ffmpeg 教程

作者：gavin 发表于：2012 年 05 月 17 日 http://lingavin.com/tag/ffmpeg

教程原地址为：http://dranger.com/ffmpeg，本人只是做了翻译和部分接口更新工作，保证其能正常工作。需要做一些预备工作：sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev libavdevice-dev libavfilter-dev libavutil-dev libpostproc-dev

教程一：视频截图（Tutorial 01: Making Screencaps）

首先我们需要了解视频文件的一些基本概念，视频文件本身被称作容器，例如 avi 或者是 quicktime，容器的类型确定了文件的信息。然后，容器里装的东西叫流（stream），通常包括视频流和音频流（“流”的意思其实就是“随着时间推移的一段连续的数据元素”）。流中的数据元素叫做“帧”。每个流由不同的编解码器来编码，编解码器定义了数据如何编码（COded）和解码（DECoded），所以叫做编解码器（CODEC）。编解码器的例子有 Divx 和 mp3。包（Packets），是从流中读取的，通过解码器解包，得到原始的帧，我们就可以对这些数据进行播放等的处理。对于我们来说，每个包包含完整的帧，或者多个音频帧。

在初级的水平，处理音视频流是非常简单的：

1. 从 video.avi 中获得视频流
2. 从视频流中解包得到帧
3. 如果帧不完整，重复第 2 步
4. 对帧进行相关操作
5. 重复第 2 步

用 ffmpeg 来处理多媒体就像上面的步骤那么简单，即使你的第 4 步可能很复杂。所以在本教程，我们先打开一个视频，读取视频流，获得帧，然后第 4 步是把帧数据存储为 PPM 文件。

打开文件

我们先来看一下怎么打开一个视频文件，首先把头文件包含进来

```
#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include <libswscale/swscale.h>
...
int main(int argc, char *argv[]){
  av_register_all();
```

av_register_all 只需要调用一次，他会注册所有可用的文件格式和编解码库，当文件被打开时他们将自动匹配相应的编解码库。如果你愿意，可以只注册个别的文件格式和编解码库。

现在真正要打开一个文件了：

```
AVFormatContext *pFormatCtx;
if(av_open_input_file(&pFormatCtx,argv[1],NULL,0,NULL)!=0)
  return -1;
```

从传入的第一个参数获得文件路径，这个函数会读取文件头信息，并把信息保存在 pFormatCtx 结构体当中。这个函数后面三个参数分别是：指定文件格式、缓存大小和格式化选项，当我们设置为 NULL 或 0 时，libavformat 会自动完成这些工作。

这个函数仅仅是获得了头信息，接下来我们要得到流信息：

```
if(av_find_steam_info(pFormatCtx)<0)
  return -1
```

这个函数填充了 pFormatCtx->streams 流信息，可以通过 dump_format 把信息打印出来：

```
dump_format(pFormatCtx, 0, argv[1], 0);
```

pFromatCtx->streams 只是大小为 pFormateCtx->nb_streams 的一系列的点，我们要从中得到视频流：

```
int i;
```

```
AVCodecContext *pCodecCtx;
// Find the first video stream
videoStream=-1;
for(i=0; i<pFormatCtx->nb_streams; i++)
  if(pFormatCtx->streams[i]->codec->codec_type==CODEC_TYPE_VIDEO) {
    videoStream=i;
    break;
  }
if(videoStream==-1)
  return -1; // Didn't find a video stream

// Get a pointer to the codec context for the video stream
pCodecCtx=pFormatCtx->streams[videoStream]->codec;
```

pCodecCtx 包含了这个流在用的编解码的所有信息，但我们仍需要通过他获得特定的解码器然后打开他。

```
AVCodec *pCodec;
pCodec=avcodec_find_decoder(pCodecCtx->codec_id);
if(pCodec==NULL) {
  fprintf(stderr, "Unsupported codec!\n");
  return -1; // Codec not found
}
// Open codec
if(avcodec_open(pCodecCtx, pCodec)<0)
  return -1; // Could not open codec
```

存储数据

现在我们需要一个地方来存储一帧：

```
AVFrame *pFrame;
pFrame=avcodec_alloc_frame();
```

我们计划存储的 PPM 文件，其存储的数据是 24 位 RGB，我们需要把得到的一帧从本地格式转换为 RGB，ffmpeg 可以帮我们完成这个工作。在很多工程里，我们都希望把原始帧转换到特定格式。现在就让我们来完成这个工作吧。

```
AVFrame *pFrameRGB;
pFrameRGB=avcodec_alloc_frame();
if(pFrameRGB==NULL)
  return -1;
```

即使分配了帧空间，我们仍然需要空间来存放转换时的 raw 数据，我们用 avpicture_get_size 来得到需要的空间，然后手动分配。

```
uint8_t *buffer;
int numBytes;
numBytes=avpicture_get_size(PIX_FMT_RGB24, pCodecCtx->width, pCodecCtx->height);
buffer=(uint8_t *)av_malloc(numBytes*sizeof(uint8_t));
```

av_malloc 是 ffmpeg 简单封装的一个分配函数，意在确保内存地址的对齐等，它不会保护内存泄漏、二次释放或其他 malloc 问题。

现在，我们使用 avpicture_fill 来关联新分配的缓冲区的帧。AVPicture 结构体是 AVFrame 结构体的一个子集，开始的 AVFrame 是和 AVPicture 相同的。

```
// Assign appropriate parts of buffer to image planes in pFrameRGB
// Note that pFrameRGB is an AVFrame, but AVFrame is a superset of AVPicture
```

```
avpicture_fill((AVPicture *)pFrameRGB, buffer, PIX_FMT_RGB24, pCodecCtx->width, pCodecCtx->height);
```

下一步我们准备读取流了！

读取数据

我们要做的是通过包来读取整个视频流，然后解码到帧当中，一但一帧完成了，将转换并保存它（这里跟教程的接口调用有不一样的地方）。

```
int frameFinished;
AVPacket packet;
i=0;
while(av_read_frame(pFormatCtx, &packet)>=0) {
  // Is this a packet from the video stream?
  if(packet.stream_index==videoStream) {
    // Decode video frame
      int result;
      avcodec_decode_video2(pCodecCtx,pFrame,&frameFinished, &packet);

    // Did we get a video frame?
    if(frameFinished) {
    //    Convert the image from its native format to RGB
    img_convert_ctx = sws_getContext(pCodecCtx->width, pCodecCtx->height, pCodecCtx->pix_fmt, pCodecCtx->width,
                                          pCodecCtx->height, PIX_FMT_RGB24, SWS_BICUBIC,NULL, NULL,NULL);
    result = sws_scale(img_convert_ctx, (const uint8_t* const*)pFrame->data, pFrame->linesize,
                0, pCodecCtx->height, pFrameRGB->data, pFrameRGB->linesize);
    printf("get result is %d~~~~~~~~~\n",result);
    // Save the frame to disk
    printf("i is %d \n",i);
    if(++i<=5)
      SaveFrame(pFrameRGB, pCodecCtx->width, pCodecCtx->height, i);
      }
    }

   // Free the packet that was allocated by av_read_frame
   av_free_packet(&packet);
}
```

现在需要做的事情就是写 SaveFrame 函数来保存数据到 PPM 文件。

```
void SaveFrame(AVFrame *pFrame, int width, int height, int iFrame) {
  FILE *pFile;
  char szFilename[32];
  int  y;
  printf("start sws_scale\n");
  // Open file
  sprintf(szFilename, "frame%d.ppm", iFrame);
  pFile=fopen(szFilename, "wb");
  if(pFile==NULL){
    printf("pFile is null");
    return;
```

```
    }

    // Write header
    fprintf(pFile, "P6\n%d %d\n255\n", width, height);

    // Write pixel data
    for(y=0; y<height; y++)
      fwrite(pFrame->data[0]+y*pFrame->linesize[0], 1, width*3, pFile);

    // Close file
    fclose(pFile);
  }
```

我们做了一些标准文件打开，然后写 RGB 数据，一次写一行文件，PPM 文件就是简单地把 RGB 信息保存为一长串。头部记录着宽和高，和 RGB 的最大尺寸。

现在回到 main 函数，读完视频流后，我们需要释放一切：

```
    // Free the RGB image
    av_free(buffer);
    av_free(pFrameRGB);

    // Free the YUV frame
    av_free(pFrame);

    // Close the codec
    avcodec_close(pCodecCtx);

    // Close the video file
    av_close_input_file(pFormatCtx);

    return 0;
```

这些就是全部代码来，现在你需要编译和运行

```
gcc -o tutorial01 tutorial01.c -lavformat -lavcodec -lswscale -lz
```

得到 tutorial01，执行以下语句可得到同级目录下的 5 个 PPM 文件

```
./tutorial01 hello.mp4
```

教程二：输出到屏幕（Tutorial 02: Outputting to the Screen）

SDL 与视频

我们使用 SDL 来把视频输出到屏幕。SDL 也就是 Simple Direct Layer，它是多媒体里一个非常棒的跨平台库，在很多项目中都有使用到。可以从官方网站获得库文件和相关文档，在里面看到中文的介绍文档。其实也可以使用 apt-get 来安装库和相应的头文件，如：sudo apt-get install libsdl1.2-dev

SDL 提供了很多把图画画到屏幕上的方法，而且有特别为视频播放到屏幕的组件，叫做 YUV 层，YUV(技术上叫 YCbCr)是一种像 RGB 格式一样的存储 原始图片的方法，粗略地说，Y 是亮度分量，U 和 V 是颜色分量(它比 RGB 复杂，因为一些颜色信息可能会被丢弃，2 个 Y 样本可能只有 1 个 U 样本和 1 个 V 样 本)。SDL 的 YUV 层放置一组 YUV 数据并将它们显示出来，它支持 4 种 YUV 格式，但显示 YV12 最快，另一种 YUV 格式 YUV420P 与 YV12 一 样，除非 U 和 V 阵列互换了。420 的意思是其二次采样比例为 4:2:0，基本的意思是 4 个亮度分量对应 1 个颜色分量，所以颜色分量是四等分的。这是节省带宽的一

种很好的方法，基于人类对与这种变化不敏感。“P”的意思是该格式是“planar”，简单来说就是 YUV 分别在单独的数组中。ffmpeg 可以把图像转换为 YUV420P，现在很多视频流格式已经是它了，或者很容易就能转换成这种格式。

那么，现在我们的计划是把教程 1 的 SaveFrame 函数替换掉，换成在屏幕中显示我们的视频，但是，首先需要了解怎么使用 SDL 库，第一步是包含头文件和初始化 SDL。

```
#include <SDL/SDL.h>
#include <SDL/SDL_thread.h>
if(SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO | SDL_INIT_TIMER)){
    fprintf(stderr, "Could not initialize SDL - %s\n", SDL_GetError());
    exit(1);
}
```

SDL_Init 本质上是告诉库我们需要使用什么功能。SDL_GetError 是一个手工除错函数。

创建显示画面

现在需要在屏幕某个区域上放上一些东西，SDL 里显示图像的区域叫做 surface：

```
SDL_Surface *screen;
screen = SDL_SetVideoMode(pCodecCtx->width, pCodecCtx->height,0,0);
if(!screen){
    fprintf(stderr, "SDL: could not set video mode - exiting\n");
    exit(1);
}
```

这就创建了一个给定长和宽的屏幕，下一个参数是屏幕的颜色深度--0 表示使用当前屏幕的颜色深度。

现在我们在屏幕创建了一个 YUV overlay，可以把视频放进去了。

```
SDL_Overlay *bmp;
bmp = SDL_CreateYUVOverlay(pCodecCtx->width, pCodecCtx->height, SDL_YV12_OVERLAY, screen);
```

就像之前说的那样，用 YV12 来显示图像。

播放图像

这些已经足够简单，现在只要播放图像就好了。让我们来看一下是如何处理完成后的帧的。我们可以摆脱之前处理 RGB 帧的方式，用播放代码代替之前的 SaveFrame 函数，为了播放图像,需要创建 AVPicture 结构体和设置其指针和初始化 YUV overlay。

```
if(frameFinished){
    SDL_LockYUVOverlay(bmp);
    AVPicture pict;
    pict.data[0] = bmp->pixels[0];
    pict.data[1] = bmp->pixels[2];
    pict.data[2] = bmp->pixels[1];

    pict.linesize[0] = bmp->pitches[0];
    pict.linesize[1] = bmp->pitches[2];
    pict.linesize[2] = bmp->pitches[1];

    // Convert the image into YUV format that SDL uses
    img_convert_ctx = sws_getContext(pCodecCtx->width, pCodecCtx->height, pCodecCtx->pix_fmt,
        pCodecCtx->width, pCodecCtx->height, PIX_FMT_YUV420P, SWS_BICUBIC,NULL, NULL,NULL);
    sws_scale(img_convert_ctx, (const uint8_t* const*)pFrame->data,
            pFrame->linesize, 0, pCodecCtx->height, pict.data, pict.linesize);
```

```
        SDL_UnlockYUVOverlay(bmp);
    }
```

首先要把图层锁住，因为我们要往上面写东西，这是一个避免以后发现问题的好习惯。就像前面所展示那样，AVPicture 结构体有一个数据指针指向一个有四个元素的数据指针，因为我们处理的 YUV420P 只有三通道，所以只要设置三组数据。其他格式可能有第四组数据来存储 alpha 值或者其他东西。linesize 就像它名字，在 YUV 层中 lineszie 与 pitches 相同（pitches 是在 SDL 里用来表示指定行数据宽度的值），所以把 pict 的 linesize 指向必要的空间地址，那样当我们向 pict 里面写东西时，实际上是写进了 overlay 里面，那里已经分配好了必要的空间。相似地，可以直接从 overlay 里得到 linesize 的信息，转换格式为 YUV420P，之后的动作就像以前一样。

绘制图像

但我们仍然需要告诉 SDL 显示已经放进去的数据，要传入一个表明电影位置、宽度、高度、缩放比例的矩形参数。这样 SDL 就可以用显卡做快速缩放。

```
    SDL_Rect rect;
    rect.x = 0;
    rect.y = 0;
    rect.w = pCodecCtx->width;
    rect.h = pCodecCtx->height;
    SDL_DisplayYUVOverlay(bmp, &rect);
```

现在，影片开始播放了。

让我们来看看 SDL 的另一个特性，事件系统，SDL 被设置为但你点击，鼠标经过或者给它一个信号的时候，它会产生一个事件，程序通过检查这些事件来处理相关的用户输入，程序也可以向 SDL 事件系统发送事件，当用 SDL 来编写多任务程序的时候特别有用，我们将会在教程 4 里面领略。在这个程序中，我们会处理完包后轮换事件（将处理 SDL_QUIT 以便于程序结束）。

```
    SDL_Event event;
    av_free_packet(&packet);
    SDL_PollEvent(&event);
    switch(event.type) {
    case SDL_QUIT:
        SDL_Quit();
        exit(0);
        break;
    default:
        break;
    }
```

让我们去掉旧的代码开始编译，首先执行：sdl-config --cflags --libs

再开始编译代码：gcc -o tutorial02 tutorial02.c -lavutil -lavformat -lavcodec -lswscale -lSDL -lz –lm

教程三：播放声音（Tutorial 03: Playing Sound）

音频

现在我们想播放音乐。SDL 同样提供输出声音的方法，SDL_OpenAudio()函数用来打开音频设备，它用 SDL_AudioSpec 作为结构体，包含了所有我们需要的音频信息。

在展示如何建立这些东西之前，首先解析一下计算机是如何处理音频的。数码音频由一长串采样流组成。每个样本值代表声音波形的一个数值。声音按照一个特定的采样率被记录着，简单来说就采样率是以多快的速度来播放每个采样，也即是每秒钟记录多少个采样点。例如采样率为 22050 和 44100 频率常用于电台和 CD。此外，大多音频不止一个通道来表示立体声或者环绕，例如，如果采样是立体声的，会同时存入两通道采样信号。当我们从电影里获取数据时，不知道可以得到多少路的采样信号，不会给我们部分采样，也就是说它不会把立体声分开处理。

SDL 播放音频的方法是这样的：你要设置好音频相关的选项，采样率（在 SDL 结构体里面叫做频率“freq”），通道数和其他参数，还设置了一个回调函数和用户数据。当开始播放音频，SDL 会持续地调用回调函数来要求它把声音缓冲数据填充进一个特定数量的字节流里面。当把这些信息写到 SDL_AudioSpec 结构体里面后，调用 SDL_OpenAudio()，它会开启声音设备和返回另一个 AudioSpec 结构体给我们。这个结构体是我们实际用到的，因为我们不能保证我要求什么就得到什么。

设置音乐

先记住上面这些，因为我们还没有关于音频流的相关信息！回到我们之前写的代码，看看是怎么找到视频流，同样也可以用同样的方法找到音频流。

```
// Find the first video stream
videoStream=-1;
audioStream=-1;
for(i=0; i<pFormatCtx->nb_streams; i++) {
  if(pFormatCtx->streams[i]->codec->codec_type==CODEC_TYPE_VIDEO && videoStream < 0) {
    videoStream=i;
  }
  if(pFormatCtx->streams[i]->codec->codec_type==CODEC_TYPE_AUDIO && audioStream < 0) {
    audioStream=i;
  }
}
if(videoStream==-1)
  return -1; // Didn't find a video stream
if(audioStream==-1)
  return -1;
```

现在可以从 AVCodecContext 得到所有我们想要的东西，就像处理视频流那样：

```
AVCodecContext  *aCodecCtx;
aCodecCtx=pFormatCtx->streams[audioStream]->codec;
```

这些编解码内容是建立音频所需要的全部内容：

```
// Set audio settings from codec info
wanted_spec.freq = aCodecCtx->sample_rate;
wanted_spec.format = AUDIO_S16SYS;
wanted_spec.channels = aCodecCtx->channels;
wanted_spec.silence = 0;
wanted_spec.samples = SDL_AUDIO_BUFFER_SIZE;
wanted_spec.callback = audio_callback;
wanted_spec.userdata = aCodecCtx;
if(SDL_OpenAudio(&wanted_spec, &spec) < 0) {
  fprintf(stderr, "SDL_OpenAudio: %s\n", SDL_GetError());
  return -1;
}
```

先来普及一下：

freq：采样率，就像之前解释的那样。

format：这个会告诉 SDL，我们会给它什么格式。“S16SYS”中的“S”是有符号的意思，16 的意思是每个样本是 16 位，“SYS”表示字节顺序按照当前系统的顺序。这些格式是从 avcodec_decode_audio2 得到以来设置到音频输入中。

channels：声音的通道数.

silence：这是用来表示静音的值。因为声音是有符号的，所以静音的值通常为 0。

samples：这个值是音频缓存，它让我们设置当 SDL 请求更多音频数据时我们应该给它多大的数据。其值为 512 到 8192 之间为佳，ffmpeg 用的值是 1024

callback：这是回调函数，这个后面我们会详细讨论。

userdata：SDL 会回调一个回调函数运行的参数。我们将让回调函数得到整个编解码的上下文；你将会在后面知道原因。

最后，我们使用 SDL_OpenAudio 来打开音频。

如果你还记得前面的教程，我们仍然需要打开声音编解码器本身，这是显然的。

```
AVCodec *aCodec;
Codec = avcodec_find_decoder(aCodecCtx->codec_id);
if(!aCodec) {
  fprintf(stderr, "Unsupported codec!\n");
  return -1;
}
avcodec_open(aCodecCtx, aCodec);
```

队列

现在准备开始把音频信息从流里面拿出来。但是我们用这些信息来干什么？我们打算持续地从电影文件里面取出包，但同时 SDL 在调用回调函数！解决方法是建立一些全局结构体，使得到的音频包有地方存放，同时声音回调函数可以从这个地方得到数据！所以接下来要做的事情就是创建一个包的队列。在 ffmpeg 中提供了一个结构体来帮助我们：AVPacketList,实际上只是一个包的链表。下面就是队列结构体：

```
typedef struct PacketQueue {
  AVPacketList *first_pkt, *last_pkt;
  int nb_packets;
  int size;
  SDL_mutex *mutex;
  SDL_cond *cond;
} PacketQueue;
```

首先，我们应当指出 nb_packets 是与 size 不一样的，size 表示从 packet->size 中得到的字节数。你会注意到结构体中有互斥量 mutex 和一个条件变量 cond。这是因为 SDL 是在一个独立的线程中做音频处理的。如果没有正确地锁定这个队列，就可能搞乱数据。我们将看到这个队列是如何运行的。每个程序员都应该知道怎么创建一个队列，但我们会包含这些以至于你可以学习到 SDL 的函数。

首先编写一个函数来初始化队列：

```
void packet_queue_init(PacketQueue *q) {
  memset(q, 0, sizeof(PacketQueue));
  q->mutex = SDL_CreateMutex();
  q->cond = SDL_CreateCond();
}
```

然后编写另外一个函数来把东西放到队列当中：

```
int packet_queue_put(PacketQueue *q, AVPacket *pkt) {
  AVPacketList *pkt1;
  if(av_dup_packet(pkt) < 0) {
    return -1;
  }
  pkt1 = av_malloc(sizeof(AVPacketList));
  if (!pkt1)
    return -1;
  pkt1->pkt = *pkt;
```

```
    pkt1->next = NULL;

    SDL_LockMutex(q->mutex);

    if (!q->last_pkt)
        q->first_pkt = pkt1;
    else
        q->last_pkt->next = pkt1;
    q->last_pkt = pkt1;
    q->nb_packets++;
    q->size += pkt1->pkt.size;
    SDL_CondSignal(q->cond);

    SDL_UnlockMutex(q->mutex);
    return 0;
}
```

SDL_LockMutex()用来锁住队列里的互斥量，这样就可以往队列里面加东西了，然后 SDL_CondSignal()会通过条件变量发送一个信号给接收函数（如果它在等待的话）来告诉它现在已经有数据了，然后解锁互斥量。

下面是相应的接收函数。注意 SDL_CondWait()是如何按照要求让函数阻塞 block 的（例如一直等到队列中有数据）。

```
int quit = 0;
static int packet_queue_get(PacketQueue *q, AVPacket *pkt, int block){
    AVPacketList *pkt1;
    int ret;
    SDL_LockMutex(q->mutex);
    for(;;) {
        if(quit) {
            ret = -1;
            break;
        }
        pkt1 = q->first_pkt;
        if (pkt1) {
            q->first_pkt = pkt1->next;
            if (!q->first_pkt)
        q->last_pkt = NULL;
            q->nb_packets--;
            q->size -= pkt1->pkt.size;
            *pkt = pkt1->pkt;
            av_free(pkt1);
            ret = 1;
            break;
        } else if (!block) {
            ret = 0;
            break;
        } else {
            SDL_CondWait(q->cond, q->mutex);
        }
```

```
    }
    SDL_UnlockMutex(q->mutex);
    return ret;
  }
```

就像你看到的那样，我们已经用一个无限循环包装了这个函数以便用阻塞的方式来得到数据。用 SDL_CondWait()来防止无限循环。基本上，所有的 CondWait 都在等待 SDL_CondSignal() (或者 SDL_CondBroadcast())发来的信号然后继续。然而，虽然看起来是互斥的，如果一直保持着这个锁，put 函数将不能往队列里面放任何东西！但是，SDL_CondWait()同样为我们解锁互斥量，然后当我们得到信号后再次锁上它。

意外情况

你同样注意到有一个全局变量 quit，用它来保证还没有设置程序退出的信号（SDL 会自动处理类似于 TERM 等的信号）。否则，这个线程会永远运行下去，除非用 kill -9 来结束它。ffmpeg 同样提供了一个回调函数用来检测是否需要退出一些被阻塞的函数：这个函数叫做 url_set_interrupt_cb。

```
int decode_interrupt_cb(void) {
  return quit;
}
...
main() {
...
  url_set_interrupt_cb(decode_interrupt_cb);
...
  SDL_PollEvent(&event);
  switch(event.type) {
  case SDL_QUIT:
    quit = 1;
...
```

填充包

剩下来的事情就只有建立队列了：

```
PacketQueue audioq;
main() {
...
  avcodec_open(aCodecCtx, aCodec);

  packet_queue_init(&audioq);
  SDL_PauseAudio(0);
```

SDL_PauseAudio() 最终启动了音频设备。没有数据的时候它是播放静音。

现在，已经建立起队列，并且已经做好了填充数据包的准备。下面就进入读包的循环了：

```
  while(av_read_frame(pFormatCtx, &packet)>=0) {
    // Is this a packet from the video stream?
    if(packet.stream_index==videoStream) {
      // Decode video frame
      ....
      }
    } else if(packet.stream_index==audioStream) {
      packet_queue_put(&audioq, &packet);
```

```
    } else {
      av_free_packet(&packet);
    }
```

要注意的是，把包放进队列之后没有释放它。我们将会在解码之后才会去释放这些包。

取包

现在写 audio_callback 函数来读取队列里面的包，回调函数必须是以下的形式 void callback(void *userdata, Uint8 *stream, int len)，用户数据就是给 SDL 的指针，stream 就是就是将要写音频数据的缓冲区，还有 len 是缓冲区的大小。以下是代码：

```
void audio_callback(void *userdata, Uint8 *stream, int len) {
  AVCodecContext *aCodecCtx = (AVCodecContext *)userdata;
  int len1, audio_size;

  static uint8_t audio_buf[(AVCODEC_MAX_AUDIO_FRAME_SIZE * 3) / 2];
  static unsigned int audio_buf_size = 0;
  static unsigned int audio_buf_index = 0;

  while(len > 0) {
    if(audio_buf_index >= audio_buf_size) {
      /* We have already sent all our data; get more */
      audio_size = audio_decode_frame(aCodecCtx, audio_buf, sizeof(audio_buf));
      if(audio_size < 0) {
        /* If error, output silence */
        audio_buf_size = 1024;
        memset(audio_buf, 0, audio_buf_size);
      } else {
        audio_buf_size = audio_size;
      }
      audio_buf_index = 0;
    }
    len1 = audio_buf_size - audio_buf_index;
    if(len1 > len)
      len1 = len;
    memcpy(stream, (uint8_t *)audio_buf + audio_buf_index, len1);
    len -= len1;
    stream += len1;
    audio_buf_index += len1;
  }
}
```

这个简单的循环会从另一个函数来读取数据，叫做 audio_decode_frame()，把数据存储在一个中间缓冲中，企图将字节转变为流，当我们数据不够的时候提供给我们，当数据塞满时帮我们保存数据以使我们以后再用。这个音频缓冲的大小是 ffmpeg 给我们的音频帧最大值的 1.5 倍，以给我们一个很好的缓冲。

最后，进行音频解码，得到真正的音频数据，audio_decode_frame：

```
int audio_decode_frame(AVCodecContext *aCodecCtx, uint8_t *audio_buf, int buf_size) {
  static AVPacket pkt;
  static uint8_t *audio_pkt_data = NULL;
  static int audio_pkt_size = 0;
```

```
  int len1, data_size;

  for(;;) {
    while(audio_pkt_size > 0) {
      data_size = buf_size;
      len1 = avcodec_decode_audio2(aCodecCtx, (int16_t *)audio_buf, &data_size, audio_pkt_data, audio_pkt_size);
      if(len1 < 0) {/* if error, skip frame */
        audio_pkt_size = 0;
        break;
      }
      audio_pkt_data += len1;
      audio_pkt_size -= len1;
      if(data_size <= 0) {/* No data yet, get more frames */
        continue;
      }
      /* We have data, return it and come back for more later */
      return data_size;
    }
    if(pkt.data)
      av_free_packet(&pkt);

    if(quit) return -1;

    if(packet_queue_get(&audioq, &pkt, 1) < 0) {
      return -1;
    }
    audio_pkt_data = pkt.data;
    audio_pkt_size = pkt.size;
  }
}
```

实际上整个流程开始朝向结束，当调用 packet_queue_get()。我们把包从队列里面拿出来和保存其信息。然后，一但得到一个包就调用 avcodec_decode_audio2()，他的功能就像姊妹函数 avcodec_decode_video()，唯一的区别是：一个包里包含不止一个帧，所以可能要多次调用来解码包中所有的数据。同时记住对 audio_buf 强制转换，因为 SDL 给出的是 8 位缓冲指针而 ffmpeg 给出的数据是 16 位的整型指针。**同时要注意 len1 和 data_size 的差别，len1 表示我们解码使用的数据在包中的大小，data_size 是实际返回的原始声音数据的大小。**

当得到一些数据后，返回来看看是否需要从队列里取得更多数据或者判断是否已完成。如果在进程中有过多数据要处理就保存它以过后才使用。如果我们完成了一个包，我们最后会释放这个包。

就是这样！我们利用主要循环从文件得到音频并送到队列中，然后被 audio_callback 读取，最后把数据送给 SDL，于是 SDL 相当于我们的声卡。编译命令如下：

```
gcc -o tutorial03 tutorial03.c -lavutil -lavformat -lavcodec -lswscale -lSDL -lz -lm
```

视频虽然还是那么快，但音频播放正常。为什么呢？因为音频信息中有采样率，我们尽可能快地填充数据到声卡缓冲中，但是声音设备会按照原来指定的采样率来进行播放。

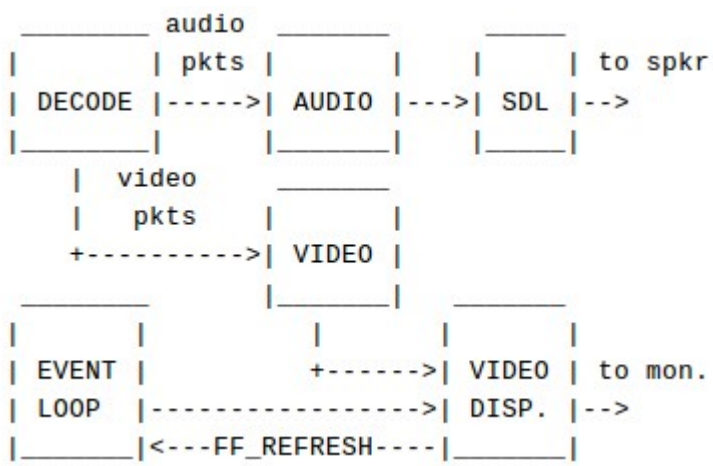
我们几乎已经准备好来开始同步音频和视频了，但首先需要一点程序的组织。用队列的方式来组织和播放音频在一个独立的线程中工作得很好：它使程序更加易于控制和模块化。在开始同步音频和视频之前，需要让代码更容易处理。

教程四：创建线程（Tutorial 04: Spawning Threads）

概要

上一次我们使用 SDL 的函数来达到支持音频播放的效果。每当 SDL 需要音频时它会启动一个线程来调用我们提供的回调函数。现在我们对视频进行同样的处理。这样会使程序更加模块化和跟容易协调工作，尤其是当我们想往代码里面加入同步功能。那么要从哪里开始呢？

首先我们注意到主函数处理太多东西了：它运行着事件循环、读取包和处理视频解码。所以我们将把这些东西分成几个部分：创建一个线程来负责解包；这个包会加入到队列里面，然后由相关的视频或者音频线程来读取这个包。音频线程之前已经按照我们的想法建立好了；由于需要自己来播放视频，因此创建视频线程会有点复杂。我们会把真正播放视频的代码放在主线程。不是仅仅在每次循环时显示视频，而是把视频播放整合到事件循环中。现在的想法是解码视频，把结果保存到另一个队列中，然后创建一个普通事件(FF_REFRESH_EVENT)加入到事件系统中，接着事件循环不断检测这个事件。他将会在这个队列里面播放下一帧。这里有一个图来解释究竟发生了什么事情；



主要目的是通过使用 SDL_Delay 线程的事件驱动来控制视频的移动，可以控制下一帧视频应该在什么时间在屏幕上显示。当我们在下一个教程中添加视频的刷新时间控制代码，就可以使视频速度播放正常了。

简化代码

我们同样会清理一些代码。我们有所有这些视频和音频编解码器的信息，将会加入队列和缓冲和所有其他的东西。所有这些东西都是为了一个逻辑单元，也就是视频。所以创建一个大结构体来装载这些信息，把它叫做 VideoState。

```
typedef struct VideoState {
  AVFormatContext *pFormatCtx;
  int             videoStream, audioStream;
  AVStream        *audio_st;
  PacketQueue     audioq;
  uint8_t         audio_buf[(AVCODEC_MAX_AUDIO_FRAME_SIZE * 3) / 2];
  unsigned int    audio_buf_size;
  unsigned int    audio_buf_index;
  AVPacket        audio_pkt;
  uint8_t         *audio_pkt_data;
  int             audio_pkt_size;
  AVStream        *video_st;
  PacketQueue     videoq;

  VideoPicture    pictq[VIDEO_PICTURE_QUEUE_SIZE];
  int             pictq_size, pictq_rindex, pictq_windex;
  SDL_mutex       *pictq_mutex;
  SDL_cond        *pictq_cond;

  SDL_Thread      *parse_tid;
  SDL_Thread      *video_tid;

  char            filename[1024];
```

```
  int                quit;
} VideoState;
```

让我们来看一下看到了什么。首先，看到基本信息：视频和音频流的格式和参数，和相应的 AVStream 对象。然后看到我们把以下音频缓冲移动到这个结构体里面。这些音频的有关信息（音频缓冲、缓冲大小等）都在附近。我们已经给视频添加了另一个队列，也为解码的帧（保存为 overlay）准备了缓冲（会用来作为队列，不需要一个花哨的队列）。VideoPicture 是我们创造的（会在以后看看里面有什么东西）。同样注意到结构体还分配指针额外创建的线程，退出标志和视频的文件名。

现在回到主函数，看看如何修改代码，首先设置 VideoState 结构体：

```
int main(int argc, char *argv[]) {
  SDL_Event       event;
  VideoState      *is;
  is = av_mallocz(sizeof(VideoState));
```

av_mallocz()函数会申请空间而且初始化为全 0。

然后要初始化为视频缓冲准备的锁（pictq）。因为一旦事件驱动调用视频函数，视频函数会从 pictq 抽出预解码帧。同时，视频解码器会把信息放进去，我们不知道那个动作会先发生。希望你认识到这是一个经典的竞争条件。所以要在开始任何线程前为其分配空间。同时把文件名放到 VideoState 当中。

```
  pstrcpy(is->filename, sizeof(is->filename), argv[1]);
  is->pictq_mutex = SDL_CreateMutex();
  is->pictq_cond = SDL_CreateCond();
```

pstrcpy（已过期）是 ffmpeg 中的一个函数，其对 strncpy 作了一些额外的检测；

第一个线程

让我们启动我们的线程使工作落到实处吧：

```
  schedule_refresh(is, 40);
  is->parse_tid = SDL_CreateThread(decode_thread, is);
  if(!is->parse_tid) {
    av_free(is);
    return -1;
  }
```

schedule_refresh 是一个将要定义的函数。它的动作是告诉系统在某个特定的毫秒数后弹出 FF_REFRESH_EVENT 事件。这将会反过来调用事件队列里的视频刷新函数。但是现在，让我们分析一下 SDL_CreateThread()。

SDL_CreateThread()做的事情是这样的，它生成一个新线程能完全访问原始进程中的内存，启动我们给的线程。它同样会运行用户定义数据的函数。在这种情况下，调用 decode_thread()并与 VideoState 结构体连接。上半部分的函数没什么新东西；它的工作就是打开文件和找到视频流和音频流的索引。唯一不同的地方是把格式内容保存到大结构体中。当找到流后，调用另一个将要定义的函数 stream_component_open()。这是一个一般的分离的方法，自从我们设置很多相似的视频和音频解码的代码，我们通过编写这个函数来重用它们。

stream_component_open()函数的作用是找到解码器，设置音频参数，保存重要信息到大结构体中，然后启动音频和视频线程。我们还会在这里设置一些其他参数，例如指定编码器而不是自动检测等等，下面就是代码：

```
int stream_component_open(VideoState *is, int stream_index) {
  AVFormatContext *pFormatCtx = is->pFormatCtx;
  AVCodecContext *codecCtx;
  AVCodec *codec;
  SDL_AudioSpec wanted_spec, spec;

  if(stream_index < 0 || stream_index >= pFormatCtx->nb_streams) {
    return -1;
  }
```

```
    // Get a pointer to the codec context for the video stream
    codecCtx = pFormatCtx->streams[stream_index]->codec;

    if(codecCtx->codec_type == CODEC_TYPE_AUDIO) {
      // Set audio settings from codec info
      wanted_spec.freq = codecCtx->sample_rate;
      /* .... */
      wanted_spec.callback = audio_callback;
      wanted_spec.userdata = is;

      if(SDL_OpenAudio(&wanted_spec, &spec) < 0) {
        fprintf(stderr, "SDL_OpenAudio: %s\n", SDL_GetError());
        return -1;
      }
    }
    codec = avcodec_find_decoder(codecCtx->codec_id);
    if(!codec || (avcodec_open(codecCtx, codec) < 0)) {
      fprintf(stderr, "Unsupported codec!\n");
      return -1;
    }

    switch(codecCtx->codec_type) {
    case CODEC_TYPE_AUDIO:
      is->audioStream = stream_index;
      is->audio_st = pFormatCtx->streams[stream_index];
      is->audio_buf_size = 0;
      is->audio_buf_index = 0;
      memset(&is->audio_pkt, 0, sizeof(is->audio_pkt));
      packet_queue_init(&is->audioq);
      SDL_PauseAudio(0);
      break;
    case CODEC_TYPE_VIDEO:
      is->videoStream = stream_index;
      is->video_st = pFormatCtx->streams[stream_index];
      packet_queue_init(&is->videoq);
      is->video_tid = SDL_CreateThread(video_thread, is);
      break;
    default:
      break;
    }
  }
```

这跟以前写的代码几乎一样，只不过现在是包括音频和视频。注意到建立了大结构体来作为音频回调的用户数据来代替了 aCodecCtx。同样保存流到 audio_st 和 video_st。像建立音频队列一样，也增加了视频队列。主要是运行视频和音频线程。就像如下：

```
  SDL_PauseAudio(0);
```

```
    break;
    /* ..... */
    is->video_tid = SDL_CreateThread(video_thread, is);
```

还记得之前 SDL_PauseAudio()的作用，还有 SDL_CreateThread()跟以前的用法一样。我们会回到 video_thread()函数。在这之前，让我们回到 decode_thread()函数的下半部分。基本上就是一个循环来读取包和把它放到相应的队列中：

```
    for(;;) {
      if(is->quit) {
        break;
      }
      // seek stuff goes here
      if(is->audioq.size > MAX_AUDIOQ_SIZE || is->videoq.size > MAX_VIDEOQ_SIZE) {
        SDL_Delay(10);
        continue;
      }
      if(av_read_frame(is->pFormatCtx, packet) < 0) {
        if(url_ferror(&pFormatCtx->pb) == 0) {
          SDL_Delay(100); /* no error; wait for user input */
          continue;
        } else {
          break;
        }
      }
      // Is this a packet from the video stream?
      if(packet->stream_index == is->videoStream) {
        packet_queue_put(&is->videoq, packet);
      } else if(packet->stream_index == is->audioStream) {
        packet_queue_put(&is->audioq, packet);
      } else {
        av_free_packet(packet);
      }
    }
```

这里没有新的东西，除了音频和视频队列定义了一个最大值，还有我们加入了检测读取错误的函数。格式内容里面有一个叫做 pb 的 ByteIOContext 结构体。ByteIOContext 是一个保存所有低级文件信息的结构体。url_ferror 检测结构体在读取文件时出现的某些错误。

经过 for 循环，我们等待程序结束或者通知我们已经结束。这些代码指导我们如何推送事件，一些我们以后用来显示视频的东西。

```
    while(!is->quit) {
        SDL_Delay(100);
     }
    fail:
     if(1){
        SDL_Event event;
        event.type = FF_QUIT_EVENT;
        event.user.data1 = is;
        SDL_PushEvent(&event);
     }
```

```
return 0;
```

我们通过 SDL 定义的一个宏来获取用户事件的值。第一个用户事件应该分配给 SDL_USEREVENT，下一个分配给 SDL_USEREVENT + 1，如此类推。FF_QUIT_EVENT 在 SDL_USEREVENT + 2 中定义。如果我们喜欢，我们同样可以传递用户事件，这里把我们的指针传递给了一个大结构体。最后调用 SDL_PushEvent()。在循环分流中，我们只是把 SDL_QUIT_EVENT 部分放进去。我们还会看到事件循环的更多细节；现在，只是保证当推送 FF_QUIT_EVENT 时，会得到它和 quit 值变为 1。

获得帧：视频线程

准备好解码后，开启视频线程。这个线程从视频队列里面读取包，把视频解码为帧，然后调用 queue_picture 函数来把帧放进 picture 队列：

```
int video_thread(void *arg) {
  VideoState *is = (VideoState *)arg;
  AVPacket pkt1, *packet = &pkt1;
  int len1, frameFinished;
  AVFrame *pFrame;
  pFrame = avcodec_alloc_frame();
  for(;;) {
    if(packet_queue_get(&is->videoq, packet, 1) < 0) {
      // means we quit getting packets
      break;
    }
    // Decode video frame
    len1 = avcodec_decode_video(is->video_st->codec, pFrame, &frameFinished,
                                packet->data, packet->size);
    // Did we get a video frame?
    if(frameFinished) {
      if(queue_picture(is, pFrame) < 0) {
        break;
      }
    }
    av_free_packet(packet);
  }
  av_free(pFrame);
  return 0;
}
```

大部分函数在这点上应该是相似的。已经把 avcodec_decode_video 函数移动到这里，只是替换了一些参数；例如，大结构体里面有 AVStream，所以从那里得到编解码器。持续地从视频队列里面取包，知道某人告诉我们该结束或者遇到错误。

帧排队

一起来看看 picture 队列里面用来存储解码帧的函数 pFrame。由于 picture 队列是 SDL overlay（大概是为了视频显示尽量少的计算），需要把转换帧存储在 picture 队列里面的数据是我们生成的：

```
typedef struct VideoPicture {
  SDL_Overlay *bmp;
  int width, height; /* source height & width */
  int allocated;
} VideoPicture;
```

大结构体有缓冲来存储他们。然而，需要自己分配 SDL_Overlay（注意到 allocated 标志用来标示是否已经分配了内存）。

使用这个队列需要两个指针：写索引和读索引。同样记录着缓冲里面实际上有多少图片。为了写队列，第一次要等待缓冲清空以保证有空间存储 VideoPicture。然后检测我们是否为写索引申请了 overlay。如果没有，我们需要申请一些空间。如果窗口的大小改变了，同样需要重新申请缓冲。然而，为了避免锁问题，不会在这里申请（我还不太确定为什么，但应该避免在不同线程调用 SDL overlay 函数）。

```
int queue_picture(VideoState *is, AVFrame *pFrame) {
  VideoPicture *vp;
  int dst_pix_fmt;
  AVPicture pict;
  /* wait until we have space for a new pic */
  SDL_LockMutex(is->pictq_mutex);
  while(is->pictq_size >= VIDEO_PICTURE_QUEUE_SIZE && !is->quit) {
    SDL_CondWait(is->pictq_cond, is->pictq_mutex);
  }
  SDL_UnlockMutex(is->pictq_mutex);

  if(is->quit)
    return -1;
  // windex is set to 0 initially
  vp = &is->pictq[is->pictq_windex];

  /* allocate or resize the buffer! */
  if(!vp->bmp || vp->width != is->video_st->codec->width || vp->height != is->video_st->codec->height) {
    SDL_Event event;

    vp->allocated = 0;
    /* we have to do it in the main thread */
    event.type = FF_ALLOC_EVENT;
    event.user.data1 = is;
    SDL_PushEvent(&event);

    /* wait until we have a picture allocated */
    SDL_LockMutex(is->pictq_mutex);
    while(!vp->allocated && !is->quit) {
      SDL_CondWait(is->pictq_cond, is->pictq_mutex);
    }
    SDL_UnlockMutex(is->pictq_mutex);
    if(is->quit) {
      return -1;
    }
  }
```

当我们想退出时，退出机制就像之前看到的那样处理。已经定义了 FF_ALLOC_EVENT 为 SDL_USEREVENT。推送事件然后等待条件变量分配函数运行。

让我们来看看我们是怎么改变事件循环的：

```
  for(;;) {
    SDL_WaitEvent(&event);
    switch(event.type) {
```

```
    /* ... */
    case FF_ALLOC_EVENT:
      alloc_picture(event.user.data1);
      break;
```

记住 event.user.data1 就是大结构体。这已经足够简单了。让我们来看看 alloc_picture()函数：

```
void alloc_picture(void *userdata) {
  VideoState *is = (VideoState *)userdata;
  VideoPicture *vp;
  vp = &is->pictq[is->pictq_windex];
  if(vp->bmp) {
    // we already have one make another, bigger/smaller
    SDL_FreeYUVOverlay(vp->bmp);
  }
  // Allocate a place to put our YUV image on that screen
  vp->bmp = SDL_CreateYUVOverlay(is->video_st->codec->width, is->video_st->codec->height,
                                 SDL_YV12_OVERLAY, screen);
  vp->width = is->video_st->codec->width;
  vp->height = is->video_st->codec->height;

  SDL_LockMutex(is->pictq_mutex);
  vp->allocated = 1;
  SDL_CondSignal(is->pictq_cond);
  SDL_UnlockMutex(is->pictq_mutex);
}
```

你应该注意到我们已经把 SDL_CreateYUVOverlay 移动到这里。此代码现在应该比较好理解了。记住我们把宽度和高度保存到 VideoPicture 里面，因为由于某些原因不想改变视频的尺寸。

好了，我们解决了所有东西，现在 YUV overlay 已经分配好内存，准备接收图片了。回到 queue_picture 来看看把帧复制到 overlay 当中，你应该记得这部分内容的：

```
int queue_picture(VideoState *is, AVFrame *pFrame) {
  /* Allocate a frame if we need it... */
  /* ... */
  /* We have a place to put our picture on the queue */

  if(vp->bmp) {
    SDL_LockYUVOverlay(vp->bmp);
    dst_pix_fmt = PIX_FMT_YUV420P;
    /* point pict at the queue */

    pict.data[0] = vp->bmp->pixels[0];
    pict.data[1] = vp->bmp->pixels[2];
    pict.data[2] = vp->bmp->pixels[1];

    pict.linesize[0] = vp->bmp->pitches[0];
    pict.linesize[1] = vp->bmp->pitches[2];
    pict.linesize[2] = vp->bmp->pitches[1];
```

```
        // Convert the image into YUV format that SDL uses
        img_convert(&pict, dst_pix_fmt, (AVPicture *)pFrame, is->video_st->codec->pix_fmt,
            is->video_st->codec->width, is->video_st->codec->height);

        SDL_UnlockYUVOverlay(vp->bmp);
        /* now we inform our display thread that we have a pic ready */
        if(++is->pictq_windex == VIDEO_PICTURE_QUEUE_SIZE) {
          is->pictq_windex = 0;
        }
        SDL_LockMutex(is->pictq_mutex);
        is->pictq_size++;
        SDL_UnlockMutex(is->pictq_mutex);
      }
      return 0;
    }
```

这部分的主要功能就是之前所用的简单地把帧填充到 YUV overlay。最后把值加到队列当中。队列的工作是持续添加直到满，和里面有什么就读取什么。因此所有东西都基于 is->pictq_size 这个值，需要锁住它。所以现在工作是增加写指针（有需要的话翻转它），然后锁住队列增加其大小。现在读索引知道队列里面有更多的信息，如果队列满了，写索引会知道的。

播放视频

这就是视频线程！现在已经包裹起所有松散的线程，除了这个，还记得调用 schedule_refresh()函数吗？让我们来看看它实际上做了什么工作：

```
    /* schedule a video refresh in 'delay' ms */
    static void schedule_refresh(VideoState *is, int delay) {
      SDL_AddTimer(delay, sdl_refresh_timer_cb, is);
    }
```

SDL_AddTimer()是一个 SDL 函数，在一个特定的毫秒数里它简单地回调了用户指定函数（可选择携带一些用户数据）。用这个函数来计划视频的更新，每次调用这个函数，它会设定一个时间，然后会触发一个事件，然后主函数会调用函数来从 picture 队列里拉出一帧然后显示它！

不过首先，让我们来触发事件。它会发送：

```
    static Uint32 sdl_refresh_timer_cb(Uint32 interval, void *opaque) {
      SDL_Event event;
      event.type = FF_REFRESH_EVENT;
      event.user.data1 = opaque;
      SDL_PushEvent(&event);
      return 0; /* 0 means stop timer */
    }
```

这里就是相似的事件推送。FF_REFRESH_EVENT 在这里的定义是 SDL_USEREVENT + 1。有一个地方需要注意的是当我们返回 0 时，SDL 会停止计时器，回调将不再起作用。

现在推送 FF_REFRESH_EVENT，我们需要在事件循环中处理它：

```
    for(;;) {
      SDL_WaitEvent(&event);
      switch(event.type) {
      /* ... */
      case FF_REFRESH_EVENT:
        video_refresh_timer(event.user.data1);
```

```
            break;
```

然后调用这个函数，将会把数据从 picture 队列里面拉出来：

```
    void video_refresh_timer(void *userdata) {
        VideoState *is = (VideoState *)userdata;
        VideoPicture *vp;
        if(is->video_st) {
            if(is->pictq_size == 0) {
                schedule_refresh(is, 1);
            } else {
                vp = &is->pictq[is->pictq_rindex];
                /* Timing code goes here */
                schedule_refresh(is, 80);
                video_display(is); /* show the picture! */

                /* update queue for next picture! */
                if(++ is->pictq_rindex == VIDEO_PICTURE_QUEUE_SIZE) {
                    is->pictq_rindex = 0;
                }
                SDL_LockMutex(is->pictq_mutex);
                is->pictq_size --;
                SDL_CondSignal(is->pictq_cond);
                SDL_UnlockMutex(is->pictq_mutex);
            }
        } else {
            schedule_refresh(is, 100);
        }
    }
```

现在，这个函数就非常简单明了了：它会从队列里面拉出数据，设置下一帧播放时间，调用 vidoe_display 来使视频显示到屏幕中，队列计数值加 1，然后减小它的尺寸。你会注意到我们没有对 vp 做任何动作，这里解析为什么：在之后，我们会使用访问时序信息来同步视频和音频。看看那个“这里的时序代码”的地方，我们会找到我们应该以多快的速度来播放视频的下一帧，然后把值传给 schedule_refresh()函数。现在只是设了一个固定值 80。技术上，你可以猜测和检验这个值，然后重编你想看的所有电影，但是：1、过一段时间它会变，2、这是很笨的方法。之后我们会回到这个地方。

我们已经差不多完成了；还剩下最后一样东西要做：播放视频！这里就是视频播放的函数：

```
    void video_display(VideoState *is) {
        SDL_Rect rect;
        VideoPicture *vp;
        AVPicture pict;
        float aspect_ratio;
        int w, h, x, y;
        int i;

        vp = &is->pictq[is->pictq_rindex];
        if(vp->bmp) {
            if(is->video_st->codec->sample_aspect_ratio.num == 0) {
                aspect_ratio = 0;
            } else {
```

```
            aspect_ratio = av_q2d(is->video_st->codec->sample_aspect_ratio) *
        is->video_st->codec->width / is->video_st->codec->height;
        }
        if(aspect_ratio <= 0.0) {
            aspect_ratio = (float)is->video_st->codec->width / (float)is->video_st->codec->height;
        }
        h = screen->h;
        w = ((int)rint(h * aspect_ratio)) & -3;
        if(w > screen->w) {
            w = screen->w;
            h = ((int)rint(w / aspect_ratio)) & -3;
        }
        x = (screen->w - w) / 2;
        y = (screen->h - h) / 2;

        rect.x = x;
        rect.y = y;
        rect.w = w;
        rect.h = h;
        SDL_DisplayYUVOverlay(vp->bmp, &rect);
      }
    }
```

由于屏幕尺寸可能为任何尺寸（我们设置为 640x480，用户可以重新设置尺寸），我们要动态指出需要多大的一个矩形区域。所以首先要指定视频的长宽比，也就是宽除以高的值。一些编解码器会有一个奇样本长宽比，也就是一个像素或者一个样本的宽高比。由于编解码的长宽值是按照像素来计算的，所以实际的宽高比等于样本宽高比某些编解码器的宽高比为 0，表示每个像素的宽高比为 1x1。然后把视频缩放到尽可能大的尺寸。这里的& -3 表示与-3 做与运算，实际上是让他们 4 字节对齐。然后我们把电影居中，然后调用 SDL_DisplayYUVOverlay()。

那么结果怎样？做完了吗？仍然要重写音频代码来使用新的 VideoStruct，但那只是琐碎的改变，你可以参考示例代码。最后需要做的事情是改变 ffmpeg 内部的退出回调函数，变为自己的退出回调函数。

```
VideoState *global_video_state;
int decode_interrupt_cb(void) {
  return (global_video_state && global_video_state->quit);
}
```

在主函数里面设置 global_video_state 这个大结构体。

这就是了！让我们来编译它：

```
sdl-config --cflags --libs
gcc -o tutorial04 tutorial04.c -lavutil -lavformat -lavcodec -lswscale -lSDL -lz -lm
```

享受你的未同步电影吧！下一节我们会使视频播放器真正地工作起来。

教程五：同步视频（Tutorial 05: Synching Video）

视频如何同步

这在这个时间里，我们已经弄好了一个基本上没什么用的视频播放器。它能播放视频，也能播放音频，但它不是通常意义上说的播放器。接下来我们应该怎样做？

PTS 和 DTS

幸运地，音频或视频流都有一些信息告诉我们，它支持以多快的速度去播放：音频流采样率，视频流帧率值。然而，如果单纯地通过帧数乘以帧率来同步视频，可能会使音频失步。作为代替，流里面的包可能会有解码时间戳（DTS）和显示时间戳（PTS）。要搞懂这两个值，你需要知道视频存储的方式。某些格式，例如 MPEG，使用叫做 B 帧的方式（B 表示双向“bidirectional”）。另外两种帧叫做“I”帧和“P”帧（“I”表示关 键"intra"，“P”表示预测“predicted”）。I 帧保存一幅完整的图像，P 帧依赖于前面的 I 帧和 P 帧，并且使用比较或者差分的方式来编码。B 帧与 P 帧类似，但依赖于前面和后面帧信息！这就解释了为什么当我们调用 avcodec_decode_video 后可能没有得到完整的一帧。

假设有一部电影，其帧排列为：I B B P。现在我们在播放 B 帧之前要知道 P 帧的信息。因为这个原因，帧的存储顺序可能是这样的：I P B B。这就是为什么我们会有一个解码时间戳和显示时间戳。解码时间戳告诉我们什么时候需要解码什么，显示时间戳告诉我们什么时候需要显示什么。所以，在这个案例中，流可能是这样的：

```
PTS: 1 4 2 3      // 显示顺序
DTS: 1 2 3 4      // 解码顺序
Stream: I P B B   // 存储顺序
```

通常只有当显示 B 帧的时候 PTS 和 DTS 才会不一样。

当我们从 av_read_frame()得到一个包，包里会包含 PTS 和 DTS 信息。但真正想要的是 PTS 是刚刚解码出来的原始帧的 PTS，这样我们才会知道应该在什么时候显示它。然而 avcodec_decode_video()给我们的帧包含的 AVFrame 没有包含有用的 PTS 信息（警告：AVFrame 包含 PTS 值，但当得到帧的时候并不总是我们需要的）。而且，ffmpeg 重新排序包以便于被 avcodec_decode_video()函数处理的包的 DTS 总是与其返回的 PTS 相同。但是，另一个警告：并不是总能得到这个信息。

不用担心，因为有另外一种方法可以找到帧的 PTS，可以让程序自己来排序包。保存一帧第一个包里面得到的 PTS：这就是整个帧的 PTS。所以当流不给我们提供 DTS 的时候，就使用这个保存了的 PTS。可以通过 avcodec_decode_video()来告诉我们那个是一帧里面的第一个包。怎样实现？每当一个包开始一帧的时候，avcodec_decode_video()会调用一个函数来为一帧申请缓冲。当然，ffmpeg 允许我们重新定义那个分配内存的函数。所以我们会创建一个新的函数来保存一个包的 pts。

当然，尽管可能还是得不到真正的 pts。我们会在后面处理它。

同步

现在，已经知道什么时候显示一个视频帧，但要怎样实现？这里有一个主意：当播放完一帧后，找出下一帧应该在什么时候播放。然后简单地设置一个定时器来重新刷新视频。可能你会想，检查 PTS 的值来而不是用系统时钟来设置延时时间。这种方法可以，但有两个问题需要解决。

首先第一个问题是要知道下一个 PTS 是什么。现在，你可能会想可以把视频速率添加到 PTS 中，这个主意不错。然而，有些电影需要帧重复。这就表示重复播放当前帧。这会使程序显示下一帧太快。所以需要计算它们。

第二个问题是现在视频和音频各自播放，一点不受同步影响。如果一切工作都好的话我们不必担心。但你的电脑可能不太好，或者很多视频文件也 不太好。所以现在有三种选择：音频同步视频，视频同步音频，或者是视频和音频同步到一个外部时钟（例如你的计算机）。从现在起，我们使用视频同步音频的方式。

编程：获得帧的时间戳

现在编写代码来完成这些东西。我们会增加更多成员进我们的大结构体中，但我们会在需要的时候才做这个事情。首先来看看视频线程。记住，就是在这里我们获得从解码线程放进队列里的包。需要做的事情是从 avcodec_decode_video 解出的帧里拿到 PTS。我们讨论的第一种方式是从上次处理的包中得到 DTS，这是很容易的：

```
double pts;
for(;;) {
  if(packet_queue_get(&is->videoq, packet, 1) < 0) {
    // means we quit getting packets
    break;
  }
  pts = 0;
  // Decode video frame
  len1 = avcodec_decode_video(is->video_st->codec, pFrame, &frameFinished, packet->data, packet->size);
```

```
if(packet->dts != AV_NOPTS_VALUE) {
   pts = packet->dts;
} else {
   pts = 0;
}
pts *= av_q2d(is->video_st->time_base);
```

如果得不到 PTS 我们就把它设成 0。

嗯，这很简单。但之前已经说了如果包里面的 DTS 帮助不了我们，我们需要使用帧里第一个包的 PTS。通过告诉 ffmpeg 来使用我们的函数来分配一帧资源来实现。下面就是函数。

```
int get_buffer(struct AVCodecContext *c, AVFrame *pic);
void release_buffer(struct AVCodecContext *c, AVFrame *pic);
```

get 函数不会告诉我们任何关于包的信息，所以每当得到一个包时，需要把其 PTS 存放到一个全局变量里面，然后 get 函数就可以读取到了。然后可以把值存放到 AVFrame 结构体不透明变量中。这是一个用户定义的变量，所以可以任意使用它。首先，这里是我们的函数实现代码：

```
uint64_t global_video_pkt_pts = AV_NOPTS_VALUE;
/* These are called whenever we allocate a frame buffer. We use this to store the global_pts in
* a frame at the time it is allocated. */
int our_get_buffer(struct AVCodecContext *c, AVFrame *pic) {
   int ret = avcodec_default_get_buffer(c, pic);
   uint64_t *pts = av_malloc(sizeof(uint64_t));
   *pts = global_video_pkt_pts;
   pic->opaque = pts;
   return ret;
}
void our_release_buffer(struct AVCodecContext *c, AVFrame *pic) {
   if(pic) av_freep(&pic->opaque);
   avcodec_default_release_buffer(c, pic);
}
```

avcodec_default_get_buffer 和 avcodec_default_release_buffer 是 ffmepg 默认用来分配缓冲的函数。av_freep 是一个内存管理函数，它不仅释放指针指向的内存，还会把指针设置为 NULL。接下来来到打开流的函数 （stream_component_open），我们加上这几行来告诉 ffmpeg 怎么做：

```
codecCtx->get_buffer = our_get_buffer;
codecCtx->release_buffer = our_release_buffer;
```

现在添加代码以达到 PTS 保存到全局变量的目的，那么就可以在需要时使用这个已经存储了的 PTS。代码就像这样：

```
for(;;) {
   if(packet_queue_get(&is->videoq, packet, 1) < 0) {
      // means we quit getting packets
      break;
   }
   pts = 0;
   global_video_pkt_pts = packet->pts;    // Save global pts to be stored in pFrame in first call
   // Decode video frame
   len1 = avcodec_decode_video(is->video_st->codec, pFrame, &frameFinished, packet->data, packet->size);
   if(packet->dts == AV_NOPTS_VALUE && pFrame->opaque && *(uint64_t*)pFrame->opaque != AV_NOPTS_VALUE) {
      pts = *(uint64_t *)pFrame->opaque;
   } else if(packet->dts != AV_NOPTS_VALUE) {
```

```
    pts = packet->dts;
  } else {
    pts = 0;
  }
  pts *= av_q2d(is->video_st->time_base);
```

技术笔记：你可能注意到我们用 int64 来装载 PTS。因为 PTS 以整形的形式来存放。这个时间戳是度量流基本时间单元的时间长度的。例如，如果流每秒钟有 24 帧，那么 PTS 为 42 时表示如果每帧的时间是 24 分之一的话，现在应该播放到 42 帧了（肯定未必是真实的）。

可以通过除以帧率而把 PTS 转换为秒数。time_base 的值其实就是 1/帧率（对于固定帧率来说），所以可以用 PTS 乘 time_base 来得到时间。

编程：使用 PTS 来同步

我们得到了 PTS。现在来解决之前所说的两个同步的问题。定义一个叫做 synchronize_video 的函数来更新同步 PTS。这个函数同样会处理当得不到 PTS 值的情况。同时需要留意什么时候需要播放下一帧以设置刷新率。可以使用一个反映视频已经播放了多长时间的内部值 video_clock 来完成这个工作。把这个值放在大结构体中。

```
typedef struct VideoState {
  double                  video_clock; ///<="" pre="">
```

这里是 synchronize_video 函数，他有很好的注释：

```
double synchronize_video(VideoState *is, AVFrame *src_frame, double pts) {
  double frame_delay;
  if(pts != 0) {
    is->video_clock = pts; /* if we have pts, set video clock to it */
  } else {
    pts = is->video_clock; /* if we aren't given a pts, set it to the clock */
  }
  /* update the video clock */
  frame_delay = av_q2d(is->video_st->codec->time_base);
  /* if we are repeating a frame, adjust clock accordingly */
  frame_delay += src_frame->repeat_pict * (frame_delay * 0.5);
  is->video_clock += frame_delay;
  return pts;
}
```

你会注意到我们会在这个函数里面计算重复帧。

让我们得到正确的帧和用 queue_picture 来队列化帧，添加一个新的时间戳参数 pts：

```
// Did we get a video frame?
if(frameFinished) {
  pts = synchronize_video(is, pFrame, pts);
  if(queue_picture(is, pFrame, pts) < 0) {
    break;
  }
}
```

queue_picture 的唯一改变是把时间戳值 pts 保存到 VideoPicture 结构体中。所以要把 pts 值添加到结构体中并增加一行代码：

```
typedef struct VideoPicture {
  ...
  double pts;
```

```
}
int queue_picture(VideoState *is, AVFrame *pFrame, double pts) {
   ... stuff ...
   if(vp->bmp) {
      ... convert picture ...
      vp->pts = pts;
      ... alert queue ...
   }
```

现在所有图像队列里面的图像都有了正确的时间戳了，就让我们看看视频刷新函数吧。你可能还记得之前用固定值 80ms 来欺骗它。现在要算出正确的值。

我们的策略是通过简单计算前一帧和现在这帧的时间戳的差。同时需要视频同步到音频。将设置音频时钟：一个内部值记录正在播放音频的位置。就像从任意 mp3 播放器中读出来数字一样。由于我们需要视频同步到音频，所以视频线程会使用这个值来计算出播放视频是快了还是慢了。

我们会在之后实现这些代码；现在假设已经有一个可以给我们音频时钟的函数 get_audio_clock。即使我们有了这个值，在视频和音频失步的时候应该怎么办？简单而笨的办法是试着用跳过正确帧或者其他方法来解决。除了这种笨办法，我们会去判断和调整下次刷新的时间值。如果 PTS 太落后于音频时间，我们加陪计算延迟。如果 PTS 太领先于音频时间，应尽量加快刷新时间。现在有了刷新时间或者是延时，我们会和电脑时钟计算出的 frame_timer 做比较。这个 frame timer 会统计出播放电影所有的延时。也就是说，这个 frame timer 告诉我们什么时候要播放下一帧。我们只是简单的给 frame timer 加上延时，然后与系统时钟做比较，然后用那个值来计划下一帧的刷新时间。这可能看起来会有点混乱，一起来细心地学习代码吧：

```
void video_refresh_timer(void *userdata) {
   VideoState *is = (VideoState *)userdata;
   VideoPicture *vp;
   double actual_delay, delay, sync_threshold, ref_clock, diff;
   if(is->video_st) {
      if(is->pictq_size == 0) {
         schedule_refresh(is, 1);
      } else {
         vp = &is->pictq[is->pictq_rindex];
         delay = vp->pts - is->frame_last_pts; /* the pts from last time */
         if(delay <= 0 || delay >= 1.0) {
          delay = is->frame_last_delay; /* if incorrect delay, use previous one */
         }
         /* save for next time */
         is->frame_last_delay = delay;
         is->frame_last_pts = vp->pts;
         /* update delay to sync to audio */
         ref_clock = get_audio_clock(is);
         diff = vp->pts - ref_clock;
         /* Skip or repeat the frame. Take delay into account FFPlay still doesn't "know if this is the best guess." */
         sync_threshold = (delay > AV_SYNC_THRESHOLD) ? delay : AV_SYNC_THRESHOLD;
         if(fabs(diff) < AV_NOSYNC_THRESHOLD) {
            if(diff <= -sync_threshold) {
               delay = 0;
            } else if(diff >= sync_threshold) {
               delay = 2 * delay;
            }
```

```
            }
            is->frame_timer += delay;
            /* computer the REAL delay */
            actual_delay = is->frame_timer - (av_gettime() / 1000000.0);
            if(actual_delay < 0.010) {
               actual_delay = 0.010; /* Really it should skip the picture instead */
            }
            schedule_refresh(is, (int)(actual_delay * 1000 + 0.5));
            /* show the picture! */
            video_display(is);
            /* update queue for next picture! */
            if(++is->pictq_rindex == VIDEO_PICTURE_QUEUE_SIZE) {
              is->pictq_rindex = 0;
            }
            SDL_LockMutex(is->pictq_mutex);
            is->pictq_size--;
            SDL_CondSignal(is->pictq_cond);
            SDL_UnlockMutex(is->pictq_mutex);
          }
        } else {
          schedule_refresh(is, 100);
        }
      }
```

这里我们做了不少检测：首先，确保现在的时间戳和上一个时间戳之间的延时是有效的。如果不是的话我们猜测着使用上次的延时。接着，保证我们有一个同步阀值，因为同步的时候并不总是完美的。ffplay 用的值是 0.01。我们也保证阀值不会比时间戳之间的间隔短。最后，把最小的刷新值设置为 10 毫秒，但我们不会去理会。

往大结构体里面加了一大串值，所以不要忘记去检查代码。同样地，不要忘记在 stream_component_open 里初始化 frame time 和 previous frame delay。

```
    is->frame_timer = (double)av_gettime() / 1000000.0;
    is->frame_last_delay = 40e-3;
```

同步：音频时钟

现在是时候来实现音频时钟了。可以在 audio_decode_frame 函数里面更新时间，也就是做音频解码的地方。现在记住调用这个函数的时候并不总是处理新包，所以要在两个地方更新时钟。一个是获得新包的地方：简单地把包的 PTS 赋值给 audio clock。然后如果一个包有多个帧，通过计算采样数和采样每秒的乘积来获得音频播放的时间。所以一旦得到包：

```
    /* if update, update the audio clock w/pts */
    if(pkt->pts != AV_NOPTS_VALUE) {
      is->audio_clock = av_q2d(is->audio_st->time_base)*pkt->pts;
    }
```

和一旦我们处理这个包：

```
      /* Keep audio_clock up-to-date */
      pts = is->audio_clock;
      *pts_ptr = pts;
      n = 2 * is->audio_st->codec->channels;
      is->audio_clock += (double)data_size / (double)(n * is->audio_st->codec->sample_rate);
```

一些细节：临时函数改变为包含 pts_ptr，所以确保你改变了它。pts_ptr 是一个用来通知 audio_callback 函数当前音频包的时间戳的指针。这个会在下次用来同步音频和视频。

现在可以实现 get_audio_clock 函数了。这不是简单地取得 is->audio_clock 值。注意每次处理它的时候设置 PTS，当如果你看看 audio_callback 函数，它花费了是将来把数据从声音包移动到输出缓冲区中。这意味着在 audio clock 中记录的时间可能会比实际的要早很多，所以需要检查还剩下多少要写入。下面是完整的代码：

```
double get_audio_clock(VideoState *is) {
  double pts;
  int hw_buf_size, bytes_per_sec, n;
  pts = is->audio_clock; /* maintained in the audio thread */
  hw_buf_size = is->audio_buf_size - is->audio_buf_index;
  bytes_per_sec = 0;
  n = is->audio_st->codec->channels * 2;
  if(is->audio_st) {
    bytes_per_sec = is->audio_st->codec->sample_rate * n;
  }
  if(bytes_per_sec) {
    pts -= (double)hw_buf_size / bytes_per_sec;
  }
  return pts;
}
```

你现在应该可以说出为什么这个函数能够工作了。

```
gcc -o tutorial05 tutorial05.c -lavutil -lavformat -lavcodec -lswscale -lSDL -lz -lm`
```

最后，你可以用你自己的视频播放器来看视频了。下一节我们来看看音频同步，然后再下一节讨论查询。

教程六：音频同步（Tutorial 06: Synching Audio）

同步音频

现在我们已经弄了一个比较像样的播放器了，让我们看看还有什么零散的东西需要做的。上一次，我们演示了一点同步的问题，就是同步视频到音频而不是使用其他方式。我们将使用视频一样的做法：做一个内部视频时钟来记录视频线程播放了多久，然后同步到音频上去。之后我们会创建把视频和音频同步到外部时钟。

生成视频时钟

现在我们想像音频时钟那样生成音频时钟：一个给出当前视频播放时间的内部值。首先，你可能会想这和使用上一帧时间戳来更新定时器一样简单。然而，别忘记当我们用毫秒来计算时间的话时间帧可能会很长。解决办法是跟踪另外一个值，我们在设置上一帧时间戳的时候的时间值。那么当前视频时间值就是 PTS_of_last_frame + (current_time - time_elapsed_since_PTS_value_was_set)。这个跟处理 get_audio_clock 时的方法很相似。所以在大结构体中，我们会加入一个双精度浮点 video_current_pts 和 64 位宽整型 video_current_pts_time。更新时间的代 码会放在 video_refresh_timer 函数里面。

```
void video_refresh_timer(void *userdata) {
  /* ... */
  if(is->video_st) {
    if(is->pictq_size == 0) {
      schedule_refresh(is, 1);
    } else {
      vp = &is->pictq[is->pictq_rindex];
      is->video_current_pts = vp->pts;
      is->video_current_pts_time = av_gettime();
```

不要忘记在 stream_component_open 时初始化代码：

```
is->video_current_pts_time = av_gettime();
```

现在我们需要做的事情是获得这些信息。

```
double get_video_clock(VideoState *is) {
  double delta;
  delta = (av_gettime() - is->video_current_pts_time) / 1000000.0;
  return is->video_current_pts + delta;
}
```

提取时钟

但是为什么要强制使用视频时钟呢？我们必须更改视频同步代码以至于音频和视频不会试着相互同步。想象以下我们把它做成像 ffplay 一样有命令行参数。让我们抽象出些东西来：我们将会做一个新的封装函数 get_master_clock，用来检测 av_sync_type 变量然后确定是使用 get_audio_clock 还是 get_video_clock，又或者是我们想使用的其他的时钟，甚至可以使用电脑时钟，这个函数叫做 get_external_clock：

```
enum {
  AV_SYNC_AUDIO_MASTER,
  AV_SYNC_VIDEO_MASTER,
  AV_SYNC_EXTERNAL_MASTER,
};
#define DEFAULT_AV_SYNC_TYPE AV_SYNC_VIDEO_MASTER
double get_master_clock(VideoState *is) {
  if(is->av_sync_type == AV_SYNC_VIDEO_MASTER) {
    return get_video_clock(is);
  } else if(is->av_sync_type == AV_SYNC_AUDIO_MASTER) {
    return get_audio_clock(is);
  } else {
    return get_external_clock(is);
  }
}
main() {
...
  is->av_sync_type = DEFAULT_AV_SYNC_TYPE;
...
}
```

同步音频

现在是最难的部分：音频来同步视频时钟。我们的策略是计算音频的位置，把它和视频时钟做比较，然后计算出需要修正多少的样本数，也就是我们需要丢弃样本来加速或者是通过插值样本的方式来放慢播放？我们将在每次处理声音样本的时候运行一个 synchronize_audio 的函数来正确收缩或者扩展声音样本。然而，我们不想每次发生偏差时都同步，因为处理音频频率比处理视频包频繁。所以我们为 synchronize_audio 设置一个最小连续值来限定需要同步的时刻，这样我们就不用总是在调整了。当然，就像上次那样，**失步**的意思是视频时钟和音频时钟的差别超过了我们设置的阀值。

所以我们使用一个分数系数，叫做 c，然后，现在我们有 N 个失步的音频样本。失去同步的数量可能会有很多的变化，所以我们要计算一下失去同步的长度的平均值。例如，第一次调用显示我们失去同步的值为 40ms，第二次为 50ms 等等。但我们不会去使用一个简单的平均值，因为最近的值比考前的值更重要。所以我们用以个分数系数 c，然后通过以下公式计算：diff_sum = new_diff + diff_sum*c。当我们准备去找平均超以的时候，我们用简单的计算方式：avg_diff = diff_sum*(1-c)。

注意：为什么会在这里？这个公式看来很神奇！它基本剩是一个使用等比级数的加权平均值。想要更多的信息请点击以下两个网址：http://www.dranger.com/ffmpeg/weightedmean.html 或者 html://www.dranger.com/ffmpeg/weightedmean.txt。

以下就是我们的函数：

```
/* Add or subtract samples to get a better sync, return new audio buffer size */
int synchronize_audio(VideoState *is, short *samples, int samples_size, double pts) {
  int n;
  double ref_clock;
  n = 2 * is->audio_st->codec->channels;
  if(is->av_sync_type != AV_SYNC_AUDIO_MASTER) {
    double diff, avg_diff;
    int wanted_size, min_size, max_size, nb_samples;

    ref_clock = get_master_clock(is);
    diff = get_audio_clock(is) - ref_clock;

    if(diff < AV_NOSYNC_THRESHOLD) {
      // accumulate the diffs
      is->audio_diff_cum = diff + is->audio_diff_avg_coef * is->audio_diff_cum;
      if(is->audio_diff_avg_count < AUDIO_DIFF_AVG_NB) {
        is->audio_diff_avg_count++;
      } else {
        avg_diff = is->audio_diff_cum * (1.0 - is->audio_diff_avg_coef);
        /* Shrinking/expanding buffer code.... */
      }
    } else {
      /* difference is TOO big; reset diff stuff */
      is->audio_diff_avg_count = 0;
      is->audio_diff_cum = 0;
    }
  }
  return samples_size;
}
```

我们已经做得很好了；我们已经近似地知道如何用视频或者其他时钟来调整音频了。所以现在来计算以下要添加或者删除多少样本，并且如何在“Shrinking/expanding buffer code”部分来编写代码：

```
if(fabs(avg_diff) >= is->audio_diff_threshold) {
  wanted_size = samples_size +
  ((int)(diff * is->audio_st->codec->sample_rate) * n);
  min_size = samples_size * ((100 - SAMPLE_CORRECTION_PERCENT_MAX) / 100);
  max_size = samples_size * ((100 + SAMPLE_CORRECTION_PERCENT_MAX) / 100);
  if(wanted_size < min_size) {
    wanted_size = min_size;
  } else if (wanted_size > max_size) {
    wanted_size = max_size;
  }
```

记住 audio_length * (sample_rate * # of channels * 2) 是 audio_length 每秒时间的样本数。因此，我们需要的样本数是我们更具声音的偏移添加或者减少后的声音样本数。我们同样可以设置一个范围来限定一次进行修正的长度，因为修正太多，用户会听到刺耳的声音。

修正样本数

现在我们要真正地修正音频。你可能注意到 synchronize_audio 函数返回一个样本大小。所以只需要调整样本数为 wanted_size 就可以了。这样可以使样本值小一些。但是如果想把它变大，我们不能只是让样本的大小变大，因为缓冲里面没有更多的数据。所以我们必须添加它。但是应该怎样添加？最笨的办法是推断声音，所以让我们用已有的数据在缓冲的末尾添加上最后的样本。

```
if(wanted_size < samples_size) {
  /* remove samples */
  samples_size = wanted_size;
} else if(wanted_size > samples_size) {
  uint8_t *samples_end, *q;
  int nb;
  /* add samples by copying final samples */
  nb = (samples_size - wanted_size);
  samples_end = (uint8_t *)samples + samples_size - n;
  q = samples_end + n;
  while(nb > 0) {
    memcpy(q, samples_end, n);
    q += n;
    nb -= n;
  }
  samples_size = wanted_size;
}
```

现在我们返回样本值，那么这个函数的功能已经完成了。我们需要做的东西是使用它。

```
void audio_callback(void *userdata, Uint8 *stream, int len) {
  VideoState *is = (VideoState *)userdata;
  int len1, audio_size;
  double pts;

  while(len > 0) {
    if(is->audio_buf_index >= is->audio_buf_size) {
      /* We have already sent all our data; get more */
      audio_size = audio_decode_frame(is, is->audio_buf, sizeof(is->audio_buf), &pts);
      if(audio_size < 0) {
    /* If error, output silence */
    is->audio_buf_size = 1024;
    memset(is->audio_buf, 0, is->audio_buf_size);
      } else {
    audio_size = synchronize_audio(is, (int16_t *)is->audio_buf, audio_size, pts);
    is->audio_buf_size = audio_size;
```

我们要做的是把函数 synchronize_audio 插入进去（同时，保证初始化了变量）。

结束之前的最后一件事：我们要加一个 if 语句来保证我们不会在视频为主时钟的时候去同步视频。

```
if(is->av_sync_type != AV_SYNC_VIDEO_MASTER) {
  ref_clock = get_master_clock(is);
  diff = vp->pts - ref_clock;
  /* Skip or repeat the frame. Take delay into account FFPlay still doesn't "know if this is the best guess." */
  sync_threshold = (delay > AV_SYNC_THRESHOLD) ? delay : AV_SYNC_THRESHOLD;
```

```
    if(fabs(diff) < AV_NOSYNC_THRESHOLD) {
      if(diff <= -sync_threshold) {
        delay = 0;
      } else if(diff >= sync_threshold) {
        delay = 2 * delay;
      }
    }
```

这样就可以了！确保初始化了所有我没有提到的变量。然后编译它：

```
gcc -o tutorial06 tutorial06.c -lavutil -lavformat -lavcodec -lswscale -lSDL -lz -lm
```

然后你可以运行它。

下次我们要做的是让你可以让电影快退和快进。

教程七：跳转（Tutorial 07: Seeking）

处理 seek 命令

现在要往播放器里面添加查找功能，因为一个播放器不能倒带还真的蛮烦人。再加上这能够展示一下 av_seek_frame 是如何使用的。我们打算设置方向键的左和右的功能是快退和快进 10 秒，上和下的功能是快退快进 60 秒。所以我们需要设置我们的主循环来捕获键值。然而，当我们获得键值时我们不能直接调用 av_seek_frame。我们必须在解码主进程 decode_thread 来处理。所以我们会向大结构体里面添加跳转位置和一些跳转标识：

```
  int             seek_req;
  int             seek_flags;
  int64_t         seek_pos;
```

现在需要在主循环里捕获按键：

```
    for(;;) {
      double incr, pos;
      SDL_WaitEvent(&event);
      switch(event.type) {
      case SDL_KEYDOWN:
      switch(event.key.keysym.sym) {
        case SDLK_LEFT:
          incr = -10.0;
          goto do_seek;
        case SDLK_RIGHT:
          incr = 10.0;
          goto do_seek;
        case SDLK_UP:
          incr = 60.0;
          goto do_seek;
        case SDLK_DOWN:
          incr = -60.0;
          goto do_seek;
        do_seek:
         if(global_video_state) {
          pos = get_master_clock(global_video_state);
          pos += incr;
          stream_seek(global_video_state, (int64_t)(pos * AV_TIME_BASE), incr);
         }
```

```
      break;
      default: break;
    }
    break;
```

为了检测按键，首先需要检查是否有 SDL_KEYDOWN 事件。

然后通过 event.key.keysym.sym 来检测那个按键被按下。一旦知道如何来跳转，通过新的函数 get_master_clock 获得的值加上增加的时间值来计算新时间。然后调用 stream_seek 函数来设置 seek_pos 等的变量。把新的时间转换成为 avcodec 中的内部时间戳单位。记得我们使用帧数而不是用秒数来计算时间戳，其公式为 seconds = frames * time_base(fps)。默认的 avcodec 值是 1,000,000fps(所以 2 秒的时间戳是 2,000,000fps)。我们在后面讨论为什么要转换这个值。这里就是 stream_seek 函数。注意我们设置了一个后退的标志。

```
void stream_seek(VideoState *is, int64_t pos, int rel) {
  if(!is->seek_req) {
    is->seek_pos = pos;
    is->seek_flags = rel < 0 ? AVSEEK_FLAG_BACKWARD : 0;
    is->seek_req = 1;
  }
}
```

让我们来到 decode_thread,这是实现跳转的地方。你会注意到已经标志了一个区域“这里实现跳转”。现在要把代码填到那里。跳转是围绕“av_seek_frame”函数的。这个函数用到一个格式内容，一个流，一个时间戳和一组标记来作为它的参数。这个函数会跳转到你给它的时间戳位置。时间戳的单位是你传递给函数的流的 time_base。然而，你不是必须要传递一个流进去（可以传入-1 代替）。如果你这样做了，time_base 将会使用内部时间戳单位，或者 1000000fps。就是为什么在设置 seek_pos 的时候把位置乘于 AV_TIME_BASE 的原因。

然而，如果传递了-1 给 av_seek_frame，播放某些文件可能会出现问题（几率较少），所以要把第一个流传递给 av_seek_frame。不要忘记还要把时间戳 timestamp 的单位进行转化。

```
if(is->seek_req) {
  int stream_index= -1;
  int64_t seek_target = is->seek_pos;
  if (is->videoStream >= 0) stream_index = is->videoStream;
  else if(is->audioStream >= 0) stream_index = is->audioStream;
  if(stream_index>=0){
    seek_target= av_rescale_q(seek_target, AV_TIME_BASE_Q, pFormatCtx->streams[stream_index]->time_base);
  }
  if(av_seek_frame(is->pFormatCtx, stream_index, seek_target, is->seek_flags) < 0) {
    fprintf(stderr, "%s: error while seeking\n", is->pFormatCtx->filename);
  } else {
     /* handle packet queues... more later... */
```

av_rescale_q(a,b,c) 函数是用来把 timestamp 的时机调整到另一个时机。其基本动作是 a8b/c，这个函数可以防止溢出。AV_TIME_BASE_Q 是 AV_TIME_BASE 作为分母的一个版本。他们是不一样的：AV_TIME_BASE * time_in_seconds = avcodec_timestamp 而 AV_TIME_BASE_Q * avcodec_timestamp = time_in_seconds(但留意 AV_TIME_BASE_Q 实际上是 AVRational 对象，所以需要用 avcodec 里特殊的 q 函数来处理 它)。

清空缓存

已经正确设置了跳转，但还没有结束。记得我们还有一个堆放了一堆包的队列。既然要跳到不同的位置，必须清空队列或者不让电影跳转。不止那样，avcodec 有它自己的缓存，我们还需要每次来清理它。

为了完成上述工作，需要写一个清理包队列的函数。然后，需要一个指导音频和视频线程来清理 avcodec 内部缓存的方法。可以通过在清理后放入一个特殊包的方法来实现它，当他们检测到这个特殊的包后，他们就会清理他们的缓存。

让我们开始编写清理缓存的函数。它比较简单，所以我只是把它显示出来：

```
static void packet_queue_flush(PacketQueue *q) {
  AVPacketList *pkt, *pkt1;
  SDL_LockMutex(q->mutex);
  for(pkt = q->first_pkt; pkt != NULL; pkt = pkt1) {
    pkt1 = pkt->next;
    av_free_packet(&pkt->pkt);
    av_freep(&pkt);
  }
  q->last_pkt = NULL;
  q->first_pkt = NULL;
  q->nb_packets = 0;
  q->size = 0;
  SDL_UnlockMutex(q->mutex);
}
```

现在队列已经清空了，让我们来放入“清空包”。但首先先来定义这个包然后创建它：

```
AVPacket flush_pkt;
main() {
  ...
  av_init_packet(&flush_pkt);
  flush_pkt.data = "FLUSH";
  ...
}
```

现在把这个包放入队列：

```
  } else {
    if(is->audioStream >= 0) {
      packet_queue_flush(&is->audioq);
      packet_queue_put(&is->audioq, &flush_pkt);
    }
    if(is->videoStream >= 0) {
      packet_queue_flush(&is->videoq);
      packet_queue_put(&is->videoq, &flush_pkt);
    }
  }
  is->seek_req = 0;
```

（这些代码片段是上面 decode_thread 片段的延续。)我们同样需要改变 packet_queue_put 以避免特别的清理包的重复。

```
int packet_queue_put(PacketQueue *q, AVPacket *pkt) {
  AVPacketList *pkt1;
  if(pkt != &flush_pkt && av_dup_packet(pkt) < 0) {
    return -1;
  }
```

然后在音频线程和视频线程中，在 packet_queue_get 后立即调用 avcodec_flush_buffers。

```
if(packet_queue_get(&is->audioq, pkt, 1) < 0) {
  return -1;
}
if(packet->data == flush_pkt.data) {
```

```
        avcodec_flush_buffers(is->audio_st->codec);
        continue;
    }
```

上面的代码片段与视频线程中的一样，只要把"audio"替换为"video"。

就是这样了！让我们来编译播放器吧：

```
gcc -o tutorial07 tutorial07.c -lavutil -lavformat -lavcodec -lswscale -lSDL -lz -lm
```