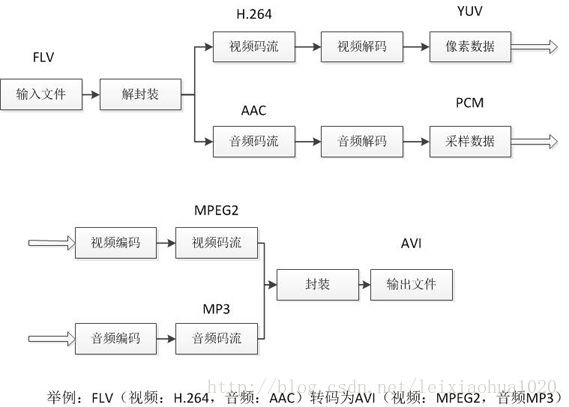
本文介绍一个简单的基于FFmpeg的转码器。它可以将一种视频格式（包括封转格式和编码格式）转换为另一种视频格式。转码器在视音频编解码处理的程序中，属于一个比较复杂的东西。因为它结合了视频的解码和编码。一个视频播放器，一般只包含解码功能；一个视频编码工具，一般只包含编码功能；而一个视频转码器，则需要先对视频进行解码，然后再对视频进行编码，因而相当于解码器和编码器的结合。下图例举了一个视频的转码流程。输入视频的封装格式是FLV，视频编码标准是H.264，音频编码标准是AAC；输出视频的封装格式是AVI，视频编码标准是MPEG2，音频编码标准是MP3。从流程中可以看出，首先从输入视频中分离出视频码流和音频压缩码流，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。

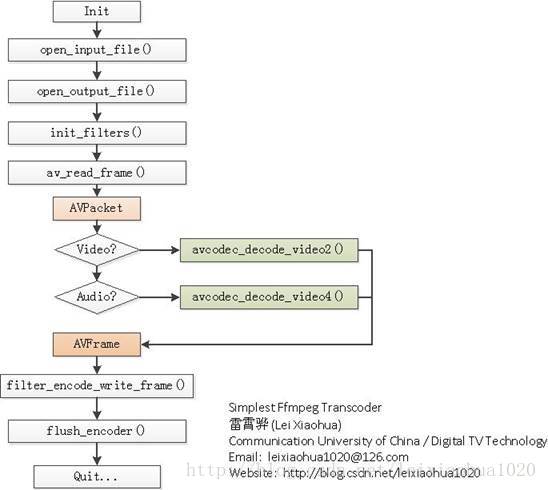


本文介绍的视频转码器正是使用FFMPEG类库从编程的角度实现了上述流程。该例子是从FFmpeg的例子改编的，平台是VC2010，类库版本是2014.5.6。

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\_video4()：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：

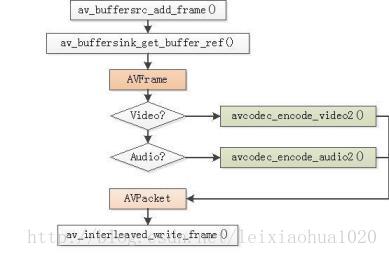
open\_input\_file()可参考：[100行代码实现最简单的基于FFMPEG+SDL的视频播放器（SDL1.x）](http://blog.csdn.net/leixiaohua1020/article/details/8652605)

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）](http://blog.csdn.net/leixiaohua1020/article/details/25430425)

init\_filters()可参考：[最简单的基于FFmpeg的AVfilter例子（水印叠加）](http://blog.csdn.net/leixiaohua1020/article/details/29368911)

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]** [view plain](http://blog.csdn.net/leixiaohua1020/article/details/26838535#) [copy](http://blog.csdn.net/leixiaohua1020/article/details/26838535#)

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\*buffersrc\_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 openinput 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 findstream 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 toopen 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 notcreate 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, "Failedallocating 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 andsupported 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.                 AVRationaltime\_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\*buffersrc = NULL;

162.     AVFilter\*buffersink = NULL;

163.     AVFilterContext\*buffersrc\_ctx = NULL;

164.     AVFilterContext\*buffersink\_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.         buffersrc =avfilter\_get\_by\_name("buffer");

174.         buffersink= avfilter\_get\_by\_name("buffersink");

175.         **if** (!buffersrc || !buffersink) {

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(&buffersrc\_ctx, buffersrc, "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(&buffersink\_ctx, buffersink, "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(buffersink\_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.         buffersrc = avfilter\_get\_by\_name("abuffer");

207.         buffersink= avfilter\_get\_by\_name("abuffersink");

208.         **if** (!buffersrc || !buffersink) {

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, "Cannotcreate 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, "Cannotcreate 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 setoutput 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 setoutput 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 setoutput sample rate\n");

252.             **goto** end;

253.         }

254.     } **else** {

255.         ret =AVERROR\_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 =AVERROR(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** AVERROR(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.         ret =init\_filter(&filter\_ctx[i], ifmt\_ctx->streams[i]->codec,

305.                ofmt\_ctx->streams[i]->codec, filter\_spec);

306.         **if** (ret)

307.             **return** ret;

308.     }

309.     **return** 0;

310. }

**311. static** intencode\_write\_frame(AVFrame \*filt\_frame, unsignedint stream\_index, **int**\*got\_frame) {

312.     **int** ret;

313.     **int** got\_frame\_local;

314.     AVPacketenc\_pkt;

315.     **int** (\*enc\_func)(AVCodecContext \*, AVPacket \*, **const** AVFrame \*, **int**\*) =

316.        (ifmt\_ctx->streams[stream\_index]->codec->codec\_type ==

317.         AVMEDIA\_TYPE\_VIDEO) ? avcodec\_encode\_video2 : avcodec\_encode\_audio2;

318.     **if** (!got\_frame)

319.         got\_frame =&got\_frame\_local;

320.     av\_log(NULL,AV\_LOG\_INFO, "Encoding frame\n");

321.     /\* encode filtered frame \*/

322.     enc\_pkt.data =NULL;

323.     enc\_pkt.size =0;

324.     av\_init\_packet(&enc\_pkt);

325.     ret =enc\_func(ofmt\_ctx->streams[stream\_index]->codec, &enc\_pkt,

326.            filt\_frame, got\_frame);

327.    av\_frame\_free(&filt\_frame);

328.     **if** (ret < 0)

329.         **return** ret;

330.     **if** (!(\*got\_frame))

331.         **return** 0;

332.     /\* prepare packet for muxing \*/

333.    enc\_pkt.stream\_index = stream\_index;

334.     enc\_pkt.dts =av\_rescale\_q\_rnd(enc\_pkt.dts,

335.            ofmt\_ctx->streams[stream\_index]->codec->time\_base,

336.            ofmt\_ctx->streams[stream\_index]->time\_base,

337.            (AVRounding)(AV\_ROUND\_NEAR\_INF|AV\_ROUND\_PASS\_MINMAX));

338.     enc\_pkt.pts =av\_rescale\_q\_rnd(enc\_pkt.pts,

339.            ofmt\_ctx->streams[stream\_index]->codec->time\_base,

340.            ofmt\_ctx->streams[stream\_index]->time\_base,

341.            (AVRounding)(AV\_ROUND\_NEAR\_INF|AV\_ROUND\_PASS\_MINMAX));

342.    enc\_pkt.duration = av\_rescale\_q(enc\_pkt.duration,

343.            ofmt\_ctx->streams[stream\_index]->codec->time\_base,

344.            ofmt\_ctx->streams[stream\_index]->time\_base);

345.     av\_log(NULL,AV\_LOG\_DEBUG, "Muxing frame\n");

346.     /\* mux encoded frame \*/

347.     ret =av\_interleaved\_write\_frame(ofmt\_ctx, &enc\_pkt);

348.     **return** ret;

349. }

**350. static** intfilter\_encode\_write\_frame(AVFrame \*frame, unsignedint stream\_index)

351. {

352.     **int** ret;

353.     AVFrame\*filt\_frame;

354.     av\_log(NULL,AV\_LOG\_INFO, "Pushing decoded frame tofilters\n");

355.     /\* push the decoded frame into the filtergraph \*/

356.     ret =av\_buffersrc\_add\_frame\_flags(filter\_ctx[stream\_index].buffersrc\_ctx,

357.             frame,0);

358.     **if** (ret < 0) {

359.        av\_log(NULL, AV\_LOG\_ERROR, "Error whilefeeding the filtergraph\n");

360.         **return** ret;

361.     }

362.     /\* pull filtered frames from the filtergraph \*/

363.     **while** (1) {

364.         filt\_frame= av\_frame\_alloc();

365.         **if** (!filt\_frame) {

366.             ret =AVERROR(ENOMEM);

367.             **break**;

368.         }

369.        av\_log(NULL, AV\_LOG\_INFO, "Pullingfiltered frame from filters\n");

370.         ret =av\_buffersink\_get\_frame(filter\_ctx[stream\_index].buffersink\_ctx,

371.                filt\_frame);

372.         **if** (ret < 0) {

373.             /\* if nomore frames for output - returns AVERROR(EAGAIN)

374.             \* if flushed and no more frames for output - returns AVERROR\_EOF

375.             \* rewrite retcode to 0 to show it as normal procedure completion

376.             \*/

377.             **if** (ret == AVERROR(EAGAIN) || ret == AVERROR\_EOF)

378.                 ret= 0;

379.            av\_frame\_free(&filt\_frame);

380.             **break**;

381.         }

382.        filt\_frame->pict\_type = AV\_PICTURE\_TYPE\_NONE;

383.         ret =encode\_write\_frame(filt\_frame, stream\_index, NULL);

384.         **if** (ret < 0)

385.             **break**;

386.     }

387.     **return** ret;

388. }

**389. static** **int** flush\_encoder(unsigned intstream\_index)

390. {

391.     **int** ret;

392.     **int** got\_frame;

393.     **if**(!(ofmt\_ctx->streams[stream\_index]->codec->codec->capabilities&

394.                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.     AVPacketpacket;

411.     AVFrame \*frame= NULL;

412.     **enum** AVMediaType type;

413.     unsigned intstream\_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[packet.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, "Flushingfilter 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, "Flushingencoder 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);

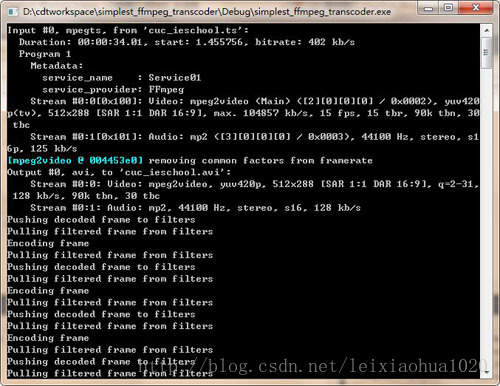
519.     **if** (ret < 0)

520.        av\_log(NULL, AV\_LOG\_ERROR, "Erroroccurred\n");

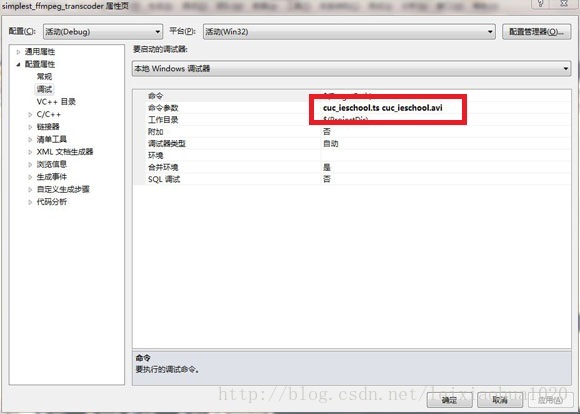
521.     **return** (ret? 1:0);

522. }

程序运行截图：



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>