ffplay播放器-1-3

1 ffplay.c的意义

2 FFplay框架分析

3 数据结构分析

- struct VideoState 播放器封装
- struct Clock 时钟封装
- struct MyAVPacketList和PacketQueue队列
  - packet_queue_init()
  - packet_queue_destroy()
  - packet_queue_start()
  - packet_queue_abort()
  - packet_queue_put()
  - packet_queue_get()
  - packet_queue_put_nullpacket()
  - packet_queue_flush()
  - PacketQueue总结
- struct Frame 和 FrameQueue队列
  - frame_queue_init() 初始化
  - frame_queue_destory()销毁
  - frame_queue_peek_writable()获取可写Frame
  - frame_queue_push()入队列
  - frame_queue_peek_readable() 获取可读Frame
  - frame_queue_next()出队列
  - frame_queue_nb_remaining()获取队列的size
  - frame_queue_peek()获取当前帧
  - frame_queue_peek_next()获取下一帧
  - frame_queue_peek_last()获取上一帧
- struct AudioParams 音频参数
- struct Decoder解码器封装

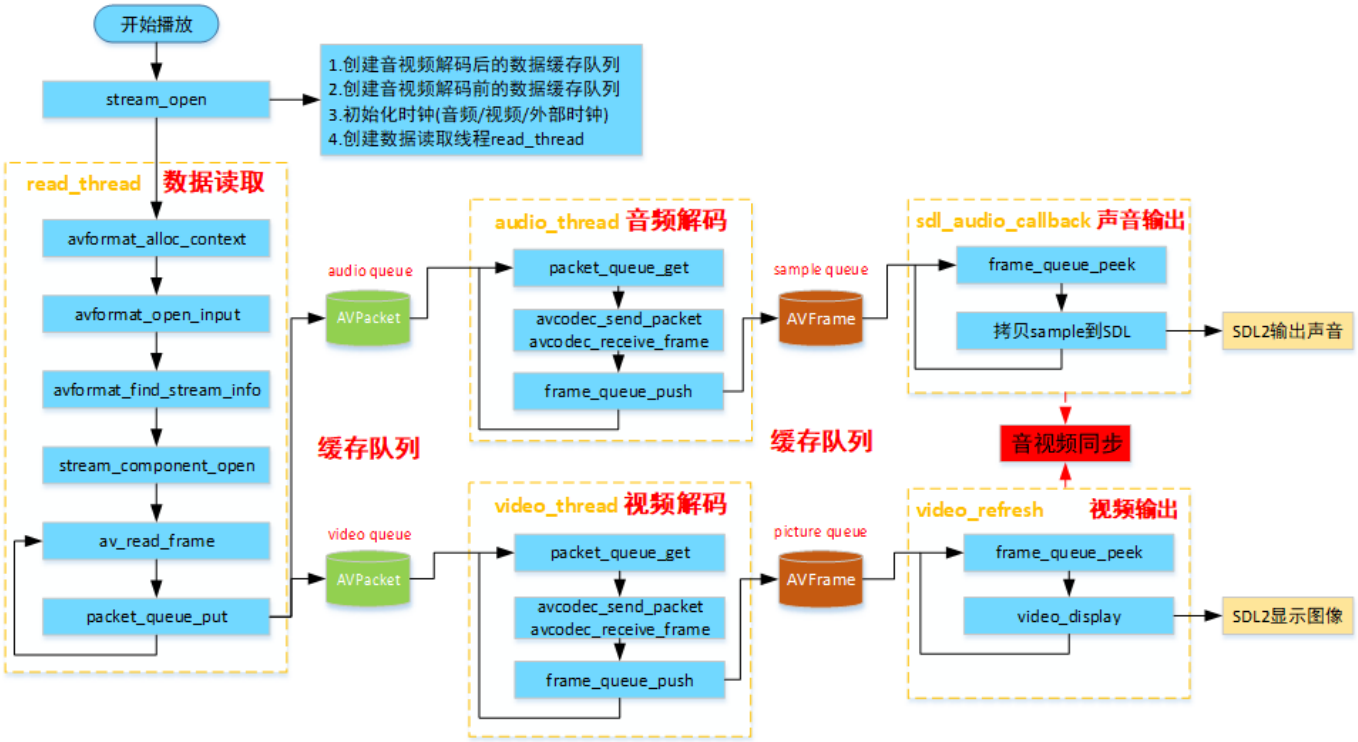
音视频流媒体高级开发教程 – Darren老师：QQ326873713
课程链接：https://ke.qq.com/course/468797?tuin=137bb271

1 ffplay.c的意义

ffplay.c是FFmpeg源码自带的播放器，调用FFmpeg和SDL API实现一个非常有用的播放器。
例如哔哩哔哩著名开源项目ijkplayer也是基于ffplay.c进行二次开发。

ffplay实现了播放器的主体功能，掌握其原理对于我们独立开发播放器非常有帮助。

2 FFplay框架分析



播放器初始化

- 初始化packet queue
- 初始化frame queue
- 初始化clock
- 创建数据读取线程

线程的划分

- 数据读取线程
  - 打开媒体文件

    - 打开对应码流的decoder以及初始化对应的audio、video、subtitle输出
    - 创建decoder线程，audio、video和subtitle的解码线程独立
    - 调用av_read_frame读取packet，并根据steam_index放入不同stream对应的packet队列
- 音频解码
    - 从packet queue读取packet，解出frame后放入frame queue
- 视频解码
    - 从packet queue读取packet，解出frame后放入frame queue
- 字幕解码
    - 从packet queue读取packet，解出frame后放入frame queue
- 音频播放（或者回调函数）
    - 从frame queue读取frame进行播放
- 视频播放（ffplay目前是在main主线程进行视频播放）
    - 从frame queue读取frame进行播放
- 字幕播放（ffplay目前是在main主线程进行字幕播放）
    - 从frame queue读取frame进行播放
- 控制响应（播放/暂停/快进/快退等）（ffplay目前是在main主线程进行播放控制）

packet队列的设计

- 线程安全，支持互斥、等待、唤醒
- 缓存数据大小
- 缓存包数
- 队列播放可持续时间
- 进队列/出队列等

frame队列的设计

- 线程安全，支持互斥、等待、唤醒
- 缓存帧数
- 支持读取数据而不出队列
- 进队列/出队列等

音视频同步

- 音频同步
- 视频同步
- 外部时钟同步

音频处理

- 音量调节
- 静音
- 重采样

视频处理

- 图像格式转换YUV->RGB等
- 图像缩放1280*720->800*480等

播放器控制

- 播放
- 暂停
- 停止
- 快进/快退
- 逐帧
- 静音

3 数据结构分析

struct VideoState 播放器封装

```
typedef struct VideoState {
    SDL_Thread  *read_tid;      // 读线程句柄
    AVInputFormat   *iformat;   // 指向demuxer
    int     abort_request;      // =1时请求退出播放
    int     force_refresh;      // =1时需要刷新画面，请求立即刷新画面的意思
    int     paused;             // =1时暂停，=0时播放
    int     last_paused;        // 暂存“暂停”/“播放”状态
    int     queue_attachments_req;
    int     seek_req;           // 标识一次seek请求
    int     seek_flags;         // seek标志，诸如AVSEEK_FLAG_BYTE等
    int64_t     seek_pos;       // 请求seek的目标位置(当前位置+增量)
    int64_t     seek_rel;       // 本次seek的位置增量
    int     read_pause_return;
    AVFormatContext *ic;        // iformat的上下文
    int     realtime;           // =1为实时流

    Clock   audclk;             // 音频时钟
    Clock   vidclk;             // 视频时钟
    Clock   extclk;             // 外部时钟

```

```
21      FrameQueue  pictq;          // 视频Frame队列
22      FrameQueue  subpq;          // 字幕Frame队列
23      FrameQueue  sampq;          // 采样Frame队列
24
25      Decoder auddec;             // 音频解码器
26      Decoder viddec;             // 视频解码器
27      Decoder subdec;             // 字幕解码器
28
29      int audio_stream ;          // 音频流索引
30
31      int av_sync_type;           // 音视频同步类型, 默认audio master
32
33      double          audio_clock;            // 当前音频帧的PTS+当前帧Du
    ration
34      int             audio_clock_serial;     // 播放序列, seek可改变此值
35      // 以下4个参数 非audio master同步方式使用
36      double          audio_diff_cum;         // used for AV differen
    ce average computation
37      double          audio_diff_avg_coef;
38      double          audio_diff_threshold;
39      int         audio_diff_avg_count;
40      // end
41
42      AVStream        *audio_st;              // 音频流
43      PacketQueue     audioq;                 // 音频packet队列
44      int         audio_hw_buf_size;          // SDL音频缓冲区的大小(字节
    为单位)
45      // 指向待播放的一帧音频数据, 指向的数据区将被拷入SDL音频缓冲区。若经过重采样
    则指向audio_buf1,
46      // 否则指向frame中的音频
47      uint8_t         *audio_buf;             // 指向需要重采样的数据
48      uint8_t         *audio_buf1;            // 指向重采样后的数据
49      unsigned int        audio_buf_size;     // 待播放的一帧音频数据(aud
    io_buf指向)的大小
50      unsigned int        audio_buf1_size;    // 申请到的音频缓冲区audio_
    buf1的实际尺寸
51      int         audio_buf_index;            // 更新拷贝位置 当前音频帧中
    已拷入SDL音频缓冲区
52                                              // 的位置索引(指向第一个待拷
    贝字节)
```

```
    // 当前音频帧中尚未拷入SDL音频缓冲区的数据量:
    // audio_buf_size = audio_buf_index + audio_write_buf_size
    int          audio_write_buf_size;
    int          audio_volume;                // 音量
    int          muted;                       // =1静音，=0则正常
    struct AudioParams audio_src;             // 音频frame的参数
#if CONFIG_AVFILTER
    struct AudioParams audio_filter_src;
#endif
    struct AudioParams audio_tgt;        // SDL支持的音频参数，重采样转换：audio_src->audio_tgt
    struct SwrContext *swr_ctx;          // 音频重采样context
    int frame_drops_early;               // 丢弃视频packet计数
    int frame_drops_late;                // 丢弃视频frame计数

    enum ShowMode {
        SHOW_MODE_NONE = -1, SHOW_MODE_VIDEO = 0, SHOW_MODE_WAVES, SHOW_MODE_RDFT, SHOW_MODE_NB
    } show_mode;

    // 音频波形显示使用
    int16_t sample_array[SAMPLE_ARRAY_SIZE];
    int sample_array_index;
    int last_i_start;
    RDFTContext *rdft;
    int rdft_bits;
    FFTSample *rdft_data;

    int xpos;
    double last_vis_time;
    SDL_Texture *vis_texture;

    SDL_Texture *sub_texture;        // 字幕显示
    SDL_Texture *vid_texture;        // 视频显示

    int subtitle_stream;             // 字幕流索引
    AVStream *subtitle_st;           // 字幕流
    PacketQueue subtitleq;           // 字幕packet队列

    double frame_timer;              // 记录最后一帧播放的时刻
```

```
    double frame_last_returned_time;
    double frame_last_filter_delay;

    int video_stream;               // 视频流索引
    AVStream *video_st;             // 视频流
    PacketQueue videoq;             // 视频队列
    double max_frame_duration;      // 一帧最大间隔. above this, we consider the jump a timestamp discontinuity
    struct SwsContext *img_convert_ctx; // 视频尺寸格式变换
    struct SwsContext *sub_convert_ctx; // 字幕尺寸格式变换
    int eof;            // 是否读取结束

    char *filename;     // 文件名
    int width, height, xleft, ytop; // 宽、高，x起始坐标，y起始坐标
    int step;           // =1 步进播放模式, =0 其他模式

#if CONFIG_AVFILTER
    int vfilter_idx;
    AVFilterContext *in_video_filter;   // the first filter in the video chain
    AVFilterContext *out_video_filter;  // the last filter in the video chain
    AVFilterContext *in_audio_filter;   // the first filter in the audio chain
    AVFilterContext *out_audio_filter;  // the last filter in the audio chain
    AVFilterGraph *agraph;              // audio filter graph
#endif
    // 保留最近的相应audio、video、subtitle流的steam index
    int last_video_stream, last_audio_stream, last_subtitle_stream;

    SDL_cond *continue_read_thread; // 当读取数据队列满了后进入休眠时，可以通过该condition唤醒读线程
} VideoState;
```

struct Clock 时钟封装

```c
typedef struct Clock {
    double  pts;            // 时钟基础, 当前帧(待播放)显示时间戳, 播放后, 当前帧变成上一帧
    // 当前pts与当前系统时钟的差值, audio、video对于该值是独立的
    double  pts_drift;      // clock base minus time at which we updated the clock
    // 当前时钟(如视频时钟)最后一次更新时间, 也可称当前时钟时间
    double  last_updated;   // 最后一次更新的系统时钟
    double  speed;          // 时钟速度控制, 用于控制播放速度
    // 播放序列, 所谓播放序列就是一段连续的播放动作, 一个seek操作会启动一段新的播放序列
    int serial;             // clock is based on a packet with this serial
    int paused;             // = 1 说明是暂停状态
    // 指向packet_serial
    int *queue_serial;      /* pointer to the current packet queue serial, used for obsolete clock detection */
} Clock;
```

struct MyAVPacketList和PacketQueue队列

ffplay用PacketQueue保存解封装后的数据，即保存AVPacket。
ffplay首先定义了一个结构体`MyAVPacketList`：

```c
typedef struct MyAVPacketList {
    AVPacket        pkt;    //解封装后的数据
    struct MyAVPacketList   *next;  //下一个节点
    int         serial;     //播放序列
} MyAVPacketList;
```

可以理解为是队列的一个节点。可以通过其`next`字段访问下一个节点。

serial字段主要用于标记当前节点的播放序列号，ffplay中多处用到serial的概念，主要用来区分是否连续数据，每做一次seek，该serial都会做+1的递增，以区分不同的播放序列。serial字段在我们ffplay的分析中应用非常广泛，谨记他是用来区分数据否连续先。

接着定义另一个结构体PacketQueue：

```
typedef struct PacketQueue {
    MyAVPacketList  *first_pkt, *last_pkt;  // 队首，队尾指针
    int     nb_packets;   // 包数量，也就是队列元素数量
    int     size;         // 队列所有元素的数据大小总和
    int64_t     duration; // 队列所有元素的数据播放持续时间
    int     abort_request; // 用户退出请求标志
    int     serial;         // 播放序列号，和MyAVPacketList的serial作用相同
    SDL_mutex   *mutex;     // 用于维持PacketQueue的多线程安全(SDL_mutex可以按pthread_mutex_t理解)
    SDL_cond    *cond;      // 用于读、写线程相互通知(SDL_cond可以按pthread_cond_t理解)
} PacketQueue;
```

该结构体内定义了“队列”自身的属性。上面的注释对每个字段作了简单的介绍，这里也看到了serial字段，MyAVPacketList的serial字段的赋值来自PacketQueue的serial，每个PacketQueue的serial是独立的。

音频、视频、字幕流都有自己独立的PacketQueue。

接下来我们也从队列的操作函数具体分析各个字段的含义。

`PacketQueue`操作提供以下方法：

- packet_queue_init：初始化
- packet_queue_destroy：销毁
- packet_queue_start：启用
- packet_queue_abort：中止
- packet_queue_get：获取一个节点
- packet_queue_put：存入一个节点
- packet_queue_put_nullpacket：存入一个空节点
- packet_queue_flush：清除队列内所有的节点

packet_queue_init()

初始化用于初始各个字段的值，并创建mutex和cond：

```
/* packet queue handling */
static int packet_queue_init(PacketQueue *q)
```

```
{
    memset(q, 0, sizeof(PacketQueue));
    q->mutex = SDL_CreateMutex();
    if (!q->mutex) {
        av_log(NULL, AV_LOG_FATAL, "SDL_CreateMutex(): %s\n", SDL_GetError());
        return AVERROR(ENOMEM);
    }
    q->cond = SDL_CreateCond();
    if (!q->cond) {
        av_log(NULL, AV_LOG_FATAL, "SDL_CreateCond(): %s\n", SDL_GetError());
        return AVERROR(ENOMEM);
    }
    q->abort_request = 1; // 在packet_queue_start和packet_queue_abort时修改到该值
    return 0;
}
```

packet_queue_destroy()

相应的，packet_queue_destroy()销毁过程负责清理mutex和cond:

```
static void packet_queue_destroy(PacketQueue *q)
{
    packet_queue_flush(q); //先清除所有的节点
    SDL_DestroyMutex(q->mutex);
    SDL_DestroyCond(q->cond);
}
```

packet_queue_start()

启动队列

```
static void packet_queue_start(PacketQueue *q)
{
    SDL_LockMutex(q->mutex);
```

```
    q->abort_request = 0;
    packet_queue_put_private(q, &flush_pkt); //这里放入了一个flush_pkt，问题：目的是什么
    SDL_UnlockMutex(q->mutex);
}
```

flush_pkt定义是`static AVPacket flush_pkt;`，是一个特殊的packet，主要用来作为非连续的两端数据的“分界”标记：

- 插入`flush_pkt`触发PacketQueue其对应的serial，加1操作
- 触发解码器清空自身缓存 avcodec_flush_buffers()，以备新序列的数据进行新解码

packet_queue_abort()

中止队列：

```
static void packet_queue_abort(PacketQueue *q)
{
    SDL_LockMutex(q->mutex);

    q->abort_request = 1;        // 请求退出

    SDL_CondSignal(q->cond);    //释放一个条件信号

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

这里SDL_CondSignal的作用在于确保当前等待该条件的线程能被激活并继续执行退出流程，并唤醒者会检测abort_request标志确定自己的退出流程。

packet_queue_put()

读、写是PacketQueue的主要方法。
先看写——往队列中放入一个节点：

```
static int packet_queue_put(PacketQueue *q, AVPacket *pkt)
{
    int ret;

```

```
    SDL_LockMutex(q->mutex);
    ret = packet_queue_put_private(q, pkt);//主要实现
    SDL_UnlockMutex(q->mutex);

    if (pkt != &flush_pkt && ret < 0)
        av_packet_unref(pkt);       //放入失败，释放AVPacket

    return ret;
}
```

主要实现在函数`packet_queue_put_private`，这里需要注意的是**如果插入失败，则需要释放AVPacke**t。

我们再分析packet_queue_put_private：

```
static int packet_queue_put_private(PacketQueue *q, AVPacket *pkt)
{
    MyAVPacketList *pkt1;

    if (q->abort_request)   //如果已中止，则放入失败
       return -1;

    pkt1 = av_malloc(sizeof(MyAVPacketList));   //分配节点内存
    if (!pkt1)  //内存不足，则放入失败
        return -1;
    // 没有做引用计数，那这里也说明av_read_frame不会释放替用户释放buffer。
    pkt1->pkt = *pkt; //拷贝AVPacket(浅拷贝，AVPacket.data等内存并没有拷贝)
    pkt1->next = NULL;
    if (pkt == &flush_pkt)//如果放入的是flush_pkt，需要增加队列的播放序列号，以区分不连续的两段数据
    {
        q->serial++;
        printf("q->serial = %d\n", q->serial++);
    }
    pkt1->serial = q->serial;   //用队列序列号标记节点
    /* 队列操作：如果last_pkt为空，说明队列是空的，新增节点为队头；
     * 否则，队列有数据，则让原队尾的next为新增节点。 最后将队尾指向新增节点
     */
```

```
    if (!q->last_pkt)
        q->first_pkt = pkt1;
    else
        q->last_pkt->next = pkt1;
    q->last_pkt = pkt1;

    //队列属性操作：增加节点数、cache大小、cache总时长，用来控制队列的大小
    q->nb_packets++;
    q->size += pkt1->pkt.size + sizeof(*pkt1);
    q->duration += pkt1->pkt.duration;

    /* XXX: should duplicate packet data in DV case */
    //发出信号，表明当前队列中有数据了，通知等待中的读线程可以取数据了
    SDL_CondSignal(q->cond);
    return 0;
}
```

对于packet_queue_put_private主要完成3件事：

- 计算serial。serial标记了这个节点内的数据是何时的。一般情况下新增节点与上一个节点的serial是一样的，但当队列中加入一个flush_pkt后，后续节点的serial会比之前大1，用来区别不同播放序列的packet.
- 节点入队列操作。
- 队列属性操作。更新队列中节点的数目、占用字节数（含AVPacket.data的大小）及其时长。**主要用来控制Packet队列的大小，我们PacketQueue链表式的队列，在内存充足的条件下我们可以无限put入packet，如果我们要控制队列大小，则需要通过其变量size、duration、nb_packets三者单一或者综合去约束队列的节点的数量，具体在read_thread进行分析。**

packet_queue_get()

从队列中取一个节点：

```
/**
 * @brief packet_queue_get
 * @param q 队列
 * @param pkt 输出参数，即MyAVPacketList.pkt
 * @param block 调用者是否需要在没节点可取的情况下阻塞等待
 * @param serial 输出参数，即MyAVPacketList.serial
 * @return <0: aborted; =0: no packet; >0: has packet
 */
static int packet_queue_get(PacketQueue *q, AVPacket *pkt, int block
```

```
, int *serial)
{
    MyAVPacketList *pkt1;
    int ret;

    SDL_LockMutex(q->mutex);    // 加锁

    for (;;) {
        if (q->abort_request) {
            ret = -1;
            break;
        }

        pkt1 = q->first_pkt;    //MyAVPacketList *pkt1; 从队头拿数据
        if (pkt1) {     //队列中有数据
            q->first_pkt = pkt1->next;  //队头移到第二个节点
            if (!q->first_pkt)
                q->last_pkt = NULL;
            q->nb_packets--;    //节点数减1
            q->size -= pkt1->pkt.size + sizeof(*pkt1);  //cache大小扣除一个节点
            q->duration -= pkt1->pkt.duration;  //总时长扣除一个节点
            //返回AVPacket，这里发生一次AVPacket结构体拷贝，AVPacket的data只拷贝了指针
            *pkt = pkt1->pkt;
            if (serial) //如果需要输出serial，把serial输出
                *serial = pkt1->serial;
            av_free(pkt1);      //释放节点内存,只是释放节点，而不是释放AVPacket
            ret = 1;
            break;
        } else if (!block) {    //队列中没有数据，且非阻塞调用
            ret = 0;
            break;
        } else {    //队列中没有数据，且阻塞调用
            //这里没有break。for循环的另一个作用是在条件变量满足后重复上述代码取出节点
            SDL_CondWait(q->cond, q->mutex);
        }
    }
```

```
45      SDL_UnlockMutex(q->mutex);  // 释放锁
46      return ret;
47  }
```

该函数整体流程：

- 加锁
- 进入for循环，如果需要退出for循环，则break；当没有数据可读且block为1时则等待
  - ret = –1 终止获取packet
  - ret = 0 没有读取到packet
  - ret = 1 获取到了packet
- 释放锁

如果有取到数据，主要分3个步骤：

1. 队列操作：出队列操作；`nb_packets相应-1; duration`的也相应减少，`size`也相应占用的字节大小（`pkt1->pkt.size + sizeof(*pkt1)`）
2. 给输出参数赋值：就是MyAVPacketList的成员传递给输出参数pkt和serial
3. 释放节点内存：释放放入队列时申请的节点内存（注意是节点内存而不是AVPacket的数据的内存）

packet_queue_put_nullpacket()

放入“空包”（nullpacket）。放入空包意味着流的结束，一般在媒体数据读取完成的时候放入空包。放入空包，目的是为了冲刷解码器，将编码器里面所有frame都读取出来：

```
1 static int packet_queue_put_nullpacket(PacketQueue *q, int stream_ind
  ex)
2 {
3     AVPacket pkt1, *pkt = &pkt1;
4     av_init_packet(pkt);
5     pkt->data = NULL;
6     pkt->size = 0;
7     pkt->stream_index = stream_index;
8     return packet_queue_put(q, pkt);
9 }
```

文件数据读取完毕后刷入空包。

packet_queue_flush()

packet_queue_flush用于将packet队列中的所有节点清除，包括节点对应的AVPacket。比如用于退出播放和seek播放：

- 退出播放，则要清空packet queue的节点
- seek播放，要清空seek之前缓存的节点数据，以便插入新节点数据

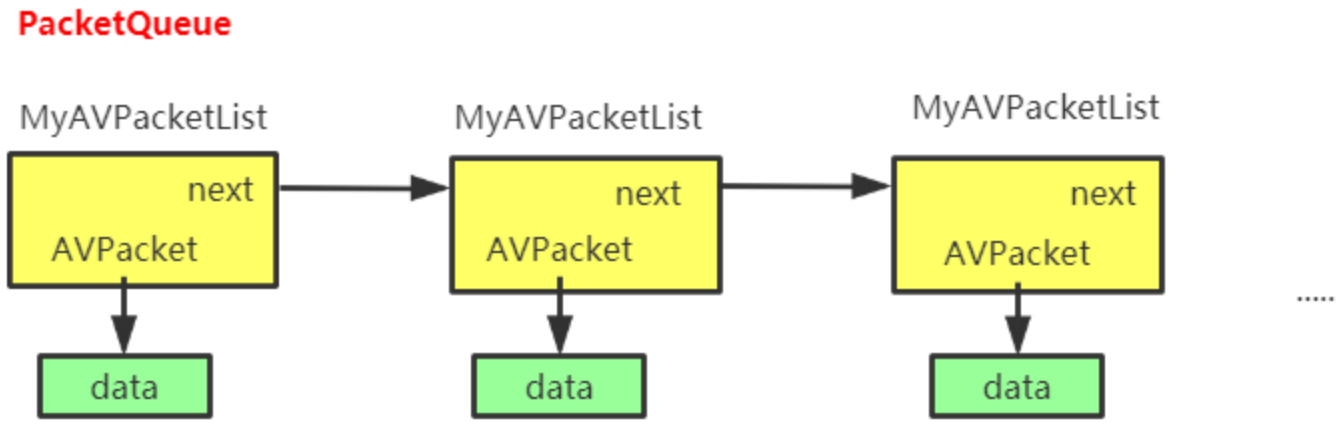
```
static void packet_queue_flush(PacketQueue *q)
{
    MyAVPacketList *pkt, *pkt1;

    SDL_LockMutex(q->mutex);
    for (pkt = q->first_pkt; pkt; pkt = pkt1) {
        pkt1 = pkt->next;
        av_packet_unref(&pkt->pkt); // 释放AVPacket的数据
        av_freep(&pkt);
    }
    q->last_pkt = NULL;
    q->first_pkt = NULL;
    q->nb_packets = 0;
    q->size = 0;
    q->duration = 0;
    SDL_UnlockMutex(q->mutex);
}
```

函数主体的for循环是队列遍历，**遍历过程释放节点和AVPacket(AVpacket对应的数据也被释放掉)**。最后将PacketQueue的属性恢复为空队列状态。

PacketQueue总结

前面我们分析了PacketQueue的实现和主要的操作方法，现在总结下两个关键的点：

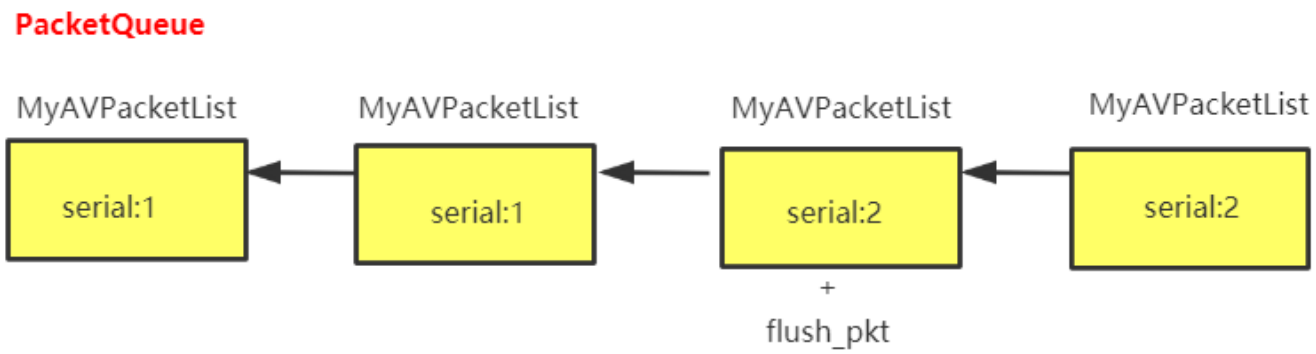
第一，PacketQueue的内存管理：



MyAVPacketList的内存是完全由PacketQueue维护的，在put的时候malloc，在get的时候free。
AVPacket分两块：

- 一部分是AVPacket结构体的内存，这部分从MyAVPacketList的定义可以看出是和MyAVPacketList共存亡的。
- 另一部分是AVPacket字段指向的内存，这部分一般通过`av_packet_unref`函数释放。一般情况下，是在get后**由调用者**负责用`av_packet_unref`函数释放。特殊的情况是当碰到`packet_queue_flush`或put失败时，这时需要队列自己处理。

第二，serial的变化过程：



如上图所示，左边是队头，右边是队尾，从左往右标注了4个节点的serial，以及放入对应节点时queue的serial。
可以看到放入flush_pkt的时候后，serial增加了1.
假设，现在要从队头取出一个节点，那么取出的节点是serial 1，而PacketQueue自身的queue已经增长到了2。

PacketQueue设计思路：

1. 设计一个多线程安全的队列，保存AVPacket，同时统计队列内已缓存的数据大小。（这个统计数据会用来后续设置要缓存的数据量）
2. 引入serial的概念，区别前后数据包是否连续，主要应用于seek操作。

3. 设计了两类特殊的packet——flush_pkt和nullpkt（类似用于多线程编程的事件模型——往队列中放入flush事件、放入null事件），我们在音频输出、视频输出、播放控制等模块时也会继续对flush_pkt和nullpkt的作用展开分析。

struct Frame 和 FrameQueue队列

1 Frame

```
typedef struct Frame {
    AVFrame  *frame;          // 指向数据帧
    AVSubtitle  sub;             // 用于字幕
    int  serial;              // 播放序列，在seek的操作时serial会变化
    double  pts;             // 时间戳，单位为秒
    double  duration;        // 该帧持续时间，单位为秒
    int64_t  pos;             // 该帧在输入文件中的字节位置
    int  width;               // 图像宽度
    int  height;              // 图像高读
    int  format;              // 对于图像为(enum AVPixelFormat),
                                 // 对于声音则为(enum AVSampleFormat)
    AVRational  sar;             // 图像的宽高比，如果未知或未指定则为0/1
    int  uploaded;            // 用来记录该帧是否已经显示过?
    int  flip_v;              // =1则旋转180,  = 0则正常播放
} Frame;
```

真正存储解码后音视频数据的结构体为AVFrame ，存储字幕则使用AVSubtitle，该Frame的设计是为了音频、视频、字幕帧通用，所以Frame结构体的设计类似AVFrame，部分成员变量只对不同类型有作用，比如sar只对视频有作用。
里面也包含了serial播放序列（每次seek时都切换serial），sar（图像的宽高比（16:9，4:3...），该值来自AVFrame结构体的sample_aspect_ratio变量）。

2 FrameQueue

```
typedef struct FrameQueue {
    Frame   queue[FRAME_QUEUE_SIZE];        // FRAME_QUEUE_SIZE  最大size, 数字太大时会占用大量的内存，需要注意该值的设置
    int  rindex;                            // 读索引。待播放时读取此帧进行播放，播放后此帧成为上一帧
```

```
    int  windex;                          // 写索引
    int  size;                            // 当前总帧数
    int  max_size;                        // 可存储最大帧数
    int  keep_last;                       // = 1说明要在队列里面保持最后一帧的数据不释放，只在销毁队列的时候才将其真正释放
    int  rindex_shown;                    // 初始化为0，配合keep_last=1使用
    SDL_mutex   *mutex;                   // 互斥量
    SDL_cond    *cond;                    // 条件变量
    PacketQueue *pktq;                    // 数据包缓冲队列
} FrameQueue;
```

FrameQueue是一个环形缓冲区(ring buffer)，是用数组实现的一个FIFO。**数组方式**的环形缓冲区适合于事先明确了缓冲区的最大容量的情形。

ffplay中创建了三个frame_queue：音频frame_queue，视频frame_queue，字幕frame_queue。每一个frame_queue一个写端一个读端，写端位于解码线程，读端位于播放线程。

FrameQueue的设计比如PacketQueue复杂，引入了读取节点但节点不出队列的操作、读取下一节点也不出队列等等的操作，FrameQueue操作提供以下方法：

- frame_queue_unref_item：释放Frame里面的AVFrame和 AVSubtitle
- frame_queue_init：初始化队列
- frame_queue_destory：销毁队列
- frame_queue_signal：发送唤醒信号
- frame_queue_peek：获取当前Frame，调用之前先调用frame_queue_nb_remaining确保有frame可读
- frame_queue_peek_next：获取当前Frame的下一Frame，调用之前先调用frame_queue_nb_remaining确保至少有2 Frame在队列
- frame_queue_peek_last：获取上一Frame
- frame_queue_peek_writable：获取一个可写Frame，可以以阻塞或非阻塞方式进行
- frame_queue_peek_readable：获取一个可读Frame，可以以阻塞或非阻塞方式进行
- frame_queue_push：更新写索引，此时Frame才真正入队列，队列节点Frame个数加1
- frame_queue_next：更新读索引，此时Frame才真正出队列，队列节点Frame个数减1，内部调用frame_queue_unref_item是否对应的AVFrame和AVSubtitle
- frame_queue_nb_remaining：获取队列Frame节点个数
- frame_queue_last_pos：获取最近播放Frame对应数据在媒体文件的位置，主要在seek时使用

frame_queue_init() 初始化

```c
static int frame_queue_init(FrameQueue *f, PacketQueue *pktq, int max_size, int keep_last)
{
    int i;
    memset(f, 0, sizeof(FrameQueue));
    if (!(f->mutex = SDL_CreateMutex())) {
        av_log(NULL, AV_LOG_FATAL, "SDL_CreateMutex(): %s\n", SDL_GetError());
        return AVERROR(ENOMEM);
    }
    if (!(f->cond = SDL_CreateCond())) {
        av_log(NULL, AV_LOG_FATAL, "SDL_CreateCond(): %s\n", SDL_GetError());
        return AVERROR(ENOMEM);
    }
    f->pktq = pktq;
    f->max_size = FFMIN(max_size, FRAME_QUEUE_SIZE);
    f->keep_last = !!keep_last;
    for (i = 0; i < f->max_size; i++)
        if (!(f->queue[i].frame = av_frame_alloc())) // 分配AVFrame结构体
            return AVERROR(ENOMEM);
    return 0;
}
```

队列初始化函数确定了队列大小，将为队列中每一个节点的frame(`f->queue[i].frame`)分配内存，注意只是分配Frame对象本身，而不关注Frame中的数据缓冲区。Frame中的数据缓冲区是AVBuffer，使用引用计数机制。

`f->max_size`是队列的大小，此处值为16（由FRAME_QUEUE_SIZE定义），实际分配的时候视频为3，音频为9，字幕为16，因为这里存储的是解码后的数据，不宜设置过大，比如视频当为1080p时，如果为YUV420p格式，一帧就有3110400字节。

- #define VIDEO_PICTURE_QUEUE_SIZE 3 // 图像帧缓存数量
- #define SUBPICTURE_QUEUE_SIZE 16 // 字幕帧缓存数量
- #define SAMPLE_QUEUE_SIZE 9 // 采样帧缓存数量
- #define FRAME_QUEUE_SIZE FFMAX(SAMPLE_QUEUE_SIZE, FFMAX(VIDEO_PICTURE_QUEUE_SIZE, SUBPICTURE_QUEUE_SIZE))

`f->keep_last`是队列中是否保留最后一次播放的帧的标志。`f->keep_last = !!keep_last`是将int取值的keep_last转换为boot取值(0或1)。

frame_queue_destory()销毁

```
1  static void frame_queue_destory(FrameQueue *f)
2  {
3      int i;
4      for (i = 0; i < f->max_size; i++) {
5          Frame *vp = &f->queue[i];
6          // 释放对vp->frame中的数据缓冲区的引用，注意不是释放frame对象本身
7          frame_queue_unref_item(vp);
8           // 释放vp->frame对象
9          av_frame_free(&vp->frame);
10     }
11     SDL_DestroyMutex(f->mutex);
12     SDL_DestroyCond(f->cond);
13 }
```

队列销毁函数对队列中的每个节点作了如下处理：

1. `frame_queue_unref_item(vp)`释放本队列对vp->frame中AVBuffer的引用
2. `av_frame_free(&vp->frame)`释放vp->frame对象本身

frame_queue_peek_writable()获取可写Frame

frame_queue_push()入队列

FrameQueue写队列的步骤和PacketQueue不同，分了3步进行：

1. 调用frame_queue_peek_writable获取可写的Frame，如果队列已满则等待
2. 获取到Frame后，设置Frame的成员变量
3. 再调用frame_queue_push更新队列的写索引，真正将Frame入队列

```
1 Frame *frame_queue_peek_writable(FrameQueue *f);     // 获取可写帧
2 void frame_queue_push(FrameQueue *f);                // 更新写索引
```

通过实例看一下写队列的用法：

```
// video AVFrame 入队列
int queue_picture(VideoState *is, AVFrame *src_frame, double pts, double duration, int64_t pos, int serial)
{
    Frame *vp;

    if (!(vp = frame_queue_peek_writable(&is->pictq))) // 检测队列是否有可写空间
        return -1;      // Frame队列满了则返回-1
    // 执行到这步说已经获取到了可写入的Frame
    vp->sar = src_frame->sample_aspect_ratio;
    vp->uploaded = 0;

    vp->width = src_frame->width;
    vp->height = src_frame->height;
    vp->format = src_frame->format;

    vp->pts = pts;
    vp->duration = duration;
    vp->pos = pos;
    vp->serial = serial;

    set_default_window_size(vp->width, vp->height, vp->sar);

    av_frame_move_ref(vp->frame, src_frame); // 将src中所有数据拷贝到dst中，并复位src。
    frame_queue_push(&is->pictq);   // 更新写索引位置
    return 0;
}
```

上面一段代码是视频解码线程向视频frame_queue中写入一帧的代码，步骤如下：

1. `frame_queue_peek_writable(&is->pictq)`向队列尾部申请一个可写的帧空间，若队列已满无空间可写，则等待(由SDL_cond *cond控制，由frame_queue_next或frame_queue_signal触发唤醒)
2. `av_frame_move_ref(vp->frame, src_frame)`将src_frame中所有数据拷贝到vp->frame并复位src_frame，vp->frame中AVBuffer使用引用计数机制，不会执行AVBuffer的拷贝动作，仅是修改指针指向值。为避免

内存泄漏，在`av_frame_move_ref(dst, src)`之前应先调用`av_frame_unref(dst)`，这里没有调用，是因为frame_queue在删除一个节点时，已经释放了frame及frame中的AVBuffer。

3. `frame_queue_push(&is->pictq)`此步仅将frame_queue中的写索引加1，实际的数据写入在此步之前已经完成。

frame_queue_peek_writable获取可写Frame指针

```
// 获取可写指针
static Frame *frame_queue_peek_writable(FrameQueue *f)
{
    /* wait until we have space to put a new frame */
    SDL_LockMutex(f->mutex);
    while (f->size >= f->max_size &&
           !f->pktq->abort_request) {   /* 检查是否需要退出 */
        SDL_CondWait(f->cond, f->mutex);
    }
    SDL_UnlockMutex(f->mutex);

    if (f->pktq->abort_request)          /* 检查是不是要退出 */
        return NULL;

    return &f->queue[f->windex];
}
```

向队列尾部申请一个可写的帧空间，若无空间可写，则等待。

这里最需要体会到的是abort_request的使用，在等待时如果播放器需要退出则将abort_request = 1，那frame_queue_peek_writable函数可以知道是正常frame可写唤醒，还是其他唤醒。

frame_queue_push()

```
// 更新写索引
static void frame_queue_push(FrameQueue *f)
{
    if (++f->windex == f->max_size)
        f->windex = 0;
    SDL_LockMutex(f->mutex);
    f->size++;
    SDL_CondSignal(f->cond);    // 当frame_queue_peek_readable在等待时则可以唤醒
```

```
    SDL_UnlockMutex(f->mutex);
}
```

向队列尾部压入一帧，只更新计数与写指针，因此调用此函数前应将帧数据写入队列相应位置。SDL_CondSignal(f->cond);可以唤醒读frame_queue_peek_readable。

frame_queue_peek_readable() 获取可读Frame

frame_queue_next()出队列

写队列中，应用程序写入一个新帧后通常总是将写索引加1。而读队列中，“读取”和“更新读索引(同时删除旧帧)”二者是独立的，可以只读取而不更新读索引，也可以只更新读索引(只删除)而不读取（只有更新读索引的时候才真正释放对应的Frame数据）。而且读队列引入了是否保留已显示的最后一帧的机制，导致读队列比写队列要复杂很多。

读队列和写队列步骤是类似的，基本步骤如下：

1. 调用frame_queue_peek_readable获取可读Frame；
2. 如果需要更新读索引（出队列该节点）则调用frame_queue_peek_next；

读队列涉及如下函数：

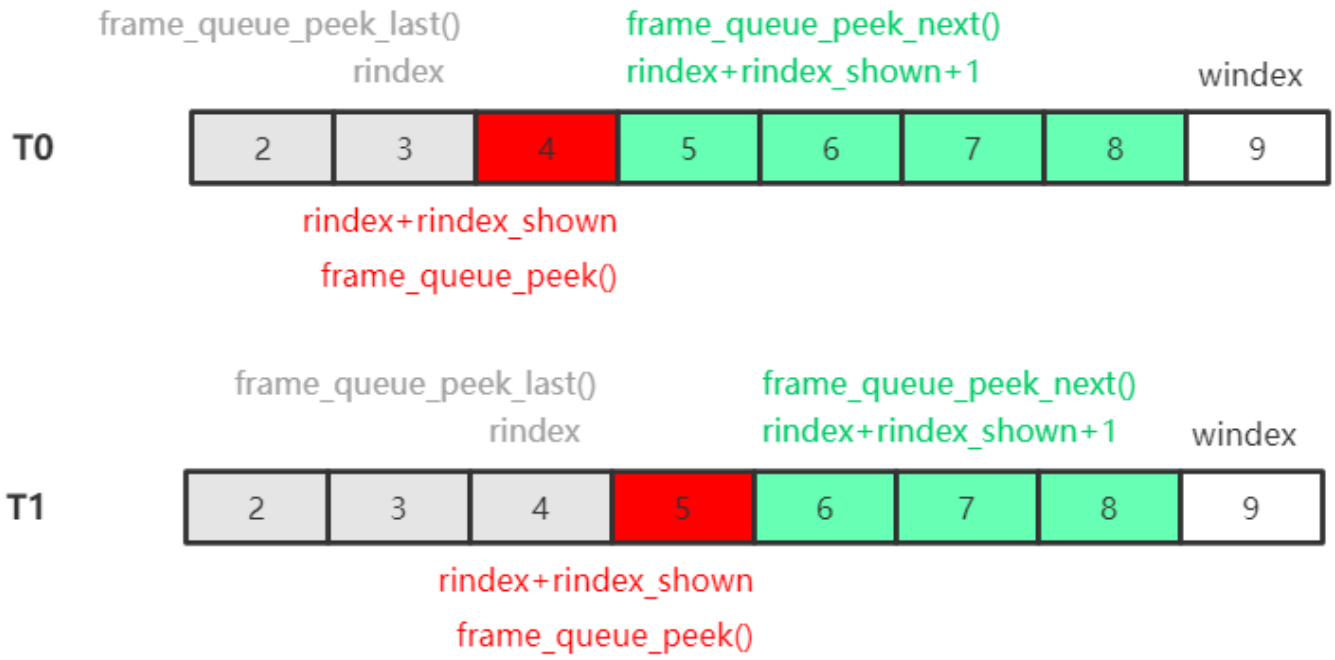
```
Frame *frame_queue_peek_readable(FrameQueue *f);      // 获取可读Frame指针(若读空则等待)
Frame *frame_queue_peek(FrameQueue *f);               // 获取当前Frame指针
Frame *frame_queue_peek_next(FrameQueue *f);          // 获取下一Frame指针
Frame *frame_queue_peek_last(FrameQueue *f);          // 获取上一Frame指针
void frame_queue_next(FrameQueue *f);                 // 更新读索引(同时删除旧frame)
```

通过实例看一下读队列的用法：

```
static void video_refresh(void *opaque, double *remaining_time)
{
    ......
    if (frame_queue_nb_remaining(&is->pictq) == 0) {    // 所有帧已显示
        // nothing to do, no picture to display in the queue
    } else {
        Frame *vp, *lastvp;
```

```
        lastvp = frame_queue_peek_last(&is->pictq);     // 上一帧：上次已显示的帧
        vp = frame_queue_peek(&is->pictq);              // 当前帧：当前待显示的帧
        frame_queue_next(&is->pictq);                   // 出队列，并更新rindex
        video_display(is)-->video_image_display()-->frame_queue_peek_last();
    }
    ......
}
```

上面一段代码是视频播放线程从视频frame_queue中读取视频帧进行显示的基本步骤，其他代码已省略，只保留了读队列部分。
记lastvp为上一次已播放的帧，vp为本次待播放的帧，下图中方框中的数字表示显示序列中帧的序号：



在启用keep_last机制后，rindex_shown值总是为1，rindex_shown确保了最后播放的一帧总保留在队列中。
假设某次进入`video_refresh()`的时刻为T0，下次进入的时刻为T1。在T0时刻，读队列的步骤如下：

1. rindex表示上一次播放的帧lastvp，本次调用`video_refresh()`中，lastvp会被删除，rindex会加1，即是当调用frame_queue_next删除的是lastvp，而不是当前的vp，当前的vp转为lastvp。
2. rindex+rindex_shown表示本次待播放的帧vp，本次调用`video_refresh()`中，vp会被读出播放图中已播放的帧是灰色方框，本次待播放的帧是红色方框，其他未播放的帧是绿色方框，队列中空位置

为白色方框。

3. rindex+rindex_shown+1表示下一帧nextvp

frame_queue_nb_remaining()获取队列的size

```
/* return the number of undisplayed frames in the queue */
static int frame_queue_nb_remaining(FrameQueue *f)
{
    return f->size - f->rindex_shown;
}
```

rindex_shown为1时，队列中总是保留了最后一帧lastvp(灰色方框)。需要注意的时候rindex_shown的值就是0或1，不存在变为2，3等的可能。**在计算队列当前Frame数量是不包含lastvp。**

rindex_shown的引入增加了读队列操作的理解难度。大多数读操作函数都会用到这个变量。
通过`FrameQueue.keep_last`和`FrameQueue.rindex_shown`两个变量实现了保留最后一次播放帧的机制。
是否启用keep_last机制是由全局变量`keep_last`值决定的，在队列初始化函数`frame_queue_init()`中有`f->keep_last = !!keep_last;`，而在更新读指针函数`frame_queue_next()`中如果启用keep_last机制，则`f->rindex_shown`值为1。

我们具体分析下`frame_queue_next()`函数：

```
/* 释放当前frame，并更新读索引rindex，
 * 当keep_last为1, rindex_show为0时不去更新rindex,也不释放当前frame */
static void frame_queue_next(FrameQueue *f)
{
    if (f->keep_last && !f->rindex_shown) {
        f->rindex_shown = 1; // 第一次调用置为1
        return;
    }
    frame_queue_unref_item(&f->queue[f->rindex]);
    if (++f->rindex == f->max_size)
        f->rindex = 0;
    SDL_LockMutex(f->mutex);
    f->size--;
    SDL_CondSignal(f->cond);
```

```
    SDL_UnlockMutex(f->mutex);
}
```

主要步骤：

- 在启用keeplast时，如果rindex_shown为0则将其设置为1，并返回。此时并不会更新读索引。也就是说keeplast机制实质上也会占用着队列Frame的size，当调用frame_queue_nb_remaining()获取size时并不能将其计算入size；
- 释放Frame对应的数据（比如AVFrame的数据），但不释放Frame本身
- 更新读索引
- 释放唤醒信号，以唤醒正在等待写入的线程。

frame_queue_peek_readable()的具体实现

```
static Frame *frame_queue_peek_readable(FrameQueue *f)
{
    /* wait until we have a readable a new frame */
    SDL_LockMutex(f->mutex);
    while (f->size - f->rindex_shown <= 0 &&
           !f->pktq->abort_request) {
        SDL_CondWait(f->cond, f->mutex);
    }
    SDL_UnlockMutex(f->mutex);

    if (f->pktq->abort_request)
        return NULL;

    return &f->queue[(f->rindex + f->rindex_shown) % f->max_size];
}
```

从队列头部读取一帧(vp)，只读取不删除，若无帧可读则等待。这个函数和`frame_queue_peek()`的区别仅仅是多了不可读时等待的操作。

frame_queue_peek()获取当前帧

```
/* 获取队列当前Frame, 在调用该函数前先调用frame_queue_nb_remaining确保有frame可读 */
static Frame *frame_queue_peek(FrameQueue *f)
```

```
{
    return &f->queue[(f->rindex + f->rindex_shown) % f->max_size];
}
```

frame_queue_peek_next()获取下一帧

```
/* 获取当前Frame的下一Frame, 此时要确保queue里面至少有2个Frame */
static Frame *frame_queue_peek_next(FrameQueue *f)
{
    return &f->queue[(f->rindex + f->rindex_shown + 1) % f->max_size]
;
}
```

frame_queue_peek_last()获取上一帧

```
/* 获取last Frame:
 * 当rindex_shown=0时, 和frame_queue_peek效果一样
 * 当rindex_shown=1时, 读取的是已经显示过的frame
 */
static Frame *frame_queue_peek_last(FrameQueue *f)
{
    return &f->queue[f->rindex];
}
```

struct AudioParams 音频参数

```
typedef struct AudioParams {
    int     freq;                   // 采样率
    int     channels;               // 通道数
    int64_t  channel_layout;          // 通道布局, 比如2.1声道, 5.1声道等
    enum AVSampleFormat fmt;             // 音频采样格式, 比如AV_SAMPLE_FM
T_S16表示为有符号16bit深度, 交错排列模式。
    int     frame_size;             // 一个采样单元占用的字节数 (比如2通道
```

```
    时，则左右通道各采样一次合成一个采样单元）
    int     bytes_per_sec;          // 一秒时间的字节数，比如采样率48Khz，2 channel，16bit，则一秒48000*2*16/8=192000
} AudioParams;
```

struct Decoder解码器封装

```
/**
 * 解码器封装
 */
typedef struct Decoder {
    AVPacket pkt;
    PacketQueue *queue;          // 数据包队列
    AVCodecContext  *avctx;      // 解码器上下文
    int     pkt_serial;          // 包序列
    int     finished;            // =0，解码器处于工作状态；=非0，解码器处于空闲状态
    int     packet_pending;      // =0，解码器处于异常状态，需要考虑重置解码器；=1，解码器处于正常状态
    SDL_cond    *empty_queue_cond;  // 检查到packet队列空时发送 signal缓存read_thread读取数据
    int64_t     start_pts;          // 初始化时是stream的start time
    AVRational  start_pts_tb;       // 初始化时是stream的time_base
    int64_t     next_pts;           // 记录最近一次解码后的frame的pts，当解出来的部分帧没有有效的pts时则使用next_pts进行推算
    AVRational  next_pts_tb;        // next_pts的单位
    SDL_Thread  *decoder_tid;       // 线程句柄
} Decoder;
```