原 最简单的基于FFmpeg的内存读写的例子：内存转码器

2014年10月05日 13:30:35　阅读数：18391

==========

最简单的基于FFmpeg的内存读写的例子系列文章列表：

最简单的基于FFmpeg的内存读写的例子：内存播放器

最简单的基于FFmpeg的内存读写的例子：内存转码器

==========

上篇文章记录了一个基于FFmpeg的内存播放器，可以使用FFmpeg读取并播放内存中的数据。这篇文章记录一个基于FFmpeg的内存转码器。该转码器可以使用FFmpeg读取内存中的数据，转码为H.264之后再将数据输出到内存。

关于如何从内存读取数据，以及如何将数据输出到内存，可以参考文章：
ffmpeg 从内存中读取数据(或将数据输出到内存)

FFmpeg读写内存的关键点有2个：
1. 初始化自定义的AVIOContext，指定自定义的回调函数。
2. 自己写回调函数。注意函数的参数和返回值（尤其是返回值）。

转码实际上就是解码和编码的结合。该方面的知识可以参考文章：
解码：100行代码实现最简单的基于FFMPEG+SDL的视频播放器（SDL1.x）

编码：最简单的基于FFMPEG的视频编码器（YUV编码为H.264）

转码：最简单的基于FFMPEG的转码程序

流程

程序的流程图如下图所示。从图中可以看出，首先分别初始化了输入和输出的AVFormatContext。然后首先解码输入的AVPacket，得到存储像素数据（YUV420P格式）的AVFrame，然后编码AVFrame为H.264的AVPacket，最后将编码后的AVPacket输出。

代码

下面直接贴上代码：

```cpp
/**
 * 最简单的基于FFmpeg的内存读写例子（内存转码器）
 * Simplest FFmpeg mem Transcoder
 *
 * 雷霄骅，张晖
 * leixiaohua1020@126.com
 * 中国传媒大学/数字电视技术
 * Communication University of China / Digital TV Technology
 * http://blog.csdn.net/leixiaohua1020
 *
 * 本程序实现了任意格式视频数据（例如MPEG2）转码为H.264码流数据。
 * 本程序并不是对文件进行处理，而是对内存中的视频数据进行处理。
 * 它从内存读取数据，并且将转码后的数据输出到内存中。
 * 是最简单的使用FFmpeg读写内存的例子。
 *
 * This software convert video bitstream (Such as MPEG2) to H.264
 * bitstream. It read video bitstream from memory (not from a file),
 * convert it to H.264 bitstream, and finally output to another memory.
 * It's the simplest example to use FFmpeg to read (or write) from
 * memory.
 *
 */
#include <stdio.h>

extern "C"
{
#include "libavcodec/avcodec.h"
#include "libavformat/avformat.h"
#include "libavutil/avutil.h"
#include "libavutil/opt.h"
#include "libavutil/pixdesc.h"
};

FILE *f
```

```
FILE *fp_open;
FILE *fp_write;

//Read File
int read_buffer(void *opaque, uint8_t *buf, int buf_size){
    if(!feof(fp_open)){
        int true_size=fread(buf,1,buf_size,fp_open);
        return true_size;
    }else{
        return -1;
    }
}

//Write File
int write_buffer(void *opaque, uint8_t *buf, int buf_size){
    if(!feof(fp_write)){
        int true_size=fwrite(buf,1,buf_size,fp_write);
        return true_size;
    }else{
        return -1;
    }
}



int flush_encoder(AVFormatContext *fmt_ctx,unsigned int stream_index)
{
    int ret;
    int got_frame;
    AVPacket enc_pkt;
    if (!(fmt_ctx->streams[stream_index]->codec->codec->capabilities &
                CODEC_CAP_DELAY))
        return 0;
    while (1) {
        av_log(NULL, AV_LOG_INFO, "Flushing stream #%u encoder\n", stream_index);
        //ret = encode_write_frame(NULL, stream_index, &got_frame);
        enc_pkt.data = NULL;
        enc_pkt.size = 0;
        av_init_packet(&enc_pkt);
        ret = avcodec_encode_video2 (fmt_ctx->streams[stream_index]->codec, &enc_pkt,
                NULL, &got_frame);
        av_frame_free(NULL);
        if (ret < 0)
            break;
        if (!got_frame)
        {ret=0;break;}
        /* prepare packet for muxing */
        enc_pkt.stream_index = stream_index;
        enc_pkt.dts = av_rescale_q_rnd(enc_pkt.dts,
                fmt_ctx->streams[stream_index]->codec->time_base,
                fmt_ctx->streams[stream_index]->time_base,
                (AVRounding)(AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
        enc_pkt.pts = av_rescale_q_rnd(enc_pkt.pts,
                fmt_ctx->streams[stream_index]->codec->time_base,
                fmt_ctx->streams[stream_index]->time_base,
                (AVRounding)(AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
        enc_pkt.duration = av_rescale_q(enc_pkt.duration,
                fmt_ctx->streams[stream_index]->codec->time_base,
                fmt_ctx->streams[stream_index]->time_base);
        av_log(NULL, AV_LOG_DEBUG, "Muxing frame\n");
        /* mux encoded frame */
        ret = av_write_frame(fmt_ctx, &enc_pkt);
        if (ret < 0)
            break;
    }
    return ret;
}


int main(int argc, char* argv[])
{
    int ret;
    AVFormatContext* ifmt_ctx=NULL;
    AVFormatContext* ofmt_ctx=NULL;
    AVPacket packet,enc_pkt;
    AVFrame *frame = NULL;
    enum AVMediaType type;
    unsigned int stream_index;
    unsigned int i=0;
    int got_frame,enc_got_frame;

    AVStream *out_stream;
    AVStream *in_stream;
    AVCodecContext *dec_ctx, *enc_ctx;
    AVCodec *encoder;

    fp_open = fopen("cuc60anniversary_start.ts", "rb"); //视频源文件
    fp_write=fopen("cuc60anniversary_start.h264","wb+"); //输出文件

    av_register_all();
    ifmt_ctx=avformat_alloc_context();
    avformat_alloc_output_context2(&ofmt_ctx, NULL, "h264", NULL);
```

```
    avformat_alloc_output_context2(&ofmt_ctx, NULL, "h264", NULL);

    unsigned char* inbuffer=NULL;
    unsigned char* outbuffer=NULL;
    inbuffer=(unsigned char*)av_malloc(32768);
    outbuffer=(unsigned char*)av_malloc(32768);

    /*open input file*/
    AVIOContext *avio_in =avio_alloc_context(inbuffer, 32768,0,NULL,read_buffer,NULL,NULL);
    if(avio_in==NULL)
        goto end;
    ifmt_ctx->pb=avio_in;
    ifmt_ctx->flags=AVFMT_FLAG_CUSTOM_IO;
    if ((ret = avformat_open_input(&ifmt_ctx, "whatever", NULL, NULL)) < 0) {
        av_log(NULL, AV_LOG_ERROR, "Cannot open input file\n");
        return ret;
    }
    if ((ret = avformat_find_stream_info(ifmt_ctx, NULL)) < 0) {
        av_log(NULL, AV_LOG_ERROR, "Cannot find stream information\n");
        return ret;
    }
    for (i = 0; i < ifmt_ctx->nb_streams; i++) {
        AVStream *stream;
        AVCodecContext *codec_ctx;
        stream = ifmt_ctx->streams[i];
        codec_ctx = stream->codec;
        /* Reencode video & audio and remux subtitles etc. */
        if (codec_ctx->codec_type == AVMEDIA_TYPE_VIDEO){
            /* Open decoder */
            ret = avcodec_open2(codec_ctx,
                avcodec_find_decoder(codec_ctx->codec_id), NULL);
            if (ret < 0) {
                av_log(NULL, AV_LOG_ERROR, "Failed to open decoder for stream #%u\n", i);
                return ret;
            }
        }
    }
    //av_dump_format(ifmt_ctx, 0, "whatever", 0);

    /*open output file*/
    AVIOContext *avio_out =avio_alloc_context(outbuffer, 32768,1,NULL,NULL,write_buffer,NULL);
    if(avio_out==NULL)
        goto end;
    //avio_out->write_packet=write_packet;
    ofmt_ctx->pb=avio_out;
    ofmt_ctx->flags=AVFMT_FLAG_CUSTOM_IO;
    for (i = 0; i < 1; i++) {
        out_stream = avformat_new_stream(ofmt_ctx, NULL);
        if (!out_stream) {
            av_log(NULL, AV_LOG_ERROR, "Failed allocating output stream\n");
            return AVERROR_UNKNOWN;
        }
        in_stream = ifmt_ctx->streams[i];
        dec_ctx = in_stream->codec;
        enc_ctx = out_stream->codec;
        if (dec_ctx->codec_type == AVMEDIA_TYPE_VIDEO)
        {
            encoder = avcodec_find_encoder(AV_CODEC_ID_H264);
            enc_ctx->height = dec_ctx->height;
            enc_ctx->width = dec_ctx->width;
            enc_ctx->sample_aspect_ratio = dec_ctx->sample_aspect_ratio;
            enc_ctx->pix_fmt = encoder->pix_fmts[0];
            enc_ctx->time_base = dec_ctx->time_base;
            //enc_ctx->time_base.num = 1;
            //enc_ctx->time_base.den = 25;
            //H264的必备选项，没有就会错
            enc_ctx->me_range=16;
            enc_ctx->max_qdiff = 4;
            enc_ctx->qmin = 10;
            enc_ctx->qmax = 51;
            enc_ctx->qcompress = 0.6;
            enc_ctx->refs=3;
            enc_ctx->bit_rate = 500000;

            ret = avcodec_open2(enc_ctx, encoder, NULL);
            if (ret < 0) {
                av_log(NULL, AV_LOG_ERROR, "Cannot open video encoder for stream #%u\n", i);
                return ret;
            }
        }
        else if (dec_ctx->codec_type == AVMEDIA_TYPE_UNKNOWN) {
            av_log(NULL, AV_LOG_FATAL, "Elementary stream #%d is of unknown type, cannot proceed\n", i);
            return AVERROR_INVALIDDATA;
        } else {
            /* if this stream must be remuxed */
            ret = avcodec_copy_context(ofmt_ctx->streams[i]->codec,
                ifmt_ctx->streams[i]->codec);
            if (ret < 0) {
                av_log(NULL, AV_LOG_ERROR, "Copying stream context failed\n");
                return ret;
            }
        }
```

```
        }
        if (ofmt_ctx->oformat->flags & AVFMT_GLOBALHEADER)
            enc_ctx->flags |= CODEC_FLAG_GLOBAL_HEADER;
    }
    //av_dump_format(ofmt_ctx, 0, "whatever", 1);
    /* init muxer, write output file header */
    ret = avformat_write_header(ofmt_ctx, NULL);
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR, "Error occurred when opening output file\n");
        return ret;
    }

    i=0;
    /* read all packets */
    while (1) {
        i++;
        if ((ret = av_read_frame(ifmt_ctx, &packet)) < 0)
            break;
        stream_index = packet.stream_index;
        if(stream_index!=0)
            continue;
        type = ifmt_ctx->streams[packet.stream_index]->codec->codec_type;
        av_log(NULL, AV_LOG_DEBUG, "Demuxer gave frame of stream_index %u\n",
            stream_index);

        av_log(NULL, AV_LOG_DEBUG, "Going to reencode the frame\n");
        frame = av_frame_alloc();
        if (!frame) {
            ret = AVERROR(ENOMEM);
            break;
        }
        packet.dts = av_rescale_q_rnd(packet.dts,
            ifmt_ctx->streams[stream_index]->time_base,
            ifmt_ctx->streams[stream_index]->codec->time_base,
            (AVRounding)(AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
        packet.pts = av_rescale_q_rnd(packet.pts,
            ifmt_ctx->streams[stream_index]->time_base,
            ifmt_ctx->streams[stream_index]->codec->time_base,
            (AVRounding)(AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
        ret = avcodec_decode_video2(ifmt_ctx->streams[stream_index]->codec, frame,
            &got_frame, &packet);
        printf("Decode 1 Packet\tsize:%d\tpts:%d\n",packet.size,packet.pts);

        if (ret < 0) {
            av_frame_free(&frame);
            av_log(NULL, AV_LOG_ERROR, "Decoding failed\n");
            break;
        }
        if (got_frame) {
            frame->pts = av_frame_get_best_effort_timestamp(frame);
            frame->pict_type=AV_PICTURE_TYPE_NONE;

            enc_pkt.data = NULL;
            enc_pkt.size = 0;
            av_init_packet(&enc_pkt);
            ret = avcodec_encode_video2 (ofmt_ctx->streams[stream_index]->codec, &enc_pkt,
                frame, &enc_got_frame);

            printf("Encode 1 Packet\tsize:%d\tpts:%d\n",enc_pkt.size,enc_pkt.pts);

            av_frame_free(&frame);
            if (ret < 0)
                goto end;
            if (!enc_got_frame)
                continue;
            /* prepare packet for muxing */
            enc_pkt.stream_index = stream_index;
            enc_pkt.dts = av_rescale_q_rnd(enc_pkt.dts,
                ofmt_ctx->streams[stream_index]->codec->time_base,
                ofmt_ctx->streams[stream_index]->time_base,
                (AVRounding)(AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
            enc_pkt.pts = av_rescale_q_rnd(enc_pkt.pts,
                ofmt_ctx->streams[stream_index]->codec->time_base,
                ofmt_ctx->streams[stream_index]->time_base,
                (AVRounding)(AV_ROUND_NEAR_INF|AV_ROUND_PASS_MINMAX));
            enc_pkt.duration = av_rescale_q(enc_pkt.duration,
                ofmt_ctx->streams[stream_index]->codec->time_base,
                ofmt_ctx->streams[stream_index]->time_base);
            av_log(NULL, AV_LOG_INFO, "Muxing frame %d\n",i);
            /* mux encoded frame */
            av_write_frame(ofmt_ctx,&enc_pkt);
            if (ret < 0)
                goto end;
        } else {
            av_frame_free(&frame);
        }

        av_free_packet(&packet);
    }

    /* flush encoders */
    for (i = 0; i < 1; i++) {
```

```
        /* flush encoder */
        ret = flush_encoder(ofmt_ctx,i);
        if (ret < 0) {
            av_log(NULL, AV_LOG_ERROR, "Flushing encoder failed\n");
            goto end;
        }
    }
    av_write_trailer(ofmt_ctx);
end:
    av_freep(avio_in);
    av_freep(avio_out);
    av_free(inbuffer);
    av_free(outbuffer);
    av_free_packet(&packet);
    av_frame_free(&frame);
    avformat_close_input(&ifmt_ctx);
    avformat_free_context(ofmt_ctx);

    fcloseall();

    if (ret < 0)
        av_log(NULL, AV_LOG_ERROR, "Error occurred\n");
    return (ret? 1:0);
}
```

结果

程序运行的结果如下图所示。

转码前的视频信息使用MediaInfo查看如下图所示。

转码后的视频信息使用MediaInfo查看如下图所示。

下载

simplest ffmpeg mem handler

项目主页

SourceForge： https://sourceforge.net/projects/simplestffmpegmemhandler/

Github： https://github.com/leixiaohua1020/simplest_ffmpeg_mem_handler

开源中国： http://git.oschina.net/leixiaohua1020/simplest_ffmpeg_mem_handler

CSDN下载地址：
http://download.csdn.net/detail/leixiaohua1020/8003731

本工程包含两个FFmpeg读写内存的例子：
simplest_ffmpeg_mem_player：基于FFmpeg的内存播放器。
simplest_ffmpeg_mem_transcoder：基于FFmpeg的内存转码器。

更新-1.1 (2015.2.13)==========

这次考虑到了跨平台的要求，调整了源代码。经过这次调整之后，源代码可以在以下平台编译通过：

VC++：打开sln文件即可编译，无需配置。

cl.exe：打开compile_cl.bat即可命令行下使用cl.exe进行编译，注意可能需要按照VC的安装路径调整脚本里面的参数。编译命令如下。

```
::VS2010 Environment
call "D:\Program Files\Microsoft Visual Studio 10.0\VC\vcvarsall.bat"
::include
@set INCLUDE=include;%INCLUDE%
::lib
@set LIB=lib;%LIB%
::compile and link
cl simplest_ffmpeg_mem_transcoder.cpp /link avcodec.lib avformat.lib avutil.lib ^
avdevice.lib avfilter.lib postproc.lib swresample.lib swscale.lib /OPT:NOREF
```

MinGW：MinGW命令行下运行compile_mingw.sh即可使用MinGW的g++进行编译。编译命令如下。

```
g++ simplest_ffmpeg_mem_transcoder.cpp -g -o simplest_ffmpeg_mem_transcoder.exe \
-I /usr/local/include -L /usr/local/lib \
-lavcodec -lavformat -lavutil -lavdevice -lavfilter -lpostproc -lswresample -lswscale
```

GCC：Linux或者MacOS命令行下运行compile_gcc.sh即可使用GCC进行编译。编译命令如下。

```
gcc simplest_ffmpeg_mem_transcoder.cpp -g -o simplest_ffmpeg_mem_transcoder.out \
-I /usr/local/include -L /usr/local/lib \
-lavcodec -lavformat -lavutil -lavdevice -lavfilter -lpostproc -lswresample -lswscale
```

PS：相关的编译命令已经保存到了工程文件夹中

CSDN下载地址： http://download.csdn.net/detail/leixiaohua1020/8445795

SourceForge上已经更新。

文章标签： ffmpeg 内存 转码 H.264

个人分类： 我的开源项目 FFMPEG

所属专栏： FFmpeg