【转】 如何同步视频

2010-07-16 10:18

转载自 [fandy586](http://hi.baidu.com/sdlyfdy)

最终编辑 [fandy586](http://hi.baidu.com/sdlyfdy)

|  |
| --- |
| 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bidrectional）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：  intget\_buffer(structAVCodecContext \*c, AVFrame \*pic);  void release\_buffer(structAVCodecContext \*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;  intour\_get\_buffer(structAVCodecContext \*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(structAVCodecContext \*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。  typedefstructVideoState {  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：  typedefstructVideoPicture {  ...  double pts;  }  intqueue\_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;  inthw\_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 里。  下面是我们的函数：  intsynchronize\_audio(VideoState \*is, short \*samples,  intsamples\_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;  intwanted\_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;  intnb;  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, intlen) {  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：  intseek\_req;  intseek\_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;  gotodo\_seek;  case SDLK\_RIGHT:  incr = 10.0;  gotodo\_seek;  case SDLK\_UP:  incr = 60.0;  gotodo\_seek;  case SDLK\_DOWN:  incr = -60.0;  gotodo\_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, intrel) {  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) {  intstream\_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：  AVPacketflush\_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：  intpacket\_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!  intqueue\_picture(VideoState \*is, AVFrame \*pFrame, double pts) {  static structSwsContext \*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。 |