

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：

【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/declavcodec.h，如下所示。

```
[cpp]
1.  /**
2.   * Initialize the AVCodecContext to use the given AVCodec. Prior to using this
3.   * function the context has to be allocated with avcodec_alloc_context3().
4.   *
5.   * The functions avcodec_find_decoder_by_name(), avcodec_find_encoder_by_name(),
6.   * avcodec_find_decoder() and avcodec_find_encoder() provide an easy way for
7.   * retrieving a codec.
8.   *
9.   * @warning This function is not thread safe!
10.  *
11.  * @code
12.  * avcodec_register_all();
13.  * av_dict_set(&opts, "b", "2.5M", 0);
14.  * codec = avcodec_find_decoder(AV_CODEC_ID_H264);
15.  * if (!codec)
16.  *     exit(1);
17.  *
18.  * context = avcodec_alloc_context3(codec);
19.  *
20.  * if (avcodec_open2(context, codec, opts) < 0)
21.  *     exit(1);
22.  * @endcode
23.  *
24.  * @param avctx The context to initialize.
25.  * @param codec The codec to open this context for. If a non-NULL codec has been
26.  *             previously passed to avcodec_alloc_context3() or
27.  *             avcodec_get_context_defaults3() for this context, then this
28.  *             parameter MUST be either NULL or equal to the previously passed
29.  *             codec.
30.  * @param options A dictionary filled with AVCodecContext and codec-private options.
31.  *               On return this object will be filled with options that were not found.
32.  *
33.  * @return zero on success, a negative value on error
34.  * @see avcodec_alloc_context3(), avcodec_find_decoder(), avcodec_find_encoder(),
35.  *      av_dict_set(), av_opt_find().
36.  */
37. 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libavcodecutils.c，如下所示。

```
[cpp]
1.  int avcodec_open2(AVCodecContext *avctx, const AVCodec *codec, AVDictionary **options)
2.  {
3.      int ret = 0;
4.      AVDictionary *tmp = NULL;
5.      //如果已经打开，直接返回
6.      if (avcodec_is_open(avctx))
7.          return 0;
8.  }
```

```

9.  if ((!codec && !avctx->codec)) {
10.      av_log(avctx, AV_LOG_ERROR, "No codec provided to avcodec_open2()\n");
11.      return AVERROR(EINVAL);
12.  }
13.  if ((codec && avctx->codec && codec != avctx->codec)) {
14.      av_log(avctx, AV_LOG_ERROR, "This AVCodecContext was allocated for %s, "
15.          "but %s passed to avcodec_open2()\n", avctx->codec->name, codec->name);
16.      return AVERROR(EINVAL);
17.  }
18.  if (!codec)
19.      codec = avctx->codec;
20.
21.  if (avctx->extradata_size < 0 || avctx->extradata_size >= FF_MAX_EXTRADATA_SIZE)
22.      return AVERROR(EINVAL);
23.
24.  if (options)
25.      av_dict_copy(&tmp, *options, 0);
26.
27.  ret = ff_lock_avcodec(avctx);
28.  if (ret < 0)
29.      return ret;
30.  //各种Malloc
31.  avctx->internal = av_mallocz(sizeof(AVCodecInternal));
32.  if (!avctx->internal) {
33.      ret = AVERROR(ENOMEM);
34.      goto end;
35.  }
36.
37.  avctx->internal->pool = av_mallocz(sizeof(*avctx->internal->pool));
38.  if (!avctx->internal->pool) {
39.      ret = AVERROR(ENOMEM);
40.      goto free_and_end;
41.  }
42.
43.  avctx->internal->to_free = av_frame_alloc();
44.  if (!avctx->internal->to_free) {
45.      ret = AVERROR(ENOMEM);
46.      goto free_and_end;
47.  }
48.
49.  if (codec->priv_data_size > 0) {
50.      if (!avctx->priv_data) {
51.          avctx->priv_data = av_mallocz(codec->priv_data_size);
52.          if (!avctx->priv_data) {
53.              ret = AVERROR(ENOMEM);
54.              goto end;
55.          }
56.          if (codec->priv_class) {
57.              *(const AVClass **)avctx->priv_data = codec->priv_class;
58.              av_opt_set_defaults(avctx->priv_data);
59.          }
60.      }
61.      if (codec->priv_class && (ret = av_opt_set_dict(avctx->priv_data, &tmp)) < 0)
62.          goto free_and_end;
63.  } else {
64.      avctx->priv_data = NULL;
65.  }
66.  //将输入的AVDictionary形式的选项设置到AVCodecContext
67.  if ((ret = av_opt_set_dict(avctx, &tmp)) < 0)
68.      goto free_and_end;
69.
70.  if (avctx->codec_whitelist && av_match_list(codec->name, avctx->codec_whitelist, ',') <= 0) {
71.      av_log(avctx, AV_LOG_ERROR, "Codec (%s) not on whitelist\n", codec->name);
72.      ret = AVERROR(EINVAL);
73.      goto free_and_end;
74.  }
75.
76.  // only call ff_set_dimensions() for non H.264/VP6F codecs so as not to overwrite previously setup dimensions
77.  if (!(avctx->coded_width && avctx->coded_height && avctx->width && avctx->height &&
78.      (avctx->codec_id == AV_CODEC_ID_H264 || avctx->codec_id == AV_CODEC_ID_VP6F))) {
79.      if (avctx->coded_width && avctx->coded_height)
80.          ret = ff_set_dimensions(avctx, avctx->coded_width, avctx->coded_height);
81.      else if (avctx->width && avctx->height)
82.          ret = ff_set_dimensions(avctx, avctx->width, avctx->height);
83.      if (ret < 0)
84.          goto free_and_end;
85.  }
86.  //检查宽和高
87.  if ((avctx->coded_width || avctx->coded_height || avctx->width || avctx->height)
88.      && ( av_image_check_size(avctx->coded_width, avctx->coded_height, 0, avctx) < 0
89.          || av_image_check_size(avctx->width, avctx->height, 0, avctx) < 0)) {
90.      av_log(avctx, AV_LOG_WARNING, "Ignoring invalid width/height values\n");
91.      ff_set_dimensions(avctx, 0, 0);
92.  }
93.  //检查宽高比
94.  if (avctx->width > 0 && avctx->height > 0) {
95.      if (av_image_check_sar(avctx->width, avctx->height,
96.          avctx->sample_aspect_ratio) < 0) {
97.          av_log(avctx, AV_LOG_WARNING, "Ignoring invalid SAR: %u/%u\n",
98.              avctx->sample_aspect_ratio.num,
99.              avctx->sample_aspect_ratio.den);

```

```

100.     avctx->sample_aspect_ratio = (AVRational){ 0, 1 };
101. }
102. }
103.
104. /* if the decoder init function was already called previously,
105.  * free the already allocated subtitle_header before overwriting it */
106. if (av_codec_is_decoder(codec))
107.     av_freep(&avctx->subtitle_header);
108.
109. if (avctx->channels > FF_SANE_NB_CHANNELS) {
110.     ret = AERROR(EINVAL);
111.     goto free_and_end;
112. }
113.
114. avctx->codec = codec;
115. if ((avctx->codec_type == AVMEDIA_TYPE_UNKNOWN || avctx->codec_type == codec->type) &&
116.     avctx->codec_id == AV_CODEC_ID_NONE) {
117.     avctx->codec_type = codec->type;
118.     avctx->codec_id = codec->id;
119. }
120. if (avctx->codec_id != codec->id || (avctx->codec_type != codec->type
121.     && avctx->codec_type != AVMEDIA_TYPE_ATTACHMENT)) {
122.     av_log(avctx, AV_LOG_ERROR, "Codec type or id mismatches\n");
123.     ret = AERROR(EINVAL);
124.     goto free_and_end;
125. }
126. avctx->frame_number = 0;
127. avctx->codec_descriptor = avcodec_descriptor_get(avctx->codec_id);
128. //检查编码器是否出于“实验”阶段
129. if (avctx->codec->capabilities & CODEC_CAP_EXPERIMENTAL &&
130.     avctx->strict_std_compliance > FF_COMPLIANCE_EXPERIMENTAL) {
131.     const char *codec_string = av_codec_is_encoder(codec) ? "encoder" : "decoder";
132.     AVCodec *codec2;
133.     av_log(avctx, AV_LOG_ERROR,
134.         "The %s '%s' is experimental but experimental codecs are not enabled, "
135.         "add '-strict %d' if you want to use it.\n",
136.         codec_string, codec->name, FF_COMPLIANCE_EXPERIMENTAL);
137.     codec2 = av_codec_is_encoder(codec) ? avcodec_find_encoder(codec->id) : avcodec_find_decoder(codec->id);
138.     if (!(codec2->capabilities & CODEC_CAP_EXPERIMENTAL))
139.         av_log(avctx, AV_LOG_ERROR, "Alternatively use the non experimental %s '%s'.\n",
140.             codec_string, codec2->name);
141.     ret = AERROR_EXPERIMENTAL;
142.     goto free_and_end;
143. }
144.
145. if (avctx->codec_type == AVMEDIA_TYPE_AUDIO &&
146.     (!avctx->time_base.num || !avctx->time_base.den)) {
147.     avctx->time_base.num = 1;
148.     avctx->time_base.den = avctx->sample_rate;
149. }
150.
151. if (!HAVE_THREADS)
152.     av_log(avctx, AV_LOG_WARNING, "Warning: not compiled with thread support, using thread emulation\n");
153.
154. if (CONFIG_FRAME_THREAD_ENCODER) {
155.     ff_unlock_avcodec(); //we will instantiate a few encoders thus kick the counter to prevent false detection of a problem
156.     ret = ff_frame_thread_encoder_init(avctx, options ? *options : NULL);
157.     ff_lock_avcodec(avctx);
158.     if (ret < 0)
159.         goto free_and_end;
160. }
161.
162. if (HAVE_THREADS
163.     && !(avctx->internal->frame_thread_encoder && (avctx->active_thread_type & FF_THREAD_FRAME))) {
164.     ret = ff_thread_init(avctx);
165.     if (ret < 0) {
166.         goto free_and_end;
167.     }
168. }
169. if (!HAVE_THREADS && !(codec->capabilities & CODEC_CAP_AUTO_THREADS))
170.     avctx->thread_count = 1;
171.
172. if (avctx->codec->max_lowres < avctx->lowres || avctx->lowres < 0) {
173.     av_log(avctx, AV_LOG_ERROR, "The maximum value for lowres supported by the decoder is %d\n",
174.         avctx->codec->max_lowres);
175.     ret = AERROR(EINVAL);
176.     goto free_and_end;
177. }
178.
179. #if FF_API_VISMV
180.     if (avctx->debug_mv)
181.         av_log(avctx, AV_LOG_WARNING, "The 'vismv' option is deprecated, "
182.             "see the codecview filter instead.\n");
183. #endif
184. //检查输入参数是否符合【编码器】要求
185. if (av_codec_is_encoder(avctx->codec)) {
186.     int i;
187.     //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
188.     if (avctx->codec->sample_fmts) {
189.         //遍历编码器支持的所有采样率
190.         for (i = 0; avctx->codec->sample_fmts[i] != AV_SAMPLE_FMT_NONE; i++) {

```

```

191.         //如果设置的采样率==编码器支持的采样率, 跳出循环。
192.         if (avctx->sample_fmt == avctx->codec->sample_fmts[i])
193.             break;
194.         if (avctx->channels == 1 &&
195.             av_get_planar_sample_fmt(avctx->sample_fmt) ==
196.             av_get_planar_sample_fmt(avctx->codec->sample_fmts[i])) {
197.             avctx->sample_fmt = avctx->codec->sample_fmts[i];
198.             break;
199.         }
200.     }
201.     //再检查一下采样率取值是否正确
202.     //注意, 此时的i值没有变化
203.     if (avctx->codec->sample_fmts[i] == AV_SAMPLE_FMT_NONE) {
204.         char buf[128];
205.         snprintf(buf, sizeof(buf), "%d", avctx->sample_fmt);
206.         av_log(avctx, AV_LOG_ERROR, "Specified sample format %s is invalid or not supported\n",
207.             (char *)av_x_if_null(av_get_sample_fmt_name(avctx->sample_fmt), buf));
208.         ret = AVERROR(EINVAL);
209.         goto free_and_end;
210.     }
211. }
212. //检查像素格式
213. if (avctx->codec->pix_fmts) {
214.     for (i = 0; avctx->codec->pix_fmts[i] != AV_PIX_FMT_NONE; i++)
215.         if (avctx->pix_fmt == avctx->codec->pix_fmts[i])
216.             break;
217.     if (avctx->codec->pix_fmts[i] == AV_PIX_FMT_NONE
218.        && !((avctx->codec_id == AV_CODEC_ID_MJPEG || avctx->codec_id == AV_CODEC_ID_LJPEG)
219.            && avctx->strict_std_compliance <= FF_COMPLIANCE_UNOFFICIAL)) {
220.         char buf[128];
221.         snprintf(buf, sizeof(buf), "%d", avctx->pix_fmt);
222.         av_log(avctx, AV_LOG_ERROR, "Specified pixel format %s is invalid or not supported\n",
223.             (char *)av_x_if_null(av_get_pix_fmt_name(avctx->pix_fmt), buf));
224.         ret = AVERROR(EINVAL);
225.         goto free_and_end;
226.     }
227.     if (avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ420P ||
228.        avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ411P ||
229.        avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ422P ||
230.        avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ440P ||
231.        avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ444P)
232.         avctx->color_range = AVCOL_RANGE_JPEG;
233. }
234. //检查采样率
235. if (avctx->codec->supported_samplerates) {
236.     for (i = 0; avctx->codec->supported_samplerates[i] != 0; i++)
237.         if (avctx->sample_rate == avctx->codec->supported_samplerates[i])
238.             break;
239.     if (avctx->codec->supported_samplerates[i] == 0) {
240.         av_log(avctx, AV_LOG_ERROR, "Specified sample rate %d is not supported\n",
241.             avctx->sample_rate);
242.         ret = AVERROR(EINVAL);
243.         goto free_and_end;
244.     }
245. }
246. //检查声道布局
247. if (avctx->codec->channel_layouts) {
248.     if (!avctx->channel_layout) {
249.         av_log(avctx, AV_LOG_WARNING, "Channel layout not specified\n");
250.     } else {
251.         for (i = 0; avctx->codec->channel_layouts[i] != 0; i++)
252.             if (avctx->channel_layout == avctx->codec->channel_layouts[i])
253.                 break;
254.         if (avctx->codec->channel_layouts[i] == 0) {
255.             char buf[512];
256.             av_get_channel_layout_string(buf, sizeof(buf), -1, avctx->channel_layout);
257.             av_log(avctx, AV_LOG_ERROR, "Specified channel layout '%s' is not supported\n", buf);
258.             ret = AVERROR(EINVAL);
259.             goto free_and_end;
260.         }
261.     }
262. }
263. //检查声道数
264. if (avctx->channel_layout && avctx->channels) {
265.     int channels = av_get_channel_layout_nb_channels(avctx->channel_layout);
266.     if (channels != avctx->channels) {
267.         char buf[512];
268.         av_get_channel_layout_string(buf, sizeof(buf), -1, avctx->channel_layout);
269.         av_log(avctx, AV_LOG_ERROR,
270.             "Channel layout '%s' with %d channels does not match number of specified channels %d\n",
271.             buf, channels, avctx->channels);
272.         ret = AVERROR(EINVAL);
273.         goto free_and_end;
274.     }
275. } else if (avctx->channel_layout) {
276.     avctx->channels = av_get_channel_layout_nb_channels(avctx->channel_layout);
277. }
278. //检查宽高
279. if (avctx->codec_type == AVMEDIA_TYPE_VIDEO) {
280.     if (avctx->width <= 0 || avctx->height <= 0) {
281.         av_log(avctx, AV_LOG_ERROR, "dimensions not set\n");

```

```

282.         ret = AVERROr(EINVAL);
283.         goto free_and_end;
284.     }
285. }
286. //检查码率
287. if ( (avctx->codec_type == AVMEDIA_TYPE_VIDEO || avctx->codec_type == AVMEDIA_TYPE_AUDIO)
288.     && avctx->bit_rate>0 && avctx->bit_rate<1000) {
289.     av_log(avctx, AV_LOG_WARNING, "Bitrate %d is extremely low, maybe you mean %dk\n", avctx->bit_rate, avctx->bit_rate);
290. }
291.
292. if (!avctx->rc_initial_buffer_occupancy)
293.     avctx->rc_initial_buffer_occupancy = avctx->rc_buffer_size * 3 / 4;
294. }
295.
296. avctx->pts_correction_num_faulty_pts =
297. avctx->pts_correction_num_faulty_dts = 0;
298. avctx->pts_correction_last_pts =
299. avctx->pts_correction_last_dts = INT64_MIN;
300. //关键:
301. //一切检查都无误之后, 调用编解码器初始化函数
302. if ( avctx->codec->init && (!avctx->active_thread_type&FF_THREAD_FRAME)
303.     || avctx->internal->frame_thread_encoder) {
304.     ret = avctx->codec->init(avctx);
305.     if (ret < 0) {
306.         goto free_and_end;
307.     }
308. }
309.
310. ret=0;
311.
312. #if FF_API_AUDIOENC_DELAY
313.     if (av_codec_is_encoder(avctx->codec))
314.         avctx->delay = avctx->initial_padding;
315. #endif
316.
317. //【解码器】
318. //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
319. if (av_codec_is_decoder(avctx->codec)) {
320.     if (!avctx->bit_rate)
321.         avctx->bit_rate = get_bit_rate(avctx);
322.     /* validate channel layout from the decoder */
323.     if (avctx->channel_layout) {
324.         int channels = av_get_channel_layout_nb_channels(avctx->channel_layout);
325.         if (!avctx->channels)
326.             avctx->channels = channels;
327.         else if (channels != avctx->channels) {
328.             char buf[512];
329.             av_get_channel_layout_string(buf, sizeof(buf), -1, avctx->channel_layout);
330.             av_log(avctx, AV_LOG_WARNING,
331.                 "Channel layout '%s' with %d channels does not match specified number of channels %d: "
332.                 "ignoring specified channel layout\n",
333.                 buf, channels, avctx->channels);
334.             avctx->channel_layout = 0;
335.         }
336.     }
337.
338.     if (avctx->channels && avctx->channels < 0 ||
339.         avctx->channels > FF_SANE_NB_CHANNELS) {
340.         ret = AVERROr(EINVAL);
341.         goto free_and_end;
342.     }
343.     if (avctx->sub_charenc) {
344.         if (avctx->codec_type != AVMEDIA_TYPE_SUBTITLE) {
345.             av_log(avctx, AV_LOG_ERROR, "Character encoding is only "
346.                 "supported with subtitles codecs\n");
347.             ret = AVERROr(EINVAL);
348.             goto free_and_end;
349.         } else if (avctx->codec_descriptor->props & AV_CODEC_PROP_BITMAP_SUB) {
350.             av_log(avctx, AV_LOG_WARNING, "Codec '%s' is bitmap-based, "
351.                 "subtitles character encoding will be ignored\n",
352.                 avctx->codec_descriptor->name);
353.             avctx->sub_charenc_mode = FF_SUB_CHARENC_MODE_DO_NOTHING;
354.         } else {
355.             /* input character encoding is set for a text based subtitle
356.              * codec at this point */
357.             if (avctx->sub_charenc_mode == FF_SUB_CHARENC_MODE_AUTOMATIC)
358.                 avctx->sub_charenc_mode = FF_SUB_CHARENC_MODE_PRE_DECODER;
359.
360.             if (avctx->sub_charenc_mode == FF_SUB_CHARENC_MODE_PRE_DECODER) {
361. #if CONFIG_ICONV
362.                 iconv_t cd = iconv_open("UTF-8", avctx->sub_charenc);
363.                 if (cd == (iconv_t)-1) {
364.                     ret = AVERROr(errno);
365.                     av_log(avctx, AV_LOG_ERROR, "Unable to open iconv context "
366.                         "with input character encoding \"%s\"\n", avctx->sub_charenc);
367.                     goto free_and_end;
368.                 }
369.                 iconv_close(cd);
370. #else
371.                 av_log(avctx, AV_LOG_ERROR, "Character encoding subtitles "
372.                     "conversion needs a libavcodec built with iconv support "
373.                     "for this codec\n");

```

```

373.         "for this codec\n");
374.         ret = AVERROR(ENOSYS);
375.         goto free_and_end;
376.     #endif
377.     }
378. }
379. }
380.
381. #if FF_API_AVCTX_TIMEBASE
382.     if (avctx->framerate.num > 0 && avctx->framerate.den > 0)
383.         avctx->time_base = av_inv_q(av_mul_q(avctx->framerate, (AVRational){avctx->ticks_per_frame, 1}));
384. #endif
385. }
386. end:
387.     ff_unlock_avcodec();
388.     if (options) {
389.         av_dict_free(options);
390.         *options = tmp;
391.     }
392.
393.     return ret;
394. free_and_end:
395.     av_dict_free(&tmp);
396.     if (codec->priv_class && codec->priv_data_size)
397.         av_opt_free(avctx->priv_data);
398.     av_freep(&avctx->priv_data);
399.     if (avctx->internal) {
400.         av_frame_free(&avctx->internal->to_free);
401.         av_freep(&avctx->internal->pool);
402.     }
403.     av_freep(&avctx->internal);
404.     avctx->codec = NULL;
405.     goto end;
406. }

```

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。

```

1. //检查像素格式
2.     if (avctx->codec->pix_fmts) {
3.         for (i = 0; avctx->codec->pix_fmts[i] != AV_PIX_FMT_NONE; i++)
4.             if (avctx->pix_fmt == avctx->codec->pix_fmts[i])
5.                 break;
6.         if (avctx->codec->pix_fmts[i] == AV_PIX_FMT_NONE
7.             && !((avctx->codec_id == AV_CODEC_ID_MJPEG || avctx->codec_id == AV_CODEC_ID_LJPEG)
8.                 && avctx->strict_std_compliance <= FF_COMPLIANCE_UNOFFICIAL)) {
9.             char buf[128];
10.            snprintf(buf, sizeof(buf), "%d", avctx->pix_fmt);
11.            av_log(avctx, AV_LOG_ERROR, "Specified pixel format %s is invalid or not supported\n",
12.                (char *)av_x_if_null(av_get_pix_fmt_name(avctx->pix_fmt), buf));
13.            ret = AVERROR(EINVAL);
14.            goto free_and_end;
15.        }
16.        if (avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ420P ||
17.            avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ411P ||
18.            avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ422P ||
19.            avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ440P ||
20.            avctx->codec->pix_fmts[i] == AV_PIX_FMT_YUVJ444P)
21.            avctx->color_range = AVCOL_RANGE_JPEG;
22.    }

```

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]
1. static const enum AVPixelFormat pix_fmts_8bit[] = {
2.     AV_PIX_FMT_YUV420P,
3.     AV_PIX_FMT_YUVJ420P,
4.     AV_PIX_FMT_YUV422P,
5.     AV_PIX_FMT_YUVJ422P,
6.     AV_PIX_FMT_YUV444P,
7.     AV_PIX_FMT_YUVJ444P,
8.     AV_PIX_FMT_NV12,
9.     AV_PIX_FMT_NV16,
10.    AV_PIX_FMT_NONE
11. };
```

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]
1. AVCodec ff_libx264_encoder = {
2.     .name           = "libx264",
3.     .long_name      = NULL_IF_CONFIG_SMALL("libx264 H.264 / AVC / MPEG-4 AVC / MPEG-4 part 10"),
4.     .type           = AVMEDIA_TYPE_VIDEO,
5.     .id             = AV_CODEC_ID_H264,
6.     .priv_data_size = sizeof(X264Context),
7.     .init            = X264_init,
8.     .encode2         = X264_frame,
9.     .close           = X264_close,
10.    .capabilities     = CODEC_CAP_DELAY | CODEC_CAP_AUTO_THREADS,
11.    .priv_class       = &x264_class,
12.    .defaults         = x264_defaults,
13.    .init_static_data = X264_init_static,
14. };
```

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]
1. static av_cold int X264_init(AVCodecContext *avctx)
2. {
3.     X264Context *x4 = avctx->priv_data;
4.     int sw,sh;
5.
6.     if (avctx->global_quality > 0)
7.         av_log(avctx, AV_LOG_WARNING, "-qscale is ignored, -crf is recommended.\n");
8.
9.     x264_param_default(&x4->params);
10.
11.    x4->params.b_deblocking_filter = avctx->flags & CODEC_FLAG_LOOP_FILTER;
12.
13.    if (x4->preset || x4->tune)
14.        if (x264_param_default_preset(&x4->params, x4->preset, x4->tune) < 0) {
15.            int i;
16.            av_log(avctx, AV_LOG_ERROR, "Error setting preset/tune %s/%s.\n", x4->preset, x4->tune);
17.            av_log(avctx, AV_LOG_INFO, "Possible presets:");
18.            for (i = 0; x264_preset_names[i]; i++)
19.                av_log(avctx, AV_LOG_INFO, " %s", x264_preset_names[i]);
20.            av_log(avctx, AV_LOG_INFO, "\n");
21.            av_log(avctx, AV_LOG_INFO, "Possible tunes:");
22.            for (i = 0; x264_tune_names[i]; i++)
23.                av_log(avctx, AV_LOG_INFO, " %s", x264_tune_names[i]);
24.            av_log(avctx, AV_LOG_INFO, "\n");
25.            return AVERROR(EINVAL);
26.        }
27.
28.    if (avctx->level > 0)
29.        x4->params.i_level_idc = avctx->level;
30.
31.    x4->params.pf_log = X264_log;
32.    x4->params.p_log_private = avctx;
33.    x4->params.i_log_level = X264_LOG_DEBUG;
34.    x4->params.i_csp = convert_pix_fmt(avctx->pix_fmt);
35.
36.    OPT_STR("weightp", x4->wpredep);
37.
38.    if (avctx->bit_rate) {
39.        x4->params.rc.i_bitrate = avctx->bit_rate / 1000;
40.        x4->params.rc.i_rc_method = X264_RC_ABR;
41.    }
42.    x4->params.rc.i_vbv_buffer_size = avctx->rc_buffer_size / 1000;
```



```

43. x4->params.rc.i_vbv_max_bitrate = avctx->rc_max_rate / 1000;
44. x4->params.rc.b_stat_write = avctx->flags & CODEC_FLAG_PASS1;
45. if (avctx->flags & CODEC_FLAG_PASS2) {
46.     x4->params.rc.b_stat_read = 1;
47. } else {
48.     if (x4->crf >= 0) {
49.         x4->params.rc.i_rc_method = X264_RC_CRF;
50.         x4->params.rc.f_rf_constant = x4->crf;
51.     } else if (x4->cqp >= 0) {
52.         x4->params.rc.i_rc_method = X264_RC_CQP;
53.         x4->params.rc.i_qp_constant = x4->cqp;
54.     }
55.
56.     if (x4->crf_max >= 0)
57.         x4->params.rc.f_rf_constant_max = x4->crf_max;
58. }
59.
60. if (avctx->rc_buffer_size && avctx->rc_initial_buffer_occupancy > 0 &&
61.     (avctx->rc_initial_buffer_occupancy <= avctx->rc_buffer_size)) {
62.     x4->params.rc.f_vbv_buffer_init =
63.         (float)avctx->rc_initial_buffer_occupancy / avctx->rc_buffer_size;
64. }
65.
66. OPT_STR("level", x4->level);
67.
68. if (avctx->i_quant_factor > 0)
69.     x4->params.rc.f_ip_factor = 1 / fabs(avctx->i_quant_factor);
70. if (avctx->b_quant_factor > 0)
71.     x4->params.rc.f_pb_factor = avctx->b_quant_factor;
72. if (avctx->chromaoffset)
73.     x4->params.analyse.i_chroma_qp_offset = avctx->chromaoffset;
74.
75. if (avctx->me_method == ME_EPZS)
76.     x4->params.analyse.i_me_method = X264_ME_DIA;
77. else if (avctx->me_method == ME_HEX)
78.     x4->params.analyse.i_me_method = X264_ME_HEX;
79. else if (avctx->me_method == ME_UHM)
80.     x4->params.analyse.i_me_method = X264_ME_UHM;
81. else if (avctx->me_method == ME_FULL)
82.     x4->params.analyse.i_me_method = X264_ME_ESA;
83. else if (avctx->me_method == ME_TESA)
84.     x4->params.analyse.i_me_method = X264_ME_TESA;
85.
86. if (avctx->gop_size >= 0)
87.     x4->params.i_keyint_max = avctx->gop_size;
88. if (avctx->max_b_frames >= 0)
89.     x4->params.i_bframe = avctx->max_b_frames;
90. if (avctx->scenechange_threshold >= 0)
91.     x4->params.i_scenecut_threshold = avctx->scenechange_threshold;
92. if (avctx->qmin >= 0)
93.     x4->params.rc.i_qp_min = avctx->qmin;
94. if (avctx->qmax >= 0)
95.     x4->params.rc.i_qp_max = avctx->qmax;
96. if (avctx->max_qdiff >= 0)
97.     x4->params.rc.i_qp_step = avctx->max_qdiff;
98. if (avctx->qblur >= 0)
99.     x4->params.rc.f_qblur = avctx->qblur; /* temporally blur quant */
100. if (avctx->qcompress >= 0)
101.     x4->params.rc.f_qcompress = avctx->qcompress; /* 0.0 => cbr, 1.0 => constant qp */
102. if (avctx->refs >= 0)
103.     x4->params.i_frame_reference = avctx->refs;
104. else if (x4->level) {
105.     int i;
106.     int mbn = FF_CEIL_RSHIFT(avctx->width, 4) * FF_CEIL_RSHIFT(avctx->height, 4);
107.     int level_id = -1;
108.     char *tail;
109.     int scale = X264_BUILD < 129 ? 384 : 1;
110.
111.     if (!strcmp(x4->level, "1b")) {
112.         level_id = 9;
113.     } else if (strlen(x4->level) <= 3) {
114.         level_id = av_strtod(x4->level, &tail) * 10 + 0.5;
115.         if (*tail)
116.             level_id = -1;
117.     }
118.     if (level_id <= 0)
119.         av_log(avctx, AV_LOG_WARNING, "Failed to parse level\n");
120.
121.     for (i = 0; i < x264_levels[i].level_idc; i++)
122.         if (x264_levels[i].level_idc == level_id)
123.             x4->params.i_frame_reference = av_clip(x264_levels[i].dpb / mbn / scale, 1, x4->params.i_frame_reference);
124. }
125.
126. if (avctx->trellis >= 0)
127.     x4->params.analyse.i_trellis = avctx->trellis;
128. if (avctx->me_range >= 0)
129.     x4->params.analyse.i_me_range = avctx->me_range;
130. if (avctx->noise_reduction >= 0)
131.     x4->params.analyse.i_noise_reduction = avctx->noise_reduction;
132. if (avctx->me_subpel_quality >= 0)
133.     x4->params.analyse.i_subpel_refine = avctx->me_subpel_quality;

```

```

134.     if (avctx->b_frame_strategy >= 0)
135.         x4->params.i_bframe_adaptive = avctx->b_frame_strategy;
136.     if (avctx->keyint_min >= 0)
137.         x4->params.i_keyint_min = avctx->keyint_min;
138.     if (avctx->coder_type >= 0)
139.         x4->params.b_cabac = avctx->coder_type == FF_CODER_TYPE_AC;
140.     if (avctx->me_cmp >= 0)
141.         x4->params.analyse.b_chroma_me = avctx->me_cmp & FF_CMP_CHROMA;
142.
143.     if (x4->aq_mode >= 0)
144.         x4->params.rc.i_aq_mode = x4->aq_mode;
145.     if (x4->aq_strength >= 0)
146.         x4->params.rc.f_aq_strength = x4->aq_strength;
147.     PARSE_X264_OPT("psy-rd", psy_rd);
148.     PARSE_X264_OPT("deblock", deblock);
149.     PARSE_X264_OPT("partitions", partitions);
150.     PARSE_X264_OPT("stats", stats);
151.     if (x4->psy >= 0)
152.         x4->params.analyse.b_psy = x4->psy;
153.     if (x4->rc_lookahead >= 0)
154.         x4->params.rc.i_lookahead = x4->rc_lookahead;
155.     if (x4->weightp >= 0)
156.         x4->params.analyse.i_weighted_pred = x4->weightp;
157.     if (x4->weightb >= 0)
158.         x4->params.analyse.b_weighted_bipred = x4->weightb;
159.     if (x4->cplxblur >= 0)
160.         x4->params.rc.f_complexity_blur = x4->cplxblur;
161.
162.     if (x4->ssim >= 0)
163.         x4->params.analyse.b_ssim = x4->ssim;
164.     if (x4->intra_refresh >= 0)
165.         x4->params.b_intra_refresh = x4->intra_refresh;
166.     if (x4->bluray_compat >= 0) {
167.         x4->params.b_bluray_compat = x4->bluray_compat;
168.         x4->params.b_vfr_input = 0;
169.     }
170.     if (x4->avcintra_class >= 0)
171. #if X264_BUILD >= 142
172.         x4->params.i_avcintra_class = x4->avcintra_class;
173. #else
174.         av_log(avctx, AV_LOG_ERROR,
175.              "x264 too old for AVC Intra, at least version 142 needed\n");
176. #endif
177.     if (x4->b_bias != INT_MIN)
178.         x4->params.i_bframe_bias = x4->b_bias;
179.     if (x4->b_pyramid >= 0)
180.         x4->params.i_bframe_pyramid = x4->b_pyramid;
181.     if (x4->mixed_refs >= 0)
182.         x4->params.analyse.b_mixed_references = x4->mixed_refs;
183.     if (x4->dct8x8 >= 0)
184.         x4->params.analyse.b_transform_8x8 = x4->dct8x8;
185.     if (x4->fast_pskip >= 0)
186.         x4->params.analyse.b_fast_pskip = x4->fast_pskip;
187.     if (x4->aud >= 0)
188.         x4->params.b_aud = x4->aud;
189.     if (x4->mbtree >= 0)
190.         x4->params.rc.b_mb_tree = x4->mbtree;
191.     if (x4->direct_pred >= 0)
192.         x4->params.analyse.i_direct_mv_pred = x4->direct_pred;
193.
194.     if (x4->slice_max_size >= 0)
195.         x4->params.i_slice_max_size = x4->slice_max_size;
196.     else {
197.         /*
198.          * Allow x264 to be instructed through AVCodecContext about the maximum
199.          * size of the RTP payload. For example, this enables the production of
200.          * payload suitable for the H.264 RTP packetization-mode 0 i.e. single
201.          * NAL unit per RTP packet.
202.          */
203.         if (avctx->rtp_payload_size)
204.             x4->params.i_slice_max_size = avctx->rtp_payload_size;
205.     }
206.
207.     if (x4->fastfirstpass)
208.         x264_param_apply_fastfirstpass(&x4->params);
209.
210.     /* Allow specifying the x264 profile through AVCodecContext. */
211.     if (!x4->profile)
212.         switch (avctx->profile) {
213.             case FF_PROFILE_H264_BASELINE:
214.                 x4->profile = av_strdup("baseline");
215.                 break;
216.             case FF_PROFILE_H264_HIGH:
217.                 x4->profile = av_strdup("high");
218.                 break;
219.             case FF_PROFILE_H264_HIGH_10:
220.                 x4->profile = av_strdup("high10");
221.                 break;
222.             case FF_PROFILE_H264_HIGH_422:
223.                 x4->profile = av_strdup("high422");
224.                 break;

```

```

225.     case FF_PROFILE_H264_HIGH_444:
226.         x4->profile = av_strdup("high444");
227.         break;
228.     case FF_PROFILE_H264_MAIN:
229.         x4->profile = av_strdup("main");
230.         break;
231.     default:
232.         break;
233. }
234.
235. if (x4->nal_hrd >= 0)
236.     x4->params.i_nal_hrd = x4->nal_hrd;
237.
238. if (x4->profile)
239.     if (x264_param_apply_profile(&x4->params, x4->profile) < 0) {
240.         int i;
241.         av_log(avctx, AV_LOG_ERROR, "Error setting profile %s.\n", x4->profile);
242.         av_log(avctx, AV_LOG_INFO, "Possible profiles:");
243.         for (i = 0; x264_profile_names[i]; i++)
244.             av_log(avctx, AV_LOG_INFO, " %s", x264_profile_names[i]);
245.         av_log(avctx, AV_LOG_INFO, "\n");
246.         return AVERROR(EINVAL);
247.     }
248.
249. x4->params.i_width      = avctx->width;
250. x4->params.i_height     = avctx->height;
251. av_reduce(&sw, &sh, avctx->sample_aspect_ratio.num, avctx->sample_aspect_ratio.den, 4096);
252. x4->params.vui.i_sar_width = sw;
253. x4->params.vui.i_sar_height = sh;
254. x4->params.i_timebase_den = avctx->time_base.den;
255. x4->params.i_timebase_num = avctx->time_base.num;
256. x4->params.i_fps_num = avctx->time_base.den;
257. x4->params.i_fps_den = avctx->time_base.num * avctx->ticks_per_frame;
258.
259. x4->params.analyse.b_psnr = avctx->flags & CODEC_FLAG_PSNR;
260.
261. x4->params.i_threads     = avctx->thread_count;
262. if (avctx->thread_type)
263.     x4->params.b_sliced_threads = avctx->thread_type == FF_THREAD_SLICE;
264.
265. x4->params.b_interlaced  = avctx->flags & CODEC_FLAG_INTERLACED_DCT;
266.
267. x4->params.b_open_gop    = !(avctx->flags & CODEC_FLAG_CLOSED_GOP);
268.
269. x4->params.i_slice_count = avctx->slices;
270.
271. x4->params.vui.b_fullrange = avctx->pix_fmt == AV_PIX_FMT_YUVJ420P ||
272.                             avctx->pix_fmt == AV_PIX_FMT_YUVJ422P ||
273.                             avctx->pix_fmt == AV_PIX_FMT_YUVJ444P ||
274.                             avctx->color_range == AVCOL_RANGE_JPEG;
275.
276. if (avctx->colorspace != AVCOL_SPC_UNSPECIFIED)
277.     x4->params.vui.i_colmatrix = avctx->colorspace;
278. if (avctx->color_primaries != AVCOL_PRI_UNSPECIFIED)
279.     x4->params.vui.i_colorprim = avctx->color_primaries;
280. if (avctx->color_trc != AVCOL_TRC_UNSPECIFIED)
281.     x4->params.vui.i_transfer = avctx->color_trc;
282.
283. if (avctx->flags & CODEC_FLAG_GLOBAL_HEADER)
284.     x4->params.b_repeat_headers = 0;
285.
286. if(x4->x264opts){
287.     const char *p= x4->x264opts;
288.     while(p){
289.         char param[256]={0}, val[256]={0};
290.         if(sscanf(p, "%255[^\]=%255[^\:]", param, val) == 1){
291.             OPT_STR(param, "1");
292.         }else
293.             OPT_STR(param, val);
294.         p= strchr(p, ':');
295.         p+=!!p;
296.     }
297. }
298.
299. if (x4->x264_params) {
300.     AVDictionary *dict = NULL;
301.     AVDictionaryEntry *en = NULL;
302.
303.     if (!av_dict_parse_string(&dict, x4->x264_params, "=", ":", 0)) {
304.         while ((en = av_dict_get(dict, "", en, AV_DICT_IGNORE_SUFFIX))) {
305.             if (x264_param_parse(&x4->params, en->key, en->value) < 0)
306.                 av_log(avctx, AV_LOG_WARNING,
307.                     "Error parsing option '%s = %s'.\n",
308.                     en->key, en->value);
309.         }
310.
311.         av_dict_free(&dict);
312.     }
313. }
314.
315. // update AVCodecContext with x264 parameters

```

```



316.     avctx->has_b_frames = x4->params.i_bframe ?
317.         x4->params.i_bframe_pyramid ? 2 : 1 : 0;
318.     if (avctx->max_b_frames < 0)
319.         avctx->max_b_frames = 0;
320.
321.     avctx->bit_rate = x4->params.rc.i_bitrate*1000;
322.
323.     x4->enc = x264_encoder_open(&x4->params);
324.     if (!x4->enc)
325.         return -1;
326.
327.     avctx->coded_frame = av_frame_alloc();
328.     if (!avctx->coded_frame)
329.         return AVERROR(ENOMEM);
330.
331.     if (avctx->flags & CODEC_FLAG_GLOBAL_HEADER) {
332.         x264_nal_t *nal;
333.         uint8_t *p;
334.         int nnal, s, i;
335.
336.         s = x264_encoder_headers(x4->enc, &nal, &nnal);
337.         avctx->extradata = p = av_malloc(s);
338.
339.         for (i = 0; i < nnal; i++) {
340.             /* Don't put the SEI in extradata. */
341.             if (nal[i].i_type == NAL_SEI) {
342.                 av_log(avctx, AV_LOG_INFO, "%s\n", nal[i].p_payload+25);
343.                 x4->sei_size = nal[i].i_payload;
344.                 x4->sei = av_malloc(x4->sei_size);
345.                 memcpy(x4->sei, nal[i].p_payload, nal[i].i_payload);
346.                 continue;
347.             }
348.             memcpy(p, nal[i].p_payload, nal[i].i_payload);
349.             p += nal[i].i_payload;
350.         }
351.         avctx->extradata_size = p - avctx->extradata;
352.     }
353.
354.     return 0;
355. }

```

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]  
1. typedef struct X264Context {
2.     AVClass      *class;
3.     x264_param_t  params;
4.     x264_t        *enc;
5.     x264_picture_t pic;
6.     uint8_t       *sei;
7.     int           sei_size;
8.     char *preset;
9.     char *tune;
10.    char *profile;
11.    char *level;
12.    int fastfirstpass;
13.    char *wpredp;
14.    char *x264opts;
15.    float crf;
16.    float crf_max;
17.    int cqp;
18.    int aq_mode;
19.    float aq_strength;
20.    char *psy_rd;
21.    int psy;
22.    int rc_lookahead;
23.    int weightp;
24.    int weightb;
25.    int ssim;
26.    int intra_refresh;
27.    int bluray_compat;
28.    int b_bias;
29.    int b_pyramid;
30.    int mixed_refs;
31.    int dct8x8;
32.    int fast_pskip;
33.    int aud;
34.    int mbtree;
35.    char *deblock;
36.    float cplxblur;
37.    char *partitions;
38.    int direct_pred;
39.    int slice_max_size;
40.    char *stats;
41.    int nal_hrd;
42.    int avcintra_class;
43.    char *x264_params;
44. } X264Context;
```

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/44117891>

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](#)

此PDF由[spygg](#)生成, 请尊重原作者版权!!!

我的邮箱: liushidc@163.com