "Warning: not compiled with thread support, using thread emulation\n");

    if (CONFIG_FRAME_THREAD_ENCODER) {
        ff_unlock_avcodec(); //we will instanciate a few encoders thus kick the counter to prevent false detection of a problem
        ret = ff_frame_thread_encoder_init(avctx, options ? *options : NULL);
        ff_lock_avcodec(avctx);
        if (ret < 0)
            goto free_and_end;
    }

    if (HAVE_THREADS
        && !(avctx->internal->frame_thread_encoder && (avctx->active_thread_type&FF_THREAD_FRAME))) {
        ret = ff_thread_init(avctx);
        if (ret < 0) {
            goto free_and_end;
        }
    }
    if (!HAVE_THREADS && !(codec->capabilities & CODEC_CAP_AUTO_THREADS))
        avctx->thread_count = 1;

    if (avctx->codec->max_lowres < avctx->lowres || avctx->lowres < 0) {
        av_log(avctx, AV_LOG_ERROR, "The maximum value for lowres supported by the decoder is %d\n",
               avctx->codec->max_lowres);
        ret = AVERROR(EINVAL);
        goto free_and_end;
    }

#if FF_API_VISMV
    if (avctx->debug_mv)
        av_log(avctx, AV_LOG_WARNING, "The 'vismv' option is deprecated, "
               "see the codecview filter instead.\n");
#endif
    //检查输入参数是否符合【编码器】要求
    if (av_codec_is_encoder(avctx->codec)) {
        int i;
        //如果包含采样率参数（表明是音频），检查采样率是否符合要求
        if (avctx->codec->sample_fmts) {
            //遍历编码器支持的所有采样率
            for (i = 0; avctx->codec->sample_fmts[i] != AV_SAMPLE_FMT_NONE; i++) {
```

```c
            //如果设置的采样率==编码器支持的采样率，跳出循环。
            if (avctx->sample_fmt == avctx->codec->sample_fmts[i])
                break;
            if (avctx->channels == 1 &&
                av_get_planar_sample_fmt(avctx->sample_fmt) ==
                av_get_planar_sample_fmt(avctx->codec->sample_fmts[i])) {
                avctx->sample_fmt = avctx->codec->sample_fmts[i];
                break;
            }
        }
        //再检查一下采样率取值是否正确
        //注意，此时的i值没有变化
        if (avctx->codec->sample_fmts[i] == AV_SAMPLE_FMT_NONE) {
            char buf[128];
            snprintf(buf, sizeof(buf), "%d", avctx->sample_fmt);
            av_log(avctx, AV_LOG_ERROR, "Specified sample format %s is invalid or not supported\n",
                   (char *)av_x_if_null(av_get_sample_fmt_name(avctx->sample_fmt), buf));
            ret = AVERROR(EINVAL);
            goto free_and_end;
        }
    }
    //检查像素格式
    if (avctx->codec->pix_fmts) {
        for (i = 0; avctx->codec->pix_fmts[i] != AV_PIX_FMT_NONE; i++)
            if (avctx->pix_fmt == avctx->codec->pix_fmts[i])
                break;
        if (avctx->codec->pix_fmts[i] == AV_PIX_FMT_NONE
            && !((avctx->codec_id == AV_CODEC_ID_MJPEG || avctx->codec_id == AV_CODEC_ID_LJPEG)
                 && avctx->strict_std_compliance <= FF_COMPLIANCE_UNOFFICIAL)) {
            char buf[128];
            snprintf(buf, sizeof(buf), "%d", avctx->pix_fmt);
            av_log(avctx, AV_LOG_ERROR, "Specified pixel format %s is invalid or not supported\n",
                   (char *)av_x_if_null(av_get_pix_fmt_name(avctx->pix_fmt), buf));
            ret = AVERROR(EINVAL);
            goto free_and_end;
        }
        if (avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ420P ||
            avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ411P ||
            avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ422P ||
            avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ440P ||
            avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ444P)
            avctx->color_range = AVCOL_RANGE_JPEG;
    }
    //检查采样率
    if (avctx->codec->supported_samplerates) {
        for (i = 0; avctx->codec->supported_samplerates[i] != 0; i++)
            if (avctx->sample_rate == avctx->codec->supported_samplerates[i])
                break;
        if (avctx->codec->supported_samplerates[i] == 0) {
            av_log(avctx, AV_LOG_ERROR, "Specified sample rate %d is not supported\n",
                   avctx->sample_rate);
            ret = AVERROR(EINVAL);
            goto free_and_end;
        }
    }
    //检查声道布局
    if (avctx->codec->channel_layouts) {
        if (!avctx->channel_layout) {
            av_log(avctx, AV_LOG_WARNING, "Channel layout not specified\n");
        } else {
            for (i = 0; avctx->codec->channel_layouts[i] != 0; i++)
                if (avctx->channel_layout == avctx->codec->channel_layouts[i])
                    break;
            if (avctx->codec->channel_layouts[i] == 0) {
                char buf[512];
                av_get_channel_layout_string(buf, sizeof(buf), -1, avctx->channel_layout);
                av_log(avctx, AV_LOG_ERROR, "Specified channel layout '%s' is not supported\n", buf);
                ret = AVERROR(EINVAL);
                goto free_and_end;
            }
        }
    }
    //检查声道数
    if (avctx->channel_layout && avctx->channels) {
        int channels = av_get_channel_layout_nb_channels(avctx->channel_layout);
        if (channels != avctx->channels) {
            char buf[512];
            av_get_channel_layout_string(buf, sizeof(buf), -1, avctx->channel_layout);
            av_log(avctx, AV_LOG_ERROR,
                   "Channel layout '%s' with %d channels does not match number of specified channels %d\n",
                   buf, channels, avctx->channels);
            ret = AVERROR(EINVAL);
            goto free_and_end;
        }
    } else if (avctx->channel_layout) {
        avctx->channels = av_get_channel_layout_nb_channels(avctx->channel_layout);
    }
    //检查宽高
    if(avctx->codec_type == AVMEDIA_TYPE_VIDEO) {
        if (avctx->width <= 0 || avctx->height <= 0) {
            av_log(avctx, AV_LOG_ERROR, "dimensions not set\n");
```

```c
                ret = AVERROR(EINVAL);
                goto free_and_end;
            }
        }
        //检查码率
        if (   (avctx->codec_type == AVMEDIA_TYPE_VIDEO || avctx->codec_type == AVMEDIA_TYPE_AUDIO)
            && avctx->bit_rate>0 && avctx->bit_rate<1000) {
            av_log(avctx, AV_LOG_WARNING, "Bitrate %d is extremely low, maybe you mean %dk\n", avctx->bit_rate, avctx->bit_rate);
        }

        if (!avctx->rc_initial_buffer_occupancy)
            avctx->rc_initial_buffer_occupancy = avctx->rc_buffer_size * 3 / 4;
    }

    avctx->pts_correction_num_faulty_pts =
    avctx->pts_correction_num_faulty_dts = 0;
    avctx->pts_correction_last_pts =
    avctx->pts_correction_last_dts = INT64_MIN;
    //关键：
    //一切检查都无误之后，调用编解码器初始化函数
    if (   avctx->codec->init && (!(avctx->active_thread_type&FF_THREAD_FRAME)
        || avctx->internal->frame_thread_encoder)) {
        ret = avctx->codec->init(avctx);
        if (ret < 0) {
            goto free_and_end;
        }
    }

    ret=0;

#if FF_API_AUDIOENC_DELAY
    if (av_codec_is_encoder(avctx->codec))
        avctx->delay = avctx->initial_padding;
#endif

    //【解码器】
    //解码器的参数大部分都是由系统自动设定而不是由用户设定，因而不怎么需要检查
    if (av_codec_is_decoder(avctx->codec)) {
        if (!avctx->bit_rate)
            avctx->bit_rate = get_bit_rate(avctx);
        /* validate channel layout from the decoder */
        if (avctx->channel_layout) {
            int channels = av_get_channel_layout_nb_channels(avctx->channel_layout);
            if (!avctx->channels)
                avctx->channels = channels;
            else if (channels != avctx->channels) {
                char buf[512];
                av_get_channel_layout_string(buf, sizeof(buf), -1, avctx->channel_layout);
                av_log(avctx, AV_LOG_WARNING,
                       "Channel layout '%s' with %d channels does not match specified number of channels %d: "
                       "ignoring specified channel layout\n",
                       buf, channels, avctx->channels);
                avctx->channel_layout = 0;
            }
        }

        if (avctx->channels && avctx->channels < 0 ||
            avctx->channels > FF_SANE_NB_CHANNELS) {
            ret = AVERROR(EINVAL);
            goto free_and_end;
        }
        if (avctx->sub_charenc) {
            if (avctx->codec_type != AVMEDIA_TYPE_SUBTITLE) {
                av_log(avctx, AV_LOG_ERROR, "Character encoding is only "
                       "supported with subtitles codecs\n");
                ret = AVERROR(EINVAL);
                goto free_and_end;
            } else if (avctx->codec_descriptor->props & AV_CODEC_PROP_BITMAP_SUB) {
                av_log(avctx, AV_LOG_WARNING, "Codec '%s' is bitmap-based, "
                       "subtitles character encoding will be ignored\n",
                       avctx->codec_descriptor->name);
                avctx->sub_charenc_mode = FF_SUB_CHARENC_MODE_DO_NOTHING;
            } else {
                /* input character encoding is set for a text based subtitle
                 * codec at this point */
                if (avctx->sub_charenc_mode == FF_SUB_CHARENC_MODE_AUTOMATIC)
                    avctx->sub_charenc_mode = FF_SUB_CHARENC_MODE_PRE_DECODER;

                if (avctx->sub_charenc_mode == FF_SUB_CHARENC_MODE_PRE_DECODER) {
#if CONFIG_ICONV
                    iconv_t cd = iconv_open("UTF-8", avctx->sub_charenc);
                    if (cd == (iconv_t)-1) {
                        ret = AVERROR(errno);
                        av_log(avctx, AV_LOG_ERROR, "Unable to open iconv context "
                               "with input character encoding \"%s\"\n", avctx->sub_charenc);
                        goto free_and_end;
                    }
                    iconv_close(cd);
#else
                    av_log(avctx, AV_LOG_ERROR, "Character encoding subtitles "
                           "conversion needs a libavcodec built with iconv support "
                           "for this codec\n");
```

```cpp
                           "for this codec\n");
                    ret = AVERROR(ENOSYS);
                    goto free_and_end;
#endif
                }
            }
        }

#if FF_API_AVCTX_TIMEBASE
        if (avctx->framerate.num > 0 && avctx->framerate.den > 0)
            avctx->time_base = av_inv_q(av_mul_q(avctx->framerate, (AVRational){avctx->ticks_per_frame, 1}));
#endif
    }
end:
    ff_unlock_avcodec();
    if (options) {
        av_dict_free(options);
        *options = tmp;
    }

    return ret;
free_and_end:
    av_dict_free(&tmp);
    if (codec->priv_class && codec->priv_data_size)
        av_opt_free(avctx->priv_data);
    av_freep(&avctx->priv_data);
    if (avctx->internal) {
        av_frame_free(&avctx->internal->to_free);
        av_freep(&avctx->internal->pool);
    }
    av_freep(&avctx->internal);
    avctx->codec = NULL;
    goto end;
}
```

avcodec_open2()的源代码量是非常长的，但是它的调用关系非常简单——它只调用了一个关键的函数，即AVCodec的init()，后文将会对这个函数进行分析。
我们可以简单梳理一下avcodec_open2()所做的工作，如下所列：

（1）为各种结构体分配内存（通过各种av_malloc()实现）。

（2）将输入的AVDictionary形式的选项设置到AVCodecContext。

（3）其他一些零零碎碎的检查，比如说检查编解码器是否处于“实验”阶段。

（4）如果是编码器，检查输入参数是否符合编码器的要求

（5）调用AVCodec的init()初始化具体的解码器。

前几步比较简单，不再分析。在这里我们分析一下第4步和第5步。

## 检查输入参数是否符合编码器要求

在这里简单分析一下第4步，即“检查输入参数是否符合编码器的要求”。这一步中检查了很多的参数，在这里我们随便选一个参数pix_fmts（像素格式）看一下，如下所示。

```cpp
//检查像素格式
        if (avctx->codec->pix_fmts) {
            for (i = 0; avctx->codec->pix_fmts[i] != AV_PIX_FMT_NONE; i++)
                if (avctx->pix_fmt == avctx->codec->pix_fmts[i])
                    break;
            if (avctx->codec->pix_fmts[i] == AV_PIX_FMT_NONE
                && !((avctx->codec_id == AV_CODEC_ID_MJPEG || avctx->codec_id == AV_CODEC_ID_LJPEG)
                     && avctx->strict_std_compliance <= FF_COMPLIANCE_UNOFFICIAL)) {
                char buf[128];
                snprintf(buf, sizeof(buf), "%d", avctx->pix_fmt);
                av_log(avctx, AV_LOG_ERROR, "Specified pixel format %s is invalid or not supported\n",
                       (char *)av_x_if_null(av_get_pix_fmt_name(avctx->pix_fmt), buf));
                ret = AVERROR(EINVAL);
                goto free_and_end;
            }
            if (avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ420P ||
                avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ411P ||
                avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ422P ||
                avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ440P ||
                avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ444P)
                avctx->color_range = AVCOL_RANGE_JPEG;
        }
```

可以看出，该代码首先进入了一个for()循环，将AVCodecContext中设定的pix_fmt与编码器AVCodec中的pix_fmts数组中的元素逐一比较。
先简单介绍一下AVCodec中的pix_fmts数组。AVCodec中的pix_fmts数组存储了该种编码器支持的像素格式，并且规定以AV_PIX_FMT_NONE（AV_PIX_FMT_NONE取值为-1）为结尾。例如，libx264的pix_fmts数组的定义位于libavcodec\libx264.c，如下所示。

[cpp]

```cpp
static const enum AVPixelFormat pix_fmts_8bit[] = {
    AV_PIX_FMT_YUV420P,
    AV_PIX_FMT_YUVJ420P,
    AV_PIX_FMT_YUV422P,
    AV_PIX_FMT_YUVJ422P,
    AV_PIX_FMT_YUV444P,
    AV_PIX_FMT_YUVJ444P,
    AV_PIX_FMT_NV12,
    AV_PIX_FMT_NV16,
    AV_PIX_FMT_NONE
};
```

从pix_fmts_8bit的定义可以看出libx264主要支持的是以YUV为主的像素格式。
现在回到"检查输入pix_fmt是否符合编码器的要求"的那段代码。如果for()循环从AVCodec->pix_fmts数组中找到了符合AVCodecContext->pix_fmt的像素格式，或者完成了AVCodec->pix_fmts数组的遍历，都会跳出循环。如果发现AVCodec->pix_fmts数组中索引为i的元素是AV_PIX_FMT_NONE（即最后一个元素，取值为-1）的时候，就认为没有找到合适的像素格式，并且最终提示错误信息。

## AVCodec->init()

avcodec_open2()中最关键的一步就是调用AVCodec的init()方法初始化具体的编码器。AVCodec的init()是一个函数指针，指向具体编解码器中的初始化函数。这里我们以libx264为例，看一下它对应的AVCodec的定义。libx264对应的AVCodec的定义位于libavcodec\libx264.c，如下所示。

[cpp]

```cpp
AVCodec ff_libx264_encoder = {
    .name             = "libx264",
    .long_name        = NULL_IF_CONFIG_SMALL("libx264 H.264 / AVC / MPEG-4 AVC / MPEG-4 part 10"),
    .type             = AVMEDIA_TYPE_VIDEO,
    .id               = AV_CODEC_ID_H264,
    .priv_data_size   = sizeof(X264Context),
    .init             = X264_init,
    .encode2          = X264_frame,
    .close            = X264_close,
    .capabilities     = CODEC_CAP_DELAY | CODEC_CAP_AUTO_THREADS,
    .priv_class       = &x264_class,
    .defaults         = x264_defaults,
    .init_static_data = X264_init_static,
};
```

可以看出在ff_libx264_encoder中init()指向X264_init()。X264_init()的定义同样位于libavcodec\libx264.c，如下所示。

[cpp]

```cpp
static av_cold int X264_init(AVCodecContext *avctx)
{
    X264Context *x4 = avctx->priv_data;
    int sw,sh;

    if (avctx->global_quality > 0)
        av_log(avctx, AV_LOG_WARNING, "-qscale is ignored, -crf is recommended.\n");

    x264_param_default(&x4->params);

    x4->params.b_deblocking_filter         = avctx->flags & CODEC_FLAG_LOOP_FILTER;

    if (x4->preset || x4->tune)
        if (x264_param_default_preset(&x4->params, x4->preset, x4->tune) < 0) {
            int i;
            av_log(avctx, AV_LOG_ERROR, "Error setting preset/tune %s/%s.\n", x4->preset, x4->tune);
            av_log(avctx, AV_LOG_INFO, "Possible presets:");
            for (i = 0; x264_preset_names[i]; i++)
                av_log(avctx, AV_LOG_INFO, " %s", x264_preset_names[i]);
            av_log(avctx, AV_LOG_INFO, "\n");
            av_log(avctx, AV_LOG_INFO, "Possible tunes:");
            for (i = 0; x264_tune_names[i]; i++)
                av_log(avctx, AV_LOG_INFO, " %s", x264_tune_names[i]);
            av_log(avctx, AV_LOG_INFO, "\n");
            return AVERROR(EINVAL);
        }

    if (avctx->level > 0)
        x4->params.i_level_idc = avctx->level;

    x4->params.pf_log               = X264_log;
    x4->params.p_log_private        = avctx;
    x4->params.i_log_level          = X264_LOG_DEBUG;
    x4->params.i_csp                = convert_pix_fmt(avctx->pix_fmt);

    OPT_STR("weightp", x4->wpredp);

    if (avctx->bit_rate) {
        x4->params.rc.i_bitrate   = avctx->bit_rate / 1000;
        x4->params.rc.i_rc_method = X264_RC_ABR;
    }
    x4->params.rc.i_vbv_buffer_size = avctx->rc_buffer_size / 1000;
```

```c
    x4->params.rc.i_vbv_max_bitrate = avctx->rc_max_rate    / 1000;
    x4->params.rc.b_stat_write      = avctx->flags & CODEC_FLAG_PASS1;
    if (avctx->flags & CODEC_FLAG_PASS2) {
        x4->params.rc.b_stat_read = 1;
    } else {
        if (x4->crf >= 0) {
            x4->params.rc.i_rc_method   = X264_RC_CRF;
            x4->params.rc.f_rf_constant = x4->crf;
        } else if (x4->cqp >= 0) {
            x4->params.rc.i_rc_method   = X264_RC_CQP;
            x4->params.rc.i_qp_constant = x4->cqp;
        }

        if (x4->crf_max >= 0)
            x4->params.rc.f_rf_constant_max = x4->crf_max;
    }

    if (avctx->rc_buffer_size && avctx->rc_initial_buffer_occupancy > 0 &&
        (avctx->rc_initial_buffer_occupancy <= avctx->rc_buffer_size)) {
        x4->params.rc.f_vbv_buffer_init =
            (float)avctx->rc_initial_buffer_occupancy / avctx->rc_buffer_size;
    }

    OPT_STR("level", x4->level);

    if (avctx->i_quant_factor > 0)
        x4->params.rc.f_ip_factor         = 1 / fabs(avctx->i_quant_factor);
    if (avctx->b_quant_factor > 0)
        x4->params.rc.f_pb_factor         = avctx->b_quant_factor;
    if (avctx->chromaoffset)
        x4->params.analyse.i_chroma_qp_offset = avctx->chromaoffset;

    if (avctx->me_method == ME_EPZS)
        x4->params.analyse.i_me_method = X264_ME_DIA;
    else if (avctx->me_method == ME_HEX)
        x4->params.analyse.i_me_method = X264_ME_HEX;
    else if (avctx->me_method == ME_UMH)
        x4->params.analyse.i_me_method = X264_ME_UMH;
    else if (avctx->me_method == ME_FULL)
        x4->params.analyse.i_me_method = X264_ME_ESA;
    else if (avctx->me_method == ME_TESA)
        x4->params.analyse.i_me_method = X264_ME_TESA;

    if (avctx->gop_size >= 0)
        x4->params.i_keyint_max         = avctx->gop_size;
    if (avctx->max_b_frames >= 0)
        x4->params.i_bframe             = avctx->max_b_frames;
    if (avctx->scenechange_threshold >= 0)
        x4->params.i_scenecut_threshold = avctx->scenechange_threshold;
    if (avctx->qmin >= 0)
        x4->params.rc.i_qp_min          = avctx->qmin;
    if (avctx->qmax >= 0)
        x4->params.rc.i_qp_max          = avctx->qmax;
    if (avctx->max_qdiff >= 0)
        x4->params.rc.i_qp_step         = avctx->max_qdiff;
    if (avctx->qblur >= 0)
        x4->params.rc.f_qblur           = avctx->qblur;     /* temporally blur quants */
    if (avctx->qcompress >= 0)
        x4->params.rc.f_qcompress       = avctx->qcompress; /* 0.0 => cbr, 1.0 => constant qp */
    if (avctx->refs >= 0)
        x4->params.i_frame_reference    = avctx->refs;
    else if (x4->level) {
        int i;
        int mbn = FF_CEIL_RSHIFT(avctx->width, 4) * FF_CEIL_RSHIFT(avctx->height, 4);
        int level_id = -1;
        char *tail;
        int scale = X264_BUILD < 129 ? 384 : 1;

        if (!strcmp(x4->level, "1b")) {
            level_id = 9;
        } else if (strlen(x4->level) <= 3){
            level_id = av_strtod(x4->level, &tail) * 10 + 0.5;
            if (*tail)
                level_id = -1;
        }
        if (level_id <= 0)
            av_log(avctx, AV_LOG_WARNING, "Failed to parse level\n");

        for (i = 0; i<x264_levels[i].level_idc; i++)
            if (x264_levels[i].level_idc == level_id)
                x4->params.i_frame_reference = av_clip(x264_levels[i].dpb / mbn / scale, 1, x4->params.i_frame_reference);
    }

    if (avctx->trellis >= 0)
        x4->params.analyse.i_trellis    = avctx->trellis;
    if (avctx->me_range >= 0)
        x4->params.analyse.i_me_range   = avctx->me_range;
    if (avctx->noise_reduction >= 0)
        x4->params.analyse.i_noise_reduction = avctx->noise_reduction;
    if (avctx->me_subpel_quality >= 0)
        x4->params.analyse.i_subpel_refine   = avctx->me_subpel_quality;
    if (avctx->b_frame_strategy >= 0)
        x4->params.i_bframe_adaptive = avctx->b_frame_strategy;
    if (avctx->keyint_min >= 0)
        x4->params.i_keyint_min = avctx->keyint_min;
    if (avctx->coder_type >= 0)
        x4->params.b_cabac = avctx->coder_type == FF_CODER_TYPE_AC;
    if (avctx->me_cmp >= 0)
        x4->params.analyse.b_chroma_me = avctx->me_cmp & FF_CMP_CHROMA;

    if (x4->aq_mode >= 0)
        x4->params.rc.i_aq_mode = x4->aq_mode;
    if (x4->aq_strength >= 0)
        x4->params.rc.f_aq_strength = x4->aq_strength;
    PARSE_X264_OPT("psy-rd", psy_rd);
    PARSE_X264_OPT("deblock", deblock);
    PARSE_X264_OPT("partitions", partitions);
    PARSE_X264_OPT("stats", stats);
    if (x4->psy >= 0)
        x4->params.analyse.b_psy  = x4->psy;
    if (x4->rc_lookahead >= 0)
        x4->params.rc.i_lookahead = x4->rc_lookahead;
    if (x4->weightp >= 0)
        x4->params.analyse.i_weighted_pred = x4->weightp;
    if (x4->weightb >= 0)
        x4->params.analyse.b_weighted_bipred = x4->weightb;
    if (x4->cplxblur >= 0)
        x4->params.rc.f_complexity_blur = x4->cplxblur;

    if (x4->ssim >= 0)
        x4->params.analyse.b_ssim = x4->ssim;
    if (x4->intra_refresh >= 0)
        x4->params.b_intra_refresh = x4->intra_refresh;
    if (x4->bluray_compat >= 0) {
        x4->params.b_bluray_compat = x4->bluray_compat;
        x4->params.b_vfr_input = 0;
    }
    if (x4->avcintra_class >= 0)
#if X264_BUILD >= 142
        x4->params.i_avcintra_class = x4->avcintra_class;
#else
        av_log(avctx, AV_LOG_ERROR,
               "x264 too old for AVC Intra, at least version 142 needed\n");
#endif
    if (x4->b_bias != INT_MIN)
        x4->params.i_bframe_bias              = x4->b_bias;
    if (x4->b_pyramid >= 0)
        x4->params.i_bframe_pyramid = x4->b_pyramid;
    if (x4->mixed_refs >= 0)
        x4->params.analyse.b_mixed_references = x4->mixed_refs;
    if (x4->dct8x8 >= 0)
        x4->params.analyse.b_transform_8x8    = x4->dct8x8;
    if (x4->fast_pskip >= 0)
        x4->params.analyse.b_fast_pskip       = x4->fast_pskip;
    if (x4->aud >= 0)
        x4->params.b_aud                      = x4->aud;
    if (x4->mbtree >= 0)
        x4->params.rc.b_mb_tree               = x4->mbtree;
    if (x4->direct_pred >= 0)
        x4->params.analyse.i_direct_mv_pred   = x4->direct_pred;

    if (x4->slice_max_size >= 0)
        x4->params.i_slice_max_size =  x4->slice_max_size;
    else {
        /*
         * Allow x264 to be instructed through AVCodecContext about the maximum
         * size of the RTP payload. For example, this enables the production of
         * payload suitable for the H.264 RTP packetization-mode 0 i.e. single
         * NAL unit per RTP packet.
         */
        if (avctx->rtp_payload_size)
            x4->params.i_slice_max_size = avctx->rtp_payload_size;
    }

    if (x4->fastfirstpass)
        x264_param_apply_fastfirstpass(&x4->params);

    /* Allow specifying the x264 profile through AVCodecContext. */
    if (!x4->profile)
        switch (avctx->profile) {
        case FF_PROFILE_H264_BASELINE:
            x4->profile = av_strdup("baseline");
            break;
        case FF_PROFILE_H264_HIGH:
            x4->profile = av_strdup("high");
            break;
        case FF_PROFILE_H264_HIGH_10:
            x4->profile = av_strdup("high10");
            break;
        case FF_PROFILE_H264_HIGH_422:
            x4->profile = av_strdup("high422");
            break;
        case FF_PROFILE_H264_HIGH_444:
            x4->profile = av_strdup("high444");
            break;
        case FF_PROFILE_H264_MAIN:
            x4->profile = av_strdup("main");
            break;
        default:
            break;
        }

    if (x4->nal_hrd >= 0)
        x4->params.i_nal_hrd = x4->nal_hrd;

    if (x4->profile)
        if (x264_param_apply_profile(&x4->params, x4->profile) < 0) {
            int i;
            av_log(avctx, AV_LOG_ERROR, "Error setting profile %s.\n", x4->profile);
            av_log(avctx, AV_LOG_INFO, "Possible profiles:");
            for (i = 0; x264_profile_names[i]; i++)
                av_log(avctx, AV_LOG_INFO, " %s", x264_profile_names[i]);
            av_log(avctx, AV_LOG_INFO, "\n");
            return AVERROR(EINVAL);
        }

    x4->params.i_width          = avctx->width;
    x4->params.i_height         = avctx->height;
    av_reduce(&sw, &sh, avctx->sample_aspect_ratio.num, avctx->sample_aspect_ratio.den, 4096);
    x4->params.vui.i_sar_width  = sw;
    x4->params.vui.i_sar_height = sh;
    x4->params.i_timebase_den = avctx->time_base.den;
    x4->params.i_timebase_num = avctx->time_base.num;
    x4->params.i_fps_num = avctx->time_base.den;
    x4->params.i_fps_den = avctx->time_base.num * avctx->ticks_per_frame;

    x4->params.analyse.b_psnr = avctx->flags & CODEC_FLAG_PSNR;

    x4->params.i_threads      = avctx->thread_count;
    if (avctx->thread_type)
        x4->params.b_sliced_threads = avctx->thread_type == FF_THREAD_SLICE;

    x4->params.b_interlaced   = avctx->flags & CODEC_FLAG_INTERLACED_DCT;

    x4->params.b_open_gop     = !(avctx->flags & CODEC_FLAG_CLOSED_GOP);

    x4->params.i_slice_count  = avctx->slices;

    x4->params.vui.b_fullrange = avctx->pix_fmt == AV_PIX_FMT_YUVJ420P ||
                                 avctx->pix_fmt == AV_PIX_FMT_YUVJ422P ||
                                 avctx->pix_fmt == AV_PIX_FMT_YUVJ444P ||
                                 avctx->color_range == AVCOL_RANGE_JPEG;

    if (avctx->colorspace != AVCOL_SPC_UNSPECIFIED)
        x4->params.vui.i_colmatrix = avctx->colorspace;
    if (avctx->color_primaries != AVCOL_PRI_UNSPECIFIED)
        x4->params.vui.i_colorprim = avctx->color_primaries;
    if (avctx->color_trc != AVCOL_TRC_UNSPECIFIED)
        x4->params.vui.i_transfer  = avctx->color_trc;

    if (avctx->flags & CODEC_FLAG_GLOBAL_HEADER)
        x4->params.b_repeat_headers = 0;

    if(x4->x264opts){
        const char *p= x4->x264opts;
        while(p){
            char param[256]={0}, val[256]={0};
            if(sscanf(p, "%255[^:=]=%255[^:]", param, val) == 1){
                OPT_STR(param, "1");
            }else
                OPT_STR(param, val);
            p= strchr(p, ':');
            p+=!!p;
        }
    }

    if (x4->x264_params) {
        AVDictionary *dict    = NULL;
        AVDictionaryEntry *en = NULL;

        if (!av_dict_parse_string(&dict, x4->x264_params, "=", ":", 0)) {
            while ((en = av_dict_get(dict, "", en, AV_DICT_IGNORE_SUFFIX))) {
                if (x264_param_parse(&x4->params, en->key, en->value) < 0)
                    av_log(avctx, AV_LOG_WARNING,
                           "Error parsing option '%s = %s'.\n",
                            en->key, en->value);
            }

            av_dict_free(&dict);
        }
    }

    // update AVCodecContext with x264 parameters
    avctx->has_b_frames = x4->params.i_bframe ?
        x4->params.i_bframe_pyramid ? 2 : 1 : 0;
    if (avctx->max_b_frames < 0)
        avctx->max_b_frames = 0;

    avctx->bit_rate = x4->params.rc.i_bitrate*1000;

    x4->enc = x264_encoder_open(&x4->params);
    if (!x4->enc)
        return -1;

    avctx->coded_frame = av_frame_alloc();
    if (!avctx->coded_frame)
        return AVERROR(ENOMEM);

    if (avctx->flags & CODEC_FLAG_GLOBAL_HEADER) {
        x264_nal_t *nal;
        uint8_t *p;
        int nnal, s, i;

        s = x264_encoder_headers(x4->enc, &nal, &nnal);
        avctx->extradata = p = av_malloc(s);

        for (i = 0; i < nnal; i++) {
            /* Don't put the SEI in extradata. */
            if (nal[i].i_type == NAL_SEI) {
                av_log(avctx, AV_LOG_INFO, "%s\n", nal[i].p_payload+25);
                x4->sei_size = nal[i].i_payload;
                x4->sei      = av_malloc(x4->sei_size);
                memcpy(x4->sei, nal[i].p_payload, nal[i].i_payload);
                continue;
            }
            memcpy(p, nal[i].p_payload, nal[i].i_payload);
            p += nal[i].i_payload;
        }
        avctx->extradata_size = p - avctx->extradata;
    }

    return 0;
}
```

X264_init()的代码以后研究X264的时候再进行细节的分析，在这里简单记录一下它做的两项工作：

（1）设置X264Context的参数。X264Context主要完成了libx264和FFmpeg对接的功能。可以看出代码主要在设置一个params结构体变量，该变量的类型即是x264中存储参数的结构体x264_param_t。

（2）调用libx264的API进行编码器的初始化工作。例如调用x264_param_default()设置默认参数，调用x264_param_apply_profile()设置profile，调用x264_encoder_open()打开编码器等等。

最后附上X264Context的定义，位于libavcodec\libx264.c，如下所示。

[cpp]

```cpp
typedef struct X264Context {
    AVClass        *class;
    x264_param_t    params;
    x264_t         *enc;
    x264_picture_t  pic;
    uint8_t        *sei;
    int             sei_size;
    char *preset;
    char *tune;
    char *profile;
    char *level;
    int fastfirstpass;
    char *wpredp;
    char *x264opts;
    float crf;
    float crf_max;
    int cqp;
    int aq_mode;
    float aq_strength;
    char *psy_rd;
    int psy;
    int rc_lookahead;
    int weightp;
    int weightb;
    int ssim;
    int intra_refresh;
    int bluray_compat;
    int b_bias;
    int b_pyramid;
    int mixed_refs;
    int dct8x8;
    int fast_pskip;
    int aud;
    int mbtree;
    char *deblock;
    float cplxblur;
    char *partitions;
    int direct_pred;
    int slice_max_size;
    char *stats;
    int nal_hrd;
    int avcintra_class;
    char *x264_params;
} X264Context;
```

**雷霄骅**
**leixiaohua1020@126.com**
**http://blog.csdn.net/leixiaohua1020**

文章标签： FFmpeg 源代码 编码器 AVCodec 初始化

个人分类： FFMPEG

所属专栏： FFmpeg