ffmpeg 开发指南[lsosa.BIT 翻译]

lsosa 2006-05-20

ffmpeg 中的Libavformat 和 libavcodec 库是访问大多数视频文件格式的一个很好的方法。不幸的是，在开发您自己的程序时，这套库基本上没有提供什么实际的文档可以用来作为参 考（至少我没有找到任何文档），并且它的例程也并没有太多的帮助。

这种情况意味着，当我在最近某个项目中需要用到 libavformat/libavcodec 库时，需要作很多试验来搞清楚怎样使用它们。这里是我所学习的－－希望我做的这些能够帮助一些人，以免他们重蹈我的覆辙，作同样的试验，遇到同样的错误。 你还可以从这里下载一个demo 程序。我将要公开的这部分代码需要 0.4.8 版本的 ffmpeg 库中的 libavformat/libavcodec 的支持（我正在写最新版本）。如果您发现以后的版本与我写的程序不能兼容，请告知我。

在这个文档里，我仅仅涉及到如何从文件中读入视频流；音频流使用几乎同样的方法可以工作的很好，不过，我并没有实际使用过它们，所以，我没于办法提供任何示例代码。

或许您会觉得奇怪，为什么需要两个库文件 libavformat 和 libavcodec ：许多视频文件格式（AVI 就是一个最好的例子）实际上并没有明确指出应该使用哪种编码来解析音频和视频数据；它们只是定义了音频流和视频流（或者，有可 能是多个音频视频流）如何被绑定在一个文件里面。这就是为什么有时候，当你打开了一个 AVI 文件时，你只能听到声音，却不能看到图象－－因为你的系统没有 安装合适的视频解码器。所以， libavformat 用来处理解析视频文件并将包含在其中的流分离出来， 而 libavcodec 则处理原始音频和视频流的解码。

打开视频文件：
首先第一件事情－－让我们来看看怎样打开一个视频文件并从中得到流。我们要做的第一件事情就是初始化 libavformat/libavcodec:

```
av_register_all();
```

这 一步注册库中含有的所有可用的文件格式和编码器，这样当打开一个文件时，它们才能够自动选择相应的文件格式和编码器。要注意你只需调用一次 av_register_all()，所以，尽可能的在你的初始代码中使用它。如果你愿意，你可以仅仅注册个人的文件格式和编码，不过，通常你不得不这么 做却没有什么原因。

下一步，打开文件：

```
AVFormatContext *pFormatCtx;
const char      *filename="myvideo.mpg";
// 打开视频文件
if(av_open_input_file(&pFormatCtx, filename, NULL, 0, NULL)!=0)
    handle_error(); // 不能打开此文件
```

最后三个参数描述了文件格式，缓冲区大小（size）和格式参数；我们通过简单地指明 NULL

或 0 告诉 libavformat 去自动探测文件格式并且使用默认的缓冲区大小。请在你的程序中用合适的出错处理函数替换掉 handle_error()。
下一步，我们需要取出包含在文件中的流信息：

```
// 取出流信息
if(av_find_stream_info(pFormatCtx)<0)
    handle_error(); // 不能够找到流信息
```

这一步会用有效的信息把 AVFormatContext 的流域（streams field）填满。作为一个可调试的诊断，我们会将这些信息全盘输出到标准错误输出中，不过你在一个应用程序的产品中并不用这么做：

```
dump_format(pFormatCtx, 0, filename, false);
```

就像在引言中提到的那样，我们仅仅处理视频流，而不是音频流。为了让这件事情更容易理解，我们只简单使用我们发现的第一种视频流：

```
int             i, videoStream;
AVCodecContext *pCodecCtx;
//  寻找第一个视频流
videoStream=-1;
for(i=0; i<pFormatCtx->nb_streams; i++)
    if(pFormatCtx->streams->codec.codec_type==CODEC_TYPE_VIDEO)
    {
        videoStream=i;
        break;
    }
if(videoStream==-1)
    handle_error(); // Didn't find a video stream

// 得到视频流编码上下文的指针
pCodecCtx=&pFormatCtx->streams[videoStream]->codec;
```

好了，我们已经得到了一个指向视频流的称之为上下文的指针。但是我们仍然需要找到真正的编码器打开它。

```
AVCodec *pCodec;

//  寻找视频流的解码器
pCodec=avcodec_find_decoder(pCodecCtx->codec_id);
if(pCodec==NULL)
    handle_error(); // 找不到解码器

// 通知解码器我们能够处理截断的 bit 流——ie，
// bit 流帧边界可以在包中
if(pCodec->capabilities & CODEC_CAP_TRUNCATED)
```

```
    pCodecCtx->flags|=CODEC_FLAG_TRUNCATED;

// 打开解码器
if(avcodec_open(pCodecCtx, pCodec)<0)
    handle_error(); // 打不开解码器
```

（那么什么是“截断 bit 流”？好的，就像一会我们看到的，视频流中的数据是被分割放入包中的。因为每个视频帧的数据的大小是可变的，那么两帧之间的边界就不一定刚好是包的边界。这里，我们告知解码器我们可以处理 bit 流。）

存储在 AVCodecContext 结构中的一个重要的信息就是视频帧速率。为了允许非整数的帧速率（比如 NTSC 的 29.97 帧），速率以分数的形式存储，分子在 pCodecCtx->frame_rate，分母在 pCodecCtx->frame_rate_base 中。在用不同的视频文件测试库时，我注意到一些编码器（很显然 ASF）似乎并不能正确的给予赋值（ frame_rate_base 用 1 代替 1000）。下面给出修复补丁：

```
//  加入这句话来纠正某些编码器产生的帧速错误
if(pCodecCtx->frame_rate>1000 && pCodecCtx->frame_rate_base==1)
    pCodecCtx->frame_rate_base=1000;
```

注意即使将来这个 bug 解决了，留下这几句话也并没有什么坏处。视频不可能拥有超过 1000fps 的帧速。

只剩下一件事情要做了：给视频帧分配空间以便存储解码后的图片：

```
AVFrame *pFrame;

pFrame=avcodec_alloc_frame();
```

就这样，现在我们开始解码这些视频。

解码视频帧
就像我前面提到过的，视频文件包含数个音频和视频流，并且他们各个独自被分开存储在固定大小的包里。我们要做的就是使用 libavformat 依次读取这些 包，过滤掉所有那些视频流中我们不感兴趣的部分，并把它们交给 libavcodec 进行解码处理。在做这件事情时，我们要注意这样一个事实，两帧之间的边界也可以在包的中间部分。
听起来很复杂？幸运的是，我们在一个例程中封装了整个过程，它仅仅返回下一帧：

```
bool GetNextFrame(AVFormatContext *pFormatCtx, AVCodecContext *pCodecCtx,
    int videoStream, AVFrame *pFrame)
{
    static AVPacket packet;
    static int      bytesRemaining=0;
```

```
    static uint8_t  *rawData;
    static bool     fFirstTime=true;
    Int bytesDecoded;
    Int frameFinished;

//  我们第一次调用时，将 packet.data 设置为 NULL 指明它不用释放了
    if(fFirstTime)
    {
        fFirstTime=false;
        packet.data=NULL;
    }

// 解码直到成功解码完整的一帧
    while(true)
    {
         //  除非解码完毕，否则一直在当前包中工作
        while(bytesRemaining > 0)
        {
        //  解码下一块数据
            bytesDecoded=avcodec_decode_video(pCodecCtx, pFrame,
                &frameFinished, rawData, bytesRemaining);

                // 出错了?
            if(bytesDecoded < 0)
            {
                fprintf(stderr, "Error while decoding frame\n");
                return false;
            }

            bytesRemaining-=bytesDecoded;
            rawData+=bytesDecoded;

                // 我们完成当前帧了吗? 接着我们返回
            if(frameFinished)
                return true;
        }

        // 读取下一包，跳过所有不属于这个流的包
        do
        {
            // 释放旧的包
            if(packet.data!=NULL)
                av_free_packet(&packet);
```

```
            // 读取新的包
            if(av_read_packet(pFormatCtx, &packet)<0)
                goto loop_exit;
        } while(packet.stream_index!=videoStream);

        bytesRemaining=packet.size;
        rawData=packet.data;
    }

loop_exit:

        // 解码最后一帧的余下部分
    bytesDecoded=avcodec_decode_video(pCodecCtx, pFrame, &frameFinished,
        rawData, bytesRemaining);

        // 释放最后一个包
    if(packet.data!=NULL)
        av_free_packet(&packet);

    return frameFinished!=0;
}
```

现 在，我们要做的就是在一个循环中，调用 GetNextFrame () 直到它返回 false。还有一处需要注意：大多数编码器返回 YUV 420 格式的图片（一个亮度和两个色度通道，色度通道只占亮度通道空间分辨率的一半(译者注:此句原句为 the chrominance channels samples at half the spatial resolution of the luminance channel））。看你打算如何对视频数据处理，或许你打算将它转换至 RGB 格式。(注意，尽管，如果你只是打算显示视频数据，那大可不必要这么做;查 看一下 X11 的 Xvideo 扩展,它可以在硬件层进行 YUV 到 RGB 转换。）幸运的是， libavcodec 提供给我们了一个转换例程 img_convert ，它可以像转换其他图象进行 YUV 和 RGB 之间的转换。这样解码视频的循环就变成这样：

```
while(GetNextFrame(pFormatCtx, pCodecCtx, videoStream, pFrame))
{
    img_convert((AVPicture *)pFrameRGB, PIX_FMT_RGB24, (AVPicture*)pFrame,
        pCodecCtx->pix_fmt, pCodecCtx->width, pCodecCtx->height);

    // 处理视频帧（存盘等等）
    DoSomethingWithTheImage(pFrameRGB);
}
```

RGB 图象 pFrameRGB （AVFrame *类型）的空间分配如下：

```
AVFrame *pFrameRGB;
int     numBytes;
```

```
uint8_t *buffer;

// 分配一个 AVFrame 结构的空间
pFrameRGB=avcodec_alloc_frame();
if(pFrameRGB==NULL)
    handle_error();

// 确认所需缓冲区大小并且分配缓冲区空间
numBytes=avpicture_get_size(PIX_FMT_RGB24, pCodecCtx->width,
    pCodecCtx->height);
buffer=new uint8_t[numBytes];

// 在 pFrameRGB 中给图象位面赋予合适的缓冲区
avpicture_fill((AVPicture *)pFrameRGB, buffer, PIX_FMT_RGB24,
    pCodecCtx->width, pCodecCtx->height);
```

清除
好了，我们已经处理了我们的视频，现在需要做的就是清除我们自己的东西：

```
// 释放 RGB 图象
delete [] buffer;
av_free(pFrameRGB);

// 释放 YUV 帧
av_free(pFrame);

// 关闭解码器（codec）
avcodec_close(pCodecCtx);

// 关闭视频文件
av_close_input_file(pFormatCtx);
```

完成!
更新（2005 年 4 月 26 号）：有个读者提出：在 Kanotix （一个 Debian 的发行版）上面编译本例程，或者直接在 Debian 上面编译，头文件中 avcodec.h 和 avformat.h 需要加上前缀“ffmpeg”，就像这样：

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

同样的， libdts 库在编译程序时也要像下面这样加入进来：

```
g++ -o avcodec_sample.0.4.9 avcodec_sample.0.4.9.cpp \
    -lavformat -lavcodec -ldts -lz
```

几个月前，我写了一篇有关使用 ffmpeg 下 libavformat 和 libavcodec 库的文章。从那以来，我收到过一些评论，并且新的 ffmpeg 预发行版(0.4.9-pre1) 最近也要出来了，增加了对在视频文件中定位的支持，新的文件格式，和简单的读取视频帧的接口。这些改变不久就会应用到 CVS 中，不过这次是我第一次在发行 版中看到它们。（顺便感谢 Silviu Minut 共享长时间学习 CVS 版的 ffmpeg 的成果——他的有关 ffmpeg 的信息和 demo 程序在这里。）

在这篇文章里，我仅仅会描述一下以前的版本(0.4.8)和最新版本之间的区别，所以，如果你是采用新的 libavformat / libavcodec ，我建议你读前面的文章。

首 先，说说有关编译新发行版吧。用我的编译器（ SuSE 上的 gcc 3.3.1 ），在编译源文件 ffv1.c 时会报一个编译器内部的错误。我怀疑这是个精简版的 gcc——我在编译 OpenCV 时也遇到了同样的事情——但是不论如何，一个快速的解决方法就是在编译此文件时不要加优化参数。最简单的方法就是作一个 make，当编译时遇到编译器错 误，进入 libavcodec 子目录（因为这也是 ffv1.c 所在之处），在你的终端中使用 gcc 命令去编译 ffv1.c，粘贴，编辑删除编译器开关（译者注：就是参数）"-O3"，然后使用那个命令运行 gcc。然 后，你可以变回 ffmpeg 主目录并且重新运行 make，这次应该可以编译了。

都有哪些更新？
有那些更新呢？从一个程序员的角度 来看，最大的变化就是尽可能的简化了从视频文件中读取个人的视频帧的操作。在 ffmpeg 0.4.8 和其早期版本中，在从一个视频文件中的包中用例程 av_read_packet()来读取数据时，一个视频帧的信息通常可以包含在几个包里，而另情况更为 复杂的是,实际上两帧之间的边界还可以存在于两个包之间。幸亏 ffmpeg 0.4.9 引入了新的叫做 av_read_frame()的例程,它可以从一个简单的包里返回一个视频帧包含的所有数据。使用 av_read_packet()读取 视频数据的老办法仍然支持，但是不赞成使用——我说：摆脱它是可喜的。

这里让我们来看看如何使用新的 API 来读取视频数据。在我原来的文章中（与 0.4.8 API 相关），主要的解码循环就像下面这样：

```
while(GetNextFrame(pFormatCtx, pCodecCtx, videoStream, pFrame))
{
    img_convert((AVPicture *)pFrameRGB, PIX_FMT_RGB24, (AVPicture*)pFrame,
        pCodecCtx->pix_fmt, pCodecCtx->width, pCodecCtx->height);

    // 处理视频帧（存盘等等）
    DoSomethingWithTheImage(pFrameRGB);
}
```

GetNextFrame() 是个有帮助的例程，它可以处理这样一个过程，这个过程汇编一个完整的视频帧所需要的所有的包。新的 API 简化了我们在主循环中实际直接读取和解码数据的操作：

```
while(av_read_frame(pFormatCtx, &packet)>=0)
{
```

```
    // 这是视频流中的一个包吗?
    if(packet.stream_index==videoStream)
    {
        // 解码视频流
        avcodec_decode_video(pCodecCtx, pFrame, &frameFinished,
            packet.data, packet.size);

        // 我们得到一帧了吗?
        if(frameFinished)
        {
            // 把原始图像转换成 RGB
            img_convert((AVPicture *)pFrameRGB, PIX_FMT_RGB24,
                (AVPicture*)pFrame, pCodecCtx->pix_fmt, pCodecCtx->width,
                pCodecCtx->height);

            // 处理视频帧（存盘等等）
            DoSomethingWithTheImage(pFrameRGB);
        }
    }

    // 释放用 av_read_frame 分配空间的包
    av_free_packet(&packet);
}
```

看第一眼，似乎看上去变得更为复杂了。但那仅仅是因为这块代码做的都是要隐藏在GetNextFrame()例程中实现的（检查包是否属于视频流，解码帧并释放包）。总的说来，因为我们能够完全排除 GetNextFrame ()，事情变得更简单了。
我已经更新了 demo 程序使用最新的 API。简单比较一下行数（老版本 222 行 Vs 新版本 169 行）显示出新的 API 大大的简化了这件事情。

0.4.9 的另一个重要的更新是能够在视频文件中定位一个时间戳。它通过函数 av_seek_frame() 来实现，此函数有三个参数：一个指向 AVFormatContext 的指针，一个流索引和定位时间戳。此函数在给定时间戳以前会去定位第一个关键帧。所有这些都来自于文档。我并没有对 av_seek_frame()进行测 试，所以这里我并不能够给出任何示例代码。如果你成功的使用 av_seek_frame() ，我很高兴听到这个消息。

捕获视频(Video4Linux and IEEE1394)
Toru Tamaki 发给我了一些使用 libavformat / libavcodec 库从 Video4Linux 或者 IEEE1394 视频设备源中抓捕视频帧的样例代码。对 Video4Linux,调用 av_open_input_file() 函数应该修改如下：

```
AVFormatParameters formatParams;
AVInputFormat *iformat;

formatParams.device = "/dev/video0";
```

```
formatParams.channel = 0;
formatParams.standard = "ntsc";
formatParams.width = 640;
formatParams.height = 480;
formatParams.frame_rate = 29;
formatParams.frame_rate_base = 1;
filename = "";
iformat = av_find_input_format("video4linux");

av_open_input_file(&ffmpegFormatContext,
                filename, iformat, 0, &formatParams);

For IEEE1394, call av_open_input_file() like this:

AVFormatParameters formatParams;
AVInputFormat *iformat;

formatParams.device = "/dev/dv1394";
filename = "";
iformat = av_find_input_format("dv1394");

av_open_input_file(&ffmpegFormatContext,
                filename, iformat, 0, &formatParams);
```

继续。。。
如果我碰巧遇到了一些有关 libavformat / libavcodec 的有趣的信息，我计划在这里公布。
所以，如果你有任何的评论，请通过这篇文章顶部给出的地址联系我。
标准弃权：我没有责任去纠正这些代码的功能和这篇文章中涉及的技术。