

=====

H.264源代码分析文章列表：

[【编码 - x264】](#)

[x264源代码简单分析：概述](#)

[x264源代码简单分析：x264命令行工具（x264.exe）](#)

[x264源代码简单分析：编码器主干部分-1](#)

[x264源代码简单分析：编码器主干部分-2](#)

[x264源代码简单分析：x264_slice_write\(\)](#)

[x264源代码简单分析：滤波（Filter）部分](#)

[x264源代码简单分析：宏块分析（Analysis）部分-帧内宏块（Intra）](#)

[x264源代码简单分析：宏块分析（Analysis）部分-帧间宏块（Inter）](#)

[x264源代码简单分析：宏块编码（Encode）部分](#)

[x264源代码简单分析：熵编码（Entropy Encoding）部分](#)

[FFmpeg与libx264接口源代码简单分析](#)

[【解码 - libavcodec H.264 解码器】](#)

[FFmpeg的H.264解码器源代码简单分析：概述](#)

[FFmpeg的H.264解码器源代码简单分析：解析器（Parser）部分](#)

[FFmpeg的H.264解码器源代码简单分析：解码器主干部分](#)

[FFmpeg的H.264解码器源代码简单分析：熵解码（EntropyDecoding）部分](#)

[FFmpeg的H.264解码器源代码简单分析：宏块解码（Decode）部分-帧内宏块（Intra）](#)

[FFmpeg的H.264解码器源代码简单分析：宏块解码（Decode）部分-帧间宏块（Inter）](#)

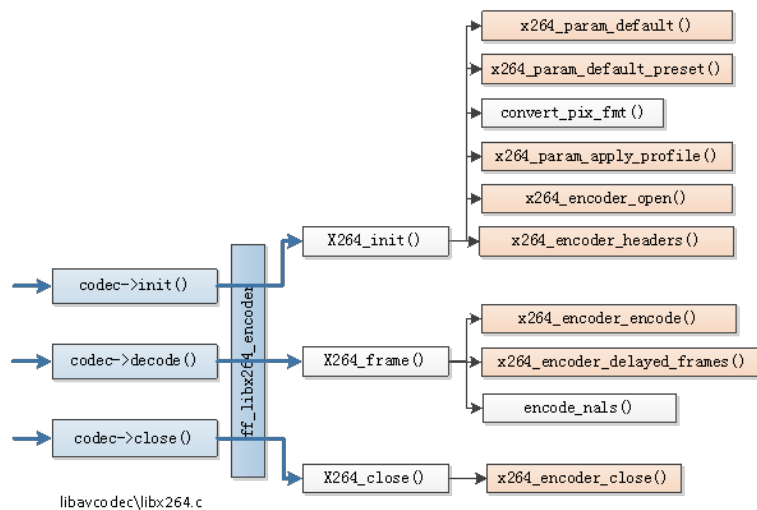
[FFmpeg的H.264解码器源代码简单分析：环路滤波（Loop Filter）部分](#)

=====

本文简单记录一下FFmpeg的libavcodec中与libx264接口部分的源代码。该部分源代码位于“libavcodec/libx264.c”中。正是有了这部分代码，使得FFmpeg可以调用libx264编码H.264视频。

函数调用关系图

FFmpeg的libavcodec中的libx264.c的函数调用关系如下图所示。



```
AVCodec* codec;
```

FFmpeg Source Analysis - Libx264.c

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

从图中可以看出，libx264对应的AVCodec结构体ff_libx264_encoder中设定编码器初始化函数是X264_init()，编码一帧数据的函数是X264_frame()，编码器关闭函数是X264_close()。

X264_init()调用了如下函数：

- [libx264 API] x264_param_default()：设置默认参数。
- [libx264 API] x264_param_default_preset()：设置默认preset。
- convert_pix_fmt()：将FFmpeg像素格式转换为libx264像素格式。
- [libx264 API] x264_param_apply_profile()：设置Profile。
- [libx264 API] x264_encoder_open()：打开编码器。
- [libx264 API] x264_encoder_headers()：需要全局头的时候，输出头信息。

X264_frame()调用了如下函数：

- [libx264 API] x264_encoder_encode()：编码一帧数据。
- [libx264 API] x264_encoder_delayed_frames()：输出编码器中缓存的数据。
- encode_nals()：将编码后得到的x264_nal_t转换为AVPacket。

X264_close()调用了如下函数：

- [libx264 API] x264_encoder_close()：关闭编码器。

下文将会分别分析X264_init()，X264_frame()和X264_close()这三个函数。

ff_libx264_encoder

ff_libx264_encoder是libx264对应的AVCodec结构体，定义如下所示。

```
1. //libx264对应的AVCodec结构体
2. AVCodec ff_libx264_encoder = {
3.     .name           = "libx264",
4.     .long_name      = NULL_IF_CONFIG_SMALL("libx264 H.264 / AVC / MPEG-4 AVC / MPEG-4 part 10"),
5.     .type           = AVMEDIA_TYPE_VIDEO,
6.     .id             = AV_CODEC_ID_H264,
7.     .priv_data_size = sizeof(X264Context),
8.     .init           = X264_init,
9.     .encode2        = X264_frame,
10.    .close           = X264_close,
11.    .capabilities    = CODEC_CAP_DELAY | CODEC_CAP_AUTO_THREADS,
12.    .priv_class      = &x264_class,
13.    .defaults        = x264_defaults,
14.    .init_static_data = X264_init_static,
15. };
```

从ff_libx264_encoder定义中可以看出：init()指向X264_init()，encode2()指向X264_frame()，close()指向X264_close()。此外priv_class指向一个x264_class静态结构体，该结构体是libx264对应的AVClass，定义如下。

```

1. static const AVClass x264_class = {
2.     .class_name = "libx264",
3.     .item_name  = av_default_item_name,
4.     .option     = options, //选项
5.     .version    = LIBAVUTIL_VERSION_INT,
6. };

```

x264_class中的option指向一个options[]静态数组，其中包含了libx264支持的AVOption选项，如下所示。

```

1. //FFmpeg针对libx264提供的可以通过AVOption设置的选项
2. #define OFFSET(x) offsetof(X264Context, x)
3. #define VE AV_OPT_FLAG_VIDEO_PARAM | AV_OPT_FLAG_ENCODING_PARAM
4. static const AVOption options[] = {
5.     { "preset",          "Set the encoding preset (cf. x264 --fullhelp)",  OFFSET(preset),          AV_OPT_TYPE_STRING, { .str = "medium" }, 0, 0, VE },
6.     { "tune",            "Tune the encoding params (cf. x264 --fullhelp)",  OFFSET(tune),            AV_OPT_TYPE_STRING, { 0 }, 0, 0, VE },
7.     { "profile",         "Set profile restrictions (cf. x264 --fullhelp)",  OFFSET(profile),         AV_OPT_TYPE_STRING, { 0 }, 0, 0, VE },
8.     { "fastfirstpass",   "Use fast settings when encoding first pass",  OFFSET(fastfirstpass),   AV_OPT_TYPE_INT,    { .i64 = 1 }, 0, 1, VE },
9.     { "level",           "Specify level (as defined by Annex A)",  OFFSET(level),           AV_OPT_TYPE_STRING, { .str=NULL }, 0, 0, VE },
10.    { "passlogfile",      "Filename for 2 pass stats",  OFFSET(stats),           AV_OPT_TYPE_STRING, { .str=NULL }, 0, 0, VE },
11.    { "wpredp",           "Weighted prediction for P-frames",  OFFSET(wpredp),          AV_OPT_TYPE_STRING, { .str=NULL }, 0, 0, VE },
12.    { "x264opts",         "x264 options",  OFFSET(x264opts),        AV_OPT_TYPE_STRING, { .str=NULL }, 0, 0, VE },
13.    { "crf",              "Select the quality for constant quality mode",  OFFSET(crf),             AV_OPT_TYPE_FLOAT,  { .dbl = -1 }, -1, FLT_MAX, VE },
14.    { "crf_max",          "In CRF mode, prevents VBV from lowering quality beyond this point.",  OFFSET(crf_max),         AV_OPT_TYPE_FLOAT,  { .dbl = -1 }, -1, FLT_MAX, VE },
15.    { "qp",               "Constant quantization parameter rate control method",  OFFSET(cqp),             AV_OPT_TYPE_INT,    { .i64 = -1 }, -1, INT_MAX, VE },
16.    { "aq-mode",          "AQ method",  OFFSET(aq_mode),         AV_OPT_TYPE_INT,    { .i64 = -1 }, -1, INT_MAX, VE, "aq_mode" },
17.    { "none",             NULL, 0, AV_OPT_TYPE_CONST, { .i64 = X264_AQ_NONE }, INT_MIN, INT_MAX, VE, "aq_mode" },
18.    { "variance",         "Variance AQ (complexity mask)", 0, AV_OPT_TYPE_CONST, { .i64 = X264_AQ_VARIANCE }, INT_MIN, INT_MAX, VE, "aq_mode" },
19.    { "autovariance",     "Auto-variance AQ (experimental)", 0, AV_OPT_TYPE_CONST, { .i64 = X264_AQ_AUTOVARIANCE }, INT_MIN, INT_MAX, VE, "aq_mode" },
20.    { "aq-strength",      "AQ strength. Reduces blocking and blurring in flat and textured areas.",  OFFSET(aq_strength),     AV_OPT_TYPE_FLOAT,  { .dbl = -1 }, -1, FLT_MAX, VE },
21.    { "psy",              "Use psychovisual optimizations.",  OFFSET(psy),             AV_OPT_TYPE_INT,    { .i64 = -1 }, -1, 1, VE },
22.    { "psy-rd",           "Strength of psychovisual optimization, in <psy-rd>:<psy-trellis> format.",  OFFSET(psy_rd),          AV_OPT_TYPE_STRING, { 0 }, 0, 0, VE },
23.    { "rc-lookahead",     "Number of frames to look ahead for frametype and ratecontrol",  OFFSET(rc_lookahead),    AV_OPT_TYPE_INT,    { .i64 = -1 }, -1, INT_MAX, VE },
24.    { "weightb",          "Weighted prediction for B-frames.",  OFFSET(weightb),         AV_OPT_TYPE_INT,    { .i64 = -1 }, -1, 1, VE },
25.    { "weightp",          "Weighted prediction analysis method.",  OFFSET(weightp),         AV_OPT_TYPE_INT,    { .i64 = -1 }, -1, INT_MAX, VE, "weightp" },
26.    { "none",             NULL, 0, AV_OPT_TYPE_CONST, { .i64 = X264_WEIGHTP_NONE }, INT_MIN, INT_MAX, VE, "weightp" },
27.    { "simple",            NULL, 0, AV_OPT_TYPE_CONST, { .i64 = X264_WEIGHTP_SIMPLE }, INT_MIN, INT_MAX, VE, "weightp" },
28.    { "smart",            NULL, 0, AV_OPT_TYPE_CONST, { .i64 = X264_WEIGHTP_SMART }, INT_MIN, INT_MAX, VE, "weightp" },
29.    { "ssim",             "Calculate and print SSIM stats.",  OFFSET(ssim),            AV_OPT_TYPE_INT,    { .i64 = -1 }, -1, 1, VE },
30.    { "intra-refresh",    "Use Periodic Intra Refresh instead of IDR frames.",  OFFSET(intra_refresh),   AV_OPT_TYPE_INT,    { .i64 = -1 }, -1, 1, VE },
31.    { "bluray-compat",    "Bluray compatibility workarounds.",  OFFSET(bluray_compat),   AV_OPT_TYPE_INT,    { .i64 = -1 }, -1, 1, VE },
32.    { "b-bias",           "Influences how often B-frames are used",  OFFSET(b_bias),          AV_OPT_TYPE_INT,    { .i64 = INT_MIN }, INT_MIN, INT_MAX, VE },
33.    { "b-pyramid",        "Keep some B-frames as references.",  OFFSET(b_pyramid),       AV_OPT_TYPE_INT,    { .i64 = -1 }, -1, INT_MAX, VE, "b_pyramid" },
34.    { "none",             NULL, 0, AV_OPT_TYPE_CONST, { .i64 = X264_B_PYRAMID_NONE }, INT_MIN, INT_MAX, VE, "b_pyramid" },
35.    { "strict",           "Strictly hierarchical pyramid", 0, AV_OPT_TYPE_CONST, { .i64 = X264_B_PYRAMID_STRICT }, INT_MIN, INT_MAX, VE, "b_pyramid" },
36.    { "normal",           "Non-strict (not Blu-ray compatible)", 0, AV_OPT_TYPE_CONST, { .i64 = X264_B_PYRAMID_NORMAL }, INT_MIN, INT_MAX, VE, "b_pyramid" },
37.    { "mixed-refs",       "One reference per partition, as opposed to one reference per macroblock",  OFFSET(mixed_refs),      AV_OPT_TYPE_INT,    { .i64 = -1 }, -1, 1, VE },
38.    { "8x8dct",           "High profile 8x8 transform.",  OFFSET(dct8x8),          AV_OPT_TYPE_INT,    { .i64 = -1 }, -1, 1, VE },
39.    { "fast-pskip",       NULL, 0, AV_OPT_TYPE_CONST, { .i64 = -1 }, -1, 1, VE },
40.    { "aud",              "Use access unit delimiters.",  OFFSET(aud),             AV_OPT_TYPE_INT,    { .i64 = -1 }, -1, 1, VE },
41.    { "mbtree",           "Use macroblock tree ratecontrol.",  OFFSET(mbtree),          AV_OPT_TYPE_INT,    { .i64 = -1 }, -1, 1, VE },
42.    { "deblock",          "Loop filter parameters, in <alpha:beta> form.",  OFFSET(deblock),         AV_OPT_TYPE_STRING, { 0 }, 0, 0, VE },
43.    { "cplxblur",         "Reduce fluctuations in QP (before curve compression)",  OFFSET(cplxblur),        AV_OPT_TYPE_FLOAT,  { .dbl = -1 }, -1, FLT_MAX, VE }

```

```

1, FLI_MAX, VE},
44. { "partitions", "A comma-separated list of partitions to consider. "
45. "Possible values: p8x8, p4x4, b8x8, i8x8, i4x4, none, all", OFFSET(partitions), AV_OPT_TYPE_STRING, { 0 }, 0,
0, VE},
46. { "direct-pred", "Direct MV prediction mode", OFFSET(direct_pred), AV_OPT_TYPE_INT, { .i64 = -1 },
-1, INT_MAX, VE, "direct-pred" },
47. { "none", NULL, 0, AV_OPT_TYPE_CONST, { .i64 = X264_DIRECT_PRED_NONE }, 0, 0, VE, "direct-pred" },
48. { "spatial", NULL, 0, AV_OPT_TYPE_CONST, { .i64 = X264_DIRECT_PRED_SPATIAL }, 0, 0, VE, "direct-pred" },
49. { "temporal", NULL, 0, AV_OPT_TYPE_CONST, { .i64 = X264_DIRECT_PRED_TEMPORAL }, 0, 0, VE, "direct-pred" },
50. { "auto", NULL, 0, AV_OPT_TYPE_CONST, { .i64 = X264_DIRECT_PRED_AUTO }, 0, 0, VE, "direct-pred" },
51. { "slice-max-size", "Limit the size of each slice in bytes", OFFSET(slice_max_size), AV_OPT_TYPE_INT, { .i64 = -1 },
-1, INT_MAX, VE },
52. { "stats", "Filename for 2 pass stats", OFFSET(stats), AV_OPT_TYPE_STRING, { 0 }, 0,
0, VE },
53. { "nal-hrd", "Signal HRD information (requires vbv-buFSIZE; "
54. "cbr not allowed in .mp4)", OFFSET(nal_hrd), AV_OPT_TYPE_INT, { .i64 = -1 },
-1, INT_MAX, VE, "nal-hrd" },
55. { "none", NULL, 0, AV_OPT_TYPE_CONST, { .i64 = X264_NAL_HRD_NONE }, INT_MIN, INT_MAX, VE, "nal-hrd" },
56. { "vbr", NULL, 0, AV_OPT_TYPE_CONST, { .i64 = X264_NAL_HRD_VBR }, INT_MIN, INT_MAX, VE, "nal-hrd" },
57. { "cbr", NULL, 0, AV_OPT_TYPE_CONST, { .i64 = X264_NAL_HRD_CBR }, INT_MIN, INT_MAX, VE, "nal-hrd" },
58. { "avcintra-class", "AVC-Intra class 50/100/200", OFFSET(avcintra_class), AV_OPT_TYPE_INT, { .i64 = -1 },
-1, 200, VE },
59. { "x264-params", "Override the x264 configuration using a :-
separated list of key=value parameters", OFFSET(x264_params), AV_OPT_TYPE_STRING, { 0 }, 0, 0, VE },
60. { NULL },
61. };

```

options[]数组中包含的选项支持在FFmpeg中通过AVOption进行设置。

X264_init()

X264_init()用于初始化libx264编码器。该函数的定义如下所示。

```

[cpp]
1. //libx264编码器初始化
2. static av_cold int X264_init(AVCodecContext *avctx)
3. {
4.     //FFmpeg中针对libx264的私有结构体
5.     X264Context *x4 = avctx->priv_data;
6.     int sw,sh;
7.
8.     if (avctx->global_quality > 0)
9.         av_log(avctx, AV_LOG_WARNING, "-qscale is ignored, -crf is recommended.\n");
10.
11.     //[[libx264 API] 设置默认参数
12.     x264_param_default(&x4->params);
13.
14.     x4->params.b_deblocking_filter = avctx->flags & CODEC_FLAG_LOOP_FILTER;
15.
16.     if (x4->preset || x4->tune)
17.         if (x264_param_default_preset(&x4->params, x4->preset, x4->tune) < 0) { //[[libx264 API] 设置preset
18.             int i;
19.             av_log(avctx, AV_LOG_ERROR, "Error setting preset/tune %s/%s.\n", x4->preset, x4->tune);
20.             av_log(avctx, AV_LOG_INFO, "Possible presets:");
21.             for (i = 0; x264_preset_names[i]; i++)
22.                 av_log(avctx, AV_LOG_INFO, " %s", x264_preset_names[i]);
23.             av_log(avctx, AV_LOG_INFO, "\n");
24.             av_log(avctx, AV_LOG_INFO, "Possible tunes:");
25.             for (i = 0; x264_tune_names[i]; i++)
26.                 av_log(avctx, AV_LOG_INFO, " %s", x264_tune_names[i]);
27.             av_log(avctx, AV_LOG_INFO, "\n");
28.             return AVERROR(EINVAL);
29.         }
30.
31.     if (avctx->level > 0)
32.         x4->params.i_level_idc = avctx->level;
33.     //libx264日志输出设置为FFmpeg的日志输出
34.     x4->params.pf_log = X264_log;
35.     x4->params.p_log_private = avctx;
36.     x4->params.i_log_level = X264_LOG_DEBUG;
37.     //FFmpeg像素格式映射到libx264
38.     x4->params.i_csp = convert_pix_fmt(avctx->pix_fmt);
39.
40.     OPT_STR("weightp", x4->wpredp);
41.
42.     //FFmpeg码率映射到libx264
43.     if (avctx->bit_rate) {
44.         x4->params.rc.i_bitrate = avctx->bit_rate / 1000;
45.         x4->params.rc.i_rc_method = X264_RC_ABR;
46.     }
47.     x4->params.rc.i_vbv_buffer_size = avctx->rc_buffer_size / 1000;
48.     x4->params.rc.i_vbv_max_bitrate = avctx->rc_max_rate / 1000;
49.     x4->params.rc.b_stat_write = avctx->flags & CODEC_FLAG_PASS1;
50.     if (avctx->flags & CODEC_FLAG_PASS2) {
51.         x4->params.rc.b_stat_read = 1;
52.     } else {
53.         if (x4->crf >= 0) {

```

```

54.         x4->params.rc.i_rc_method = X264_RC_CRF;
55.         x4->params.rc.f_rf_constant = x4->crf;
56.     } else if (x4->cqp >= 0) {
57.         x4->params.rc.i_rc_method = X264_RC_CQP;
58.         x4->params.rc.i_qp_constant = x4->cqp;
59.     }
60.
61.     if (x4->crf_max >= 0)
62.         x4->params.rc.f_rf_constant_max = x4->crf_max;
63. }
64.
65. if (avctx->rc_buffer_size && avctx->rc_initial_buffer_occupancy > 0 &&
66.     (avctx->rc_initial_buffer_occupancy <= avctx->rc_buffer_size)) {
67.     x4->params.rc.f_vbv_buffer_init =
68.         (float)avctx->rc_initial_buffer_occupancy / avctx->rc_buffer_size;
69. }
70.
71. OPT_STR("level", x4->level);
72.
73. if (avctx->i_quant_factor > 0)
74.     x4->params.rc.f_ip_factor = 1 / fabs(avctx->i_quant_factor);
75. if (avctx->b_quant_factor > 0)
76.     x4->params.rc.f_pb_factor = avctx->b_quant_factor;
77. if (avctx->chromaoffset)
78.     x4->params.analyse.i_chroma_qp_offset = avctx->chromaoffset;
79. //FFmpeg运动估计方法映射到Libx264
80. if (avctx->me_method == ME_EPZS)
81.     x4->params.analyse.i_me_method = X264_ME_DIA;
82. else if (avctx->me_method == ME_HEX)
83.     x4->params.analyse.i_me_method = X264_ME_HEX;
84. else if (avctx->me_method == ME_UHM)
85.     x4->params.analyse.i_me_method = X264_ME_UHM;
86. else if (avctx->me_method == ME_FULL)
87.     x4->params.analyse.i_me_method = X264_ME_ESA;
88. else if (avctx->me_method == ME_TESA)
89.     x4->params.analyse.i_me_method = X264_ME_TESA;
90.
91. //把AVCodecContext的值（主要是编码时候的一些通用选项）映射到x264_param_t
92. if (avctx->gop_size >= 0)
93.     x4->params.i_keyint_max = avctx->gop_size;
94. if (avctx->max_b_frames >= 0)
95.     x4->params.i_bframe = avctx->max_b_frames;
96. if (avctx->scenechange_threshold >= 0)
97.     x4->params.i_scenecut_threshold = avctx->scenechange_threshold;
98. if (avctx->qmin >= 0)
99.     x4->params.rc.i_qp_min = avctx->qmin;
100. if (avctx->qmax >= 0)
101.     x4->params.rc.i_qp_max = avctx->qmax;
102. if (avctx->max_qdiff >= 0)
103.     x4->params.rc.i_qp_step = avctx->max_qdiff;
104. if (avctx->qblur >= 0)
105.     x4->params.rc.f_qblur = avctx->qblur; /* temporally blur quant */
106. if (avctx->qcompress >= 0)
107.     x4->params.rc.f_qcompress = avctx->qcompress; /* 0.0 => cbr, 1.0 => constant qp */
108. if (avctx->refs >= 0)
109.     x4->params.i_frame_reference = avctx->refs;
110. else if (x4->level) {
111.     int i;
112.     int mbn = FF_CEIL_RSHIFT(avctx->width, 4) * FF_CEIL_RSHIFT(avctx->height, 4);
113.     int level_id = -1;
114.     char *tail;
115.     int scale = X264_BUILD < 129 ? 384 : 1;
116.
117.     if (!strcmp(x4->level, "1b")) {
118.         level_id = 9;
119.     } else if (strlen(x4->level) <= 3){
120.         level_id = av_strtod(x4->level, &tail) * 10 + 0.5;
121.         if (*tail)
122.             level_id = -1;
123.     }
124.     if (level_id <= 0)
125.         av_log(avctx, AV_LOG_WARNING, "Failed to parse level\n");
126.
127.     for (i = 0; i < x264_levels[i].level_idc; i++)
128.         if (x264_levels[i].level_idc == level_id)
129.             x4->params.i_frame_reference = av_clip(x264_levels[i].dpb / mbn / scale, 1, x4->params.i_frame_reference);
130. }
131.
132. if (avctx->trellis >= 0)
133.     x4->params.analyse.i_trellis = avctx->trellis;
134. if (avctx->me_range >= 0)
135.     x4->params.analyse.i_me_range = avctx->me_range;
136. if (avctx->noise_reduction >= 0)
137.     x4->params.analyse.i_noise_reduction = avctx->noise_reduction;
138. if (avctx->me_subpel_quality >= 0)
139.     x4->params.analyse.i_subpel_refine = avctx->me_subpel_quality;
140. if (avctx->b_frame_strategy >= 0)
141.     x4->params.i_bframe_adaptive = avctx->b_frame_strategy;
142. if (avctx->keyint_min >= 0)
143.     x4->params.i_keyint_min = avctx->keyint_min;
144. if (avctx->coder_type >= 0)

```

```

145.     x4->params.b_cabac = avctx->coder_type == FF_CODER_TYPE_AC;
146.     if (avctx->me_cmp >= 0)
147.         x4->params.analyse.b_chroma_me = avctx->me_cmp & FF_CMP_CHROMA;
148.
149.     //把X264Context中的信息（主要是针对于libx264的一些选项）映射到x264_param_t
150.     if (x4->aq_mode >= 0)
151.         x4->params.rc.i_aq_mode = x4->aq_mode;
152.     if (x4->aq_strength >= 0)
153.         x4->params.rc.f_aq_strength = x4->aq_strength;
154.     PARSE_X264_OPT("psy-rd", psy_rd);
155.     PARSE_X264_OPT("deblock", deblock);
156.     PARSE_X264_OPT("partitions", partitions);
157.     PARSE_X264_OPT("stats", stats);
158.     if (x4->psy >= 0)
159.         x4->params.analyse.b_psy = x4->psy;
160.     if (x4->rc_lookahead >= 0)
161.         x4->params.rc.i_lookahead = x4->rc_lookahead;
162.     if (x4->weightp >= 0)
163.         x4->params.analyse.i_weighted_pred = x4->weightp;
164.     if (x4->weightb >= 0)
165.         x4->params.analyse.b_weighted_bipred = x4->weightb;
166.     if (x4->cplxblur >= 0)
167.         x4->params.rc.f_complexity_blur = x4->cplxblur;
168.
169.     if (x4->ssim >= 0)
170.         x4->params.analyse.b_ssim = x4->ssim;
171.     if (x4->intra_refresh >= 0)
172.         x4->params.b_intra_refresh = x4->intra_refresh;
173.     if (x4->bluray_compat >= 0) {
174.         x4->params.b_bluray_compat = x4->bluray_compat;
175.         x4->params.b_vfr_input = 0;
176.     }
177.     if (x4->avcintra_class >= 0)
178. #if X264_BUILD >= 142
179.         x4->params.i_avcintra_class = x4->avcintra_class;
180. #else
181.         av_log(avctx, AV_LOG_ERROR,
182.              "x264 too old for AVC Intra, at least version 142 needed\n");
183. #endif
184.     if (x4->b_bias != INT_MIN)
185.         x4->params.i_bframe_bias = x4->b_bias;
186.     if (x4->b_pyramid >= 0)
187.         x4->params.i_bframe_pyramid = x4->b_pyramid;
188.     if (x4->mixed_refs >= 0)
189.         x4->params.analyse.b_mixed_references = x4->mixed_refs;
190.     if (x4->dct8x8 >= 0)
191.         x4->params.analyse.b_transform_8x8 = x4->dct8x8;
192.     if (x4->fast_pskip >= 0)
193.         x4->params.analyse.b_fast_pskip = x4->fast_pskip;
194.     if (x4->aud >= 0)
195.         x4->params.b_aud = x4->aud;
196.     if (x4->mbtree >= 0)
197.         x4->params.rc.b_mb_tree = x4->mbtree;
198.     if (x4->direct_pred >= 0)
199.         x4->params.analyse.i_direct_mv_pred = x4->direct_pred;
200.
201.     if (x4->slice_max_size >= 0)
202.         x4->params.i_slice_max_size = x4->slice_max_size;
203.     else {
204.         /*
205.          * Allow x264 to be instructed through AVCodecContext about the maximum
206.          * size of the RTP payload. For example, this enables the production of
207.          * payload suitable for the H.264 RTP packetization-mode 0 i.e. single
208.          * NAL unit per RTP packet.
209.          */
210.         if (avctx->rtp_payload_size)
211.             x4->params.i_slice_max_size = avctx->rtp_payload_size;
212.     }
213.
214.     if (x4->fastfirstpass)
215.         x264_param_apply_fastfirstpass(&x4->params);
216.
217.     /* Allow specifying the x264 profile through AVCodecContext. */
218.     //设置Profile
219.     if (!x4->profile)
220.         switch (avctx->profile) {
221.             case FF_PROFILE_H264_BASELINE:
222.                 x4->profile = av_strdup("baseline");
223.                 break;
224.             case FF_PROFILE_H264_HIGH:
225.                 x4->profile = av_strdup("high");
226.                 break;
227.             case FF_PROFILE_H264_HIGH_10:
228.                 x4->profile = av_strdup("high10");
229.                 break;
230.             case FF_PROFILE_H264_HIGH_422:
231.                 x4->profile = av_strdup("high422");
232.                 break;
233.             case FF_PROFILE_H264_HIGH_444:
234.                 x4->profile = av_strdup("high444");
235.                 break;

```

```

236.     case FF_PROFILE_H264_MAIN:
237.         x4->profile = av_strdup("main");
238.         break;
239.     default:
240.         break;
241.     }
242.
243.     if (x4->nal_hrd >= 0)
244.         x4->params.i_nal_hrd = x4->nal_hrd;
245.     //
246.     if (x4->profile)
247.         if (x264_param_apply_profile(&x4->params, x4->profile) < 0) {
248.             int i;
249.             av_log(avctx, AV_LOG_ERROR, "Error setting profile %s.\n", x4->profile);
250.             av_log(avctx, AV_LOG_INFO, "Possible profiles:");
251.             for (i = 0; x264_profile_names[i]; i++)
252.                 av_log(avctx, AV_LOG_INFO, " %s", x264_profile_names[i]);
253.             av_log(avctx, AV_LOG_INFO, "\n");
254.             return AVERROR(EINVAL);
255.         }
256.     //宽高, 帧率等
257.     x4->params.i_width = avctx->width;
258.     x4->params.i_height = avctx->height;
259.     av_reduce(&sw, &sh, avctx->sample_aspect_ratio.num, avctx->sample_aspect_ratio.den, 4096);
260.     x4->params.vui.i_sar_width = sw;
261.     x4->params.vui.i_sar_height = sh;
262.     x4->params.i_timebase_den = avctx->time_base.den;
263.     x4->params.i_timebase_num = avctx->time_base.num;
264.     x4->params.i_fps_num = avctx->time_base.den;
265.     x4->params.i_fps_den = avctx->time_base.num * avctx->ticks_per_frame;
266.
267.     x4->params.analyse.b_psnr = avctx->flags & CODEC_FLAG_PSNR;
268.
269.     x4->params.i_threads = avctx->thread_count;
270.     if (avctx->thread_type)
271.         x4->params.b_sliced_threads = avctx->thread_type == FF_THREAD_SLICE;
272.
273.     x4->params.b_interlaced = avctx->flags & CODEC_FLAG_INTERLACED_DCT;
274.
275.     x4->params.b_open_gop = !(avctx->flags & CODEC_FLAG_CLOSED_GOP);
276.
277.     x4->params.i_slice_count = avctx->slices;
278.
279.     x4->params.vui.b_fullrange = avctx->pix_fmt == AV_PIX_FMT_YUVJ420P ||
280.                                avctx->pix_fmt == AV_PIX_FMT_YUVJ422P ||
281.                                avctx->pix_fmt == AV_PIX_FMT_YUVJ444P ||
282.                                avctx->color_range == AVCOL_RANGE_JPEG;
283.
284.     if (avctx->colorspace != AVCOL_SPC_UNSPECIFIED)
285.         x4->params.vui.i_colmatrix = avctx->colorspace;
286.     if (avctx->color_primaries != AVCOL_PRI_UNSPECIFIED)
287.         x4->params.vui.i_colorprim = avctx->color_primaries;
288.     if (avctx->color_trc != AVCOL_TRC_UNSPECIFIED)
289.         x4->params.vui.i_transfer = avctx->color_trc;
290.
291.     if (avctx->flags & CODEC_FLAG_GLOBAL_HEADER)
292.         x4->params.b_repeat_headers = 0;
293.
294.     if (x4->x264opts){
295.         const char *p= x4->x264opts;
296.         while(p){
297.             char param[256]={0}, val[256]={0};
298.             if(sscanf(p, "%255[^\r\n]=%255[^\r\n]", param, val) == 1){
299.                 OPT_STR(param, "1");
300.             }else
301.                 OPT_STR(param, val);
302.             p= strchr(p, ':');
303.             p+=!!p;
304.         }
305.     }
306.
307.     if (x4->x264_params) {
308.         AVDictionary *dict = NULL;
309.         AVDictionaryEntry *en = NULL;
310.
311.         if (!av_dict_parse_string(&dict, x4->x264_params, "=", ":", 0)) {
312.             while ((en = av_dict_get(dict, "", en, AV_DICT_IGNORE_SUFFIX))) {
313.                 if (x264_param_parse(&x4->params, en->key, en->value) < 0)
314.                     av_log(avctx, AV_LOG_WARNING,
315.                             "Error parsing option '%s = %s'.\n",
316.                             en->key, en->value);
317.             }
318.
319.             av_dict_free(&dict);
320.         }
321.     }
322.
323.     // update AVCodecContext with x264 parameters
324.     avctx->has_b_frames = x4->params.i_bframe ?
325.         x4->params.i_bframe_pyramid ? 2 : 1 : 0;
326.     if (avctx->max_b_frames < 0)
327.         avctx->max_b_frames = 0;

```

```

327.         avctx->max_d_frames = 0;
328.
329.         avctx->bit_rate = x4->params.rc.i_bitrate*1000;
330.
331.         //-----
332.         //设置完参数后, 打开编码器
333.         x4->enc = x264_encoder_open(&x4->params);
334.         if (!x4->enc)
335.             return -1;
336.
337.         avctx->coded_frame = av_frame_alloc();
338.         if (!avctx->coded_frame)
339.             return AVERROR(ENOMEM);
340.         //如果需要全局头
341.         if (avctx->flags & CODEC_FLAG_GLOBAL_HEADER) {
342.             x264_nal_t *nal;
343.             uint8_t *p;
344.             int nnal, s, i;
345.
346.             s = x264_encoder_headers(x4->enc, &nal, &nnal);
347.             avctx->extradata = p = av_malloc(s);
348.
349.             for (i = 0; i < nnal; i++) {
350.                 /* Don't put the SEI in extradata. */
351.                 if (nal[i].i_type == NAL_SEI) {
352.                     av_log(avctx, AV_LOG_INFO, "%s\n", nal[i].p_payload+25);
353.                     x4->sei_size = nal[i].i_payload;
354.                     x4->sei = av_malloc(x4->sei_size);
355.                     memcpy(x4->sei, nal[i].p_payload, nal[i].i_payload);
356.                     continue;
357.                 }
358.                 memcpy(p, nal[i].p_payload, nal[i].i_payload);
359.                 p += nal[i].i_payload;
360.             }
361.             avctx->extradata_size = p - avctx->extradata;
362.         }
363.
364.         return 0;
365.     }

```

从源代码可以看出, X264_init() 主要将各种选项值传递给libx264。这些选项有两个来源: AVCodecContext和X264Context。AVCodecContext中包含了编码器的一些通用选项, 而X264Context包含了一些libx264特有的选项。在这里需要注意, FFmpeg中的一些选项的单位和libx264中对应选项的单位是不一样的, 因此需要做一些转换。例如像素格式的转换函数convert_pix_fmt()就是完成了这个功能。该函数的定义如下所示。

```

[cpp]
1. //映射FFmpeg和libx264的像素格式
2. static int convert_pix_fmt(enum AVPixelFormat pix_fmt)
3. {
4.     switch (pix_fmt) {
5.         case AV_PIX_FMT_YUV420P:
6.         case AV_PIX_FMT_YUVJ420P:
7.         case AV_PIX_FMT_YUV420P9:
8.         case AV_PIX_FMT_YUV420P10: return X264_CSP_I420;
9.         case AV_PIX_FMT_YUV422P:
10.        case AV_PIX_FMT_YUVJ422P:
11.        case AV_PIX_FMT_YUV422P10: return X264_CSP_I422;
12.        case AV_PIX_FMT_YUV444P:
13.        case AV_PIX_FMT_YUVJ444P:
14.        case AV_PIX_FMT_YUV444P9:
15.        case AV_PIX_FMT_YUV444P10: return X264_CSP_I444;
16.        #ifdef X264_CSP_BGR
17.        case AV_PIX_FMT_BGR24:
18.            return X264_CSP_BGR;
19.
20.        case AV_PIX_FMT_RGB24:
21.            return X264_CSP_RGB;
22.        #endif
23.        case AV_PIX_FMT_NV12: return X264_CSP_NV12;
24.        case AV_PIX_FMT_NV16:
25.        case AV_PIX_FMT_NV20: return X264_CSP_NV16;
26.    };
27.    return 0;
28. }

```

可以看出convert_pix_fmt()将AV_PIX_FMT_XXX转换成了X264_CSP_XXX。在一切参数设置完毕后, X264_init()会调用x264_encoder_open()打开编码器, 完成初始化工作。

X264_frame()

X264_frame()用于编码一帧视频数据。该函数的定义如下所示。

```

[cpp]
1. //libx264编码1帧数据
2. //
3. // AVFrame --> x264_picture_t --> x264_nal_t --> AVPacket

```



```

4. //
5. static int X264_frame(AVCodecContext *ctx, AVPacket *pkt, const AVFrame *frame,
6.                     int *got_packet)
7. {
8.     X264Context *x4 = ctx->priv_data;
9.     x264_nal_t *nal;
10.    int nnal, i, ret;
11.    x264_picture_t pic_out = {0};
12.    AVFrameSideData *side_data;
13.
14.    x264_picture_init( &x4->pic );
15.    x4->pic.img.i_csp = x4->params.i_csp;
16.    if (x264_bit_depth > 8)
17.        x4->pic.img.i_csp |= X264_CSP_HIGH_DEPTH;
18.    x4->pic.img.i_plane = avfmt2_num_planes(ctx->pix_fmt);
19.
20.    if (frame) {
21.        //将AVFrame中的数据赋值给x264_picture_t
22.        //
23.        // AVFrame --> x264_picture_t
24.        //
25.        for (i = 0; i < x4->pic.img.i_plane; i++) {
26.            x4->pic.img.plane[i] = frame->data[i];
27.            x4->pic.img.i_stride[i] = frame->linesize[i];
28.        }
29.
30.        x4->pic.i_pts = frame->pts;
31.        //设置帧类型
32.        x4->pic.i_type =
33.            frame->pict_type == AV_PICTURE_TYPE_I ? X264_TYPE_KEYFRAME :
34.            frame->pict_type == AV_PICTURE_TYPE_P ? X264_TYPE_P :
35.            frame->pict_type == AV_PICTURE_TYPE_B ? X264_TYPE_B :
36.            X264_TYPE_AUTO;
37.        //检查参数设置是否正确, 不正确就重新设置
38.        if (x4->avcintra_class < 0) {
39.            if (x4->params.b_interlaced && x4->params.b_tff != frame->top_field_first) {
40.                x4->params.b_tff = frame->top_field_first;
41.                x264_encoder_reconfig(x4->enc, &x4->params);
42.            }
43.            if (x4->params.vui.i_sar_height != ctx->sample_aspect_ratio.den ||
44.                x4->params.vui.i_sar_width != ctx->sample_aspect_ratio.num) {
45.                x4->params.vui.i_sar_height = ctx->sample_aspect_ratio.den;
46.                x4->params.vui.i_sar_width = ctx->sample_aspect_ratio.num;
47.                x264_encoder_reconfig(x4->enc, &x4->params);
48.            }
49.
50.            if (x4->params.rc.i_vbv_buffer_size != ctx->rc_buffer_size / 1000 ||
51.                x4->params.rc.i_vbv_max_bitrate != ctx->rc_max_rate / 1000) {
52.                x4->params.rc.i_vbv_buffer_size = ctx->rc_buffer_size / 1000;
53.                x4->params.rc.i_vbv_max_bitrate = ctx->rc_max_rate / 1000;
54.                x264_encoder_reconfig(x4->enc, &x4->params);
55.            }
56.
57.            if (x4->params.rc.i_rc_method == X264_RC_ABR &&
58.                x4->params.rc.i_bitrate != ctx->bit_rate / 1000) {
59.                x4->params.rc.i_bitrate = ctx->bit_rate / 1000;
60.                x264_encoder_reconfig(x4->enc, &x4->params);
61.            }
62.
63.            if (x4->crf >= 0 &&
64.                x4->params.rc.i_rc_method == X264_RC_CRF &&
65.                x4->params.rc.f_rf_constant != x4->crf) {
66.                x4->params.rc.f_rf_constant = x4->crf;
67.                x264_encoder_reconfig(x4->enc, &x4->params);
68.            }
69.
70.            if (x4->params.rc.i_rc_method == X264_RC_CQP &&
71.                x4->cqp >= 0 &&
72.                x4->params.rc.i_qp_constant != x4->cqp) {
73.                x4->params.rc.i_qp_constant = x4->cqp;
74.                x264_encoder_reconfig(x4->enc, &x4->params);
75.            }
76.
77.            if (x4->crf_max >= 0 &&
78.                x4->params.rc.f_rf_constant_max != x4->crf_max) {
79.                x4->params.rc.f_rf_constant_max = x4->crf_max;
80.                x264_encoder_reconfig(x4->enc, &x4->params);
81.            }
82.        }
83.
84.        side_data = av_frame_get_side_data(frame, AV_FRAME_DATA_STEREO3D);
85.        if (side_data) {
86.            AVStereo3D *stereo = (AVStereo3D *)side_data->data;
87.            int fpa_type;
88.
89.            switch (stereo->type) {
90.            case AV_STEREO3D_CHECKERBOARD:
91.                fpa_type = 0;
92.                break;
93.            case AV_STEREO3D_COLUMNS:
94.                fpa_type = 1;

```

```

95.         break;
96.     case AV_STEREO3D_LINES:
97.         fpa_type = 2;
98.         break;
99.     case AV_STEREO3D_SIDEBYSIDE:
100.        fpa_type = 3;
101.        break;
102.     case AV_STEREO3D_TOPBOTTOM:
103.        fpa_type = 4;
104.        break;
105.     case AV_STEREO3D_FRAMESEQUENCE:
106.        fpa_type = 5;
107.        break;
108.     default:
109.        fpa_type = -1;
110.        break;
111.    }
112.
113.    if (fpa_type != x4->params.i_frame_packing) {
114.        x4->params.i_frame_packing = fpa_type;
115.        x264_encoder_reconfig(x4->enc, &x4->params);
116.    }
117.    }
118.    }
119.    do {
120.        /*[libx264 API] 编码
121.        //
122.        // x264_picture_t --> x264_nal_t
123.        //
124.        if (x264_encoder_encode(x4->enc, &nal, &nnal, frame? &x4->pic: NULL, &pic_out) < 0)
125.            return -1;
126.
127.        //把x264_nal_t赋值给AVPacket
128.        //
129.        // x264_nal_t --> AVPacket
130.        //
131.        ret = encode_nals(ctx, pkt, nal, nnal);
132.        if (ret < 0)
133.            return -1;
134.    } while (!ret && !frame && x264_encoder_delayed_frames(x4->enc));
135.
136.    //赋值AVPacket相关的字段
137.    pkt->pts = pic_out.i_pts;
138.    pkt->dts = pic_out.i_dts;
139.
140.    switch (pic_out.i_type) {
141.    case X264_TYPE_IDR:
142.    case X264_TYPE_I:
143.        ctx->coded_frame->pict_type = AV_PICTURE_TYPE_I;
144.        break;
145.    case X264_TYPE_P:
146.        ctx->coded_frame->pict_type = AV_PICTURE_TYPE_P;
147.        break;
148.    case X264_TYPE_B:
149.    case X264_TYPE_BREF:
150.        ctx->coded_frame->pict_type = AV_PICTURE_TYPE_B;
151.        break;
152.    }
153.
154.    pkt->flags |= AV_PKT_FLAG_KEY*pic_out.b_keyframe;
155.    if (ret)
156.        ctx->coded_frame->quality = (pic_out.i_qpplus1 - 1) * FF_QP2LAMBDA;
157.
158.    *got_packet = ret;
159.    return 0;
160.    }

```

从源代码可以看出，X264_frame()调用x264_encoder_encode()完成了编码工作。x264_encoder_encode()的输入是x264_picture_t，输出是x264_nal_t；而X264_frame()的输入是AVFrame，输出是AVPacket。因此X264_frame()在调用编码函数前将AVFrame转换成了x264_picture_t，而在调用编码函数之后调用encode_nals()将x264_nal_t转换成了AVPacket。转换函数encode_nals()的定义如下所示。

```

1. //把x264_nal_t赋值给AVPacket
2. //
3. // x264_nal_t --> AVPacket
4. //
5. static int encode_nals(AVCodecContext *ctx, AVPacket *pkt,
6.                        const x264_nal_t *nals, int nnal)
7. {
8.     X264Context *x4 = ctx->priv_data;
9.     uint8_t *p;
10.    int i, size = x4->sei_size, ret;
11.
12.    if (!nnal)
13.        return 0;
14.    //NALU的大小
15.    //可能有多个NALU
16.    for (i = 0; i < nnal; i++)
17.        size += nals[i].i_payload;
18.
19.    if ((ret = ff_alloc_packet2(ctx, pkt, size)) < 0)
20.        return ret;
21.
22.    //p指向AVPacket的数据
23.    p = pkt->data;
24.
25.    /* Write the SEI as part of the first frame. */
26.    if (x4->sei_size > 0 && nnal > 0) {
27.        if (x4->sei_size > size) {
28.            av_log(ctx, AV_LOG_ERROR, "Error: nal buffer is too small\n");
29.            return -1;
30.        }
31.        memcpy(p, x4->sei, x4->sei_size);
32.        p += x4->sei_size;
33.        x4->sei_size = 0;
34.        av_freep(&x4->sei);
35.    }
36.    //拷贝x264_nal_t的数据至AVPacket的数据
37.    //可能有多个NALU
38.    for (i = 0; i < nnal; i++){
39.        memcpy(p, nals[i].p_payload, nals[i].i_payload);
40.        p += nals[i].i_payload;
41.    }
42.
43.    return 1;
44. }

```

从源代码可以看出，encode_nals()的作用就是将多个x264_nal_t合并为一个AVPacket。

X264_close()

X264_close()用于关闭libx264解码器。该函数的定义如下所示。

```

1. //libx264关闭解码器
2. static av_cold int X264_close(AVCodecContext *avctx)
3. {
4.     X264Context *x4 = avctx->priv_data;
5.
6.     av_freep(&avctx->extradata);
7.     av_freep(&x4->sei);
8.
9.     //[[libx264 API] 关闭解码器
10.    if (x4->enc)
11.        x264_encoder_close(x4->enc);
12.
13.    av_frame_free(&avctx->coded_frame);
14.
15.    return 0;
16. }

```

可以看出X264_close()调用x264_encoder_close()关闭了libx264编码器。

雷霄骅

leixiaohua1020@126.com

<http://blog.csdn.net/leixiaohua1020>

文章标签：[libx264](#) [FFmpeg](#) [H.264](#) [视频编码](#) [AVCodec](#)

个人分类：[x264](#) [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