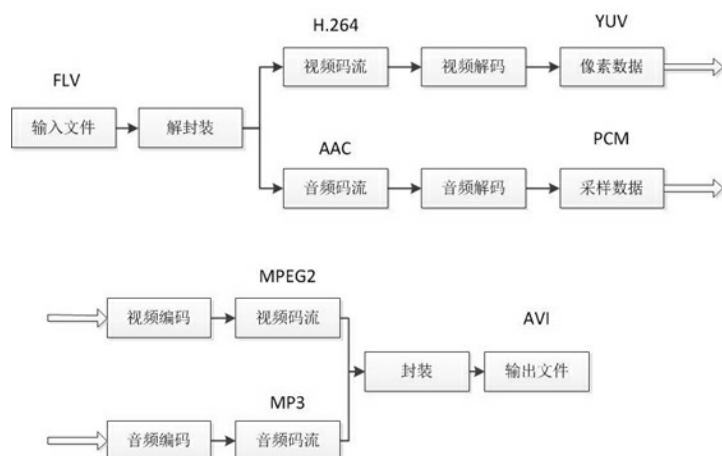


本文介绍一个简单的基于FFmpeg的转码器。它可以将一种视频格式（包括封装格式和编码格式）转换为另一种视频格式。转码器在视音频编解码处理的程序中，属于一个比较复杂的东西。因为它结合了视频的解码和编码。一个视频播放器，一般只包含解码功能；一个视频编码工具，一般只包含编码功能；而一个视频转码器，则需要先对视频进行解码，然后再对视频进行编码，因而相当于解码器和编码器的结合。下图例举了一个视频的转码流程。输入视频的封装格式是FLV，视频编码标准是H.264，音频编码标准是AAC；输出视频的封装格式是AVI，视频编码标准是MPEG2，音频编码标准是MP3。从流程中可以看出，首先从输入视频中分离出视频码流和音频压缩码流，然后分别将视频码流和音频码流进行解码，获取到非压缩的像素数据/音频采样数据，接着将非压缩的像素数据/音频采样数据重新进行编码，获得重新编码后的视频码流和音频码流，最后将视频码流和音频码流重新封装成一个文件。



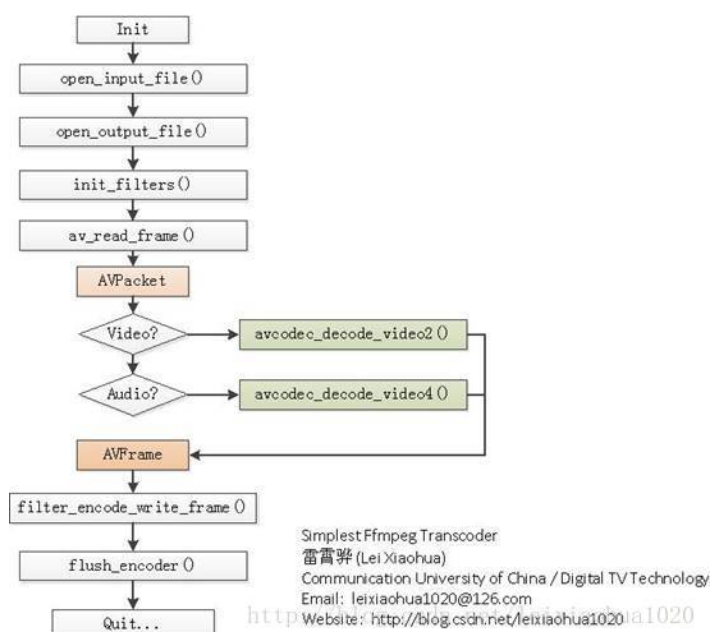
举例：FLV（视频：H.264，音频：AAC）转码为AVI（视频：MPEG2，音频MP3）

本文介绍的视频转码器正是使用FFMPEG类库从编程的角度实现了上述流程。该例子是从FFmpeg的例子改编的，平台是VC2010，类库版本是2014.5.6。

流程图（2014.9.29更新）

下面附两张使用FFmpeg转码视频的流程图。图中使用浅绿色标出了视频的编码、解码函数。从代码中可以看出，使用了AVFilter的不少东西，因此建议先学习AVFilter的内容后再看这个转码器的源代码。

PS：实际上，转码器不是一定依赖AVFilter的。因此打算有时间对这个转码器进行进一步的简化，使学习的人无需AVFilter的基础也可以理解转码器。



简单介绍一下流程中各个函数的意义：

open_input_file()：打开输入文件，并初始化相关的结构体。

open_output_file()：打开输出文件，并初始化相关的结构体。

init_filters()：初始化AVFilter相关的结构体。

av_read_frame(): 从输入文件中读取一个AVPacket。

avcodec_decode_video2(): 解码一个视频AVPacket (存储H.264等压缩码流数据) 为AVFrame (存储YUV等非压缩的像素数据)。

avcodec_decode_audio4(): 解码一个音频AVPacket (存储MP3等压缩码流数据) 为AVFrame (存储PCM采样数据)。

filter_encode_write_frame(): 编码一个AVFrame。

flush_encoder(): 输入文件读取完毕后, 输出编码器中剩余的AVPacket。

以上函数中open_input_file(), open_output_file(), init_filters()中的函数在其他文章中都有所叙述, 在这里不再重复:

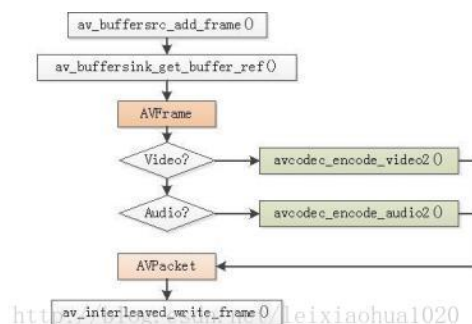
open_input_file()可参考: [100行代码实现最简单的基于FFmpeg+SDL的视频播放器 \(SDL1.x\)](#)

open_output_file()可参考: [最简单的基于FFmpeg的视频编码器 \(YUV编码为H.264\)](#)

init_filters()可参考: [最简单的基于FFmpeg的AVfilter例子 \(水印叠加\)](#)

在这里介绍一下其中编码的函数filter_encode_write_frame()。filter_encode_write_frame()函数的流程如下图所示, 它完成了视频/音频的编码功能。

PS: 视频和音频的编码流程中除了编码函数avcodec_encode_video2()和avcodec_encode_audio2()不一样之外, 其他部分几乎完全一样。



简单介绍一下filter_encode_write_frame()中各个函数的意义:

av_buffersrc_add_frame(): 将解码后的AVFrame加入Filtergraph。

av_buffersink_get_buffer_ref(): 从Filtergraph中取一个AVFrame。

avcodec_encode_video2(): 编码一个视频AVFrame为AVPacket。

avcodec_encode_audio2(): 编码一个音频AVFrame为AVPacket。

av_interleaved_write_frame(): 将编码后的AVPacket写入文件。

代码

贴上代码

```
[cpp]
1.  /*
2.  *最简单的基于FFmpeg的转码器
3.  *Simplest FFmpeg Transcoder
4.  *
5.  *雷霄骅 Lei Xiaohua
6.  *leixiaohua1020@126.com
7.  *中国传媒大学/数字电视技术
8.  *Communication University of China / DigitalTV Technology
9.  *http://blog.csdn.net/leixiaohua1020
10. *
11. *本程序实现了视频格式之间的转换。是一个最简单的视频转码程序。
12. *
13. */
14.
15. #include "stdafx.h"
16. extern "C"
17. {
18. #include "libavcodec/avcodec.h"
19. #include "libavformat/avformat.h"
20. #include "libavfilter/avfiltergraph.h"
21. #include "libavfilter/avcodec.h"
22. #include "libavfilter/buffersink.h"
23. #include "libavfilter/buffersrc.h"
24. #include "libavutil/avutil.h"
25. #include "libavutil/opt.h"
26. #include "libavutil/pixdesc.h"
27. };
28.
29.
```

```

30.
31. static AVFormatContext *ifmt_ctx;
32. static AVFormatContext *ofmt_ctx;
33. typedef struct FilteringContext{
34.     AVFilterContext*buffersink_ctx;
35.     AVFilterContext*buffer_src_ctx;
36.     AVFilterGraph*filter_graph;
37. } FilteringContext;
38. static FilteringContext *filter_ctx;
39. static int open_input_file(const char *filename)
40. {
41.     int ret;
42.     unsigned int i;
43.     ifmt_ctx = NULL;
44.     if ((ret = avformat_open_input(&ifmt_ctx, filename, NULL, NULL)) < 0) {
45.         av_log(NULL, AV_LOG_ERROR, "Cannot open input file\n");
46.         return ret;
47.     }
48.     if ((ret = avformat_find_stream_info(ifmt_ctx, NULL)) < 0) {
49.         av_log(NULL, AV_LOG_ERROR, "Cannot find stream information\n");
50.         return ret;
51.     }
52.     for (i = 0; i < ifmt_ctx->nb_streams; i++) {
53.         AVStream*stream;
54.         AVCodecContext *codec_ctx;
55.         stream = ifmt_ctx->streams[i];
56.         codec_ctx = stream->codec;
57.         /* Reencode video & audio and remux subtitles etc. */
58.         if (codec_ctx->codec_type == AVMEDIA_TYPE_VIDEO
59.             || codec_ctx->codec_type == AVMEDIA_TYPE_AUDIO) {
60.             /* Open decoder */
61.             ret = avcodec_open2(codec_ctx,
62.                               avcodec_find_decoder(codec_ctx->codec_id), NULL);
63.             if (ret < 0) {
64.                 av_log(NULL, AV_LOG_ERROR, "Failed to open decoder for stream #%u\n", i);
65.                 return ret;
66.             }
67.         }
68.     }
69.     av_dump_format(ifmt_ctx, 0, filename, 0);
70.     return 0;
71. }
72. static int open_output_file(const char *filename)
73. {
74.     AVStream*out_stream;
75.     AVStream*in_stream;
76.     AVCodecContext*dec_ctx, *enc_ctx;
77.     AVCodec*encoder;
78.     int ret;
79.     unsigned int i;
80.     ofmt_ctx = NULL;
81.     avformat_alloc_output_context2(&ofmt_ctx, NULL, NULL, filename);
82.     if (!ofmt_ctx) {
83.         av_log(NULL, AV_LOG_ERROR, "Could not create output context\n");
84.         return AVERROR_UNKNOWN;
85.     }
86.     for (i = 0; i < ifmt_ctx->nb_streams; i++) {
87.         out_stream = avformat_new_stream(ofmt_ctx, NULL);
88.         if (!out_stream) {
89.             av_log(NULL, AV_LOG_ERROR, "Failed allocating output stream\n");
90.             return AVERROR_UNKNOWN;
91.         }
92.         in_stream = ifmt_ctx->streams[i];
93.         dec_ctx = in_stream->codec;
94.         enc_ctx = out_stream->codec;
95.         if (dec_ctx->codec_type == AVMEDIA_TYPE_VIDEO
96.             || dec_ctx->codec_type == AVMEDIA_TYPE_AUDIO) {
97.             /* In this example, we choose transcoding to same codec */
98.             encoder = avcodec_find_encoder(dec_ctx->codec_id);
99.             /* In this example, we transcode to same properties (picture size,
100.              * sample rate etc.). These properties can be changed for output
101.              * streams easily using filters */
102.             if (dec_ctx->codec_type == AVMEDIA_TYPE_VIDEO) {
103.                 enc_ctx->height = dec_ctx->height;
104.                 enc_ctx->width = dec_ctx->width;
105.                 enc_ctx->sample_aspect_ratio = dec_ctx->sample_aspect_ratio;
106.                 /* take first format from list of supported formats */
107.                 enc_ctx->pix_fmt = encoder->pix_fmts[0];
108.                 /* video time_base can be set to whatever is handy and supported by encoder */
109.                 enc_ctx->time_base = dec_ctx->time_base;
110.             } else {
111.                 enc_ctx->sample_rate = dec_ctx->sample_rate;
112.                 enc_ctx->channel_layout = dec_ctx->channel_layout;
113.                 enc_ctx->channels = av_get_channel_layout_nb_channels(enc_ctx->channel_layout);
114.                 /* take first format from list of supported formats */
115.                 enc_ctx->sample_fmt = encoder->sample_fmts[0];
116.                 AVRational time_base = {1, enc_ctx->sample_rate};
117.                 enc_ctx->time_base = time_base;
118.             }
119.             /* Third parameter can be used to pass settings to encoder */
120.             ret = avcodec_open2(enc_ctx, encoder, NULL);

```

```

121.         if (ret < 0) {
122.             av_log(NULL, AV_LOG_ERROR, "Cannot openvideo encoder for stream #u\n", i);
123.             return ret;
124.         }
125.     } else if(dec_ctx->codec_type == AVMEDIA_TYPE_UNKNOWN) {
126.         av_log(NULL, AV_LOG_FATAL, "Elementarystream #d is of unknown type, cannot proceed\n", i);
127.         return AVERROR_INVALIDDATA;
128.     } else {
129.         /* if this stream must be remuxed */
130.         ret =avcodec_copy_context(ofmt_ctx->streams[i]->codec,
131.                                 ifmt_ctx->streams[i]->codec);
132.         if (ret < 0) {
133.             av_log(NULL, AV_LOG_ERROR, "Copyingstream context failed\n");
134.             return ret;
135.         }
136.     }
137.     if (ofmt_ctx->oformat->flags &AVFMT_GLOBALHEADER)
138.         enc_ctx->flags |= CODEC_FLAG_GLOBAL_HEADER;
139. }
140. av_dump_format(ofmt_ctx, 0, filename, 1);
141. if (!(ofmt_ctx->oformat->flags &AVFMT_NOFILE)) {
142.     ret =avio_open(&ofmt_ctx->pb, filename, AVIO_FLAG_WRITE);
143.     if (ret < 0) {
144.         av_log(NULL, AV_LOG_ERROR, "Could notopen output file '%s'", filename);
145.         return ret;
146.     }
147. }
148. /* init muxer, write output file header */
149. ret =avformat_write_header(ofmt_ctx, NULL);
150. if (ret < 0) {
151.     av_log(NULL,AV_LOG_ERROR, "Error occurred when openingoutput file\n");
152.     return ret;
153. }
154. return 0;
155. }
156. static intinit_filter(FilteringContext* fctx, AVCodecContext *dec_ctx,
157.                      AVCodecContext *enc_ctx, const char *filter_spec)
158. {
159.     char args[512];
160.     int ret = 0;
161.     AVFilter*buffer_src = NULL;
162.     AVFilter*buffer_sink = NULL;
163.     AVFilterContext*buffer_src_ctx = NULL;
164.     AVFilterContext*buffer_sink_ctx = NULL;
165.     AVFilterInOut*outputs = avfilter_inout_alloc();
166.     AVFilterInOut*inputs = avfilter_inout_alloc();
167.     AVFilterGraph*filter_graph = avfilter_graph_alloc();
168.     if (!outputs || !inputs || !filter_graph) {
169.         ret =AVERROR(ENOMEM);
170.         goto end;
171.     }
172.     if (dec_ctx->codec_type == AVMEDIA_TYPE_VIDEO) {
173.         buffer_src =avfilter_get_by_name("buffer");
174.         buffer_sink= avfilter_get_by_name("buffersink");
175.         if (!buffer_src || !buffer_sink) {
176.             av_log(NULL, AV_LOG_ERROR, "filteringsource or sink element not found\n");
177.             ret = AVERROR_UNKNOWN;
178.             goto end;
179.         }
180.         snprintf(args, sizeof(args),
181.                  "video_size=%dx%d:pix_fmt=%d:time_base=%d/%d:pixel_aspect=%d/%d",
182.                  dec_ctx->width, dec_ctx->height, dec_ctx->pix_fmt,
183.                  dec_ctx->time_base.num,dec_ctx->time_base.den,
184.                  dec_ctx->sample_aspect_ratio.num,
185.                  dec_ctx->sample_aspect_ratio.den);
186.         ret =avfilter_graph_create_filter(&buffer_src_ctx, buffer_src, "in",
187.                                           args, NULL, filter_graph);
188.         if (ret < 0) {
189.             av_log(NULL, AV_LOG_ERROR, "Cannotcreate buffer source\n");
190.             goto end;
191.         }
192.         ret =avfilter_graph_create_filter(&buffer_sink_ctx, buffer_sink, "out",
193.                                           NULL, NULL, filter_graph);
194.         if (ret < 0) {
195.             av_log(NULL, AV_LOG_ERROR, "Cannotcreate buffer sink\n");
196.             goto end;
197.         }
198.         ret =av_opt_set_bin(buffer_sink_ctx, "pix_fmts",
199.                             (uint8_t*)&enc_ctx->pix_fmt, sizeof(enc_ctx->pix_fmt),
200.                             AV_OPT_SEARCH_CHILDREN);
201.         if (ret < 0) {
202.             av_log(NULL, AV_LOG_ERROR, "Cannot setoutput pixel format\n");
203.             goto end;
204.         }
205.     } else if(dec_ctx->codec_type == AVMEDIA_TYPE_AUDIO) {
206.         buffer_src = avfilter_get_by_name("abuffer");
207.         buffer_sink= avfilter_get_by_name("abuffersink");
208.         if (!buffer_src || !buffer_sink) {
209.             av_log(NULL, AV_LOG_ERROR, "filteringsource or sink element not found\n");
210.             ret =AVERROR_UNKNOWN;
211.             goto end;

```

```

212.     }
213.     if (!dec_ctx->channel_layout)
214.         dec_ctx->channel_layout =
215.             av_get_default_channel_layout(dec_ctx->channels);
216.     _snprintf(args, sizeof(args),
217.         "time_base=%d/%d:sample_rate=%d:sample_fmt=%s:channel_layout=0x%I64x",
218.         dec_ctx->time_base.num, dec_ctx->time_base.den, dec_ctx->sample_rate,
219.         av_get_sample_fmt_name(dec_ctx->sample_fmt),
220.         dec_ctx->channel_layout);
221.     ret = avfilter_graph_create_filter(&buffersrc_ctx, buffersrc, "in",
222.         args, NULL, filter_graph);
223.     if (ret < 0) {
224.         av_log(NULL, AV_LOG_ERROR, "Cannot create audio buffer source\n");
225.         goto end;
226.     }
227.     ret = avfilter_graph_create_filter(&buffersink_ctx, buffersink, "out",
228.         NULL, NULL, filter_graph);
229.     if (ret < 0) {
230.         av_log(NULL, AV_LOG_ERROR, "Cannot create audio buffer sink\n");
231.         goto end;
232.     }
233.     ret = av_opt_set_bin(buffersink_ctx, "sample_fmts",
234.         (uint8_t*)&enc_ctx->sample_fmt, sizeof(enc_ctx->sample_fmt),
235.         AV_OPT_SEARCH_CHILDREN);
236.     if (ret < 0) {
237.         av_log(NULL, AV_LOG_ERROR, "Cannot set output sample format\n");
238.         goto end;
239.     }
240.     ret = av_opt_set_bin(buffersink_ctx, "channel_layouts",
241.         (uint8_t*)&enc_ctx->channel_layout,
242.         sizeof(enc_ctx->channel_layout), AV_OPT_SEARCH_CHILDREN);
243.     if (ret < 0) {
244.         av_log(NULL, AV_LOG_ERROR, "Cannot set output channel layout\n");
245.         goto end;
246.     }
247.     ret = av_opt_set_bin(buffersink_ctx, "sample_rates",
248.         (uint8_t*)&enc_ctx->sample_rate, sizeof(enc_ctx->sample_rate),
249.         AV_OPT_SEARCH_CHILDREN);
250.     if (ret < 0) {
251.         av_log(NULL, AV_LOG_ERROR, "Cannot set output sample rate\n");
252.         goto end;
253.     }
254. } else {
255.     ret = AERROR_UNKNOWN;
256.     goto end;
257. }
258. /* Endpoints for the filter graph. */
259. outputs->name = av_strdup("in");
260. outputs->filter_ctx = buffersrc_ctx;
261. outputs->pad_idx = 0;
262. outputs->next = NULL;
263. inputs->name = av_strdup("out");
264. inputs->filter_ctx = buffersink_ctx;
265. inputs->pad_idx = 0;
266. inputs->next = NULL;
267. if (!outputs->name || !inputs->name) {
268.     ret = AERROR_ENOMEM;
269.     goto end;
270. }
271. if ((ret = avfilter_graph_parse_ptr(filter_graph, filter_spec,
272.     &inputs, &outputs, NULL)) < 0)
273.     goto end;
274. if ((ret = avfilter_graph_config(filter_graph, NULL)) < 0)
275.     goto end;
276. /* Fill FilteringContext */
277. fctx->buffersrc_ctx = buffersrc_ctx;
278. fctx->buffersink_ctx = buffersink_ctx;
279. fctx->filter_graph = filter_graph;
280. end:
281. avfilter_inout_free(&inputs);
282. avfilter_inout_free(&outputs);
283. return ret;
284. }
285. static int init_filters(void)
286. {
287.     const char* filter_spec;
288.     unsigned int i;
289.     int ret;
290.     filter_ctx = (FilteringContext *)av_malloc_array(ifmt_ctx->nb_streams, sizeof(*filter_ctx));
291.     if (!filter_ctx)
292.         return AERROR_ENOMEM;
293.     for (i = 0; i < ifmt_ctx->nb_streams; i++) {
294.         filter_ctx[i].buffersrc_ctx = NULL;
295.         filter_ctx[i].buffersink_ctx = NULL;
296.         filter_ctx[i].filter_graph = NULL;
297.         if (!(ifmt_ctx->streams[i]->codec->codec_type == AVMEDIA_TYPE_AUDIO
298.             || ifmt_ctx->streams[i]->codec->codec_type == AVMEDIA_TYPE_VIDEO))
299.             continue;
300.         if (ifmt_ctx->streams[i]->codec->codec_type == AVMEDIA_TYPE_VIDEO)
301.             filter_spec = "null"; /* passthrough (dummy) filter for video */
302.         else
303.             filter_spec = "anull"; /* passthrough (dummy) filter for audio */

```

```

304.         filter_spec = NULL; // pass through (dummy) filter for audio /
305.         ret = init_filter(&filter_ctx[i], ifmt_ctx->streams[i]->codec,
306.             ofmt_ctx->streams[i]->codec, filter_spec);
307.         if (ret)
308.             return ret;
309.     }
310.     return 0;
311. }
312. static int encode_write_frame(AVFrame *filt_frame, unsigned int stream_index, int*got_frame) {
313.     int ret;
314.     int got_frame_local;
315.     AVPacket enc_pkt;
316.     int (*enc_func)(AVCodecContext *, AVPacket *, const AVFrame *, int*) =
317.         (ifmt_ctx->streams[stream_index]->codec->codec_type ==
318.          AVMEDIA_TYPE_VIDEO) ? avcodec_encode_video2 : avcodec_encode_audio2;
319.     if (!got_frame)
320.         got_frame = &got_frame_local;
321.     av_log(NULL, AV_LOG_INFO, "Encoding frame\n");
322.     /* encode filtered frame */
323.     enc_pkt.data = NULL;
324.     enc_pkt.size = 0;
325.     av_init_packet(&enc_pkt);
326.     ret = enc_func(ofmt_ctx->streams[stream_index]->codec, &enc_pkt,
327.         filt_frame, got_frame);
328.     av_frame_free(&filt_frame);
329.     if (ret < 0)
330.         return ret;
331.     if (!(*got_frame))
332.         return 0;
333.     /* prepare packet for muxing */
334.     enc_pkt.stream_index = stream_index;
335.     enc_pkt.dts = av_rescale_q_rnd(enc_pkt.dts,
336.         ofmt_ctx->streams[stream_index]->codec->time_base,
337.         ofmt_ctx->streams[stream_index]->time_base,
338.         (AVRounding)(AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
339.     enc_pkt.pts = av_rescale_q_rnd(enc_pkt.pts,
340.         ofmt_ctx->streams[stream_index]->codec->time_base,
341.         ofmt_ctx->streams[stream_index]->time_base,
342.         (AVRounding)(AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
343.     enc_pkt.duration = av_rescale_q(enc_pkt.duration,
344.         ofmt_ctx->streams[stream_index]->codec->time_base,
345.         ofmt_ctx->streams[stream_index]->time_base);
346.     av_log(NULL, AV_LOG_DEBUG, "Muxing frame\n");
347.     /* mux encoded frame */
348.     ret = av_interleaved_write_frame(ofmt_ctx, &enc_pkt);
349.     return ret;
350. }
351. static int filter_encode_write_frame(AVFrame *frame, unsigned int stream_index)
352. {
353.     int ret;
354.     AVFrame *filt_frame;
355.     av_log(NULL, AV_LOG_INFO, "Pushing decoded frame to filters\n");
356.     /* push the decoded frame into the filtergraph */
357.     ret = av_buffersrc_add_frame_flags(filter_ctx[stream_index].buffersrc_ctx,
358.         frame, 0);
359.     if (ret < 0) {
360.         av_log(NULL, AV_LOG_ERROR, "Error while feeding the filtergraph\n");
361.         return ret;
362.     }
363.     /* pull filtered frames from the filtergraph */
364.     while (1) {
365.         filt_frame = av_frame_alloc();
366.         if (!filt_frame) {
367.             ret = AVERROR(ENOMEM);
368.             break;
369.         }
370.         av_log(NULL, AV_LOG_INFO, "Pulling filtered frame from filters\n");
371.         ret = av_buffersink_get_frame(filter_ctx[stream_index].buffersink_ctx,
372.             filt_frame);
373.         if (ret < 0) {
374.             /* if no more frames for output - returns AVERROR(EAGAIN)
375.              * if flushed and no more frames for output - returns AVERROR_EOF
376.              * rewrite retcode to 0 to show it as normal procedure completion
377.              */
378.             if (ret == AVERROR(EAGAIN) || ret == AVERROR_EOF)
379.                 ret = 0;
380.             av_frame_free(&filt_frame);
381.             break;
382.         }
383.         filt_frame->pict_type = AV_PICTURE_TYPE_NONE;
384.         ret = encode_write_frame(filt_frame, stream_index, NULL);
385.         if (ret < 0)
386.             break;
387.     }
388.     return ret;
389. }
390. static int flush_encoder(unsigned int stream_index)
391. {
392.     int ret;
393.     int got_frame;
394.     if (!(ofmt_ctx->streams[stream_index]->codec->capabilities &
        CODEC_CAP_DELAY))

```

```

395.     return 0;
396. while (1) {
397.     av_log(NULL, AV_LOG_INFO, "Flushingstream #%u encoder\n", stream_index);
398.     ret = encode_write_frame(NULL, stream_index, &got_frame);
399.     if (ret < 0)
400.         break;
401.     if (!got_frame)
402.         return 0;
403. }
404. return ret;
405. }
406.
407. int_tmain(int argc, _TCHAR* argv[])
408. {
409.     int ret;
410.     AVPacket packet;
411.     AVFrame *frame= NULL;
412.     enum AVMediaType type;
413.     unsigned int stream_index;
414.     unsigned int i;
415.     int got_frame;
416.     int (*dec_func)(AVCodecContext *, AVFrame *, int *, const AVPacket*);
417.     if (argc != 3) {
418.         av_log(NULL, AV_LOG_ERROR, "Usage: %s<input file> <output file>\n", argv[0]);
419.         return 1;
420.     }
421.     av_register_all();
422.     avfilter_register_all();
423.     if ((ret = open_input_file(argv[1])) < 0)
424.         goto end;
425.     if ((ret = open_output_file(argv[2])) < 0)
426.         goto end;
427.     if ((ret = init_filters()) < 0)
428.         goto end;
429.     /* read all packets */
430.     while (1) {
431.         if ((ret= av_read_frame(ifmt_ctx, &packet)) < 0)
432.             break;
433.         stream_index = packet.stream_index;
434.         type = ifmt_ctx->streams[stream_index]->codec->codec_type;
435.         av_log(NULL, AV_LOG_DEBUG, "Demuxergave frame of stream_index %u\n",
436.             stream_index);
437.         if (filter_ctx[stream_index].filter_graph) {
438.             av_log(NULL, AV_LOG_DEBUG, "Going toreencode&filter the frame\n");
439.             frame = av_frame_alloc();
440.             if (!frame) {
441.                 ret = AVERROR(ENOMEM);
442.                 break;
443.             }
444.             packet.dts = av_rescale_q_rnd(packet.dts,
445.                 ifmt_ctx->streams[stream_index]->time_base,
446.                 ifmt_ctx->streams[stream_index]->codec->time_base,
447.                 (AVRounding)(AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
448.             packet.pts = av_rescale_q_rnd(packet.pts,
449.                 ifmt_ctx->streams[stream_index]->time_base,
450.                 ifmt_ctx->streams[stream_index]->codec->time_base,
451.                 (AVRounding)(AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
452.             dec_func = (type == AVMEDIA_TYPE_VIDEO) ? avcodec_decode_video2 :
453.                 avcodec_decode_audio4;
454.             ret = dec_func(ifmt_ctx->streams[stream_index]->codec, frame,
455.                 &got_frame, &packet);
456.             if (ret < 0) {
457.                 av_frame_free(&frame);
458.                 av_log(NULL, AV_LOG_ERROR, "Decodingfailed\n");
459.                 break;
460.             }
461.             if (got_frame) {
462.                 frame->pts = av_frame_get_best_effort_timestamp(frame);
463.                 ret = filter_encode_write_frame(frame, stream_index);
464.                 av_frame_free(&frame);
465.                 if (ret < 0)
466.                     goto end;
467.             } else {
468.                 av_frame_free(&frame);
469.             }
470.         } else {
471.             /* remux this frame without reencoding */
472.             packet.dts = av_rescale_q_rnd(packet.dts,
473.                 ifmt_ctx->streams[stream_index]->time_base,
474.                 ofmt_ctx->streams[stream_index]->time_base,
475.                 (AVRounding)(AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
476.             packet.pts = av_rescale_q_rnd(packet.pts,
477.                 ifmt_ctx->streams[stream_index]->time_base,
478.                 ofmt_ctx->streams[stream_index]->time_base,
479.                 (AVRounding)(AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
480.             ret = av_interleaved_write_frame(ofmt_ctx, &packet);
481.             if (ret < 0)
482.                 goto end;
483.         }
484.         av_free_packet(&packet);
485.     }

```



```

486.     /* flush filters and encoders */
487.     for (i = 0; i < ifmt_ctx->nb_streams; i++) {
488.         /* flush filter */
489.         if (!filter_ctx[i].filter_graph)
490.             continue;
491.         ret = filter_encode_write_frame(NULL, i);
492.         if (ret < 0) {
493.             av_log(NULL, AV_LOG_ERROR, "Flushing filter failed\n");
494.             goto end;
495.         }
496.         /* flush encoder */
497.         ret = flush_encoder(i);
498.         if (ret < 0) {
499.             av_log(NULL, AV_LOG_ERROR, "Flushing encoder failed\n");
500.             goto end;
501.         }
502.     }
503.     av_write_trailer(ofmt_ctx);
504. end:
505.     av_free_packet(&packet);
506.     av_frame_free(&frame);
507.     for (i = 0; i < ifmt_ctx->nb_streams; i++) {
508.         avcodec_close(ifmt_ctx->streams[i]->codec);
509.         if (ofmt_ctx && ofmt_ctx->nb_streams > i && ofmt_ctx->streams[i] && ofmt_ctx->streams[i]->codec)
510.             avcodec_close(ofmt_ctx->streams[i]->codec);
511.         if (filter_ctx && filter_ctx[i].filter_graph)
512.             avfilter_graph_free(&filter_ctx[i].filter_graph);
513.     }
514.     av_free(filter_ctx);
515.     avformat_close_input(&ifmt_ctx);
516.     if (ofmt_ctx && !(ofmt_ctx->oformat->flags & AVFMT_NOFILE))
517.         avio_close(ofmt_ctx->pb);
518.     avformat_free_context(ofmt_ctx);
519.     if (ret < 0)
520.         av_log(NULL, AV_LOG_ERROR, "Error occurred\n");
521.     return (ret? 1:0);
522. }

```

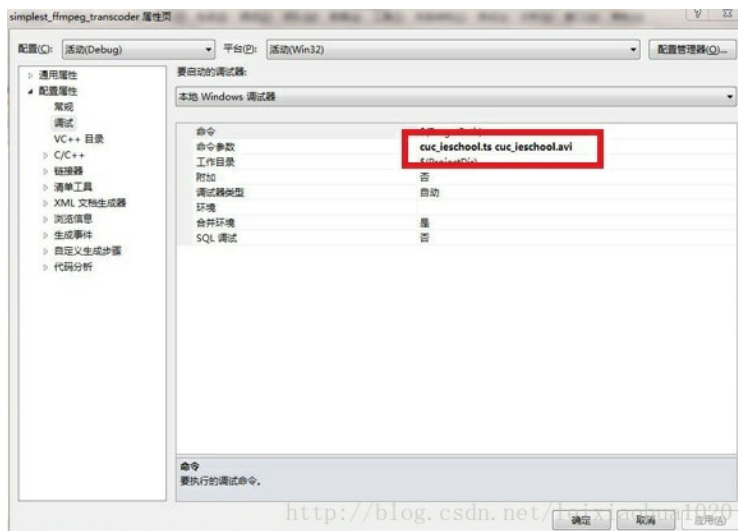
程序运行截图：

```

D:\cdtworkspace\simplest_ffmpeg_transcoder\Debug\simplest_ffmpeg_transcoder.exe
Input #00, mpegts, from 'cuc_ieschool.ts':
  Duration: 00:00:34.01, start: 1.455756, bitrate: 402 kb/s
  Program 1
  Metadata:
    service_name       : Service01
    service_provider   : FFmpeg
  Stream #0:0[0x100]: Video: mpeg2video (Main) (21)[0][0][0] / 0x00002, yuv420p(tv), 512x288 [SAR 1:1 DAR 16:9], max. 104857 kb/s, 15 fps, 15 tbr, 90k tbn, 30 tbc
  Stream #0:1[0x101]: Audio: mp2 (13)[0][0][0] / 0x00003, 44100 Hz, stereo, s16p, 125 kb/s
[mpeg2video @ 004453e0] removing common factors from framerate
Output #00, avi, to 'cuc_ieschool.avi':
  Stream #0:0: Video: mpeg2video, yuv420p, 512x288 [SAR 1:1 DAR 16:9], q=2-31, 128 kb/s, 90k tbn, 30 tbc
  Stream #0:1: Audio: mp2, 44100 Hz, stereo, s16, 128 kb/s
Pushing decoded frame to filters
Pulling filtered frame from filters
Encoding frame
Pulling filtered frame from filters
Pushing decoded frame to filters
Pulling filtered frame from filters
Encoding frame
Pulling filtered frame from filters
Pushing decoded frame to filters
Pulling filtered frame from filters
Encoding frame
Pulling filtered frame from filters
Pushing decoded frame to filters
Pulling filtered frame from filters

```

默认情况下运行程序，会将“cuc_ieschool.ts”转换为“cuc_ieschool.avi”。调试的时候，可以修改“配置属性->调试->命令参数”中的参数，即可改变转码的输入输出文件。



工程下载地址（VC2010）：<http://download.csdn.net/detail/leixiaohua1020/7394649>

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/26838535>

文章标签：[ffmpeg](#) [视频](#) [音频](#) [转码](#) [VC](#)

个人分类：[FFMPEG](#)

所属专栏：[FFmpeg](#)

此PDF由[spyyg](#)生成,请尊重原作者版权!!!

我的邮箱:liushidc@163.com