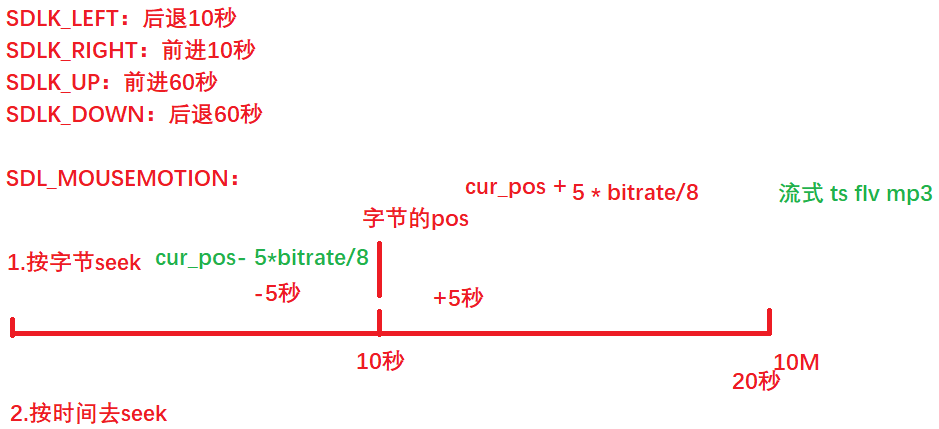
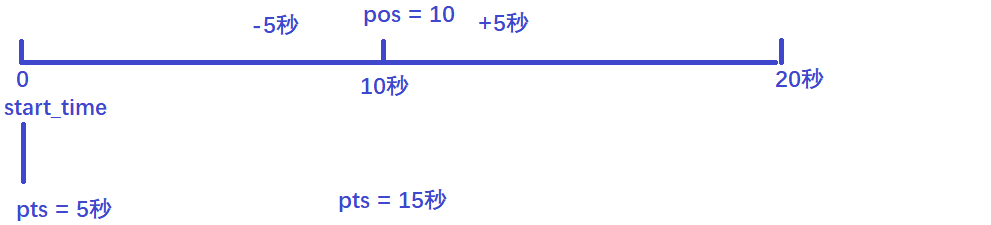
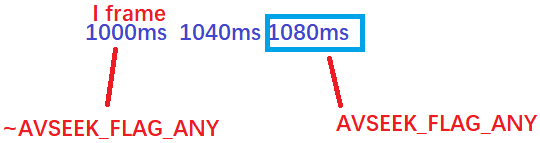
ffplay播放器-17-快进快退seek

《FFmpeg/WebRTC/RTMP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

上课演示图







(1) 清空packet queue；
(2) 开启新的播放序列；
(3)清空解码器缓存
(4)seek_req重置为0；
(5)eof = 0
(6)如果是暂停暂停，那要把seek后的第一帧播放出来

	字节	时间戳
ts	1 不太好	1 不太好
flv	1 支持不太好	1
mp4	0	1

快进快退，seek

快进、快退、seek在ffplay的实现是一样的。

- **快进和快退的本质是seek到某个点重新开始播放。**
  - 跳转到指定的数据位置avformat_seek_file
  - 清空packet队列
  - 清空frame队列（在ffplay里面是通过serial去控制）

- 插入flush_pkt以便冲刷解码器
- 切换时钟序列（ffplay）
- 清空解码器

seek操作

具体操作：

SDLK_LEFT：后退10秒

SDLK_RIGHT：前进10秒

SDLK_UP：前进60秒

SDLK_DOWN：后退60秒

最终都调用的是do_seek -> stream_seek()

SDL_MOUSEMOTION：鼠标右键按下，seek到指定的位置，最终也是调用stream_seek()

重点内容

重点内容

1. seek_target位置的计算
2. avformat_seek_file 接口

注意：**不同的容器（比如MP4和FLV）seek的机制是不一样的**。有些容器seek的时间会快些，有些则相对耗时。这个和容器的存储结构有关系。

```
/**
 * Seek to timestamp ts. 搜索时间戳TS
 * Seeking will be done so that the point from which all active stre
ams
 * can be presented successfully will be closest to ts and within mi
n/max_ts.

 * Active streams are all streams that have AVStream.discard < AVDIS
CARD_ALL.
 *
 * If flags contain AVSEEK_FLAG_BYTE, then all timestamps are in byt
```

```
es and
 9  * are the file position (this may not be supported by all demuxer
   s).
10  * If flags contain AVSEEK_FLAG_FRAME, then all timestamps are in fr
   ames
11  * in the stream with stream_index (this may not be supported by all
   demuxers).
12  * Otherwise all timestamps are in units of the stream selected by s
   tream_index
13  * or if stream_index is -1, in AV_TIME_BASE units.
14  * If flags contain AVSEEK_FLAG_ANY, then non-keyframes are treated
    as
15  * keyframes (this may not be supported by all demuxers).
16  * If flags contain AVSEEK_FLAG_BACKWARD, it is ignored.
17  *
18  * @param s media file handle
19  * @param stream_index index of the stream which is used as time bas
   e reference
20  * @param min_ts smallest acceptable timestamp
21  * @param ts target timestamp
22  * @param max_ts largest acceptable timestamp
23  * @param flags flags
24  * @return >=0 on success, error code otherwise
25  *
26  * @note This is part of the new seek API which is still under const
   ruction.
27  *       Thus do not use this yet. It may change at any time, do not
   expect
28  *       ABI compatibility yet!
29  */
30 int avformat_seek_file(AVFormatContext *s, int stream_index, int64_t
   min_ts, int64_t ts, int64_t max_ts, int flags);
```

数据结构及SEEK标志

相关数据变量定义如下：

```
1 typedef struct VideoState {
2     ......
```

```
    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;               // SEEK的目标位置(当前位置+增量)
    int64_t seek_rel;               // 本次SEEK的位置增量
    ......
} VideoState;
```

SEEK操作的实现

在解复用线程主循环中处理了SEEK操作。

```
static int read_thread(void *arg)
{
    ......
    for (;;) {
        if (is->seek_req) {
            int64_t seek_target = is->seek_pos;
            int64_t seek_min    = is->seek_rel > 0 ? seek_target - is->seek_rel + 2: INT64_MIN;
            int64_t seek_max    = is->seek_rel < 0 ? seek_target - is->seek_rel - 2: INT64_MAX;
// FIXME the +-2 is due to rounding being not done in the correct direction in generation
//      of the seek_pos/seek_rel variables
            ret = avformat_seek_file(is->ic, -1, seek_min, seek_target, seek_max, is->seek_flags);
            if (ret < 0) {
                av_log(NULL, AV_LOG_ERROR,
                       "%s: error while seeking\n", is->ic->url);
            } else {
                if (is->audio_stream >= 0) {
                    packet_queue_flush(&is->audioq);
                    packet_queue_put(&is->audioq, &flush_pkt);
                }
                if (is->subtitle_stream >= 0) {
                    packet_queue_flush(&is->subtitleq);
                    packet_queue_put(&is->subtitleq, &flush_pkt);
                }
                if (is->video_stream >= 0) {
```

```
                    packet_queue_flush(&is->videoq);
                    packet_queue_put(&is->videoq, &flush_pkt);
                }
                if (is->seek_flags & AVSEEK_FLAG_BYTE) {
                   set_clock(&is->extclk, NAN, 0);
                } else {
                   set_clock(&is->extclk, seek_target / (double)AV_TIME_BASE, 0);
                }
            }
            is->seek_req = 0;
            is->queue_attachments_req = 1;
            is->eof = 0;
            if (is->paused)
                step_to_next_frame(is);
        }
    }
    ......
}
```

上述代码中的SEEK操作执行如下步骤：

[1]. 调用`avformat_seek_file()`完成解复用器中的SEEK点切换操作

```
// 函数原型
int avformat_seek_file(AVFormatContext *s, int stream_index, int64_t min_ts, int64_t ts, int64_t max_ts, int flags);
// 调用代码
ret = avformat_seek_file(is->ic, -1, seek_min, seek_target, seek_max, is->seek_flags);
```

这个函数会等待SEEK操作完成才返回。实际的播放点力求最接近参数`ts`，并确保在[min_ts, max_ts]区间内，之所以播放点不一定在`ts`位置，是因为`ts`位置未必能正常播放。

函数与SEEK点相关的三个参数(实参“seek_min”，“seek_target”，“seek_max”)取值方式与SEEK标志有关(实参“is->seek_flags”)，此处“is->seek_flags”值为0，对应7.4.1节中的第[4]中情况。

[2]. 冲洗各解码器缓存帧，使当前播放序列中的帧播放完成，然后再开始新的播放序列(播放序列由各数据结构中的“serial”变量标志，此处不展开)。代码如下：

```
if (is->video_stream >= 0) {
    packet_queue_flush(&is->videoq);
    packet_queue_put(&is->videoq, &flush_pkt);
}
```

[3]. 清除本次SEEK请求标志`is->seek_req = 0;`

mp4和ts文件的seek原理

以对mp4 和ts的seek 逻辑完全不一样

最重要的区别在于mp4 可以seek到指定的时间戳， ts 是 seek到文件的某个position，而不能直接seek到指定的时间点，下面结合ffplay的代码看一下实际的seek逻辑。

对于ts，具体seek操作调用函数关系为 avformat_seek_file()=> av_seek_frame() => seek_frame_internal() => seek_frame_byte()

对于mp4，具体seek操作调用函数关系为 avformat_seek_file()=> av_seek_frame() => seek_frame_internal() =>mov_read_seek()

所以 ts 和 mp4 seek的差别就在最后一个环节， seek_frame_byte() 与 mov_read_seek() 的差别。

下面贴出这几个函数的相关代码，稍加注释：

avformat_seek_file

```
int avformat_seek_file(AVFormatContext *s, int stream_index, int64_t min_ts,
                       int64_t ts, int64_t max_ts, int flags)
{
    if (min_ts > ts || max_ts < ts)
        return -1;
    if (stream_index < -1 || stream_index >= (int)s->nb_streams)
        return AVERROR(EINVAL);

    if (s->seek2any>0)
        flags |= AVSEEK_FLAG_ANY;
    flags &= ~AVSEEK_FLAG_BACKWARD;

    if (s->iformat->read_seek2) {
        int ret;
        ff_read_frame_flush(s);

        if (stream_index == -1 && s->nb_streams == 1) {
```

```
18            AVRational time_base = s->streams[0]->time_base;
19            ts = av_rescale_q(ts, AV_TIME_BASE_Q, time_base);
20            min_ts = av_rescale_rnd(min_ts, time_base.den,
21                                    time_base.num * (int64_t)AV_TIME
   _BASE,
22                                    AV_ROUND_UP   | AV_ROUND_PASS_MI
   NMAX);
23            max_ts = av_rescale_rnd(max_ts, time_base.den,
24                                    time_base.num * (int64_t)AV_TIME
   _BASE,
25                                    AV_ROUND_DOWN | AV_ROUND_PASS_MI
   NMAX);
26            stream_index = 0;
27        }
28
29        ret = s->iformat->read_seek2(s, stream_index, min_ts,
30                                     ts, max_ts, flags);
31
32        if (ret >= 0)
33            ret = avformat_queue_attached_pictures(s);
34        return ret;
35    }
36
37    if (s->iformat->read_timestamp) {
38        // try to seek via read_timestamp()
39    }
40
41    // Fall back on old API if new is not implemented but old is.
42    // Note the old API has somewhat different semantics.
43    if (s->iformat->read_seek || 1) {
44        int dir = (ts - (uint64_t)min_ts > (uint64_t)max_ts - ts ? A
   VSEEK_FLAG_BACKWARD : 0);
45        int ret = av_seek_frame(s, stream_index, ts, flags | dir);
46        if (ret<0 && ts != min_ts && max_ts != ts) {
47            ret = av_seek_frame(s, stream_index, dir ? max_ts : min_
   ts, flags | dir);
48            if (ret >= 0)
49                ret = av_seek_frame(s, stream_index, ts, flags | (di
   r^AVSEEK_FLAG_BACKWARD));
50        }
```

```
        return ret;
    }

    // try some generic seek like seek_frame_generic() but with new
 ts semantics
    return -1; //unreachable
}
```

av_seek_frame

```

int av_seek_frame(AVFormatContext *s, int stream_index,
                  int64_t timestamp, int flags)
{
    int ret;

    if (s->iformat->read_seek2 && !s->iformat->read_seek) {
        int64_t min_ts = INT64_MIN, max_ts = INT64_MAX;
        if ((flags & AVSEEK_FLAG_BACKWARD))
            max_ts = timestamp;
        else
            min_ts = timestamp;
        return avformat_seek_file(s, stream_index, min_ts, timestamp
, max_ts,
                                  flags & ~AVSEEK_FLAG_BACKWARD);
    }

    ret = seek_frame_internal(s, stream_index, timestamp, flags);

    if (ret >= 0)
        ret = avformat_queue_attached_pictures(s);

    return ret;
}
```

intseek_frame_internal

```
static int seek_frame_internal(AVFormatContext *s, int stream_index,
                               int64_t timestamp, int flags)
{
    int ret;
    AVStream *st;

    if (flags & AVSEEK_FLAG_BYTE) { // ts文件走这个if条件的逻辑
        if (s->iformat->flags & AVFMT_NO_BYTE_SEEK)
            return -1;
        ff_read_frame_flush(s);
        return seek_frame_byte(s, stream_index, timestamp, flags);
    }
    // mp4文件走下面的逻辑
    if (stream_index < 0) {
        stream_index = av_find_default_stream_index(s);
        if (stream_index < 0)
            return -1;

        st = s->streams[stream_index];
        /* timestamp for default must be expressed in AV_TIME_BASE u
nits */
        timestamp = av_rescale(timestamp, st->time_base.den,
                               AV_TIME_BASE * (int64_t) st->time_bas
e.num);
    }

    /* first, we try the format specific seek */
    if (s->iformat->read_seek) {
        ff_read_frame_flush(s);
        ret = s->iformat->read_seek(s, stream_index, timestamp, flag
s);
    } else
        ret = -1;
    if (ret >= 0)
        return 0;

    if (s->iformat->read_timestamp &&
        !(s->iformat->flags & AVFMT_NOBINSEARCH)) {
        ff_read_frame_flush(s);
        return ff_seek_frame_binary(s, stream_index, timestamp, flag
```

```
s);
38      } else if (!(s->iformat->flags & AVFMT_NOGENSEARCH)) {
39          ff_read_frame_flush(s);
40          return seek_frame_generic(s, stream_index, timestamp, flags)
;
41      } else
42          return -1;
43  }
```

seek_frame_byte

seek_frame_byte函数：直接seek到文件指定的position，ts 的 seek 调用此函数。

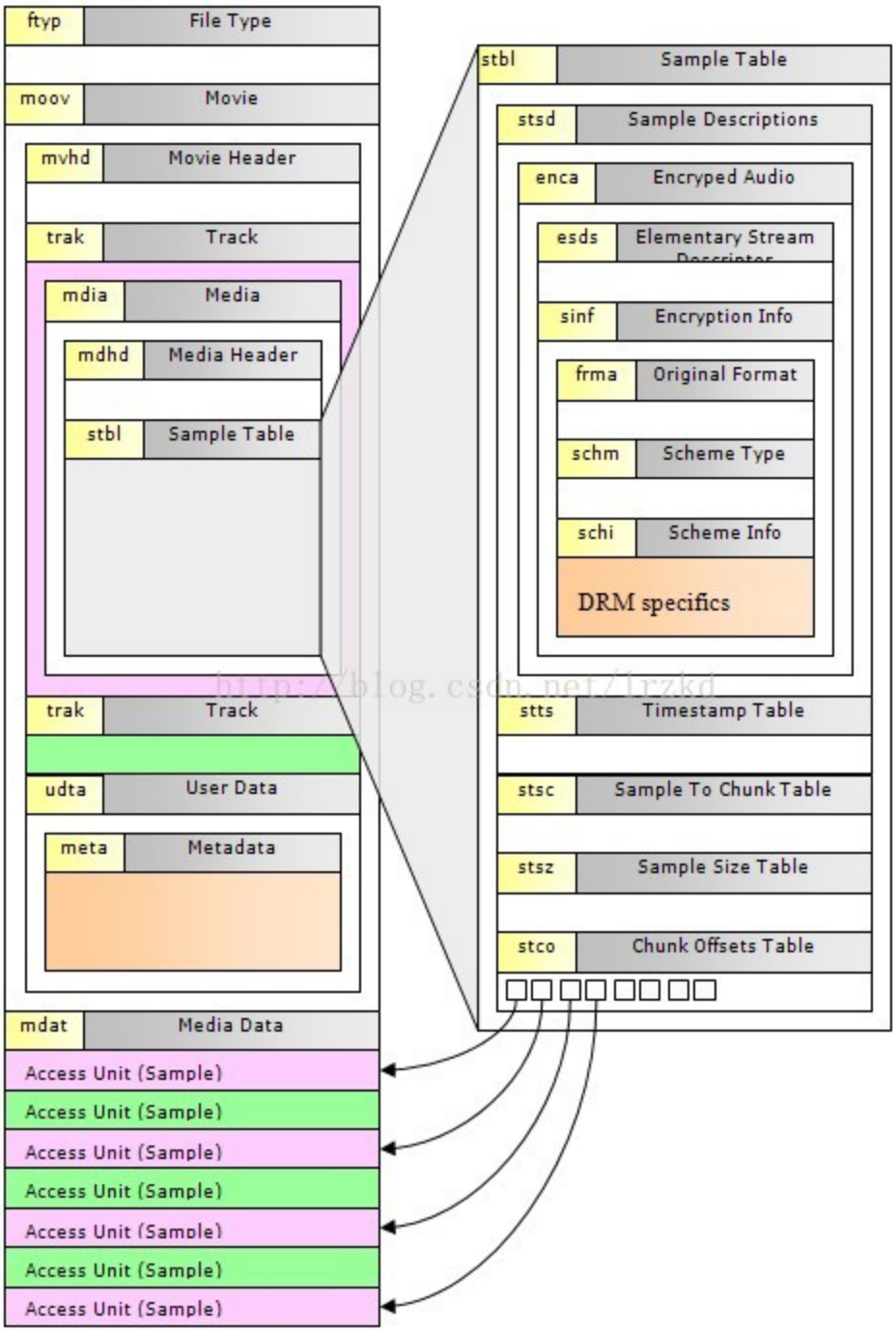
```
1
2  static int seek_frame_byte(AVFormatContext *s, int stream_index,
3                             int64_t pos, int flags)
4  {
5      int64_t pos_min, pos_max;
6
7      pos_min = s->internal->data_offset;
8      pos_max = avio_size(s->pb) - 1;
9
10     if (pos < pos_min)
11         pos = pos_min;
12     else if (pos > pos_max)
13         pos = pos_max;
14
15     avio_seek(s->pb, pos, SEEK_SET);
16
17     s->io_repositioned = 1;
18
19     return 0;
20 }
```

所以 ts seek 逻辑是：给定一个文件位置，直接将文件指针指向该位置。接下来调用 read_packet() 读取一个 ts 包(188字节)时，由于之前进行了seek操作，文件指针很可能没有指到一个 ts packet 的包头位置（包头以0x47 byte打头的），这时候需要调用 mpegts_resync() 进行重新同步找到包头，然后再重新读取一个完整 ts packet。

mov_read_seek

mp4 的 seek 操作逻辑是：给定一个 seek的目标时间戳(timestamp)，根据mp4 里每个包的索引信息，找到时间戳对应的包就可以了。根据下面的 mp4 的文件组织结构，利用Sample Table，可以快速找到任意给定时间戳的 video audio 数据包。

mp4 的文件组织结构如下图



总结

1、对 mp4 文件来说，由于有索引表，可以快速找到某个时间戳所对应的数据，所以 seek 操作可以快速完成。
2、ts 文件没有时间戳和数据包位置的对应关系，所以对播放器来说，给定seek 的时间戳ts_seek，首先应该根据文件的码率估算一个位置pos，然后获取该位置的数据包的时间戳ts_actual，如果ts_actual < ts_seek ，则需要继续往后读取数据包；如果ts_actual> ts_seek，则需要往前读取数据包，直到读到 ts_seek 对应的数据包。所以 ts 文件的操作可能更加耗时； 如果 ts 包含的是 CBR 码流，则 ts_actual 与 ts_seek 一般差别不大， seek 相对较快； 如果 ts 包含的 VBR 码流， 则 ts_actual 与 ts_seek 可能相差甚远， 则 seek 相对较慢。

退出播放

•关闭流
•销毁队列资源
•销毁线程
•关注线程是怎么退出的
•退出：do_exit()