FFPLAY 的原理

目录

概要

电影文件有很多基本的组成部分。首先，文件本身被称为容器 Container，容器的类型决定了信息被存放在文件中的位置。AVI 和 Quicktime 就是容器的例子。接着，你有一组流，例如，你经常有的是一个音频流和一个视频流。(一个流只是一种想像出来的词语，用来表示一连串的通过时间来串连的数据元素)。在流中的数据元素被称为帧 Frame。每个流是由不同的编码 器来编码生成的。编解码器描述了实际的数据是如何被编码 Coded 和解码 DECoded 的，因此它的名字叫做 CODEC。Divx 和 MP3 就是编解码器的例子。接着从流中被读出来的叫做包 Packets。包是一段数据，它包含了一段可以被解码成方便我们最后在应用程序中操作的原始帧的数据。根据我们的目的，每个包包含了完整的帧或者对于音频来说是许多格式的完整帧。

基本上来说，处理视频和音频流是很容易的：

10 从 video.avi 文件中打开视频流 video_stream
20 从视频流中读取包到帧中
30 如果这个帧还不完整，跳到 20
40 对这个帧进行一些操作
50 跳回到 20

在这个程序中使用 ffmpeg 来处理多种媒体是相当容易的，虽然很多程序可能在对帧进行操作的时候非常的复杂。因此在这篇指导中，我们将打开一个文件，读取里面的视频流，而且我们对帧的操作将是把这个帧写到一个 PPM 文件中。

打开文件

首先，来看一下我们如何打开一个文件。通过 ffmpeg，你必需先初始化这个库。(注意在某些系统中必需用<ffmpeg/avcodec.h>和<ffmpeg/avformat.h>来替换)

```
#include <avcodec.h>
#include <avformat.h>
...

int main(int argc, charg *argv[])
{
    av_register_all();
```

这里注册了所有的文件格式和编解码器的库，所以它们将被自动的使用在被打开的合适格式的文件上。注意你只需要调用 av_register_all()一次，因此我们在主函数 main()中来调用它。如果你喜欢，也可以只注册特定的格式和编解码器，但是通常你没有必要这样做。

现在我们可以真正的打开文件：

```
AVFormatContext *pFormatCtx;
// Open video file
if(av_open_input_file(&pFormatCtx, argv[1], NULL, 0, NULL)!=0)
    return -1; // Couldn't open file
```

我们通过第一个参数来获得文件名。这个函数读取文件的头部并且把信息保存到我们给的 AVFormatContext 结构体中。最后三个参数用来指定特殊的文件格式，缓冲大小和格式参数，但如果把它们设置为空 NULL 或者 0，libavformat 将自动检测这些参数。

这个函数只是检测了文件的头部，所以接着我们需要检查在文件中的流的信息：

```
// Retrieve stream information
if(av_find_stream_info(pFormatCtx)<0)
    return -1; // Couldn't find stream information
```

这个函数为 pFormatCtx->streams 填充上正确的信息。我们引进一个手工调试的函数来看一下里面有什么：

```
// Dump information about file onto standard error
dump_format(pFormatCtx, 0, argv[1], 0);
```

现在 pFormatCtx->streams 仅仅是一组大小为 pFormatCtx->nb_streams 的指针，所以让我们先跳过它直到我们找到一个视频流。

```
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;
```

流中关于编解码器的信息就是被我们叫做"codec context"（编解码器上下文）的东西。这里面包含了

流中所使用的关于编解码器的所有信息，现在我们有了一个指向他的指针。但是我们必需要找到真正的编解码器并且打开它：

```
AVCodec *pCodec;

// Find the decoder for the video stream
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
```

有些人可能会从旧的指导中记得有两个关于这些代码其它部分：添加 CODEC_FLAG_TRUNCATED 到 pCodecCtx->flags 和添加一个 hack 来粗糙的修正帧率。这两个修正已经不在存在于 ffplay.c 中。因此，我必需假设它们不再必 要。我们移除了那些代码后还有一个需要指出的不同点：pCodecCtx->time_base 现在已经保存了帧率的信息。time_base 是一个结构体，它里面有一个分子和分母 (AVRational)。我们使用分数的方式来表示帧率是因为很多编解码器使用非整数的帧率（例如 NTSC 使用 29.97fps）。

保存数据

现在我们需要找到一个地方来保存帧：

```
AVFrame *pFrame;

// Allocate video frame
pFrame=avcodec_alloc_frame();
```

因为我们准备输出保存 24 位 RGB 色的 PPM 文件，我们必需把帧的格式从原来的转换为 RGB。FFMPEG 将为我们做这些转换。在大多数项目中（包括我们的这个）我们都想把原始的帧转换成一个特定的格式。让我们先为转换来申请一帧的内存。

```
// Allocate an AVFrame structure

pFrameRGB=avcodec_alloc_frame();

if(pFrameRGB==NULL)
```

```
    return -1;
```

即使我们申请了一帧的内存，当转换的时候，我们仍然需要一个地方来放置原始的数据。我们使用 avpicture_get_size 来获得我们需要的大小，然后手工申请内存空间：

```
uint8_t *buffer;
int numBytes;

// Determine required buffer size and allocate buffer
numBytes = avpicture_get_size( PIX_FMT_RGB24, pCodecCtx->width, pCodecCtx->height );
buffer = (uint8_t *)av_malloc( numBytes*sizeof(uint8_t) );
```

av_malloc 是 ffmpeg 的 malloc，用来实现一个简单的 malloc 的包装，这样来保证内存地址是对齐的（4 字节对齐或者 2 字节对齐）。它并不能保护你不被内存泄漏，重复释放或者其它 malloc 的问题所困扰。

现在我们使用 avpicture_fill 来把帧和我们新申请的内存来结合。关于 AVPicture 的结成：AVPicture 结构体是 AVFrame 结构体的子集——AVFrame 结构体的开始部分与 AVPicture 结构体是一样的。

```
// 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 );
```

最后，我们已经准备好来从流中读取数据了。

读取数据

我们将要做的是通过读取包来读取整个视频流，然后把它解码成帧，最好后转换格式并且保存。

```
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
        avcodec_decode_video(pCodecCtx, pFrame, &frameFinished,packet.data, packet.size);
```

```
        // Did we get a video frame
        if(frameFinished)
        {
            // Convert the image from its native format to RGB
            img_convert ( (AVPicture *)pFrameRGB, PIX_FMT_RGB24,(AVPicture*)pFrame,
                        pCodecCtx->pix_fmt, pCodecCtx->width, pCodecCtx->height);

            // Save the frame to disk
            if(++i<=5)
                SaveFrame(pFrameRGB, pCodecCtx->width,pCodecCtx->height, i);

        }
    }

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

这个循环过程是比较简单的：

av_read_frame()读取一个包并且把它保存到 AVPacket 结构体中。注意我们仅仅申请了一个包的结构体,ffmpeg 为我们申请了内部的数据的内存并通过 packet.data 指针来指向它。这些数据可以在后面通过 av_free_packet()来释 放。

avcodec_decode_video()把包转换为帧。然而当解码一个包的时候，我们可能没有得到我们需要的关于帧的信息。因此，当我们得到下一帧的时候，avcodec_decode_video()为我们设置了帧结束标志 frameFinished。

img_convert()来把帧从原始格式（pCodecCtx->pix_fmt）转换成为 RGB 格式。要记住，你可以把一个 AVFrame 结构体的指针转换为 AVPicture 结构体的指针。最后，我们把帧和高度宽度信息传递给我们的 SaveFrame 函数。

关于包 Packets 的注释

从技术上讲一个包可以包含部分或者其它的数据，但是 ffmpeg 的解释器保证了我们得到的包 Packets 包含的要么是完整的要么是多种完整的帧。

现在我们需要做的是让 SaveFrame 函数能把 RGB 信息定稿到一个 PPM 格式的文件中。我们将生成一个简单的 PPM 格式文件，请相信，它是可以工作的。

```
void SaveFrame(AVFrame *pFrame, int width, int height, int iFrame)
{
    FILE *pFile;
    char szFilename[32];
    int y;
```

```
    // Open file
    sprintf(szFilename, "frame%d.ppm", iFrame);
    pFile=fopen(szFilename, "wb");
    if(pFile==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);

}
```

我们做了一些标准的文件打开动作，然后写入 RGB 数据。我们一次向文件写入一行数据。PPM 格式文件的是一种包含一长串的 RGB 数据的文件。如果你了解 HTML 色彩表示的方式，那么它就类似于把每个像素的颜色头对头的展开，就像#ff0000#ff0000....就表示了了个红色的屏幕。（它被保存成 二进制方式并且没有分隔符，但是你自己是知道如何分隔的）。文件的头部表示了图像的宽度和高度以及最大的 RGB 值的大小。

现在，回顾我们的 main()函数。一旦我们开始读取完视频流，我们必需清理一切：

```
// 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;
```

你会注意到我们使用 av_free 来释放我们使用 avcode_alloc_fram 和 av_malloc 来分配的内存。上面的就是代码！下面，我们将使用 Linux 或者其它类似的平台，你将运行：

gcc -o tutorial01 tutorial01.c -lavutil -lavformat -lavcodec -lz -lavutil -lm

如果你使用的是老版本的 ffmpeg，你可以去掉-lavutil 参数：

gcc -o tutorial01 tutorial01.c -lavutil -lavformat -lavcodec -lz -lm

大多数的图像处理函数可以打开 PPM 文件。可以使用一些电影文件来进行测试。

输出到屏幕

SDL 和视频

为了在屏幕上显示，我们将使用 SDL.SDL 是 Simple Direct Layer 的缩写。它是一个出色的多媒体库，适用于多平台，并且被用在许多工程中。你可以从它的官方网站的网址 http://www.libsdl.org/ 上来得到这个库的源代码或者如果有可能的话你可以直接下载开发包到你的操作系统中。按照这个指导，你将需要编译这个库。（剩下的几个指导中也是一样）

SDL 库中有许多种方式来在屏幕上绘制图形，而且它有一个特殊的方式来在屏幕上显示图像——这种方式叫做 YUV 覆盖。YUV（从技术上来讲并不叫 YUV 而是叫做 YCbCr）是一种类似于 RGB 方式的存储原始图像的格式。粗略的讲，Y 是亮度分量，U 和 V 是色度分量。（这种格式比 RGB 复杂的多，因为 很多的颜色信息被丢弃了，而且你可以每 2 个 Y 有 1 个 U 和 1 个 V）。

SDL 的 YUV 覆盖使用一组原始的 YUV 数据并且在屏幕上显示出他们。它可以允许 4 种不 同的 YUV 格式，但是其中的 YV12 是最快的一种。还有一个叫做 YUV420P 的 YUV 格式，它和 YV12 是一样的，除了 U 和 V 分量的位置被调换了以外。 420 意味着它以 4：2：0 的比例进行了二次抽样，基本上就意味着 1 个颜色分量对应着 4 个亮度分量。所以它的色度信息只有原来的 1/4。这是一种节省带宽 的好方式，因为人眼感觉不到这种变化。在名称中的 P 表示这种格式是平面的——简单的说就是 Y，U 和 V 分量分别在不同的数组中。

FFMPEG 可以把图像格式 转换为 YUV420P，但是现在很多视频流的格式已经是 YUV420P 的了或者可以被很容易的转换成 YUV420P 格式。

于是，我们现在计划把指导 1 中的 SaveFrame()函数替换掉，让它直接输出我们的帧到屏幕上去。但一开始我们必需要先看一下如何使用 SDL 库。首先我们必需先包含 SDL 库的头文件并且初始化它。

```
#include <SDL.h>
#include <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()函数告诉了 SDL 库，哪些特性我们将要用到。当然 SDL_GetError()是一个用来手工除错的函数。

创建一个显示

现在我们需要在屏幕上的一个地方放上一些东西。在 SDL 中显示图像的基本区域叫做面 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高度和宽度的屏幕。下一个选项是屏幕的颜色深度——0 表示使用和当前一样的深度。（这个在 OS X 系统上不能正常工作，原因请看源代码）

现在我们在屏幕上来创建一个 YUV 覆盖以便于我们输入视频上去：

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

正如前面我们所说的，我们使用 YV12 来显示图像。

显示图像

前面那些都是很简单的。现在我们需要来显示图像。让我们看一下是如何来处理完成后的帧的。我们将原来对 RGB 处理的方式，并且替换 SaveFrame() 为显示到屏幕上的代码。为了显示到屏幕上，我们将先建立一个 AVPicture 结构体并且设置其数据指针和行尺寸来为我们的 YUV 覆盖服务：

```
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(&pict, PIX_FMT_YUV420P, (AVPicture*)pFrame, pCodecCtx->pix_fmt,
        pCodecCtx->width, pCodecCtx->height);

    SDL_UnlockYUVOverlay(bmp);

}
```

首先，我们锁定这个覆盖，因为我们将要去改写它。这是一个避免以后发生问题的好习惯。正如前面所示的，这个 AVPicture 结构体有一个数据指针指向一个有 4 个元素的指针数据。由于我们处理的是 YUV420P，所以我们只需要 3 个通道即只要三组数据。其它的格式可能需要第四个指针来表示 alpha 通道或者其它参数。行尺寸正如它的名字表示的意义一样。在 YUV 覆盖中相同功能的结构体是像素 pixel 和程度 pitch。（程度 pitch 是在 SDL 里用来表示指定行数据宽度的值）。所以我们现在做的是让我们的覆盖中的 pict.data 中的三个指针有一个指向必要的空间的地址。类似的，我们可以直接从覆盖中得到行尺寸信息。像前面一样我们使用 img_convert 来把格式转换成 PIX_FMT_YUV420P。

绘制图像

但我们仍然需要告诉 SDL 如何来实际显示我们给的数据。我们也会传递一个表明电影位置、宽度、高度和缩放大小的矩形参数给 SDL 的函数。这样，SDL 为我们做缩放并且它可以通过显卡的帮忙来进行快速缩放。

```
SDL_Rect rect;

if(frameFinished)
{

    // Convert the image into YUV format that SDL uses
    img_convert(&pict, PIX_FMT_YUV420P,(AVPicture *)pFrame, pCodecCtx->pix_fmt,
        pCodecCtx->width, pCodecCtx->height);

    SDL_UnlockYUVOverlay(bmp);

    rect.x = 0;
```

```
    rect.y = 0;
    rect.w = pCodecCtx->width;
    rect.h = pCodecCtx->height;

    SDL_DisplayYUVOverlay(bmp, &rect);

}
```

让我们再花一点时间来看一下 SDL 的特性：它的事件驱动系统。SDL 被设置成当你在 SDL 中点击或者移动鼠标或者向它发送一个信号它都将产生一个事件的驱 动方式。如果你的程序想要处理用户输入的话，它就会检测这些事件。你的程序也可以产生事件并且传递给 SDL 事件系统。当使用 SDL 进行多线程编程的时候， 这相当有用，这方面代码我们可以在指导 4 中看到。在这个程序中，我们将在处理完包以后就立即轮询事件。现在而言，我们将处理 SDL_QUIT 事件以便于我们退出：

```
SDL_Event event;

av_free_packet(&packet);
SDL_PollEvent(&event);

switch(event.type)
{
    case SDL_QUIT:
        SDL_Quit();
        exit(0);
        break;
    default:
        break;
}
```

让我们去掉旧的冗余代码，开始编译。如果你使用的是 Linux 或者其变体，使用 SDL 库进行编译的最好方式为：

```
gcc -o tutorial02 tutorial02.c -lavutil -lavformat -lavcodec -lz -lm `sdl-config --cflags --libs`
```

这里的 sdl-config 命令会打印出用于 gcc 编译的包含正确 SDL 库的适当参数。为了进行编译，在你自己的平台你可能需要做的有点不同：请查阅一下 SDL 文档中关于你的系统的那部分。一旦可以编译，就马上运行它。

当运行这个程序的时候会发生什么呢？电影简直跑疯了！实际上，我们只是以我们能从文件中解码帧的最快速度显示了所有的电影的帧。现在我们没有任何代码来计 算出我们什么时候需要显示电影的帧。最后（在指导 5），我们将花足够的时间来探讨同步问题。但一开始我们会先忽略这个，因为我们有更加重要的事情要处理：音频！

播放声音

现在我们要来播放声音。SDL 也为我们准备了输出声音的方法。函数 SDL_OpenAudio()本身就是用来打开声音设备的。它使用一个叫做 SDL_AudioSpec 结构体作为参数，这个结构体中包含了我们将要输出的音频的所有信息。

在我们展示如何建立之前，让我们先解释一下电脑是如何处理音频的。数字音频是由一长串的样本流组成的。每个样本表示声音波形中的一个值。声音按照一个特定 的采样率来进行录制，采样率表示以多快的速度来播放这段样本流，它的表示方式为每秒多少次采样。例如 22050 和 44100 的采样率就是电台和 CD 常用的 采样率。此外，大多音频有不只一个通道来表示立体声或者环绕。例如，如果采样是立体声，那么每次的采样数就为 2 个。当我们从一个电影文件中等到数据的时 候，我们不知道我们将得到多少个样本，但是 ffmpeg 将不会给我们部分的样本——这意味着它将不会把立体声分割开来。

SDL 播放声音的方式是这样的：你先设置声音的选项：采样率（在 SDL 的结构体中被叫做 freq 的表示频率 frequency），声音通道数和其它的参 数，然后我们设置一个回调函数和一些用户数据 userdata。当开始播放音频的时候，SDL 将不断地调用这个回调函数并且要求它来向声音缓冲填入一个特 定的数量的字节。当我们把这些信息放到 SDL_AudioSpec 结构体中后，我们调用函数 SDL_OpenAudio()就会打开声音设备并且给我们送 回另外一个 AudioSpec 结构体。这个结构体是我们实际上用到的——因为我们不能保证得到我们所要求的。

设置音频

目前先把讲的记住，因为我们实际上还没有任何关于声音流的信息。让我们回过头来看一下我们的代码，看我们是如何找到视频流的，同样我们也可以找到声音流。

```
// 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;
```

从这里我们可以从描述流的 AVCodecContext 中得到我们想要的信息，就像我们得到视频流的信息一样。

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

包含在编解码上下文中的所有信息正是我们所需要的用来建立音频的信息：

```
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;
}
```

让我们浏览一下这些：

• freq 前面所讲的采样率

• format 告诉 SDL 我们将要给的格式。在“S16SYS”中的 S 表示有符号的 signed，16 表示每个样本是 16 位长的，SYS 表示大小头的顺序是与使用的系统相同的。这些格式是由 avcodec_decode_audio2 为我们给出来的输入音频的格式。

• channels 声音的通道数

• silence 这是用来表示静音的值。因为声音采样是有符号的，所以 0 当然就是这个值。

• samples 这是当我们想要更多声音的时候，我们想让 SDL 给出来的声音缓冲区的尺寸。一个比较合适的值在 512 到 8192 之间；ffplay 使用 1024。

• callback 这个是我们的回调函数。我们后面将会详细讨论。

• userdata 这个是 SDL 供给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；你将在后面知道原因。

最后，我们使用 SDL_OpenAudio 函数来打开声音。

如果你还记得前面的指导，我们仍然需要打开声音编解码器本身。这是很显然的。

```
AVCodec *aCodec;

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

avcodec_open(aCodecCtx, aCodec);
```

队列

嗯！现在我们已经准备好从流中取出声音信息。但是我们如何来处理这些信息呢？我们将会不断地从文件中得到这些包，但同时 SDL 也将调用回调函数。解决方法 为创建一个全局的结构体变量以便于我们从文件中得到的声音包有地方存放同时也保证 SDL 中的声音回调函数 audio_callback 能从这个地方得到声 音数据。所以我们要做的是创建一个包的队列 queue。在 ffmpeg 中有一个叫 AVPacketList 的结构体可以帮助我们，这个结构体实际是一串包 的链表。下面就是我们的队列结构体：

```
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 表示我们从 packet->size 中得到的字节数。你会注意到我们有一 个互斥量 mutex 和一个条件变量 cond 在结构体里面。这是因为 SDL 是在一个独立的线程中来进行音频处理的。如果我们没有正确的锁定这个队列，我们有 可能把数据搞乱。我们将来看一个这个队列是如何来运行的。每一个程序员应当知道如何来生成的一个队列，但是我们将把这部分也来讨论从而可以学习到 SDL 的 函数。

一开始我们先创建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();
}
```

接着我们再做一个函数来给队列中填入东西：

```
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;
}
```

函数 SDL_LockMutex()锁定队列的互斥量以便于我们向队列中添加东西，然后函数 SDL_CondSignal()通过我们的条件变量为一个接收函数（如果它在等待）发出一个信号来告诉它现在已经有数据了，接着就会解锁互斥量并让队列可以自由访问。

下面是相应的接收函数。注意函数 SDL_CondWait()是如何按照我们的要求让函数阻塞 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;
```

```
}
```

正如你所看到的，我们已经用一个无限循环包装了这个函数以便于我们想用阻塞的方式来得到数据。我们通过使用 SDL 中的函数 SDL_CondWait()来避免无限循环。基本上，所有的 CondWait 只等待从 SDL_CondSignal()函数（或者 SDL_CondBroadcast()函数）中发出的信号，然后再继续执行。然而，虽然看起来我们陷入了我们的互斥体中——如果我们一直保持着这个锁， 我们的函数将永远无法把数据放入到队列中去！但是，SDL_CondWait()函数也为我们做了解锁互斥量的动作然后才尝试着在得到信号后去重新锁定它。

意外情况

你们将会注意到我们有一个全局变量 quit，我们用它来保证还没有设置程序退出的信号（SDL 会自动处理 TERM 类似的信号）。否则，这个线程将不停地运 行直到我们使用 kill -9 来结束程序。FFMPEG 同样也提供了一个函数来进行回调并检查我们是否需要退出一些被阻塞的函数：这个函数就是 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;
        ...
```

当然，这仅仅是用来给 ffmpeg 中的阻塞情况使用的，而不是 SDL 中的。我们还必需要设置 quit 标志为 1。

为队列提供包

剩下的我们唯一需要为队列所做的事就是提供包了：

```
PacketQueue audioq;

main()
{
    ...
    avcodec_open(aCodecCtx, aCodec);
    packet_queue_init(&audioq);
    SDL_PauseAudio(0);
```

函数 SDL_PauseAudio()让音频设备最终开始工作。如果没有立即供给足够的数据，它会播放静音。我们已经建立好我们的队列，现在我们准备为它提供包。先看一下我们的读取包的循环：

```
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);
    }
```

注意：我们没有在把包放到队列里的时候释放它，我们将在解码后来释放它。

取出包

现在，让我们最后让声音回调函数 audio_callback 来从队列中取出包。回调函数的格式必需为 void callback(void *userdata, Uint8 *stream, int len)，这里的 userdata 就是我们给到 SDL 的指针，stream 是

我们要把声音数据写入的缓冲区指针，len 是缓冲区的大小。下面就是代码：

```
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)
        {
            audio_size = audio_decode_frame(aCodecCtx, audio_buf,sizeof(audio_buf));
            if(audio_size < 0)
            {
                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;
    }
}
```

这基本上是一个简单的从另外一个我们将要写的 audio_decode_frame()函数中获取数据的循环，这个循环把结果写入到中间缓冲区，尝试着向流中写入 len 字节并且在我们没有足够的数据的时候会获取更多的数据或者当我们有多余数据的时候保存下来为后面使用。这个 audio_buf 的大小为 1.5 倍的声音帧的大小以便于有一个比较好的缓冲，这个声音帧的大小是 ffmpeg 给出的。

最后解码音频

让我们看一下解码器的真正部分：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)
            {
                audio_pkt_size = 0;
                break;
            }
            audio_pkt_data += len1;
            audio_pkt_size -= len1;
            if(data_size <= 0)
            {
                continue;
            }
            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;
        }
    }
```

整个过程实际上从函数的尾部开始，在这里我们调用了 packet_queue_get()函数。我们从队列中取出包，并且保存它的信息。然后，一旦我们有了可以使用的包，我们就调用函数 avcodec_decode_audio2()，它的功能就像它的姐妹函数 avcodec_decode_video()一样，唯一的区别是它的一个包里可能有不止一个声音帧，所以你可能要调用很多次来解码出包中所有的数据。同时也要记住进行指针 audio_buf 的强制转换，因为 SDL 给出的是 8 位整型缓冲指针而 ffmpeg 给出的数据是 16 位的整型指针。你应该也会注意到 len1 和 data_size 的不同，len1 表示解码使用的数据的在包中的大小，data_size 表示实际返回的原始声音数据的大小。

当我们得到一些数据的时候，我们立刻返回来看一下是否仍然需要从队列中得到更加多的数据或者我们已经完成了。如果我们仍然有更加多的数据要处理，我们把它保存到下一次。如果我们完成了一个包的处理，我们最后要释放它。

就是这样。我们利用主的读取队列循环从文件得到音频并送到队列中，然后被 audio_callback 函数从队列中读取并处理，最后把数据送给 SDL，于是 SDL 就相当于我们的声卡。让我们继续并且编译：

```
gcc -o tutorial03 tutorial03.c -lavutil -lavformat -lavcodec -lz –lm `sdl-config --cflags --libs`
```

啊哈！视频虽然还是像原来那样快，但是声音可以正常播放了。这是为什么呢？因为声音信息中的采样率——虽然我们把声音数据尽可能快的填充到声卡缓冲中，但是声音设备却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，但是首先我们需要的是一点程序的组织。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的很好：它使得程 序更加更加易于控制和模块化。在我们开始同步音视频之前，我们需要让我们的代码更加容易处理。所以下次要讲的是：创建一个线程。

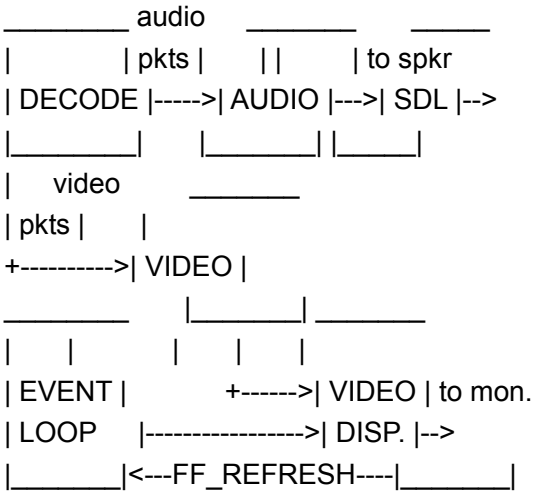
创建线程

Spawning Threads
Overview

Last time we added audio support by taking advantage of SDL's audio functions. SDL started a thread that made callbacks to a function we defined every time it needed audio. Now we're going to do the same sort of thing with the video display. This makes the code more modular and easier to work with - especially when we want to add syncing. So where do we start?

First we notice that our main function is handling an awful lot: it's running through the event loop,

reading in packets, and decoding the video. So what we're going to do is split all those apart: we're going to have a thread that will be responsible for decoding the packets; these packets will then be added to the queue and read by the corresponding audio and video threads. The audio thread we have already set up the way we want it; the video thread will be a little more complicated since we have to display the video ourselves. We will add the actual display code to the main loop. But instead of just displaying video every time we loop, we will integrate the video display into the event loop. The idea is to decode the video, save the resulting frame in another queue, then create a custom event (FF_REFRESH_EVENT) that we add to the event system, then when our event loop sees this event, it will display the next frame in the queue. Here's a handy ASCII art illustration of what is going on:



The main purpose of moving controlling the video display via the event loop is that using an SDL_Delay thread, we can control exactly when the next video frame shows up on the screen. When we finally sync the video in the next tutorial, it will be a simple matter to add the code that will schedule the next video refresh so the right picture is being shown on the screen at the right time.

Simplifying Code

We're also going to clean up the code a bit. We have all this audio and video codec information, and we're going to be adding queues and buffers and who knows what else. All this stuff is for one logical unit, viz. the movie. So we're going to make a large struct that will hold all that information called the 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;
```

Here we see a glimpse of what we're going to get to. First we see the basic information - the format context and the indices of the audio and video stream, and the corresponding AVStream objects. Then we can see that we've moved some of those audio buffers into this structure. These (audio_buf, audio_buf_size, etc.) were all for information about audio that was still lying around (or the lack thereof). We've added another queue for the video, and a buffer (which will be used as a queue; we don't need any fancy queueing stuff for this) for the decoded frames (saved as an overlay). The VideoPicture struct is of our own creations (we'll see what's in it when we come to it). We also notice that we've allocated pointers for the two extra threads we will create, and the quit flag and the filename of the movie.

So now we take it all the way back to the main function to see how this changes our program. Let's set up our VideoState struct:

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

av_mallocz() is a nice function that will allocate memory for us and zero it out.

Then we'll initialize our locks for the display buffer (pictq), because since the event loop calls our display function - the display function, remember, will be pulling pre-decoded frames from pictq. At the same time, our video decoder will be putting information into it - we don't know who will get there first. Hopefully you recognize that this is a classic race condition. So we allocate it now before we start any threads. Let's also copy the filename of our movie into our VideoState.

```
pstrcpy(is->filename, sizeof(is->filename), argv[1]);

is->pictq_mutex = SDL_CreateMutex();
is->pictq_cond = SDL_CreateCond();
```

pstrcpy is a function from ffmpeg that does some extra bounds checking beyond strncpy.

Our First Thread

Now let's finally launch our threads and get the real work done:

```
schedule_refresh(is, 40);

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

schedule_refresh is a function we will define later. What it basically does is tell the system to push a FF_REFRESH_EVENT after the specified number of milliseconds. This will in turn call the video refresh function when we see it in the event queue. But for now, let's look at SDL_CreateThread().

SDL_CreateThread() does just that - it spawns a new thread that has complete access to all the memory of the original process, and starts the thread running on the function we give it. It will also pass that function user-defined data. In this case, we're calling decode_thread() and with our VideoState struct attached. The first half of the function has nothing new; it simply does the work of opening the file and finding the index of the audio and video streams. The only thing we do different is save the format context in our big struct. After we've found our stream indices, we call another function that we will define, stream_component_open(). This is a pretty natural way to split things up, and since we do a lot of similar things to set up the video and audio codec, we reuse some code by making this a function.

The stream_component_open() function is where we will find our codec decoder, set up our audio options, save important information to our big struct, and launch our audio and video threads. This is where we would also insert other options, such as forcing the codec instead of autodetecting it and so forth. Here it is:

```
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;
    }
}
```

This is pretty much the same as the code we had before, except now it's generalized for audio and

video. Notice that instead of aCodecCtx, we've set up our big struct as the userdata for our audio callback. We've also saved the streams themselves as audio_st and video_st. We also have added our video queue and set it up in the same way we set up our audio queue. Most of the point is to launch the video and audio threads. These bits do it:

```
SDL_PauseAudio(0);
break;
/* ..... */
is->video_tid = SDL_CreateThread(video_thread, is);
```

We remember SDL_PauseAudio() from last time, and SDL_CreateThread() is used as in the exact same way as before. We'll get back to our video_thread() function.

Before that, let's go back to the second half of our decode_thread() function. It's basically just a for loop that will read in a packet and put it on the right queue:

```
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);
    }
}
```

这里没有什么新东西，除了我们给音频和视频队列限定了一个最大值并且我们添加一个检测读错误的函数。格式上下文里面有一个叫做 pb 的 ByteIOContext 类型结构体。这个结构体是用来保存一些低级的文件信息。函数 url_ferror 用来检测结构体并发现是否有些读取文件错误。

在循环以后，我们的代码是用等待其余的程序结束和提示我们已经结束的。这些代码是有益的，因为它指示出了如何驱动事件——后面我们将显示影像。

```
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;
```

我们使用 SDL 常量 SDL_USEREVENT 来从用户事件中得到值。第一个用户事件的值应当是 SDL_USEREVENT，下一个是 SDL_USEREVENT＋1 并且依此类推。在我们的程序中 FF_QUIT_EVENT 被定义成 SDL_USEREVENT＋2。如果喜欢，我们也可以 传递用户数据，在这里我们传递的是大结构体的指针。最后我们调用 SDL_PushEvent()函数。在我们的事件分支中，我们只是像以前放入 SDL_QUIT_EVENT 部分一样。我们将在自己的事件队列中详细讨论，现在只是确保我们正确放入了 FF_QUIT_EVENT 事件，我们将在后面捕 捉到它并且设置我们的退出标志 quit。

得到帧：video_thread

当我们准备好解码器后，我们开始视频线程。这个线程从视频队列中读取包，把它解码成视频帧，然后调用 queue_picture 函数把处理好的帧放入到图片队列中:

```
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;
}
```

在这里的很多函数应该很熟悉吧。我们把 avcodec_decode_video 函数移到了这里，替换了一些参数，例如：我们把 AVStream 保存在我 们自己的大结构体中，所以我们可以从那里得到编解码器的信息。我们仅仅是不断的从视频队列中取包一直到有人告诉我们要停止或者出错为止。

把帧队列化

让我们看一下保存解码后的帧 pFrame 到图像队列中去的函数。因为我们的图像队列是 SDL 的覆盖的集合（基本上不用让视频显示函数再做计算了），我们需要把帧转换成相应的格式。我们保存到图像队列

中的数据是我们自己做的一个结构体。

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

我们的大结构体有一个可以保存这些缓冲区。然而，我们需要自己来申请 SDL_Overlay（注意：allocated 标志会指明我们是否已经做了这个申请的动作与否）。

为了使用这个队列，我们有两个指针——写入指针和读取指针。我们也要保证一定数量的实际数据在缓冲中。要写入到队列中，我们先要等待缓冲清空以便于有位置 来保存我们的 VideoPicture。然后我们检查看我们是否已经申请到了一个可以写入覆盖的索引号。如果没有，我们要申请一段空间。我们也要重新申请 缓冲如果窗口的大小已经改变。然而，为了避免被锁定，尽量避免在这里申请（我现在还不太清楚原因；我相信是为了避免在其它线程中调用 SDL 覆盖函数的原 因）。

```
int queue_picture(VideoState *is, AVFrame *pFrame)
{
    VideoPicture *vp;
    int dst_pix_fmt;
    AVPicture pict;

    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];
    if(!vp->bmp ||vp->width != is->video_st->codec->width ||vp->height != is->video_st->codec->height)
    {
        SDL_Event event;

        vp->allocated = 0;
        event.type = FF_ALLOC_EVENT;
        event.user.data1 = is;
```

```
        SDL_PushEvent(&event);
        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;
        }
}
```

这里的事件机制与前面我们想要退出的时候看到的一样。我们已经定义了事件 FF_ALLOC_EVENT 作为 SDL_USEREVENT。我们把事件发到事件队列中然后等待申请内存的函数设置好条件变量。

让我们来看一看如何来修改事件循环：

```
for(;;)
{
    SDL_WaitEvent(&event);
    switch(event.type)
    {
        case FF_ALLOC_EVENT:
            alloc_picture(event.user.data1);
            break;
```

记住 event.user.data1 是我们的大结构体。就这么简单。让我们看一下 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);
}
```

你可以看到我们把 SDL_CreateYUVOverlay 函数从主循环中移到了这里。这段代码应该完全可以自我注释。记住我们把高度和宽度保存到 VideoPicture 结构体中因为我们需要保存我们的视频的大小没有因为某些原因而改变。

好，我们几乎已经全部解决并且可以申请到 YUV 覆盖和准备好接收图像。让我们回顾一下 queue_picture 并看一个拷贝帧到覆盖的代码。你应该能认出其中的一部分：

```
int queue_picture(VideoState *is, AVFrame *pFrame)
{
    if(vp->bmp)
    {
        SDL_LockYUVOverlay(vp->bmp);

        dst_pix_fmt = PIX_FMT_YUV420P;
        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);
        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;
}
```

这部分代码和前面用到的一样，主要是简单的用我们的帧来填充 YUV 覆盖。最后一点只是简单的给队列加 1。这个队列在写的时候会一直写入到满为止，在读的时 候会一直读空为止。因此所有的都依赖于 is->pictq_size 值，这要求我们必需要锁定它。这里我们做的是增加写指针（在必要的时候采用轮转 的方式），然后锁定队列并且增加尺寸。现在我们的读者函数将会知道队列中有了更多的信息，当队列满的时候，我们的写入函数也会知道。

显示视频

这就是我们的视频线程。现在我们看过了几乎所有的线程除了一个——记得我们调用 schedule_refresh()函数吗？让我们看一下实际中是如何做的：

```
static void schedule_refresh(VideoState *is, int delay)
{
    SDL_AddTimer(delay, sdl_refresh_timer_cb, is);
}
```

函数 SDL_AddTimer()是 SDL 中的一个定时（特定的毫秒）执行用户定义的回调函数（可以带一些参数 user data）的简单函数。我们将用这个函数来定时刷新视频——每次我们调用这个函数的时候，它将设置一个定时器来触发定时事件来把一帧从图像队列中显示到屏 幕上。

但是，让我们先触发那个事件。

```
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;
}
```

这里向队列中写入了一个现在很熟悉的事件。FF_REFRESH_EVENT 被定义成 SDL_USEREVENT+1。要注意的一件事是当返回 0 的时候，SDL 停止定时器，于是回调就不会再发生。

现在我们产生了一个 FF_REFRESH_EVENT 事件，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;
```

于是我们就运行到了这个函数，在这个函数中会把数据从图像队列中取出：

```
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];
            schedule_refresh(is, 80);
            video_display(is);
            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，这只是一个极其简单的函数：当队列中有数据的时候，他从其中获得数据，为下一帧设置定时器，调用 video_display 函数来真正显示图像到屏 幕上，然后把队列读索引值加 1，并且把队列的尺寸 size 减 1。你可能会注意到在这个函数中我们并没有真正对 vp 做一些实际的动作，原因是这样的：我们将 在后面处理。我们将在后面同步音频和视频的时候用它来访问时间信息。你会在这里看到这个注释信息“ timing 密码 here ”。那里我们将讨论什么时候显 示下一帧视频，然后把相应的值写入到

schedule_refresh()函数中。现在我们只是随便写入一个值 80。从技术上来讲，你可以猜测并验证这个 值，并且为每个电影重新编译程序，但是：1）过一段时间它会漂移；2）这种方式是很笨的。我们将在后面来讨论它。

我们几乎做完了；我们仅仅剩了最后一件事：显示视频！下面就是 video_display 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);

        }
    }
```

因为我们的屏幕可以是任意尺寸（我们设置为 640x480 并且用户可以自己来改变尺寸），我们需要动态计算出我们显示的图像的矩形大小。所以一开始我们需 要计算出电影的纵横比 aspect ratio，表示方式为宽度除以高度。某些编解码器会有奇数采样纵横比，只是简单表示了一个像素或者一个采样的宽度除以高度的比例。因为宽度和高度在我们 的编解码器中是用像素为单位的，所以实际的纵横比与纵横比乘以样本纵横比相同。某些编解码器会显示纵横比为 0，这表示每个像素的纵横比为 1x1。然后我们 把电影缩放到适合屏幕的尽可能大的尺寸。这里的& -3 表示与-3 做与运算，实际上是让它们 4 字节对齐。然后我们把电影移到中心位置，接着调用 SDL_DisplayYUVOverlay()函数。

结果是什么？我们做完了吗？嗯，我们仍然要重新改写声音部分的代码来使用新的 VideoStruct 结构体，但是那些只是尝试着改变，你可以看一下那些参考示例代码。最后我们要做的是改变 ffmpeg 提供的默认退出回调函数为我们的退出回调函数。

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

我们在主函数中为大结构体设置了 global_video_state。
这就是了！让我们编译它：

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

请享受一下没有经过同步的电影！下次我们将编译一个可以最终工作的电影播放器。

如何同步视频

PTS 和 DTS

幸运的是，音频和视频流都有一些关于以多快速度和什么时间来播放它们的信息在里面。音频流有采样，视频流有每秒的帧率。然而，如果我们只是简单的通过数帧 和乘以帧率的方式来同步视频，那么就很有可能会失去同步。于是作为一种补充，在流中的包有种叫做 DTS（解码时间戳）和 PTS（显示时间戳）的机制。为了 这两个参数，你需要了解电影存放的方式。像 MPEG 等格式，使用被叫做 B 帧（B 表示双向 bidirectional）的方式。另外两种帧被叫做 I 帧和 P 帧 （I 表示关键帧，P 表示预测帧）。I 帧包含了某个特定的完整图像。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 帧有点类似，但 是它是依赖于前面和后面的帧的信息的。这也就解释了为什么我们可能在调用 avcodec_decode_video 以后会得不到一帧图像。

所以对于一个电影，帧是这样来显示的：I B B P。现在我们需要在显示 B 帧之前知道 P 帧中的信息。因此，帧可能会按照这样的方式来存储：IPBB。这就是为什么我们会有一个解码时间戳和一个显示时间戳的原因。解码时间戳告诉我们什么时候需要解码，显示时间戳告诉我们什么时候需要显示。所以，在这种情况下，我们的流可以是这样的：

PTS: 1 4 2 3
DTS: 1 2 3 4
Stream: I P B B

通常 PTS 和 DTS 只有在流中有 B 帧的时候会不同。

当我们调用 av_read_frame()得到一个包的时候，PTS 和 DTS 的信息也会保存在包中。但是我们真正想要的 PTS 是我们刚刚解码出来的原始帧 的 PTS，这样我们才能知道什么时候来显示它。然而，我们从 avcodec_decode_video()函数中得到的帧只是一个 AVFrame，其中并 没有包含有用的 PTS 值（注意：AVFrame 并没有包含时间戳信息，但当我们等到帧的时候并不是我们想要的样子）。然而，ffmpeg 重新排序包以便于 被 avcodec_decode_video()函数处理的包的 DTS 可以总是与其返回的 PTS 相同。但是，另外的一个警告是：我们也并不是总能得到这个 信息。

不用担心，因为有另外一种办法可以找到帧的 PTS，我们可以让程序自己来重新排序包。我们保存一帧的第一个包的 PTS：这将作为整个这一帧的 PTS。我们 可以通过函数 avcodec_decode_video()来计算出哪个包是一帧的第一个包。怎样实现呢？任何时候当一个包开始一帧的时 候，avcodec_decode_video()将调用一个函数来为一帧申请一个缓冲。当然，ffmpeg 允许我们重新定义那个分配内存的函数。所以我 们制作了一个新的函数来保存一个包的时间戳。

当然，尽管那样，我们可能还是得不到一个正确的时间戳。我们将在后面处理这个问题。

同步

现在，知道了什么时候来显示一个视频帧真好，但是我们怎样来实际操作呢？这里有个主意：当我们显示了一帧以后，我们计算出下一帧显示的时间。然后我们简单 的设置一个新的定时器来。你可能会想，我们检查下一帧的 PTS 值而不是系统时钟来看超时是否会到。这种方式可以工作，但是有两种情况要处理。

首先，要知道下一个 PTS 是什么。现在我们能添加视频速率到我们的 PTS 中——太对了！然而，有些电影需要帧重复。这意味着我们重复播放当前的帧。这将导致程序显示下一帧太快了。所以我们需要计算它们。

第二，正如程序现在这样，视频和音频播放很欢快，一点也不受同步的影响。如果一切都工作得很好的话，我们不必担心。但是，你的电脑并不是最好的，很多视频 文件也不是完好的。所以，我们有三种选择：同步音频到视频，同步视频到音频，或者都同步到外部时钟（例如你的电脑时钟）。从现在开始，我们将同步视频到音频。

写代码：获得帧的时间戳

现在让我们到代码中来做这些事情。我们将需要为我们的大结构体添加一些成员，但是我们会根据需要来做。首先，让我们看一下视频线程。记住，在这里我们得到 了解码线程输出到队列中的包。这里我们需要的是从 avcodec_decode_video 函数中得到帧的时间戳。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);
```

如果我们得不到 PTS 就把它设置为 0。

好，那是很容易的。但是我们所说的如果包的 DTS 不能帮到我们，我们需要使用这一帧的第一个包的 PTS。我们通过让 ffmpeg 使用我们自己的申请帧程序来实现。下面的是函数的格式：

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

申请函数没有告诉我们关于包的任何事情，所以我们要自己每次在得到一个包的时候把 PTS 保存到一个全局变量中去。我们自己以读到它。然后，我们把值保存到 AVFrame 结构体难理解的变量中去。所以一开始，这就是我们的函数：

```
uint64_t global_video_pkt_pts = AV_NOPTS_VALUE;
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);
}
```

函数 avcodec_default_get_buffer 和 avcodec_default_release_buffer 是 ffmpeg 中默认的申请缓冲的函数。函数 av_freep 是一个内存管理函数，它不但把内存释放而且把指针设置为 NULL。

现在到了我们流打开的函数（stream_component_open），我们添加这几行来告诉 ffmpeg 如何去做：

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

现在我们必需添加代码来保存 PTS 到全局变量中，然后在需要的时候来使用它。我们的代码现在看起来应该是这样子：

```
for(;;)
{
    if(packet_queue_get(&is->videoq, packet, 1) < 0)
    {
        // means we quit getting packets
        break;
    }
    pts = 0;

    // Save global pts to be stored in pFrame in first call
    global_video_pkt_pts = packet->pts;

    // 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);
```

技术提示：你可能已经注意到我们使用 int64 来表示 PTS。这是因为 PTS 是以整型来保存的。这个值是一个时间戳相当于时间的度量，用来以流的 time_base 为单位进行时间度量。例如，如果一个流是 24 帧每秒，值为 42 的 PTS 表示这一帧应该排在第 42 个帧的位置如果我们每秒有 24 帧（这里 并不完全正确）。

我们可以通过除以帧率来把这个值转化为秒。流中的 time_base 值表示 1/framerate（对于固定帧率来说），所以得到了以秒为单位的 PTS，我们需要乘以 time_base。

写代码：使用 PTS 来同步

现在我们得到了 PTS。我们要注意前面讨论到的两个同步问题。我们将定义一个函数叫做 synchronize_video，它可以更新同步的 PTS。这个函数也能最终处理我们得不到 PTS 的情况。同时我们要知道下一帧的时间以便于正确设置刷新速率。我们可以使用内部的反映当前视频已经播放时间 的时钟 video_clock 来完成这个功能。我们把这些值添加到大结构体中。

```
typedef struct VideoState
{
    double video_clock; ///
```

下面的是函数 synchronize_video，它可以很好的自我注释：

```
double synchronize_video(VideoState *is, AVFrame *src_frame, double pts)
{
    double frame_delay;
    if(pts != 0)
    {
        is->video_clock = pts;
    }
    Else
    {
        pts = is->video_clock;
    }
```

```
    frame_delay = av_q2d(is->video_st->codec->time_base);
    frame_delay += src_frame->repeat_pict * (frame_delay * 0.5);
    is->video_clock += frame_delay;
    return pts;
}
```

你也会注意到我们也计算了重复的帧。
现在让我们得到正确的 PTS 并且使用 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;
    }
}
```

对于 queue_picture 来说唯一改变的事情就是我们把时间戳值 pts 保存到 VideoPicture 结构体中，我们必需添加一个时间戳变量到结构体中并且添加一行代码：

```
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 ...
    }
```

现在我们的图像队列中的所有图像都有了正确的时间戳值，所以让我们看一下视频刷新函数。你会记得上次我们用 80ms 的刷新时间来欺骗它。那么，现在我们将会算出实际的值。

我们的策略是通过简单计算前一帧和现在这一帧的时间戳来预测出下一个时间戳的时间。同时，我们需要同步视频到音频。我们将设置一个音频时间 audio clock；一个内部值记录了我们正在播放的音频的位置。就像从任意的 mp3 播放器中读出来的数字一样。既然我们把视频同步到音频，视频线程使用这个值来 算出是否太快还是太慢。

我们将在后面来实现这些代码；现在我们假设我们已经有一个可以给我们音频时间的函数 get_audio_clock。一旦我们有了这个值，我们在音频和视 频失去同步的时候应该做些什么呢？简单而有点笨的办法是试着用跳过正确帧或者其它的方式来解决。作为一种替代的手段，我们会调整下次刷新的值；如果时间戳 太落后于音频时间，我们加倍计算延迟。如果时间戳太领先于音频时间，我们将尽可能快的刷新。既然我们有了调整过的时间和延迟，我们将把它和我们通过 frame_timer 计算出来的时间进行比较。这个帧时间 frame_timer 将会统计出电影播放中所有的延时。换句话说，这个 frame_timer 就是指我们什么时候来显示下一帧。我们简单的添加新的帧定时器延时，把它和电脑的系统时间进行比较，然后使用那个值来调度下一次刷 新。这可能有点难以理解，所以请认真研究代码：

```
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;
            if(delay <= 0 || delay >= 1.0)
            {
                delay = is->frame_last_delay;
            }
            is->frame_last_delay = delay;
            is->frame_last_pts = vp->pts;
            ref_clock = get_audio_clock(is);
            diff = vp->pts - ref_clock;
            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;
            actual_delay = is->frame_timer - (av_gettime() / 1000000.0);
            if(actual_delay < 0.010)
            {
                actual_delay = 0.010;
            }
            schedule_refresh(is, (int)(actual_delay * 1000 + 0.5));
            video_display(is);
            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);
    }
}
```

我们在这里做了很多检查：首先，我们保证现在的时间戳和上一个时间戳之间的处以 delay 是有意义的。如果不是的话，我们就猜测着用上次的延迟。接着，我 们有一个同步阈值，因为在同步的时候事情并不总是那么完美的。在 ffplay 中使用 0.01 作为它的值。我们也保证阈值不会比时间戳之间的间隔短。最后， 我们把最小的刷新值设置为 10 毫秒。

（这句不知道应该放在哪里）事实上这里我们应该跳过这一帧，但是我们不想为此而烦恼。

我们给大结构体添加了很多的变量，所以不要忘记检查一下代码。同时也不要忘记在函数 streame_component_open 中初始化帧时间 frame_timer 和前面的帧延迟 frame delay：

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

同步：声音时钟

现在让我们看一下怎样来得到声音时钟。我们可以在声音解码函数 audio_decode_frame 中更新时钟时间。现在，请记住我们并不是每次调用这个 函数的时候都在处理新的包，所以有我们要在两个地方更

新时钟。第一个地方是我们得到新的包的时候：我们简单的设置声音时钟为这个包的时间戳。然后，如果一 个包里有许多帧，我们通过样本数和采样率来计算，所以当我们得到包的时候：

```
if(pkt->pts != AV_NOPTS_VALUE)
{
    is->audio_clock = av_q2d(is->audio_st->time_base)*pkt->pts;
}
```

然后当我们处理这个包的时候：

```
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);
```

一点细节：临时函数被改成包含 pts_ptr，所以要保证你已经改了那些。这时的 pts_ptr 是一个用来通知 audio_callback 函数当前声音包的时间戳的指针。这将在下次用来同步声音和视频。

现在我们可以最后来实现我们的 get_audio_clock 函数。它并不像得到 is->audio_clock 值那样简单。注意我们会在每次处理 它的时候设置声音时间戳，但是如果你看了 audio_callback 函数，它花费了时间来把数据从声音包中移到我们的输出缓冲区中。这意味着我们声音时 钟中记录的时间比实际的要早太多。所以我们必须要检查一下我们还有多少没有写入。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;
    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;
}
```

你应该知道为什么这个函数可以正常工作了;)
这就是了！让我们编译它：

```
gcc -o tutorial05 tutorial05.c -lavutil -lavformat -lavcodec -lz -lm`sdl-config --cflags --libs`
```

最后，你可以使用我们自己的电影播放器来看电影了。下次我们将看一下声音同步，然后接下来的指导我们会讨论查询。

同步音频

现在我们已经有了一个比较像样的播放器。所以让我们看一下还有哪些零碎的东西没处理。上次，我们掩饰了一点同步问题，也就是同步音频到视频而不是其它的同 步方式。我们将采用和视频一样的方式：做一个内部视频时钟来记录视频线程播放了多久，然后同步音频到上面去。后面我们也来看一下如何推而广之把音频和视频 都同步到外部时钟。

生成一个视频时钟

现在我们要生成一个类似于上次我们的声音时钟的视频时钟：一个给出当前视频播放时间的内部值。开始，你可能会想这和使用上一帧的时间戳来更新定时器一样简 单。但是，不要忘了视频帧之间的时间间隔是很长的，以毫秒为计量的。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。于 是 当 前 视 频 时 间 值 就 是 PTS_of_last_frame + (current_time - time_elapsed_since_PTS_value_was_set)。这种解决方式与我们在函数 get_audio_clock 中的方式很类似。

所在在我们的大结构体中，我们将放上一个双精度浮点变量 video_current_pts 和一个 64 位宽整型变量 video_current_pts_time。时钟更新将被放在 video_refresh_timer 函数中。

```
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 函数中初始化它：

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

现在我们需要一种得到信息的方式：

```
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？我们更改视频同步代码以致于音频和视频不会试着去相互同步。想像一下我们让它像 ffplay 一样有一个命令行参数。所以 让我们抽象一样这件事情：我们将做一个新的封装函数 get_master_clock，用来检测 av_sync_type 变量然后决定调用 get_audio_clock 还是 get_video_clock 或者其它的想使用的获得时钟的函数。我们甚至可以使用电脑时钟，这个函数我们叫做 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：同步音频到视频时钟。我们的策略是测量声音的位置，把它与视频时间比较然后算出我们需要修正多少的样本数，也就是说：我们是否需要通过丢弃样本的方式来加速播放还是需要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 的函数来正确的收缩或者扩展声音样本。然而，我们不想在每次发现有偏差的时候 都进行同步，因为这样会使同步音频多于视频包。所以我们为函数 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 会总是在调整了。当然，就像上次那样，“失去同步”意味着声音时钟和视频时钟的差异大于我们的阈值。

所以我们将使用一个分数系数，叫 c，所以现在可以说我们得到了 N 个失去同步的声音样本。失去同步的数量可能会有很多变化，所以我们要计算一下失去同步的长 度的均值。例如，第一次调用的时候，显示出来我们失去同步的长度为 40ms，下次变为 50ms 等等。但是我们不会使用一个简单的均值，因为距离现在最近的 值比靠前的值要重要的多。所以我们将使用一个分数系统，叫 c，然后用这样的公式来计算差异：diff_sum = new_diff + diff_sum*c。当我们准备好去找平均差异的时候，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！嗯，它基本上是一个使用等比级数的加权平均值。我不知道这是否有名字（我甚至查过维基百科!），但是如果想要更多的信息，这里是一个解释

http://www.dranger.com/ffmpeg/weightedmean.html 或者在
http://www.dranger.com/ffmpeg/weightedmean.txt

下面是我们的函数：

```
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);
                }
            }
            else
            {
                is->audio_diff_avg_count = 0;
                is->audio_diff_cum = 0;
            }
        }
        return samples_size;
}
```

现在我们已经做得很好；我们已经近似的知道如何用视频或者其它的时钟来调整音频了。所以让我们来计算一下要在添加和砍掉多少样本，并且如何在 “Shrinking/expanding buffer code”部分来写上代码：

```
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)就是 audio_length 秒时间的声音的样本数。所以，我们想要的样本数就是我们根据声音偏移添加或者减少后的声音样本数。我们也可以设置一个范 围来限定我们一次进行修正的长度，因为如果我们改变的太多，用户会听到刺耳的声音。

修正样本数

现在我们要真正的修正一下声音。你可能会注意到我们的同步函数 synchronize_audio 返回了一个样本数，这可以告诉我们有多少个字节被送到流 中。所以我们只要调整样本数为 wanted_size 就可以了。这会让样本更小一些。但是如果我们想让它变大，我们不能只是让样本大小变大，因为在缓冲区 中没有多余的数据！所以我们必需添加上去。但是我们怎样来添加呢？最笨的办法就是试着来推算声音，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)
{
    samples_size = wanted_size;
}
else if(wanted_size > samples_size)
{
    uint8_t *samples_end, *q;
    int nb;

    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;
}
```

现在我们通过这个函数返回的是样本数。我们现在要做的是使用它：

```
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)
        {
            audio_size = audio_decode_frame(is, is->audio_buf, sizeof(is->audio_buf), &pts);
            if(audio_size < 0)
```

```
        {
            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，保证在初始化上面变量的时候检查一下代码，这些我没有赘述）。

结束之前的最后一件事情：我们需要添加一个 if 语句来保证我们不会在视频为主时钟的时候也来同步视频。

```
if(is->av_sync_type != AV_SYNC_VIDEO_MASTER)
{
    ref_clock = get_master_clock(is);
    diff = vp->pts - ref_clock;
    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;
        }
    }
}
```

添加后就可以了。要保证整个程序中我没有赘述的变量都被初始化过了。然后编译它：

```
gcc -o tutorial06 tutorial06.c -lavutil -lavformat -lavcodec -lz -lm`sdl-config --cflags --libs`
```

然后你就可以运行它了。

快进快退

处理快进快退命令

现在我们来为我们的播放器加入一些快进和快退的功能，因为如果你不能全局搜索一部电影是很让人讨厌的。同时，这将告诉你 av_seek_frame 函数是多么容易使用。

我们将在电影播放中使用左方向键和右方向键来表示向后和向前一小段，使用向上和向下键来表示向前和向后一大段。这里一小段是 10 秒，一大段是 60 秒。所以我们需要设置我们的主循环来捕捉键盘事件。然而当我们捕捉到键盘事件后我们不能直接调用 av_seek_frame 函数。我们要主要的解码线程 decode_thread 的循环中做这些。所以，我们要添加一些变量到大结构体中，用来包含新的跳转位置和一些跳转标志：

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

现在让我们在主循环中捕捉按键：

```
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，我们先查了一下是否有 SDL_KEYDOWN 事件。然后我们使用 event.key.keysym.sym 来判断哪个按键被按下。一旦我们 知道了如何来跳转，我们就来计算新的时间，方法为把增加的时间值加到从函数 get_master_clock 中得到的时间值上。然后我们调用 stream_seek 函数来设置 seek_pos 等变量。我们把新的时间转换成为 avcodec 中的内部时间戳单位。在流中调用那个时间戳将使用帧而不 是用秒来计算，公式为 seconds = frames * time_base(fps)。默认的 avcodec 值为 1,000,000fps（所以 2 秒的内部时间戳为 2,000,000）。在后面我们来看一下为 什么要把这个值进行一下转换。

这就是我们的 stream_seek 函数。请注意我们设置了一个标志为后退服务：

```
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;
    }
}
```

现在让我们看一下如果在 decode_thread 中实现跳转。你会注意到我们已经在源文件中标记了一个叫做“seek stuff goes here”的部分。现在我们将把代码写在这里。

跳转是围绕着 av_seek_frame 函数的。这个函数用到了一个格式上下文，一个流，一个时间戳和一组标记来作为它的参数。这个函数将会跳转到你所给 的时间戳的位置。时间戳的单位是你传递给函数的流的时基 time_base。然而，你并不是必需要传给它一个流（流可以用-1 来代替）。如果你这样做了， 时基 time_base 将会是 avcodec 中的内部时间戳单位，或者是 1000000fps。这就是为什么我们在设置 seek_pos 的时候会把位置乘 以 AV_TIME_BASER 的原因。

但是，如果给 av_seek_frame 函数的 stream 参数传递传-1,你有时会在播放某些文件的时候遇到问题（比较少见），所以我们会取文件中的第一个流并且把它传递到 av_seek_frame 函数。不要忘记我们也要把时间戳 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
    {
```

这里 av_rescale_q(a,b,c)是用来把时间戳从一个时基调整到另外一个时基时候用的函数。它基本的动作是计算 a*b/c，但是这个函数还是 必需的，因为直接计算会有溢出的情况发生。AV_TIME_BASE_Q 是 AV_TIME_BASE 作为分母后的版本。它们是很不相同 的：AV_TIME_BASE * time_in_seconds = avcodec_timestamp 而 AV_TIME_BASE_Q * avcodec_timestamp = time_in_seconds （注意 AV_TIME_BASE_Q 实际上是一个 AVRational 对象，所以你必需使用 avcodec 中特定的 q 函数 来处理它）。

清空我们的缓冲

我们已经正确设定了跳转位置，但是我们还没有结束。记住我们有一个堆放了很多包的队列。既然我们跳到了不同的位置，我们必需把队列中的内容清空否则电影是不会跳转的。不仅如此，avcodec 也有它自己的内部缓冲，也需要每次被清空。

要实现这个，我们需要首先写一个函数来清空我们的包队列。然后我们需要一种命令声音和视频线程来清空 avcodec 内部缓冲的办法。我们可以在清空队列后把特定的包放入到队列中，然后当它们检测到特定的包的时候，它们就会把自己的内部缓冲清空。

让我们开始写清空函数。其实很简单的，所以我直接把代码写在下面：

```
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);
}
```

既然队列已经清空了，我们放入“清空包”。但是开始我们要定义和创建这个包：

```
AVPacket flush_pkt;

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

现在我们把这个包放到队列中：

```


    }
    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;
}
```

（这些代码片段是接着前面 decode_thread 中的代码片段的）我们也需要修改 packet_queue_put 函数才不至于直接简单复制了这个包：

```
int packet_queue_put(PacketQueue *q, AVPacket *pkt)
```

```
{
    AVPacketList *pkt1;
    if(pkt != &flush_pkt && av_dup_packet(pkt) < 0)
    {
        return -1;
    }
```

然后在声音线程和视频线程中，我们在 packet_queue_get 后立即调用函数 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”换成“video”。
就这样，让我们编译我们的播放器：

```
gcc -o tutorial07 tutorial07.c -lavutil -lavformat -lavcodec -lz -lm`sdl-config --cflags --libs`
```

试一下！我们几乎已经都做完了；下次我们只要做一点小的改动就好了，那就是检测 ffmpeg 提供的小的软件缩放采样。

软件缩放

软件缩放库 libswscale

近来 ffmpeg 添加了新的接口：libswscale 来处理图像缩放。

但是在前面我们使用 img_convert 来把 RGB 转换成 YUV12，我们现在使用新的接口。新接口更加标准和快速，而且我相信里面有了 MMX 优化代码。换句话说，它是做缩放更好的方式。

我们将用来缩放的基本函数是 sws_scale。但一开始，我们必需建立一个 SwsContext 的概念。这将让我们进行想要的转换，然后把它传递给 sws_scale 函数。类似于在 SQL 中的预备阶段或者是在 Python 中编译的规则表达式 regexp。要准备这个上下文，我们使用 sws_getContext 函数，它需要我们源的宽度和高度，我们想要的宽度和高度，源的格式和想要转换成的格式，同时还有一些其它的参数和标志。然后 我们像使用 img_convert 一样来使用 sws_scale 函数，唯一不同的是我们传递给的是 SwsContext：

```
#include <ffmpeg/swscale.h> // include the header!
```

```
int queue_picture(VideoState *is, AVFrame *pFrame, double pts)
{
    static struct SwsContext *img_convert_ctx;
    ...
    if(vp->bmp)
    {
        SDL_LockYUVOverlay(vp->bmp);
        dst_pix_fmt = PIX_FMT_YUV420P;
        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
        if(img_convert_ctx == NULL)
        {
            int w = is->video_st->codec->width;
            int h = is->video_st->codec->height;
            img_convert_ctx = sws_getContext(w, h,is->video_st->codec->pix_fmt,w, h, dst_pix_fmt,
                        SWS_BICUBIC,NULL, NULL, NULL);
            if(img_convert_ctx == NULL)
            {
                fprintf(stderr, "Cannot initialize the conversion context!\n");
                exit(1);
            }
        }
        sws_scale(img_convert_ctx, pFrame->data,pFrame->linesize, 0,
                is->video_st->codec->height,pict.data, pict.linesize);
```

我们把新的缩放器放到了合适的位置。希望这会让你知道 libswscale 能做什么。
就这样，我们做完了！编译我们的播放器：

```
gcc -o tutorial08 tutorial08.c -lavutil -lavformat -lavcodec -lz -lm `sdl-config --cflags --libs`
```

享受我们用 C 写的少于 1000 行的电影播放器吧。

当然，还有很多事情要做。

现在还要做什么？

我们已经有了一个可以工作的播放器，但是它肯定还不够好。我们做了很多，但是还有很多要添加的性能：

• 错误处理。我们代码中的错误处理是无穷的，多处理一些会更好。

• 暂停。我们不能暂停电影，这是一个很有用的功能。我们可以在大结构体中使用一个内部暂停变量，当用户暂停的时候就设置它。然后我们的音频，视频和解码线 程检测到它后就不再输出任何东西。我们也使用 av_read_play 来支持网络。这很容易解释，但是你却不能明显的计算出，所以把这个作为一个家庭作 业，如果你想尝试的话。提示，可以参考 ffplay.c。

• 支持视频硬件特性。一个参考的例子，请参考 Frame Grabbing 在 Martin 的旧的指导中的相关部分。http://www.inb.uni-luebeck.de/~boehme/libavcodec_update.html

• 按字节跳转。如果你可以按照字节而不是秒的方式来计算出跳转位置，那么对于像 VOB 文件一样的有不连续时间戳的视频文件来说，定位会更加精确。

• 丢弃帧。如果视频落后的太多，我们应当把下一帧丢弃掉而不是设置一个短的刷新时间。

• 支持网络。现在的电影播放器还不能播放网络流媒体。

• 支持像 YUV 文件一样的原始视频流。如果我们的播放器支持的话，因为我们不能猜测出时基和大小，我们应该加入一些参数来进行相应的设置。

• 全屏。

• 多种参数，例如：不同图像格式；参考 ffplay.c 中的命令开关。

• 其它事情，例如：在结构体中的音频缓冲区应该对齐。