mplayer源代码解析 （资料汇总）——audio

**MPlayer流程**

int c\_mplay\_main(int argc,char\* argv[])  
{  
\*\*调用 AddExcept（）注册异常处理函数  
\*\* initmplayer();   //初始化，创建快进和暂停的信号量  
\*\*InitTimer();初始化计时器  
\*\*mp\_msg\_init();初始化消息系统  
\*\*set\_path\_env();设置路径、环境  
\*\*ipu\_image\_start();ipu初始化  
\*\*mplayer\_showmode(1);设置显示模式  
\*\*parse\_codec\_cfg(NULL);解析codec配置寄存器  
\*\*打开数据流  
\*\*分析播放树  
\*\*添加播放树列表  
\*\*初始化预填充缓存  
\*\*打开播放的文件  
\*\*创建buffer  
\*\*打开数据流  
\*\*检测数据流类型（音频格式和视频格式）  
\*\*分析音频流视频流的信息（原始视频尺寸、分辨率、帧频率、码流大小）  
\*\*启动相应的分离器  
\*\*分析剪辑信息  
\*\*初始化codec（多媒体数字信号编解码器）  
\*\*选择打开相应的视频解码器  
\*\*初始化视频解码器，分析视频流信息  
\*\*选择打开相应的音频解码器  
\*\*初始化音频解码器、PCM，分析音频信息  
\*\*同步音频视频输出  
\*\*开始播放  
  
}  
   main函数中的入口如下~   
/\*========================== PLAY AUDIO ============================\*/   
if (mpctx->sh\_audio)   
     if (!fill\_audio\_out\_buffers())   
     // at eof, all audio at least written to ao   
     //由mpctx->sh\_audio引出，我想重点强调一下sh\_audio\_t结构，这是音频流头部结构~   
// Stream headers:   
typedef struct {   
   int aid;   
   demux\_stream\_t \*ds;   
   struct codecs\_st \*codec;   
   unsigned int format;   
   int initialized;   
   float stream\_delay; // number of seconds stream should be delayed (according   
to dwStart or similar)   
   // output format:   
   int sample\_format;   
   int samplerate;   
   int samplesize;   
   int channels;   
   int o\_bps; // == samplerate\*samplesize\*channels   (uncompr. bytes/sec)   
   int i\_bps; // == bitrate   (compressed bytes/sec)   
   // in buffers:   
   int audio\_in\_minsize;         // max. compressed packet size (== min. in buffer size   
)   
   char\* a\_in\_buffer;   
   int a\_in\_buffer\_len;   
   int a\_in\_buffer\_size;   
   // decoder buffers:   
   int audio\_out\_minsize; // max. uncompressed packet size (==min. out buffsize   
)   
   char\* a\_buffer;   
   int a\_buffer\_len;   
   int a\_buffer\_size;   
   // output buffers:   
   char\* a\_out\_buffer;   
   int a\_out\_buffer\_len;   
   int a\_out\_buffer\_size;   
   struct af\_stream\_s \*afilter;           // the audio filter stream   
   struct ad\_functions\_s\* ad\_driver;   
#ifdef DYNAMIC\_PLUGINS   
   void \*dec\_handle;   
#endif   
   // win32-compatible codec parameters:   
   AVIStreamHeader audio;   
   WAVEFORMATEX\* wf;   
   // codec-specific:   
   void\* context; // codec-specific stuff (usually HANDLE or struct pointer)   
   unsigned char\* codecdata; // extra header data passed from demuxer to codec   
  
   int codecdata\_len;   
   double pts;   // last known pts value in output from decoder   
   int pts\_bytes; // bytes output by decoder after last known pts   
   char\* lang; // track language   
   int default\_track;   
} sh\_audio\_t;   
  
三.step by step   
   1.下面我们来分析fill\_audio\_out\_buffers()函数   
   static int fill\_audio\_out\_buffers(void)   
  
     1.1查看音频驱动有多大的空间可以填充解码后的音频数据   
     bytes\_to\_write = mpctx->audio\_out->get\_space();   
  
     这里是查看音频驱动还有多少空闲空间（即pcm缓冲区队列的可用空间），如果有空闲的空间，就调用ao->play()  
函数来填充这个buffer，并播放音频数据。（pcm缓冲区的数据会被dma自动送去播放，我们不能让pcm缓冲区为空，否则声音就没了！）这个音频驱动的buffer的大小要设置合适，太小会导致一帧图像还没播放完，buffer就已经空了，会导致音视频严重不同步；太大会导致Mplayer需要不停地解析媒体文件来填充这个音频buffer，而视频可能还来不及播放。。。   
  
     1.2 对音频流进行解码   
     if (decode\_audio(sh\_audio, playsize) < 0)   
     注：这里的decode\_audio只是一个接口，并不是具体的解码api。这个函数从sh\_audio->a\_out\_buffer获得至少playsize bytes的解码或过滤后的音频数据,成功返回0，失败返回-1   
  
     1.3 将解码后的音频数据进行播放   
     playsize = mpctx->audio\_out->play(sh\_audio->a\_out\_buffer, playsize, playflags);   
     注：从sh\_audio->a\_out\_buffer中copy playsize bytes数据，注意这里一定要copy出来，因为当play调用后，这部分数据就会被覆盖了。这些数据不一定要用完，返回的值是用掉的数据长度。另外当flags|AOPLAY\_FINAL\_CHUNK的值是真时，说明音频文件快结束了。（play（）函数把音频数据写到pcm缓冲区 。）  
  
  
     1.4 if (playsize > 0) {   
             sh\_audio->a\_out\_buffer\_len -= playsize;   
             memmove(sh\_audio->a\_out\_buffer, &sh\_audio->a\_out\_buffer[playsize],   
                     sh\_audio->a\_out\_buffer\_len);   
             mpctx->delay += playback\_speed\*playsize/(double)ao\_data.bps;   
         }   
     播放成功后就从sh\_audio->a\_out\_buffer中移出这段数据，同时减去sh\_audio->a\_out\_buffer\_len的长度 .   
  
  
     2.刚才说了，decode\_audio()函数并不是真正的解码函数，它只是提供一个接口，下面我们就来看看这个函数到底做了哪些事情.   
     int decode\_audio(sh\_audio\_t \*sh\_audio, int minlen)   
  
     2.1 解码后的视频数据都被cut成大小相同的一个个区间~   
     int unitsize = sh\_audio->channels \* sh\_audio->samplesize \* 16;   
  
     2.2 如果解码器设置了audio\_out\_minsize，解码可以等价于   
     while (output\_len < target\_len) output\_len += audio\_out\_minsize;   
     因此我们必需保证a\_buffer\_size大于我们需要的解码数据长度加上audio\_out\_minsize，所以我们最大解码长度也就有了如下的限制：(是不是有点绕口？~~)   
     int max\_decode\_len = sh\_audio->a\_buffer\_size - sh\_audio->audio\_out\_minsize;   
  
     2.3 如果a\_out\_buffer中没有我们需要的足够多的解码后数据，我们当然要继续解码撒~ 只不过我们要注意，   
a\_out\_buffer中的数据是经过filter过滤了的(长度发生了变化)，这里会有一个过滤系数，我们反方向求过滤前的数据长度，所以要除以这个系数。   
     while (sh\_audio->a\_out\_buffer\_len < minlen) {   
         int declen = (minlen - sh\_audio->a\_out\_buffer\_len) / filter\_multiplier   
             + (unitsize << 5); // some extra for possible filter buffering   
         declen -= declen % unitsize;   
         if (filter\_n\_bytes(sh\_audio, declen) < 0)   
             return -1;   
     }   
  
  
     3.我们来看看这个filter\_n\_bytes函数   
   static int filter\_n\_bytes(sh\_audio\_t \*sh, int len)   
  
     3.1 还记得我们刚才说的那个最大解码长度吗？ 这里是要确保不会发生解码后数据溢出。   
  
     assert(len-1 + sh->audio\_out\_minsize <= sh->a\_buffer\_size);   
  
     3.2 按需要解码更多的数据   
     while (sh->a\_buffer\_len < len) {   
         unsigned char \*buf = sh->a\_buffer + sh->a\_buffer\_len;   
         int minlen = len - sh->a\_buffer\_len;   
         int maxlen = sh->a\_buffer\_size - sh->a\_buffer\_len;   
         //这里才是调用之前确定的音频解码器进行真正音频解码   
         int ret = sh->ad\_driver->decode\_audio(sh, buf, minlen, maxlen);   
         if (ret <= 0) {   
             error = -1;   
             len = sh->a\_buffer\_len;   
             break;   
         }   
         //解码之后a\_buffer\_len增加   
         sh->a\_buffer\_len += ret;   
     }   
  
     3.3 在前面我们提到过过滤这个步骤，下面的图描述了整个过程   
     filter\_output = af\_play(sh->afilter, &filter\_input);   
     注：这里实际上做的是过滤这部分的工作   
     ------------         ----------           ------------   
     |             |   解码 |           |   过滤   |             |   播放   
     | 未解码数据 | ----->|解码后数据| ------>| 播放的数据 | ------>   
     | a\_in\_buffer|       | a\_buffer |         |a\_out\_buffer|   
     ------------         ----------           ------------   
  
     3.4 if (sh->a\_out\_buffer\_size < sh->a\_out\_buffer\_len + filter\_output->len)   
     注：如果过滤后的数据长度太长，需要扩展a\_out\_buffer\_size   
  
     3.5 将过滤后的数据从decoder buffer移除:   
     sh->a\_buffer\_len -= len;   
     memmove(sh->a\_buffer, sh->a\_buffer + len, sh->a\_buffer\_len);   
  
四.结语   
     回顾一下今天的内容，我们从sh\_audio\_t结构体出发，依次分析了fill\_audio\_out\_buffers()函数，   
decode\_audio()函数，filter\_n\_bytes()函数，但是并没有分析具体的decode和play函数。   
     顺便说点题外话，看Mplayer的代码，结构化模块化的特点很突出，通常是先定义一组   
接口，然后具体的程序去实现这些接口，这样子对代码的维护和扩展都很有好处~

这里再补充一下Mplayer的目录结构和子文件夹说明：

原文地址：<http://qzone.qq.com/blog/81182980-1235106011>

libavcodec libavformat libavutil三个文件夹来自ffmpeg的库 ；  
libfaad2 libao2 liba52 libmpg2 mp3lib vidix几个文件夹是其它的三方库 ；  
libmpcodecs libmpdemux 文件夹中为mplayer 的 demux 和codecs。 ；  
其中demux\_XXX.c为处理各种不同的container.  
vd\_XXX.c为mplayer的内置视频解码器，ad\_xxx.c是音频解码器。  
vf\_XXX.c 和af\_XXX.c 为视，音频的各种filter。  
vo\_XXX.c 为解码后视频输出。  
libvo 视频输出模块的 Make 文件配置部分头  
libao2 音频输出模块的 Make 文件配置部分头  
drivers    使用 VIDIX 技术用到的直接硬件访问驱动程序

怎么样，还是很注重条理的吧？不过，vo\_XXX.c是在libvo下