Technology Tarena 达内科技

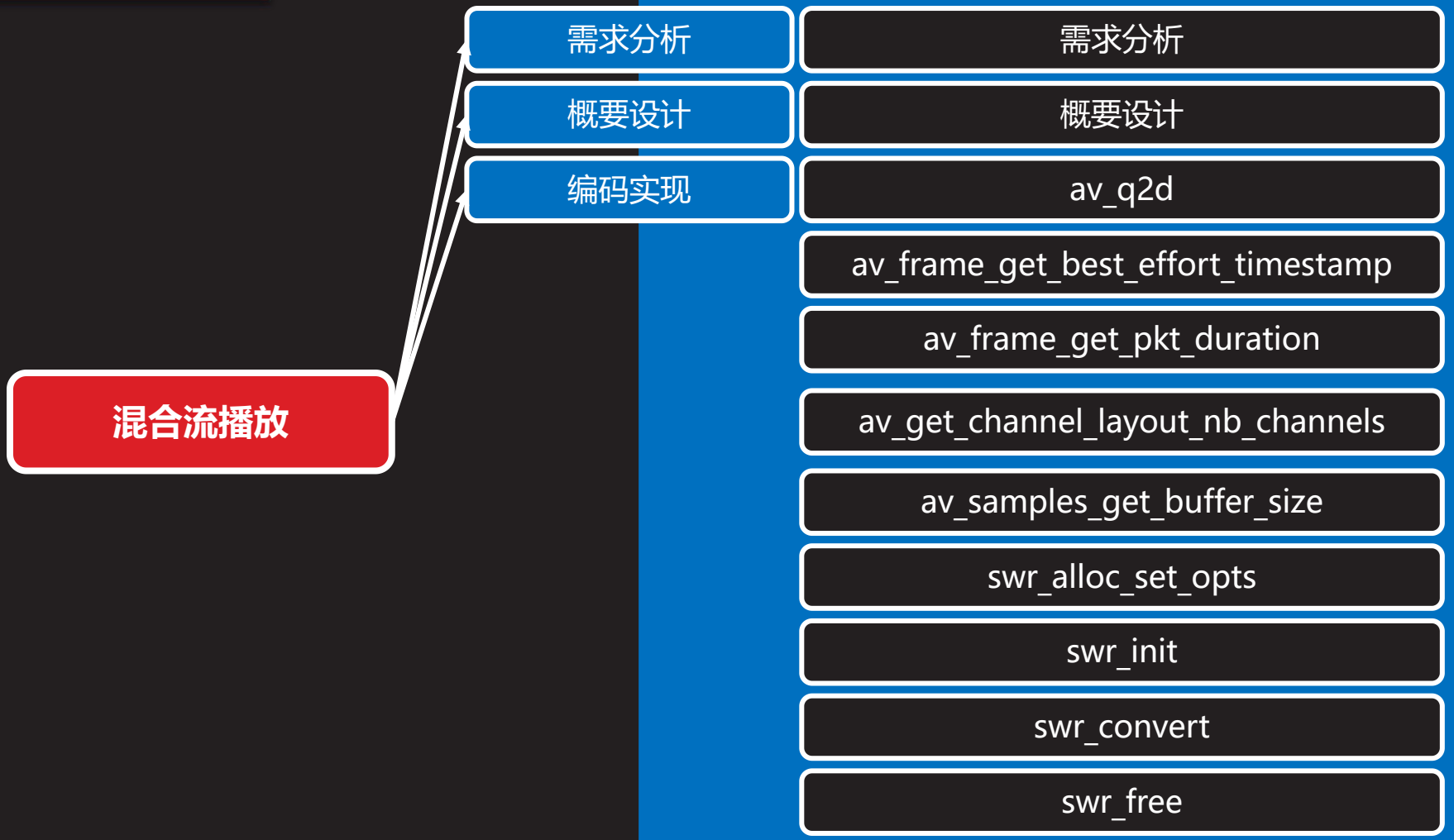
# 流媒体高级编程

STREAMING MEDIA DAY03

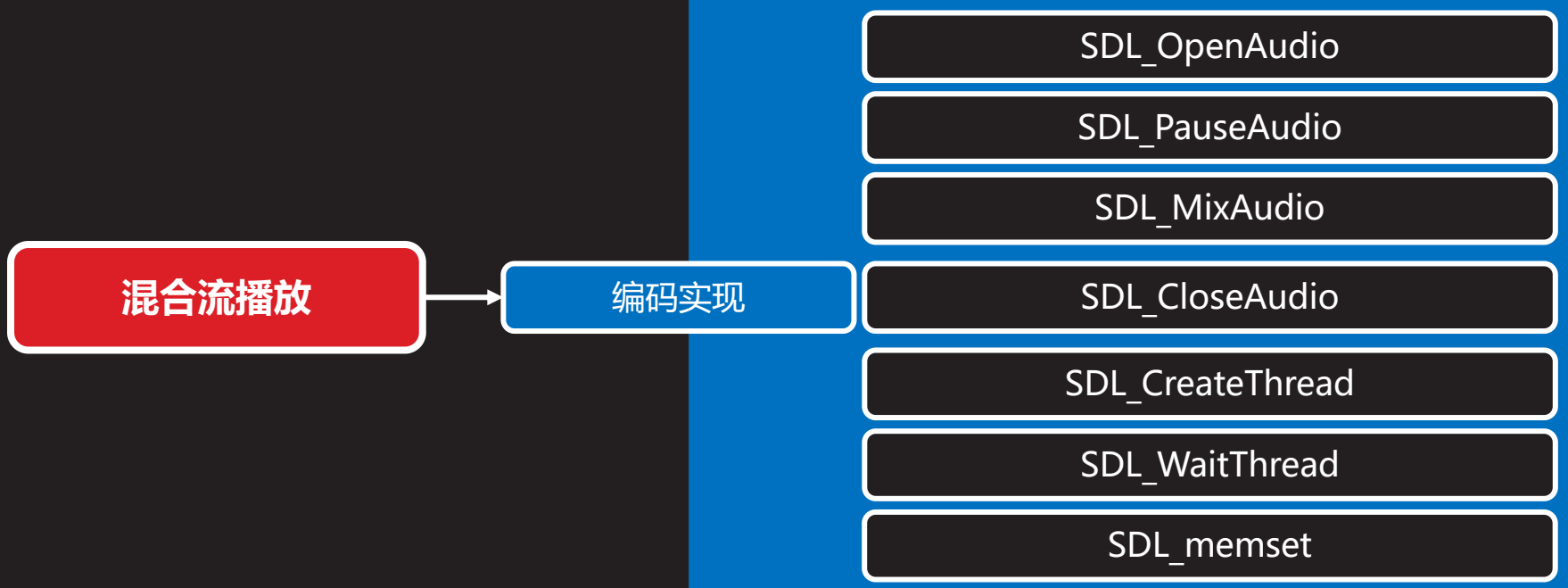
# 内容

| | | |
|---|---|---|
| 上午 | 09:00 ~ 09:30 | 作业讲解和回顾 |
| | 09:30 ~ 10:20 | 混合流播放 |
| | 10:30 ~ 11:20 | |
| | 11:30 ~ 12:20 | |
| 下午 | 14:00 ~ 14:50 | 混合流广播 |
| | 15:00 ~ 15:50 | |
| | 16:00 ~ 16:50 | |
| | 17:00 ~ 17:30 | 总结和答疑 |

# 混合流播放


混合流播放
需求分析
概要设计
编码实现
需求分析
概要设计
av_q2d
av_frame_get_best_effort_timestamp
av_frame_get_pkt_duration
av_get_channel_layout_nb_channels
av_samples_get_buffer_size
swr_alloc_set_opts
swr_init
swr_convert
swr_free

# 混合流播放


混合流播放
编码实现
SDL_OpenAudio
SDL_PauseAudio
SDL_MixAudio
SDL_CloseAudio
SDL_CreateThread
SDL_WaitThread
SDL_memset

Technology Tarena 达内科技

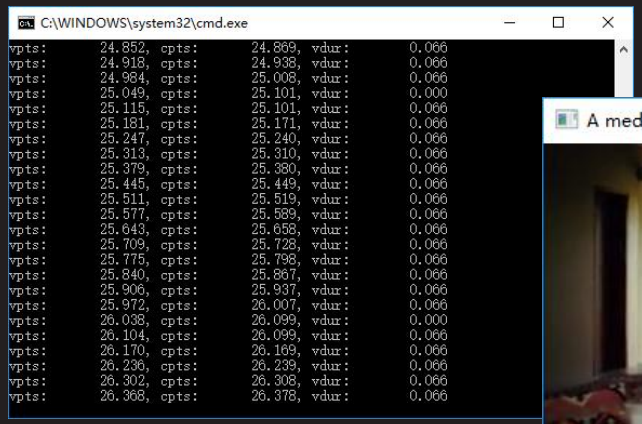
# 需求分析

# 需求分析

Tarena Technology 达内科技

知识讲解

- 播放音视频混合流
  - 分别在独立的线程中，以并发的方式，播放音频和视频流，避免因解码过程的相互等待，而导致画面卡顿和声音断续
  - 将读取数据包从播放过程中分离出来，构成一个独立的线程，避免在接收直播媒体流的过程中，音视频间相互干扰
  - 以音频播放时间(PTS)为基准，动态调节每帧视频画面的存留延时，尽可能减小音视频间的时间偏差，做到音画同步



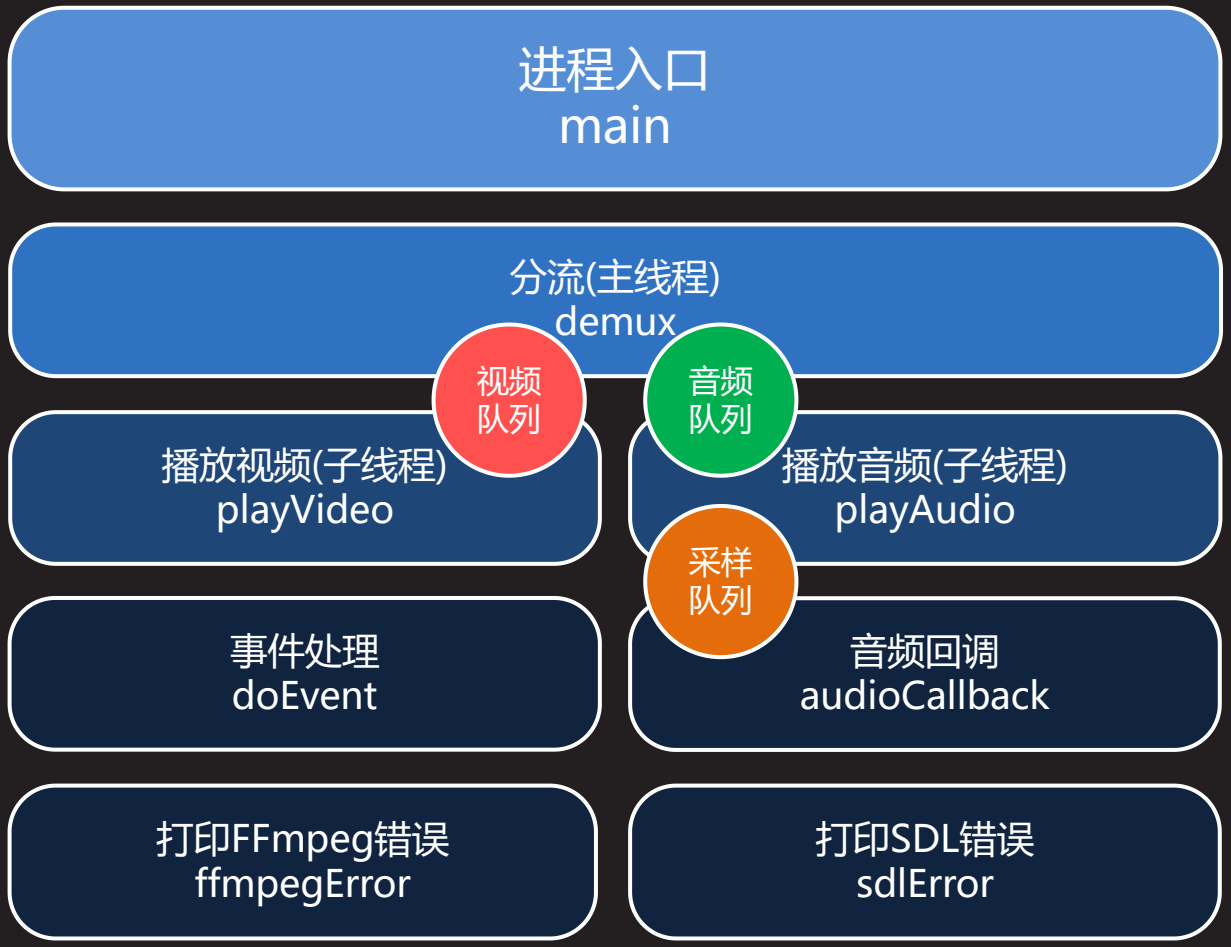

Technology Tarena 达内科技

# 概要设计

# 概要设计



知识讲解



Technology Tarena 达内科技

# 编码实现

# av_q2d



- 将AVRational类型的分数转换为double类型的小数
  - `#include <libavutil/rational.h>`
    `double av_q2d (`
    `  AVRational a); // 分数`
  - 返回小数
  - `struct AVRational {`
    `  int num; // 分子`
    `  int den; // 分母`
    `};`

知识讲解

# av_frame_get_best_effort_timestamp



- 获取音视频帧的播放时间戳(PTS)
  - `#include <libavutil/frame.h>`

    ```
    int64_t av_frame_get_best_effort_timestamp (
      const AVFrame* frame); // 音视频帧
    ```
  - 返回播放时间戳
  - 时间戳的单位取自流对象的time_base成员

    ```
    struct AVFormatContext {
      ... AVStream** streams; ... };
    struct AVStream {
      ... AVRational time_base; ... };
    pts = av_frame_get_best_effort_timestamp (
      frame) * av_q2d (
      fmtCtx->streams[index]->time_base); // 秒
    ```

知识讲解

# av_frame_get_pkt_duration



- 获取音视频帧的播放时长
  - `#include <libavutil/frame.h>`

```
int64_t av_frame_get_pkt_duration (
  const AVFrame* frame); // 音视频帧
```

  - 返回播放时长
  - 时长的单位取自流对象的time_base成员

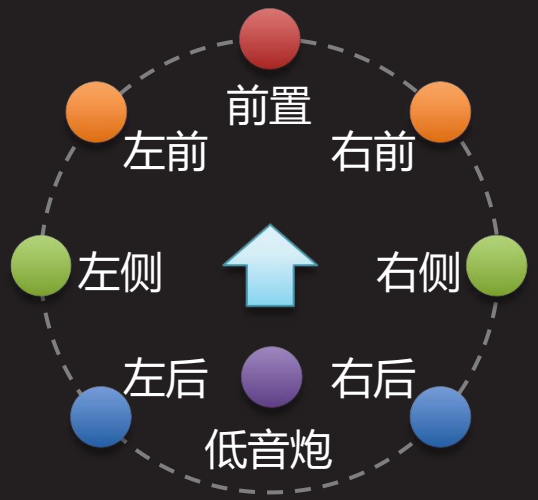
```
struct AVFormatContext {
  ... AVStream** streams; ... };
struct AVStream {
  ... AVRational time_base; ... };
duration = av_frame_get_pkt_duration (
  frame) * av_q2d (
  fmtCtx->streams[index]->time_base); // 秒
```

知识讲解

# av_get_channel_layout_nb_channels



- 根据声道布局计算声道数
  - `#include <libavutil/channel_layout.h>`
    `int av_get_channel_layout_nb_channels (`
    `  uint64_t channel_layout); // 声道布局`
  - 返回声道数
  - channel_layout参数可取如下值:
    - ✓ `AV_CH_LAYOUT_MONO` - 单声道
    - ✓ `AV_CH_LAYOUT_STEREO` - 立体声
    - ✓ `AV_CH_LAYOUT_SURROUND` - 环绕
    - ✓ `AV_CH_LAYOUT_5POINT0` - 5.0环绕
    - ✓ `AV_CH_LAYOUT_5POINT1` - 5.1环绕
    - ✓ `AV_CH_LAYOUT_7POINT1` - 7.1环绕
    - ✓ ...




知识讲解

# av_samples_get_buffer_size



- 根据声道数、采样数和采样格式计算采样缓冲区的字节数
  - 
```
#include <libavutil/samplefmt.h>
int av_samples_get_buffer_size (
    int*                 linesize,    // 输出行长
    int                  nb_channels, // 声道数
    int                  nb_samples,  // 采样数
    enum AVSampleFormat  sample_fmt,  // 采样格式
    int                  align);      // 对齐方式
```
  - 成功返回采样缓冲区字节数，失败返回错误码(<0)
  - sample_fmt参数可取如下值：
    - ✓ AV_SAMPLE_FMT_U8 - 无符号8位整数/采样
    - ✓ AV_SAMPLE_FMT_S16 - 有符号16位整数/采样
    - ✓ AV_SAMPLE_FMT_S16P - 有符号16位整数双位面/采样
    - ✓ ...

# swr_alloc_set_opts



- 创建采样器上下文对象
  - 
```
#include <libswresample/swresample.h>
struct SwrContext* swr_alloc_set_opts (
  struct SwrContext*  s,                // 采样器上下文，可置NULL
  int64_t             out_ch_layout,    // 目标声道布局
  enum AVSampleFormat out_sample_fmt,   // 目标采样格式
  int                 out_sample_rate,  // 目标采样频率
  int64_t             in_ch_layout,     // 源声道布局
  enum AVSampleFormat in_sample_fmt,    // 源采样格式
  int                 in_sample_rate,   // 源采样频率
  int                 log_offset,       // 日志等级偏移，可取0
  void*               log_ctx);         // 日志上下文，可置NULL
```
  - 成功返回指向所创建采样器上下文对象的指针，失败返回NULL
  - 所创建采样器上下文对象，在使用完以后，必须通过swr_free函数销毁
- SwrContext类型的采样器上下文对象，可以理解为是对音频采样器的抽象，可用于对解码后的音频帧做重采样操作

# swr_init

- 初始化采样器上下文对象
  - `#include <libswresample/swresample.h>`
    `int swr_init (`
    `  struct SwrContext* s); // 采样器上下文`
  - 成功返回0，失败返回错误码(<0)

知识讲解

# swr_convert



- 重采样
  - #include <libswresample/swresample.h>

```
int swr_convert (
    struct SwrContext* s,          // 采样器上下文
    uint8_t**          out,        // 采样缓冲区
    int                out_count,  // 采样缓冲区大小
    const uint8_t**    in,         // 音频帧数据
    int                in_count);  // 音频帧采样数
```

  - 成功返回采样缓冲区中的采样数，失败返回错误码(<0)
  - 所返回的采样数并非放到采样缓冲区中的字节数，还需要通过av_samples_get_buffer_size函数，结合声道数和采样格式，计算出采样缓冲区中的字节数



# swr_free



- 销毁采样器上下文对象，同时将指向该对象的指针置空
  - `#include <libswresample/swresample.h>`
    `void swr_free (`
    `  struct SwrContext** s); // 采样器上下文`
  - 该函数的参数为指向某个已被创建的采样器上下文对象的指针的地址，待函数返回后，该对象的指针即变为空指针

知识讲解

# SDL_OpenAudio



- 用期望的参数打开音频设备，并返回实际的参数
  - `#include <SDL_audio.h>`
    `int SDL_OpenAudio (`
    `SDL_AudioSpec* desired, // 期望参数`
    `SDL_AudioSpec* obtained); // 实际参数，可置NULL`
  - 成功返回0，失败返回-1
  - `struct SDL_AudioSpec {`
    `SDL_AudioCallback callback; // 音频回调`
    `void* userdata; // 回调参数`
    `... };`
  - `typedef void (*SDL_AudioCallback) (`
    `void* userdata, // 回调参数`
    `Uint8* stream, // 音频缓冲区`
    `int len); // 音频缓冲区字节数`

知识讲解

# SDL_PauseAudio



- 启动/暂停/继续音频播放
  - `#include <SDL_audio.h>`
    `void SDL_PauseAudio (`
    `int pause_on); // 1 - 暂停，0 - 启动/继续`
  - 暂停播放期间，音频回调不会被执行
  - 通过SDL_OpenAudio函数打开的音频设备，默认为暂停状态，需要用0调用此函数，以启动音频播放

知识讲解

# SDL_MixAudio



- 混音
  - `#include <SDL_audio.h>`

```
void SDL_MixAudio (
    Uint8*       dst,     // 音频缓冲区
    const Uint8* src,     // 采样缓冲区
    Uint32       len,     // 音频缓冲区字节数
    int          volume); // 音量(0-128)
```

  - 将采样缓冲区中的采样数据，按照指定的音量，混音到音频缓冲区中，后者可被提交给音频设备驱动，以播放声音
  - volume参数所表示的音量与硬件音量无关

知识讲解

# SDL_CloseAudio

- 终止音频处理，关闭音频设备
  - `#include <SDL_audio.h>`
    `void SDL_CloseAudio (void);`

知识讲解

# SDL_CreateThread



- 创建线程
  - 
```
#include <SDL_thread.h>
SDL_Thread* SDL_CreateThread (
    SDL_ThreadFunction fn,    // 线程过程
    const char*        name,  // 线程名称
    void*              data); // 线程参数
```
  - 成功返回线程对象指针，失败返回NULL
  - 线程过程的函数原型：

```
typedef int (*SDL_ThreadFunction) (
    void* data); // 线程参数
```

线程过程的返回值可通过SDL_WaitThread函数获得

知识讲解

# SDL_WaitThread



- 等待线程结束，回收线程资源，获取线程过程的返回值
  - 
```
#include <SDL_thread.h>
void SDL_WaitThread (
    SDL_Thread* thread,   // 线程对象
    int*        status); // 线程状态
```
  - status参数所指向的变量被该函数填入线程过程的返回值
  - 任何非分离线程(未调用SDL_DetachThread函数)，如不做回收，其资源将被系统保留，形成所谓的“线程僵尸”
  - 对分离线程(调用过SDL_DetachThread函数) 的回收操作将导致未定义的后果

知识讲解

# SDL_memset



- 设置内存块
  - 
```
#include <SDL_stdinc.h>
void* SDL_memset (
    void*  dst,  // 被设置首地址
    int    c,    // 被设置字节值
    size_t len); // 被设置字节数
```
  - 返回被设置内存块的首地址，即第一个参数dst
  - 将内存中从dst开始的len个字节的值，全部设置为c

知识讲解

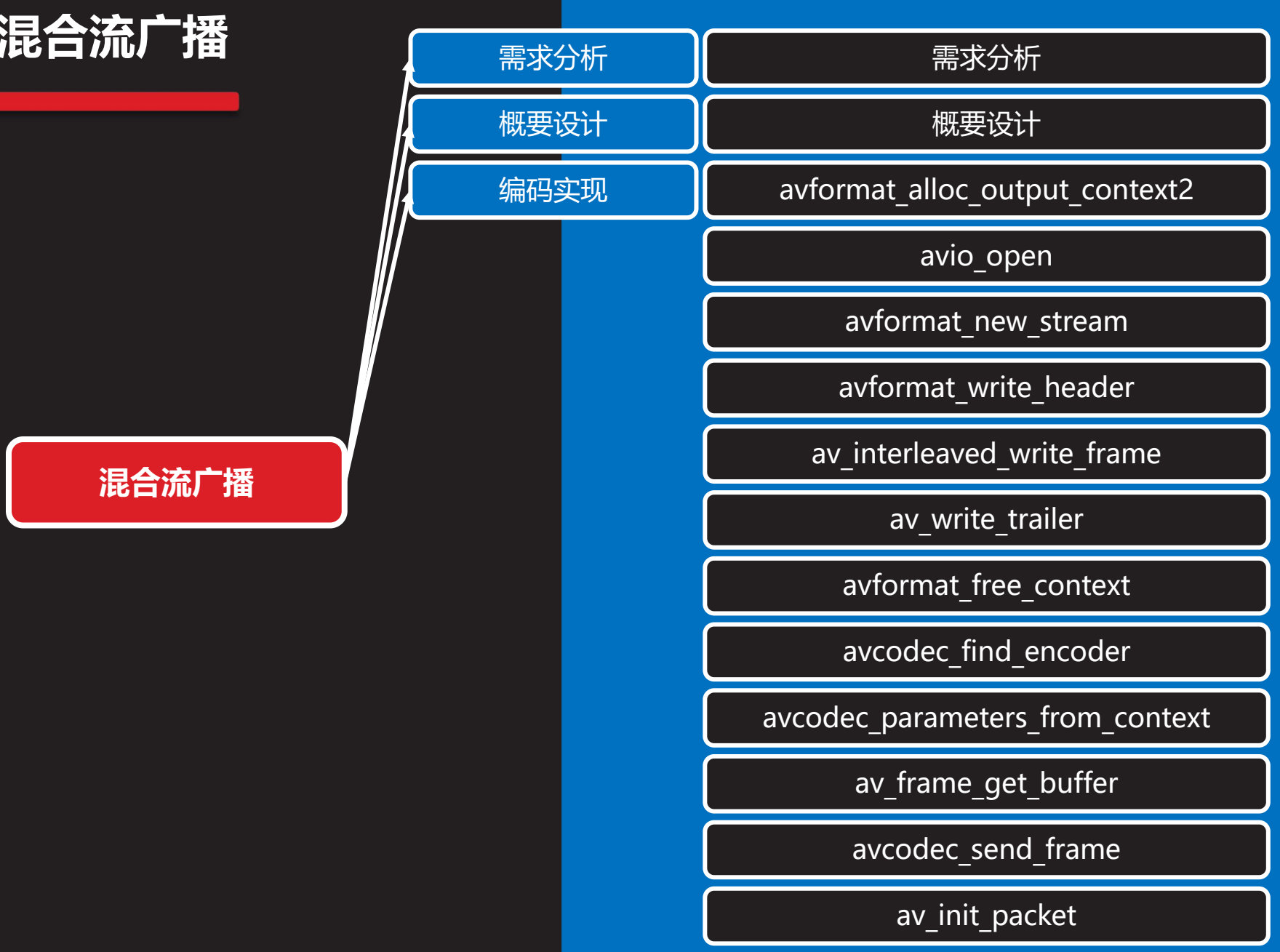
# SplitterPlayer

【参见：FFmpeg/Primer/SplitterPlayer】

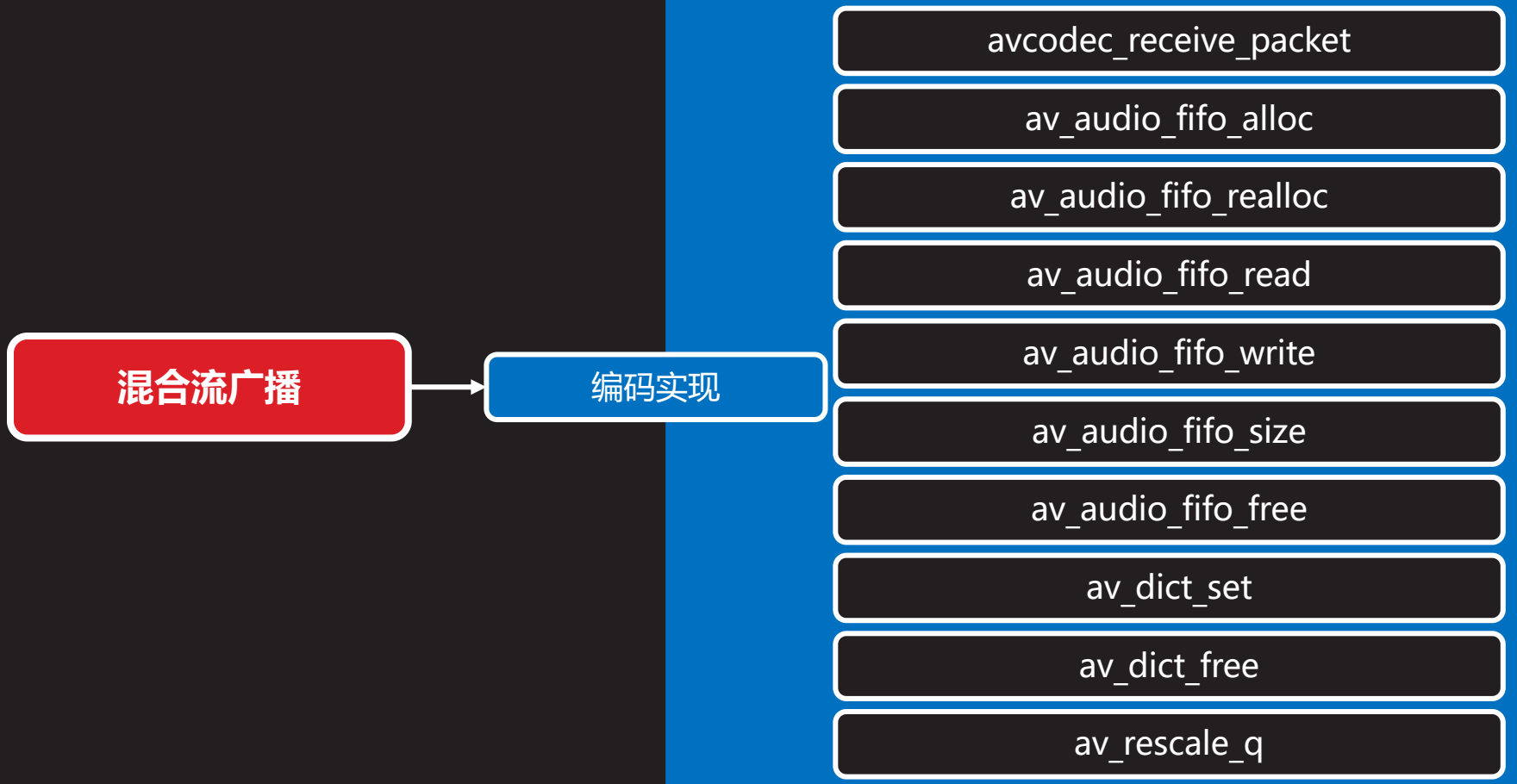
课堂练习

- 播放音视频混合流
  - 分别在独立的线程中，以并发的方式，播放音频和视频流，避免因解码过程的相互等待，而导致画面卡顿和声音断续
  - 将读取数据包从播放过程中分离出来，构成一个独立的线程，避免在接收直播媒体流的过程中，音视频间相互干扰
  - 以音频播放时间(PTS)为基准，动态调节每帧视频画面的存留延时，尽可能减小音视频间的时间偏差，做到音画同步

# 混合流广播


混合流广播
需求分析
概要设计
编码实现
需求分析
概要设计
avformat_alloc_output_context2
avio_open
avformat_new_stream
avformat_write_header
av_interleaved_write_frame
av_write_trailer
avformat_free_context
avcodec_find_encoder
avcodec_parameters_from_context
av_frame_get_buffer
avcodec_send_frame
av_init_packet

# 混合流广播


混合流广播
编码实现
avcodec_receive_packet
av_audio_fifo_alloc
av_audio_fifo_realloc
av_audio_fifo_read
av_audio_fifo_write
av_audio_fifo_size
av_audio_fifo_free
av_dict_set
av_dict_free
av_rescale_q

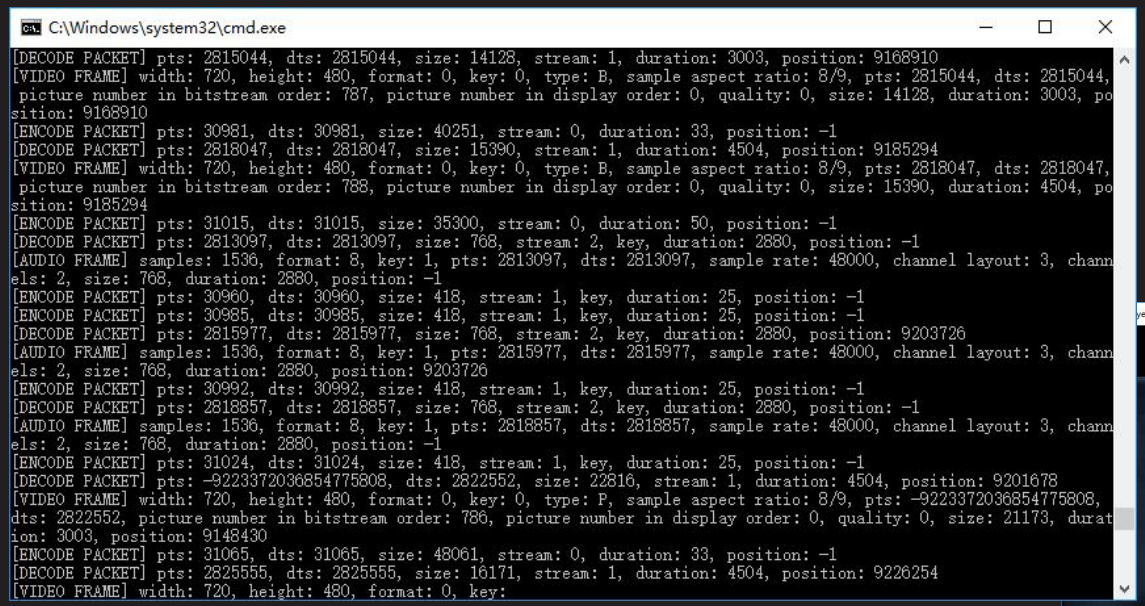
Technology Tarena 达内科技

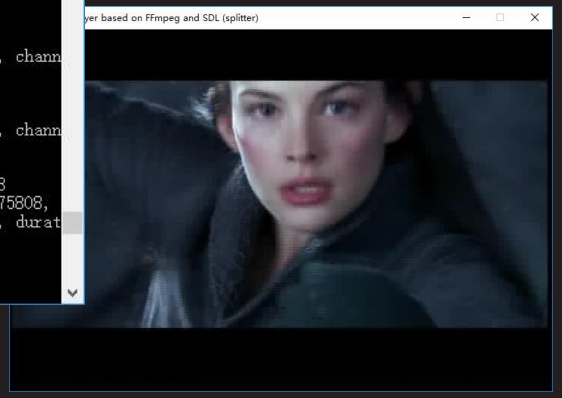
# 需求分析

# 需求分析

- 以同步方式广播音视频混合流
  - 本地转码：SyncCaster mayitbe.vob mayitbe.flv
  - 网络广播：SyncCaster mayitbe.vob rtmp://192.168.1.166/live/1

知识讲解



Technology Tarena 达内科技

# 概要设计

# 概要设计



知识讲解

进程入口
main

广播
cast

广播视频和音频
castAV

打印源格式信息
printSrcFmt

打印解码包信息
printDecPkt

打印视频帧信息
printVidFrm

打印目标格式信息
printDstFmt

打印编码包信息
printEncPkt

打印音频帧信息
printAudFrm

打印FFmpeg错误
ffmpegError

打印SDL错误
sdlError

事件处理
doEvent

Technology Tarena 达内科技

# 编码实现

# avformat_alloc_output_context2



- 创建目标格式上下文(AVFormatContext)对象
  - 
```
#include <libavformat/avformat.h>
int avformat_alloc_output_context2 (
    AVFormatContext** ctx,           // 目标格式上下文
    AVOutputFormat*   oformat,       // 输出格式
    const char*       format_name, // 输出格式名
    const char*       filename);   // 输出文件名
```
  - 成功返回0，失败返回错误码(<0)
  - 所创建的目标格式上下文对象，在使用完以后，必须通过avformat_free_context函数销毁
- AVFormatContext类型的目标格式上下文对象，可以理解为是对流媒体目标的抽象。它既可以表示一个本地输出文件，也可以表示一个远程拉流应用

知识讲解

# avio_open

- 打开统一资源定位符(URL)
  - `#include <libavformat/avio.h>`

```
int avio_open (
    AVIOContext** s,      // 输入输出上下文
    const char*   url,    // 统一资源定位符
    int           flags); // 输入输出标志位
```

  - 成功返回0，失败返回错误码(<0)
  - flags参数可取如下值：
    - ✓ `AVIO_FLAG_READ` - 只读
    - ✓ `AVIO_FLAG_WRITE` - 只写
    - ✓ `AVIO_FLAG_READ_WRITE` - 读写
  - 所打开的统一资源定位符(URL)，通常以输入输出上下文(AVIOContext)对象的形式，保存在格式上下文(AVFormatContext)对象的pb成员中，以备后续读写操作

知识讲解

# avformat_new_stream



- 在格式中新增一个流
  - `#include <libavformat/avformat.h>`

```
AVStream* avformat_new_stream (
    AVFormatContext* s,  // 包含该流的格式上下文
    const AVCodec*   c); // 用于该流的编解码器
```

  - 成功返回指向新增流对象的指针，失败返回NULL
  - 解包过程中，该函数被解包器在read_header函数中调用；打包过程中，该函数被用户在调用avformat_write_header函数之前调用

知识讲解

# avformat_write_header



- 为目标格式中的每个流分配的私有数据并写入头部信息
  - 
```
#include <libavformat/avformat.h>
int avformat_write_header (
    AVFormatContext* s,          // 目标格式上下文
    AVDictionary**   options); // 特殊选项
```
  - 成功返回AVSTREAM_INIT_IN_WRITE_HEADER或AVSTREAM_INIT_IN_INIT_OUTPUT，失败返回错误码(<0)

知识讲解

# av_interleaved_write_frame



- 将编码包以正确的交错方式写入目标格式的相应流中
  - 
```
#include <libavformat/avformat.h>
int av_interleaved_write_frame (
    AVFormatContext* s,     // 目标格式上下文
    AVPacket*        pkt); // 编码包
```
  - 成功返回0，失败返回错误码(<0)
  - 所写编码包的播放时间戳(AVPacket.pts)、解码时间戳(AVPacket.dts)和播放时长(AVPacket.duration)必须被设置为正确的值，且解码时间戳必须严格递增

知识讲解

# av_write_trailer



- 为目标格式中的每个流写入尾部信息并释放其私有数据
  - `#include <libavformat/avformat.h>`
    `int av_write_trailer (`
    `AVFormatContext* s); //` 目标格式上下文
  - 成功返回0，失败返回错误码(<0)
  - 只有在avformat_write_header函数返回成功以后，才能调用此函数

知识讲解

# avformat_free_context



- 销毁目标格式上下文对象及其所有流对象
  - `#include <libavformat/avformat.h>`

    `void avformat_free_context (`

    `AVFormatContext* s); //` 目标格式上下文
  - 该函数的参数并非目标格式上下文对象指针的地址，函数返回后，该对象的指针也不会自动变为空指针

知识讲解

# avcodec_find_encoder



- 根据编码器ID查找编码器对象
  - `#include <libavcodec/avcodec.h>`

    `AVCodec* avcodec_find_encoder (`

    `enum AVCodecID id); // 编码器ID`
  - 成功返回与参数编码器ID相对应的编码器对象指针，失败返回NULL
  - `enum AVCodecID {`

    `...`

    `AV_CODEC_ID_PNG, // PNG图像`

    `...`

    `AV_CODEC_ID_MPEG2VIDEO, // DVD视频`

    `...`

    `AV_CODEC_ID_MP3, // MP3音频`

    `...`

    `};`

知识讲解

# avcodec_parameters_from_context



- 根据编解码器上下文设置编解码器参数
  - ```
    #include <libavcodec/avcodec.h>
    int avcodec_parameters_from_context (
      AVCodecParameters*    par,     // 编解码器参数
      const AVCodecContext* codec); // 编解码器上下文
    ```
  - 成功返回0，失败返回错误码(<0)
  - 根据编解码器上下文设置的编解码器参数，通常来自于格式上下文中的流对象
    struct AVFormatContext { ... AVStream** streams; ... };
    struct AVStream { ... AVCodecParameters* codecpar; ... };

知识讲解
    ```

# av_frame_get_buffer



- 为音视频帧分配数据缓冲区
  - `#include <libavutil/frame.h>`

```
int av_frame_get_buffer (
    AVFrame* frame,   // 音视频帧
    int      align); // 对齐方式
```

  - 成功返回0，失败返回错误码(<0)
  - 在调用此函数前，音视频帧中的如下成员必须先设置好：
    - ✓ `AVFrame.channel_layout` - 声道布局
    - ✓ `AVFrame.nb_samples` - 采样数
    - ✓ `AVFrame.format` - 采样格式/像素格式
    - ✓ `AVFrame.width` - 图像宽度
    - ✓ `AVFrame.height` - 图像高度
  - 对已分配缓冲区的音视频帧，调用此函数将导致内存泄漏

知识讲解

# avcodec_send_frame



- 向编码器发送待编码数据帧
  - `#include <libavcodec/avcodec.h>`

    ```
    int avcodec_send_frame (
        AVCodecContext* avctx,   // 编码器上下文
        const AVFrame*  frame); // 待编码数据帧
    ```
  - 成功返回0，失败返回错误码(<0)，其中AVERROR(EAGAIN)表示暂时不能发送，待接收后重发，AVERROR_EOF表示编码器已被刷流，不能再发送数据帧
  - 以NULL指针作为第二个参数，向编码器发送空数据帧，表示刷流操作的开始，编码器将在后续avcodec_receive_packet调用中给出缓冲区中残存的尾包

知识讲解

# av_init_packet



- 初始化数据包为默认状态
  - `#include <libavcodec/avcodec.h>`
    `void av_init_packet (`
    `AVPacket* pkt); // 数据包`
  - 该函数并不涉及AVPacket的data和size成员，它们必须被单独初始化

知识讲解

# avcodec_receive_packet



- 从编码器接收已编码数据包
  - `#include <libavcodec/avcodec.h>`

```
int avcodec_receive_packet (
    AVCodecContext* avctx,  // 编码器上下文
    AVPacket*       avpkt); // 已编码数据包
```

  - 成功返回0，失败返回错误码(<0)，其中AVERROR(EAGAIN)表示暂时无包可取，需要新的输入，AVERROR_EOF表示编码器已被刷流，再也收不到数据包
  - 数据包中的数据缓冲区带有引用计数，该函数在做任何操作之前，会先通过av_packet_unref函数，解除数据包对其数据缓冲区的引用，并在操作完成后重建新的引用

知识讲解

# av_audio_fifo_alloc



- 创建采样队列(AVAudioFifo)对象
  - `#include <libavutil/audio_fifo.h>`

```
AVAudioFifo* av_audio_fifo_alloc (
    enum AVSampleFormat sample_fmt,  // 采样格式
    int                 channels,    // 声道数
    int                 nb_samples); // 采样数
```

  - 成功返回采样队列对象指针，失败返回NULL
  - 所创建的采样队列对象，在使用完以后，必须通过av_audio_fifo_free函数销毁
- 使用采样队列的意义在于，音频编码器每次所能接收的采样数，与音频解码器每次所能提供的采样数，并不一致。利用采样队列作为音频编码和解码过程间的缓冲区



# av_audio_fifo_realloc

- 根据采样数调整采样队列的大小
  - `#include <libavutil/audio_fifo.h>`

```
int av_audio_fifo_realloc (
    AVAudioFifo* af,              // 采样队列
    int          nb_samples); // 采样数
```

  - 成功返回0，失败返回错误码(<0)

知识讲解

# av_audio_fifo_read



- 读取采样队列
  - `#include <libavutil/audio_fifo.h>`

```
int av_audio_fifo_read (
    AVAudioFifo* af,           // 采样队列
    void**       data,         // 音频位面指针数组
    int          nb_samples); // 期望读取的采样数
```

  - 成功返回实际读到的采样数，失败返回错误码(<0)
  - 实际读到的采样数可能比期望值(nb_samples参数)少，除非采样队列中有足够多的采样，但绝不可能超过期望值



# av_audio_fifo_write



- 写入采样队列
  - `#include <libavutil/audio_fifo.h>`

```
int av_audio_fifo_write (
    AVAudioFifo* af,           // 采样队列
    void**       data,         // 音频位面指针数组
    int          nb_samples); // 期望写入的采样数
```

  - 成功返回实际写入的采样数，失败返回错误码(<0)
  - 实际写入的采样数一定与期望值(nb_samples参数)严格相等，除非写入过程中发生了错误，导致函数提前返回



# av_audio_fifo_size

- 获取采样队列中当前可读采样数
  - `#include <libavutil/audio_fifo.h>`

    `int av_audio_fifo_size (`

    `AVAudioFifo* af); //` 采样队列
  - 返回采样队列当前可读采样数

知识讲解

# av_audio_fifo_free

- 销毁采样队列对象
  - `#include <libavutil/audio_fifo.h>`

```
void av_audio_fifo_free (
  AVAudioFifo* af); // 采样队列
```

知识讲解

# av_dict_set

- 设置字典条目
  - `#include <libavutil/dict.h>`

```
int av_dict_set (
    AVDictionary** pm,      // 字典，取NULL内部创建
    const char*    key,     // 键
    const char*    value,   // 值，取NULL删除条目
    int            flags); // 标志位，缺省0
```

  - 成功返回0，失败返回错误码(<0)
  - flags参数可取如下值：
    - ✓ `AV_DICT_DONT_OVERWRITE` - 若键已存在，则返回失败
    - ✓ `AV_DICT_APPEND` - 若键已存在，则追加其值
    - ✓ `AV_DICT_MULTIKEY` - 允许不同条目拥有相同键
    - ✓ …

知识讲解

# av_dict_free

- 销毁字典对象，同时将指向该对象的指针置空
  - `#include <libavutil/dict.h>`

    `void av_dict_free (`

    `AVDictionary** m); // 字典`
  - 该函数的参数为指向某个已被创建的字典对象的指针的地址，待函数返回后，该对象的指针即变为空指针

知识讲解

# av_rescale_q



- 修改单位
  - `#include <libavutil/mathematics.h>`

```
int64_t av_rescale_q (
    int64_t    a,   // 原数值
    AVRational bq,  // 原单位
    AVRational cq); // 新单位
```

  - 返回新数值 = 原数值 x 原单位 / 新单位，按四舍五入取整

知识讲解

Technology Tarena 达内科技

# SyncCaster

【参见：FFmpeg/Primer/SyncCaster】

课堂练习

- 以同步方式广播音视频混合流
  - 本地转码：SyncCaster mayitbe.vob mayitbe.flv
  - 网络广播：SyncCaster mayitbe.vob rtmp://192.168.1.166/live/1

Technology Tarena 达内科技

# 总结和答疑