# FFmpeg之重采样demo解析！

## **一、重采样：**

### **1、什么是重采样？**

通俗的讲，重采样就是改变音频的采样率、sample format(采样格式)、声道数(channel)等参数，使之按照我们期望的参数输出。

### **2、为什么需要重采样？**

做什么事情之前，我们都要问一个为什么，也就是知道原理！那么为什么需要重采样呢？那是因为当原有的音频参数不满足我们实际要求时，比如说在FFmpeg解码音频的时候，不同的音源有不同的格式和采样率等，所以在解码后的数据中的这些参数也会不一致(最新的FFmpeg解码音频后，音频格式为AV\_SAMPLE\_FMT\_TLTP);如果我们接下来需要使用解码后的音频数据做其它操作的话，然而这些参数的不一致会导致有很多额外工作，此时直接对其进行重采样的话，获取我们制定的音频参数，就会方便很多。

再比如说，在将音频进行SDL播放的时候，因为当前的SDL2.0不支持plannar格式，也不支持浮点型的，而最新的FFpemg会将音频解码为AV\_SAMPLE\_FMT\_FLTP，这个时候进行对它重采样的话，就可以在SDL2.0上进行播放这个音频了。

### **3、重采样参数解析：**

sample rate(采样率)：采样设备每秒抽取样本的次数

sample format(采样格式)和量化精度：这个应该好理解，就是采用什么格式进行采集数据；每种⾳频格式有不同的量化精度（位宽），位数越多，表示值就越精确，声⾳表现⾃然就越精准。

channel layout（通道布局，也就是声道数）：这个就是采样的声道数

这里多补充一下:

在FFmpeg里面主要有两种采样格式:floating-point formats 和  planar sample formats；具体采样参数如下(在（libavutil/samplefmt.h头文件里面)：

enum AVSampleFormat {  
    AV\_SAMPLE\_FMT\_NONE = -1,  
    AV\_SAMPLE\_FMT\_U8,          ///< unsigned 8 bits  
    AV\_SAMPLE\_FMT\_S16,         ///< signed 16 bits  
    AV\_SAMPLE\_FMT\_S32,         ///< signed 32 bits  
    AV\_SAMPLE\_FMT\_FLT,         ///< float  
    AV\_SAMPLE\_FMT\_DBL,         ///< double  
  
    AV\_SAMPLE\_FMT\_U8P,         ///< unsigned 8 bits, planar  
    AV\_SAMPLE\_FMT\_S16P,        ///< signed 16 bits, planar  
    AV\_SAMPLE\_FMT\_S32P,        ///< signed 32 bits, planar  
    AV\_SAMPLE\_FMT\_FLTP,        ///< float, planar  
    AV\_SAMPLE\_FMT\_DBLP,        ///< double, planar  
    AV\_SAMPLE\_FMT\_S64,         ///< signed 64 bits  
    AV\_SAMPLE\_FMT\_S64P,        ///< signed 64 bits, planar  
  
    AV\_SAMPLE\_FMT\_NB           ///< Number of sample formats. DO NOT USE if linking dynamically  
};

上半部分就是floating-point formats,下半部分就是planar格式，关于这两种格式的介绍：

* floating-point formats：

\* - The floating-point formats are based on full volume being in the range  
  
 \*   [-1.0, 1.0]. Any values outside this range are beyond full volume level.

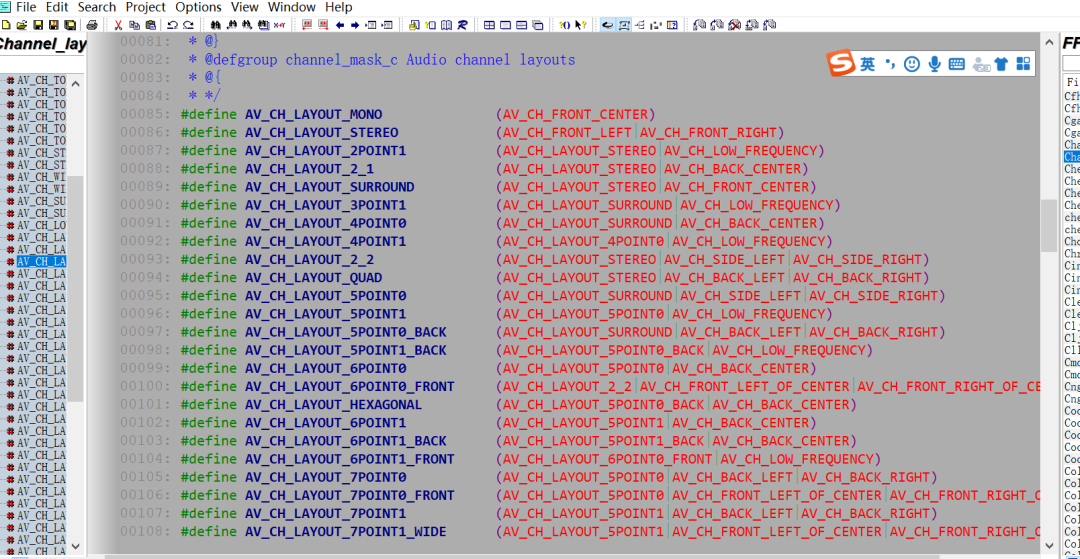
* planar sample formats：

For planar sample formats, each audio channel is in a separate data plane,  
 \* and linesize is the buffer size, in bytes, for a single plane. All data  
 \* planes must be the same size. For packed sample formats, only the first data  
 \* plane is used, and samples for each channel are interleaved. In this case,  
 \* linesize is the buffer size, in bytes, for the 1 plane.

还有就是声道分布参数，这个在FFmpeg也有说明(声道分布在FFmpeg\libavutil\channel\_layout.h中有定义，⼀般来说⽤的⽐较多的是 AV\_CH\_LAYOUT\_STEREO（双声道）和AV\_CH\_LAYOUT\_SURROUND（三声道），这两者的定义如下)：

*#define AV\_CH\_LAYOUT\_STEREO (AV\_CH\_FRONT\_LEFT|AV\_CH\_FRONT\_RIGHT)*  
*#define AV\_CH\_LAYOUT\_SURROUND (AV\_CH\_LAYOUT\_STEREO|AV\_CH\_FRONT\_CENTER)*

下面是其他声道数参数：



### **4、分⽚（plane）和打包（packed）：**

以双声道为例，带P（plane）的数据格式在存储时，其左声道和右声道的数据是分开存储的，左声道的 数据存储在data[0]，右声道的数据存储在data[1]，每个声道的所占⽤的字节数为linesize[0]和 linesize[1]；

不带P（packed）的⾳频数据在存储时，是按照LRLRLR...的格式交替存储在data[0]中，linesize[0] 表示总的数据量。

这个可以在下面的代码里面可以看到用法，这里简单提一下。

### **5、⾳频帧的数据量计算：**

⼀帧⾳频的数据量（字节）=channel数 \* nb\_samples样本数 \* 每个样本占⽤的字节数 如果该⾳频帧是FLTP格式的PCM数据，包含1024个样本，双声道，那么该⾳频帧包含的⾳频数据量是：

2\*1024\*4=8192字节

### **6、⾳频播放时间计算：**

以采样率44100Hz来计算，每秒44100个sample，⽽正常⼀帧为1024个sample，可知每帧播放时 间/1024=1000ms/44100，得到每帧播放时间=1024\*1000/44100=23.2ms （更精确的是 23.21995464852608）

但是要注意：

* ⼀帧播放时间（毫秒） = nb\_samples样本数 *1000/采样率 = 1024*1000/44100=23.21995464852608ms；约等于 23.2ms，精度损失了 0.011995464852608ms，如果累计10万帧，误差>1199毫秒，如果有视频⼀起的就会有⾳视频同步的问题。这个误差太可怕了。。。。

## **二、FFmpeg之api讲解：**

* 分配⾳频重采样的上下⽂：

av\_cold struct SwrContext \*swr\_alloc(void){  
    SwrContext \*s= av\_mallocz(sizeof(SwrContext));  
    if(s){  
        s->av\_class= &av\_class;  
        av\_opt\_set\_defaults(s);  
    }  
    return s;  
}

struct SwrContext {  
    const AVClass \*av\_class;                        ///< AVClass used for AVOption and av\_log()  
    int log\_level\_offset;                           ///< logging level offset  
    void \*log\_ctx;                                  ///< parent logging context  
    enum AVSampleFormat  in\_sample\_fmt;             ///< input sample format  
    enum AVSampleFormat int\_sample\_fmt;             ///< internal sample format (AV\_SAMPLE\_FMT\_FLTP or AV\_SAMPLE\_FMT\_S16P)  
    enum AVSampleFormat out\_sample\_fmt;             ///< output sample format  
    int64\_t  in\_ch\_layout;                          ///< input channel layout  
    int64\_t out\_ch\_layout;                          ///< output channel layout  
    int      in\_sample\_rate;                        ///< input sample rate  
    int     out\_sample\_rate;                        ///< output sample rate  
    int flags;                                      ///< miscellaneous flags such as SWR\_FLAG\_RESAMPLE  
    float slev;                                     ///< surround mixing level  
    float clev;                                     ///< center mixing level  
    float lfe\_mix\_level;                            ///< LFE mixing level  
    float rematrix\_volume;                          ///< rematrixing volume coefficient  
    float rematrix\_maxval;                          ///< maximum value for rematrixing output  
    int matrix\_encoding;                            /\*\*< matrixed stereo encoding \*/  
    const int \*channel\_map;                         ///< channel index (or -1 if muted channel) map  
    int used\_ch\_count;                              ///< number of used input channels (mapped channel count if channel\_map, otherwise in.ch\_count)  
    int engine;  
  
    int user\_in\_ch\_count;                           ///< User set input channel count  
    int user\_out\_ch\_count;                          ///< User set output channel count  
    int user\_used\_ch\_count;                         ///< User set used channel count  
    int64\_t user\_in\_ch\_layout;                      ///< User set input channel layout  
    int64\_t user\_out\_ch\_layout;                     ///< User set output channel layout  
    enum AVSampleFormat user\_int\_sample\_fmt;        ///< User set internal sample format  
    int user\_dither\_method;                         ///< User set dither method  
  
    struct DitherContext dither;  
  
    int filter\_size;                                /\*\*< length of each FIR filter in the resampling filterbank relative to the cutoff frequency \*/  
    int phase\_shift;                                /\*\*< log2 of the number of entries in the resampling polyphase filterbank \*/  
    int linear\_interp;                              /\*\*< if 1 then the resampling FIR filter will be linearly interpolated \*/  
    int exact\_rational;                             /\*\*< if 1 then enable non power of 2 phase\_count \*/  
    double cutoff;                                  /\*\*< resampling cutoff frequency (swr: 6dB point; soxr: 0dB point). 1.0 corresponds to half the output sample rate \*/  
    int filter\_type;                                /\*\*< swr resampling filter type \*/  
    double kaiser\_beta;                                /\*\*< swr beta value for Kaiser window (only applicable if filter\_type == AV\_FILTER\_TYPE\_KAISER) \*/  
    double precision;                               /\*\*< soxr resampling precision (in bits) \*/  
    int cheby;                                      /\*\*< soxr: if 1 then passband rolloff will be none (Chebyshev) & irrational ratio approximation precision will be higher \*/  
  
    float min\_compensation;                         ///< swr minimum below which no compensation will happen  
    float min\_hard\_compensation;                    ///< swr minimum below which no silence inject / sample drop will happen  
    float soft\_compensation\_duration;               ///< swr duration over which soft compensation is applied  
    float max\_soft\_compensation;                    ///< swr maximum soft compensation in seconds over soft\_compensation\_duration  
    float async;                                    ///< swr simple 1 parameter async, similar to ffmpegs -async  
    int64\_t firstpts\_in\_samples;                    ///< swr first pts in samples  
  
    int resample\_first;                             ///< 1 if resampling must come first, 0 if rematrixing  
    int rematrix;                                   ///< flag to indicate if rematrixing is needed (basically if input and output layouts mismatch)  
    int rematrix\_custom;                            ///< flag to indicate that a custom matrix has been defined  
  
    AudioData in;                                   ///< input audio data  
    AudioData postin;                               ///< post-input audio data: used for rematrix/resample  
    AudioData midbuf;                               ///< intermediate audio data (postin/preout)  
    AudioData preout;                               ///< pre-output audio data: used for rematrix/resample  
    AudioData out;                                  ///< converted output audio data  
    AudioData in\_buffer;                            ///< cached audio data (convert and resample purpose)  
    AudioData silence;                              ///< temporary with silence  
    AudioData drop\_temp;                            ///< temporary used to discard output  
    int in\_buffer\_index;                            ///< cached buffer position  
    int in\_buffer\_count;                            ///< cached buffer length  
    int resample\_in\_constraint;                     ///< 1 if the input end was reach before the output end, 0 otherwise  
    int flushed;                                    ///< 1 if data is to be flushed and no further input is expected  
    int64\_t outpts;                                 ///< output PTS  
    int64\_t firstpts;                               ///< first PTS  
    int drop\_output;                                ///< number of output samples to drop  
    double delayed\_samples\_fixup;                   ///< soxr 0.1.1: needed to fixup delayed\_samples after flush has been called.  
  
    struct AudioConvert \*in\_convert;                ///< input conversion context  
    struct AudioConvert \*out\_convert;               ///< output conversion context  
    struct AudioConvert \*full\_convert;              ///< full conversion context (single conversion for input and output)  
    struct ResampleContext \*resample;               ///< resampling context  
    struct Resampler const \*resampler;              ///< resampler virtual function table  
  
    double matrix[SWR\_CH\_MAX][SWR\_CH\_MAX];          ///< floating point rematrixing coefficients  
    float matrix\_flt[SWR\_CH\_MAX][SWR\_CH\_MAX];       ///< single precision floating point rematrixing coefficients  
    uint8\_t \*native\_matrix;  
    uint8\_t \*native\_one;  
    uint8\_t \*native\_simd\_one;  
    uint8\_t \*native\_simd\_matrix;  
    int32\_t matrix32[SWR\_CH\_MAX][SWR\_CH\_MAX];       ///< 17.15 fixed point rematrixing coefficients  
    uint8\_t matrix\_ch[SWR\_CH\_MAX][SWR\_CH\_MAX+1];    ///< Lists of input channels per output channel that have non zero rematrixing coefficients  
    mix\_1\_1\_func\_type \*mix\_1\_1\_f;  
    mix\_1\_1\_func\_type \*mix\_1\_1\_simd;  
  
    mix\_2\_1\_func\_type \*mix\_2\_1\_f;  
    mix\_2\_1\_func\_type \*mix\_2\_1\_simd;  
  
    mix\_any\_func\_type \*mix\_any\_f;  
  
    /\* TODO: callbacks for ASM optimizations \*/  
};

* 初始化SwrContext结构体：

int swr\_init(struct SwrContext \*s);

* 分配SwrContext并设置/重置常⽤的参数：

struct SwrContext \*swr\_alloc\_set\_opts(struct SwrContext \*s, // ⾳频重采样上下⽂   
int64\_t out\_ch\_layout, // 输出的layout, 如：5.1声道   
enum AVSampleFormat out\_sample\_fmt, // 输出的采样格式。  
Float, S16,⼀般 选⽤是s16 绝⼤部分声卡⽀持   
int out\_sample\_rate, //输出采样率   
int64\_t in\_ch\_layout, // 输⼊的layout enum AVSampleFormat in\_sample\_fmt, // 输⼊的采样格式   
int in\_sample\_rate, // 输⼊的采样率   
int log\_offset, // ⽇志相关，不⽤管先，直接为0 void \*log\_ctx // ⽇志相关，不⽤管先，直接为NULL   
);

* 将输⼊的⾳频按照定义的参数进⾏转换并输出：

int swr\_convert(struct SwrContext \*s, // ⾳频重采样的上下⽂   
uint8\_t \*\*out, // 输出的指针。传递的输出的数组 int out\_count, //输出的样本数量，不是字节数。单通道的样本数量。   
const uint8\_t \*\*in , //输⼊的数组，AVFrame解码出来的DATA   
int in\_count // 输⼊的单通道的样本数量。  
);

释放掉SwrContext结构体并将此结构体置为NULL：

void swr\_free(struct SwrContext \*\*s)

* ⾳频重采样，采样格式转换和混合库：

与lswr的交互是通过SwrContext完成的，SwrContext被分配给swr\_alloc（）或 swr\_alloc\_set\_opts（）。它是不透明的，所以所有参数必须使⽤AVOptions API设置。为了使⽤lswr，你需要做的第⼀件事就是分配SwrContext。这可以使⽤swr\_alloc（）或 swr\_alloc\_set\_opts（）来完成。如果您使⽤前者，则必须通过AVOptions API设置选项。后⼀个函数 提供了相同的功能，但它允许您在同⼀语句中设置⼀些常⽤选项。例如，以下代码将设置从平⾯浮动样本格式到交织的带符号16位整数的转换，从48kHz到44.1kHz的下采 样，以及从5.1声道到⽴体声的下混合（使⽤默认混合矩阵）。这是使⽤swr\_alloc（）函数：

1 SwrContext \*swr = swr\_alloc();   
2 av\_opt\_set\_channel\_layout(swr, "in\_channel\_layout", AV\_CH\_LAYOUT\_ 5POINT1, 0);   
3 av\_opt\_set\_channel\_layout(swr, "out\_channel\_layout", AV\_CH\_LAYOUT\_ STEREO, 0);   
4 av\_opt\_set\_int(swr, "in\_sample\_rate", 48000, 0) ;   
5 av\_opt\_set\_int(swr, "out\_sample\_rate", 44100, 0) ;  
6 av\_opt\_set\_sample\_fmt(swr, "in\_sample\_fmt", AV\_SAMPLE\_FMT\_FLTP, 0 );   
7 av\_opt\_set\_sample\_fmt(swr, "out\_sample\_fmt", AV\_SAMPLE\_FMT\_S16, 0 );

同样的⼯作也可以使⽤swr\_alloc\_set\_opts（）：

SwrContext \*swr = swr\_alloc\_set\_opts(NULL, // we're allocating a new context   
2 AV\_CH\_LAYOUT\_STEREO, // out\_ch\_layout  
3 AV\_SAMPLE\_FMT\_S16, // out\_sample\_fmt   
4 44100, // out\_sample\_rate  
5 AV\_CH\_LAYOUT\_5POINT1, // in\_ch\_layout  
6 AV\_SAMPLE\_FMT\_FLTP, // in\_sample\_fmt   
7 48000, // in\_sample\_rate   
8 0, // log\_offset  
9 NULL  
); // log\_ctx

⼀旦设置了所有值，它必须⽤swr\_init（）初始化。如果需要更改转换参数，可以使⽤ AVOptions来更改参数，如上⾯第⼀个例⼦所述; 或者使⽤swr\_alloc\_set\_opts（），但是第 ⼀个参数是分配的上下⽂。您必须再次调⽤swr\_init（）。

转换本身通过重复调⽤swr\_convert（）来完成。请注意，如果提供的输出空间不⾜或采样率转换完成 后，样本可能会在swr中缓冲，这需要“未来”样本。可以随时通过使⽤swr\_convert（）（in\_count可以 设置为0）来检索不需要将来输⼊的样本。在转换结束时，可以通过调⽤具有NULL in和in incount的 swr\_convert（）来刷新重采样缓冲区。

下面是一般重采样的流程：

分配：swr\_alloc()

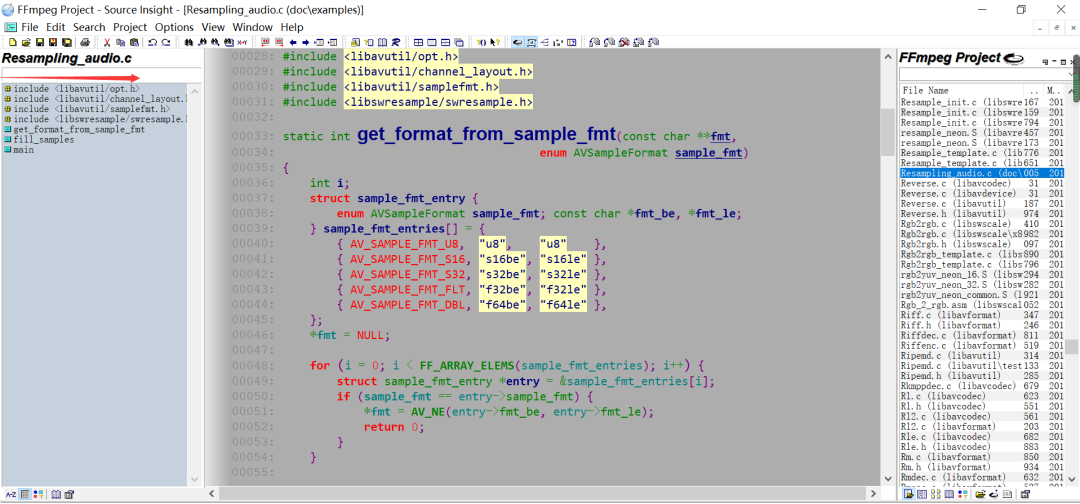
设置参数:av\_opt\_set\_channel\_layout();av\_opt\_set\_int();av\_opt\_set\_sample\_fmt()

初始化:swr\_init()

转换: swr\_convert()

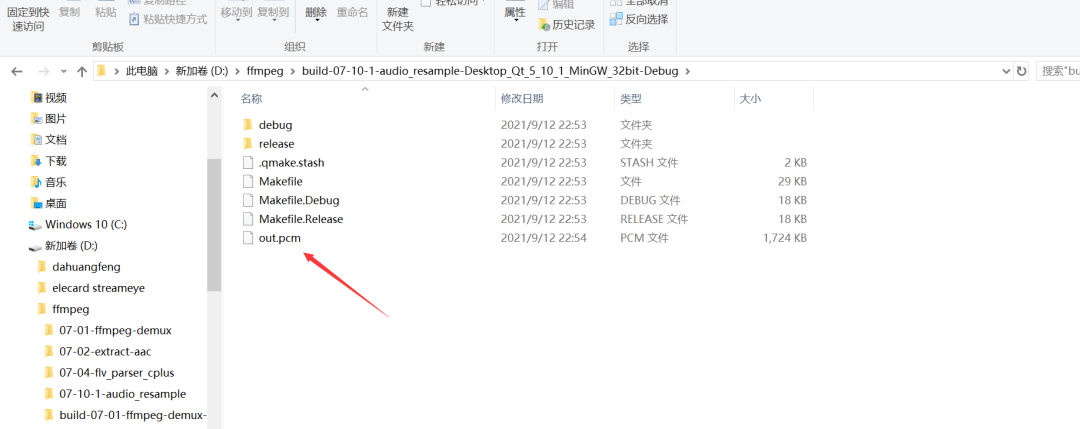
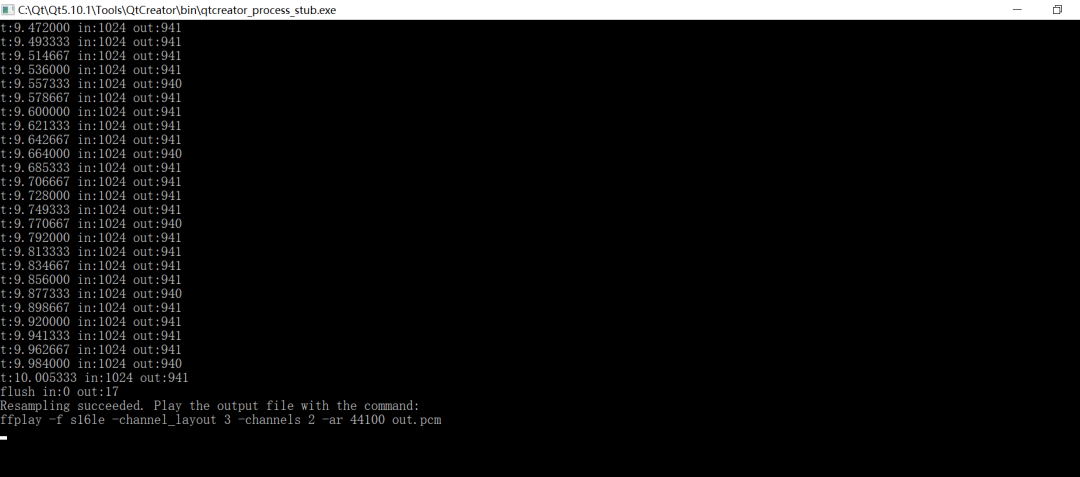
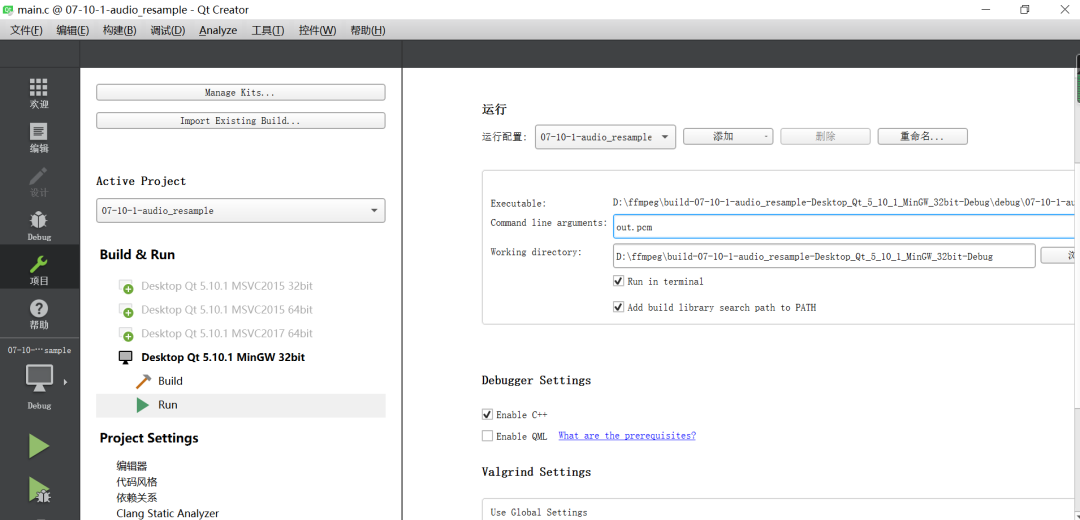
## **三、重采样demo:**

说明一下，这里代码有参考FFmpeg给的demo哈：



/\*  
 \* Copyright (c) 2012 Stefano Sabatini  
 \*  
 \* Permission is hereby granted, free of charge, to any person obtaining a copy  
 \* of this software and associated documentation files (the "Software"), to deal  
 \* in the Software without restriction, including without limitation the rights  
 \* to use, copy, modify, merge, publish, distribute, sublicense, and/or sell  
 \* copies of the Software, and to permit persons to whom the Software is  
 \* furnished to do so, subject to the following conditions:  
 \*  
 \* The above copyright notice and this permission notice shall be included in  
 \* all copies or substantial portions of the Software.  
 \*  
 \* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR  
 \* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,  
 \* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL  
 \* THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER  
 \* LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,  
 \* OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN  
 \* THE SOFTWARE.  
 \*/  
  
/\*\*  
 \* @example resampling\_audio.c  
 \* libswresample API use example.  
 \*/  
  
*#include <libavutil/opt.h>*  
*#include <libavutil/channel\_layout.h>*  
*#include <libavutil/samplefmt.h>*  
*#include <libswresample/swresample.h>*  
  
static int get\_format\_from\_sample\_fmt(const char \*\*fmt,  
                                      enum AVSampleFormat sample\_fmt)  
{  
    int i;  
    struct sample\_fmt\_entry {  
        enum AVSampleFormat sample\_fmt; const char \*fmt\_be, \*fmt\_le;  
    } sample\_fmt\_entries[] = {  
    { AV\_SAMPLE\_FMT\_U8,  "u8",    "u8"    },  
    { AV\_SAMPLE\_FMT\_S16, "s16be", "s16le" },  
    { AV\_SAMPLE\_FMT\_S32, "s32be", "s32le" },  
    { AV\_SAMPLE\_FMT\_FLT, "f32be", "f32le" },  
    { AV\_SAMPLE\_FMT\_DBL, "f64be", "f64le" },  
};  
    \*fmt = NULL;  
  
    for (i = 0; i < FF\_ARRAY\_ELEMS(sample\_fmt\_entries); i++) {  
        struct sample\_fmt\_entry \*entry = &sample\_fmt\_entries[i];  
        if (sample\_fmt == entry->sample\_fmt) {  
            \*fmt = AV\_NE(entry->fmt\_be, entry->fmt\_le);  
            return 0;  
        }  
    }  
  
    fprintf(stderr,  
            "Sample format %s not supported as output format\n",  
            av\_get\_sample\_fmt\_name(sample\_fmt));  
    return AVERROR(EINVAL);  
}  
  
/\*\*  
 \* Fill dst buffer with nb\_samples, generated starting from t. 交错模式的  
 \*/  
static void fill\_samples(double \*dst, int nb\_samples, int nb\_channels, int sample\_rate, double \*t)  
{  
    int i, j;  
    double tincr = 1.0 / sample\_rate, \*dstp = dst;  
    const double c = 2 \* M\_PI \* 440.0;  
  
    /\* generate sin tone with 440Hz frequency and duplicated channels \*/  
    for (i = 0; i < nb\_samples; i++) {  
        \*dstp = sin(c \* \*t);  
        for (j = 1; j < nb\_channels; j++)  
            dstp[j] = dstp[0];  
        dstp += nb\_channels;  
        \*t += tincr;  
    }  
}  
  
int main(int argc, char \*\*argv)  
{  
    // 输入参数  
    int64\_t src\_ch\_layout = AV\_CH\_LAYOUT\_STEREO;  
    int src\_rate = 48000;  
    enum AVSampleFormat src\_sample\_fmt = AV\_SAMPLE\_FMT\_DBL;  
    int src\_nb\_channels = 0;  
    uint8\_t \*\*src\_data = NULL;  // 二级指针  
    int src\_linesize;  
    int src\_nb\_samples = 1024;  
  
  
    // 输出参数  
    int64\_t dst\_ch\_layout = AV\_CH\_LAYOUT\_STEREO;  
    int dst\_rate = 44100;  
    enum AVSampleFormat dst\_sample\_fmt = AV\_SAMPLE\_FMT\_S16;  
    int dst\_nb\_channels = 0;  
    uint8\_t \*\*dst\_data = NULL;  //二级指针  
    int dst\_linesize;  
    int dst\_nb\_samples;  
    int max\_dst\_nb\_samples;  
  
    // 输出文件  
    const char \*dst\_filename = NULL;    // 保存输出的pcm到本地，然后播放验证  
    FILE \*dst\_file;  
  
  
    int dst\_bufsize;  
    const char \*fmt;  
  
    // 重采样实例  
    struct SwrContext \*swr\_ctx;  
  
    double t;  
    int ret;  
  
    if (argc != 2) {  
        fprintf(stderr, "Usage: %s output\_file\n"  
                        "API example program to show how to resample an audio stream with libswresample.\n"  
                        "This program generates a series of audio frames, resamples them to a specified "  
                        "output format and rate and saves them to an output file named output\_file.\n",  
                argv[0]);  
        exit(1);  
    }  
    dst\_filename = argv[1];  
  
    dst\_file = fopen(dst\_filename, "wb");  
    if (!dst\_file) {  
        fprintf(stderr, "Could not open destination file %s\n", dst\_filename);  
        exit(1);  
    }  
  
    // 创建重采样器  
    /\* create resampler context \*/  
    swr\_ctx = swr\_alloc();  
    if (!swr\_ctx) {  
        fprintf(stderr, "Could not allocate resampler context\n");  
        ret = AVERROR(ENOMEM);  
        goto end;  
    }  
  
    // 设置重采样参数  
    /\* set options \*/  
    // 输入参数  
    av\_opt\_set\_int(swr\_ctx, "in\_channel\_layout",    src\_ch\_layout, 0);  
    av\_opt\_set\_int(swr\_ctx, "in\_sample\_rate",       src\_rate, 0);  
    av\_opt\_set\_sample\_fmt(swr\_ctx, "in\_sample\_fmt", src\_sample\_fmt, 0);  
    // 输出参数  
    av\_opt\_set\_int(swr\_ctx, "out\_channel\_layout",    dst\_ch\_layout, 0);  
    av\_opt\_set\_int(swr\_ctx, "out\_sample\_rate",       dst\_rate, 0);  
    av\_opt\_set\_sample\_fmt(swr\_ctx, "out\_sample\_fmt", dst\_sample\_fmt, 0);  
  
    // 初始化重采样  
    /\* initialize the resampling context \*/  
    if ((ret = swr\_init(swr\_ctx)) < 0) {  
        fprintf(stderr, "Failed to initialize the resampling context\n");  
        goto end;  
    }  
  
    /\* allocate source and destination samples buffers \*/  
    // 计算出输入源的通道数量  
    src\_nb\_channels = av\_get\_channel\_layout\_nb\_channels(src\_ch\_layout);  
    // 给输入源分配内存空间  
    ret = av\_samples\_alloc\_array\_and\_samples(&src\_data, &src\_linesize, src\_nb\_channels,  
                                             src\_nb\_samples, src\_sample\_fmt, 0);  
    if (ret < 0) {  
        fprintf(stderr, "Could not allocate source samples\n");  
        goto end;  
    }  
  
    /\* compute the number of converted samples: buffering is avoided  
     \* ensuring that the output buffer will contain at least all the  
     \* converted input samples \*/  
    // 计算输出采样数量  
    max\_dst\_nb\_samples = dst\_nb\_samples =  
            av\_rescale\_rnd(src\_nb\_samples, dst\_rate, src\_rate, AV\_ROUND\_UP);  
  
    /\* buffer is going to be directly written to a rawaudio file, no alignment \*/  
    dst\_nb\_channels = av\_get\_channel\_layout\_nb\_channels(dst\_ch\_layout);  
    // 分配输出缓存内存  
    ret = av\_samples\_alloc\_array\_and\_samples(&dst\_data, &dst\_linesize, dst\_nb\_channels,  
                                             dst\_nb\_samples, dst\_sample\_fmt, 0);  
    if (ret < 0) {  
        fprintf(stderr, "Could not allocate destination samples\n");  
        goto end;  
    }  
  
    t = 0;  
    do {  
        /\* generate synthetic audio \*/  
        // 生成输入源  
        fill\_samples((double \*)src\_data[0], src\_nb\_samples, src\_nb\_channels, src\_rate, &t);  
  
        /\* compute destination number of samples \*/  
        int64\_t delay = swr\_get\_delay(swr\_ctx, src\_rate);  
        dst\_nb\_samples = av\_rescale\_rnd(delay + src\_nb\_samples, dst\_rate, src\_rate, AV\_ROUND\_UP);  
        if (dst\_nb\_samples > max\_dst\_nb\_samples) {  
            av\_freep(&dst\_data[0]);  
            ret = av\_samples\_alloc(dst\_data, &dst\_linesize, dst\_nb\_channels,  
                                   dst\_nb\_samples, dst\_sample\_fmt, 1);  
            if (ret < 0)  
                break;  
            max\_dst\_nb\_samples = dst\_nb\_samples;  
        }  
        //        int fifo\_size = swr\_get\_out\_samples(swr\_ctx,src\_nb\_samples);  
        //        printf("fifo\_size:%d\n", fifo\_size);  
        //        if(fifo\_size < 1024)  
        //            continue;  
  
        /\* convert to destination format \*/  
        //        ret = swr\_convert(swr\_ctx, dst\_data, dst\_nb\_samples, (const uint8\_t \*\*)src\_data, src\_nb\_samples);  
        ret = swr\_convert(swr\_ctx, dst\_data, dst\_nb\_samples, (const uint8\_t \*\*)src\_data, src\_nb\_samples);  
        if (ret < 0) {  
            fprintf(stderr, "Error while converting\n");  
            goto end;  
        }  
        dst\_bufsize = av\_samples\_get\_buffer\_size(&dst\_linesize, dst\_nb\_channels,  
                                                 ret, dst\_sample\_fmt, 1);  
        if (dst\_bufsize < 0) {  
            fprintf(stderr, "Could not get sample buffer size\n");  
            goto end;  
        }  
        printf("t:%f in:%d out:%d\n", t, src\_nb\_samples, ret);  
        fwrite(dst\_data[0], 1, dst\_bufsize, dst\_file);  
    } while (t < 10);  
  
    ret = swr\_convert(swr\_ctx, dst\_data, dst\_nb\_samples, NULL, 0);  
    if (ret < 0) {  
        fprintf(stderr, "Error while converting\n");  
        goto end;  
    }  
    dst\_bufsize = av\_samples\_get\_buffer\_size(&dst\_linesize, dst\_nb\_channels,  
                                             ret, dst\_sample\_fmt, 1);  
    if (dst\_bufsize < 0) {  
        fprintf(stderr, "Could not get sample buffer size\n");  
        goto end;  
    }  
    printf("flush in:%d out:%d\n", 0, ret);  
    fwrite(dst\_data[0], 1, dst\_bufsize, dst\_file);  
  
  
    if ((ret = get\_format\_from\_sample\_fmt(&fmt, dst\_sample\_fmt)) < 0)  
        goto end;  
    fprintf(stderr, "Resampling succeeded. Play the output file with the command:\n"  
                    "ffplay -f %s -channel\_layout %"PRId64" -channels %d -ar %d %s\n",  
            fmt, dst\_ch\_layout, dst\_nb\_channels, dst\_rate, dst\_filename);  
  
end:  
    fclose(dst\_file);  
  
    if (src\_data)  
        av\_freep(&src\_data[0]);  
    av\_freep(&src\_data);  
  
    if (dst\_data)  
        av\_freep(&dst\_data[0]);  
    av\_freep(&dst\_data);  
  
    swr\_free(&swr\_ctx);  
    return ret < 0;  
}

输出out.pcm:



然后进行播放一下：

ffplay -f s16le -channel\_layout 3 -channels 2 -ar 44100 out.pcm

