第 5 章

CHAPTER 5

AVUtil 通用工具层理论及案例实战

libavutil 是一个实用库，用于辅助多媒体编程。此库包含安全的可移植字符串函数、随机数生成器、数据结构、附加数学函数、加密和多媒体相关功能（如像素和样本格式的枚举）等。常用的 libavcodec 和 libavformat 两个库并不依赖此库。本章重点介绍与该库相关的数据结构、枚举、API 及几个应用案例。

2min

5.1 AVUtil 库及相关 API 简介

libavutil 库包含 200 多个文件，如图 5-1 所示，主要包括以下几种功能：

图 5-1　libavutil 库的目录结构

（1）数学函数。

（2）字符串操作。

（3）内存管理相关。

(4) 数据结构相关。

(5) 错误码及错误处理。

(6) 日志输出。

(7) 其他辅助信息,例如密钥、哈希值、宏、库版本、常量等。

1. 数学函数

这部分主要提供与基本数学概念相关的功能,例如有理数的定义代码如下:

```
//chapter5/5.1.help.txt
struct AVRational{
    int num; //< Numerator 分子
    int den; //< Denominator 分母
};
```

近似取值的枚举值(enum AVRounding),如表 5-1 所示。

表 5-1 AVRational 的近似值

枚 举 名	值	描 述
AV_ROUND_ZERO	0	向零取整
AV_ROUND_INF	1	向非零取整
AV_ROUND_DOWN	2	向负无穷取整
AV_ROUND_UP	3	向正无穷取整
AV_ROUND_NEAR_INF	5	向无穷取整,包含中点位置
AV_ROUND_PASS_MINMAX	8192	透传 INT64_MIN/MAX,避免出现 AV_NOPTS_VALUE

其他的数学函数(包括但不限于),代码如下(详见注释信息):

```
//chapter5/5.1.help.txt
//获得 a 和 b 的最大公约数
int64_t av_gcd (int64_t a, int64_t b);

//a * bq / cq, 有理数,按照 rnd 标志取整
int64_t av_rescale_q (int64_t a, AVRational bq, AVRational cq);
int64_t av_rescale_q_rnd (int64_t a, AVRational bq, AVRational cq, enum AVRounding rnd);

//a * b / c,按照 rnd 标志取整
int64_t av_rescale (int64_t a, int64_t b, int64_t c);
int64_t av_rescale_rnd (int64_t a, int64_t b, int64_t c, enum AVRounding rnd);

//比较余数大小,即 a % mod 与 b % mod,mod 必须是 2 的倍数
int64_t av_compare_mod (uint64_t a, uint64_t b, uint64_t mod)

//比较 ts_a/tb_a 与 ts_b/tb_b 的大小. -1:前者小;0:相等;1:后者小
int av_compare_ts (int64_t ts_a, AVRational tb_a, int64_t ts_b, AVRational tb_b);
```

```
//创建 AVRational
static AVRational av_make_q (int num, int den);

//将 AVRational 转换为 double
static double av_q2d (AVRational a);

//浮点转 AVRational,分子、分母的最大值是 max
AVRational av_d2q (double d, int max) av_const

//AVRational 转 float,返回以 32 位表示的浮点数(float)值
uint32_t av_q2intfloat (AVRational q)

//比较两个 AVRational.0:a == b; 1: a > b; -1: a < b
static int  av_cmp_q (AVRational a, AVRational b);

//分数的约分,分子分母需要小于 max
int av_reduce (int * dst_num, int * dst_den, int64_t num, int64_t den, int64_t max)

//分数相加,return b + c
AVRational av_add_q (AVRational b, AVRational c) av_const

//分数相减,return b - c
AVRational av_sub_q (AVRational b, AVRational c) av_const

//分数相乘,return b * c
AVRational av_mul_q (AVRational b, AVRational c) av_const

//分数相除,return b/c
AVRational av_div_q (AVRational b, AVRational c) av_const

//求倒数,return 1/q
static AVRational av_inv_q (AVRational q)

//找到离 q 最近的值.0:等距离;-1:q1 离 q 近;1:q2 离 q 近
int av_nearer_q (AVRational q, AVRational q1, AVRational q2)

//找到指定数组中离 q 最近的值的索引
int av_find_nearest_q_idx (AVRational q, const AVRational * q_list)

//在特定时间戳上累加一个值,返回为 ts + inc / inc_tb * ts_tb
int64_t av_add_stable (AVRational ts_tb, int64_t ts, AVRational inc_tb, int64_t inc)
```

2. 字符串函数

与字符串操作相关的宏,代码如下:

```
//chapter5/5.1.help.txt
//转译所有空格,包括位于字符串中间的空格
#define     AV_ESCAPE_FLAG_WHITESPACE   (1 << 0)

//仅对特殊标记字符转译.如果没有该标志,则将对 av_get_token()返回的特殊字符做转译
#define     AV_ESCAPE_FLAG_STRICT   (1 << 1)

//支持大于 0x10FFFF 的 codepoints
#define     AV_UTF8_FLAG_ACCEPT_INVALID_BIG_CODES   1

//支持非字符(0xFFFE 和 0xFFFF)
#define     AV_UTF8_FLAG_ACCEPT_NON_CHARACTERS   2

//支持 UTF - 16 的 surrogates codes
#define     AV_UTF8_FLAG_ACCEPT_SURROGATES   4

//排除不被 XML 支持的控制码
#define     AV_UTF8_FLAG_EXCLUDE_XML_INVALID_CONTROL_CODES   8

#define     AV_UTF8_FLAG_ACCEPT_ALL   AV_UTF8_FLAG_ACCEPT_INVALID_BIG_CODES|AV_UTF8_FLAG_
ACCEPT_NON_CHARACTERS|AV_UTF8_FLAG_ACCEPT_SURROGATES
```

与字符串操作相关的枚举,如表 5-2 所示。

表 5-2　字符串相关的枚举

枚　举　名	值	描　　述
AV_ESCAPE_MODE_AUTO	0	使用自动选择的反转译模式
AV_ESCAPE_MODE_BACKSLASH	1	使用反斜杠转译
AV_ESCAPE_MODE_QUOTE	2	使用单引号转译

与字符串操作相关的 API,代码如下:

```
//chapter5/5.1.help.txt
//标准库中 strstr 的替代版本,区分大小写字母的字符串匹配
char * av_stristr (const char * haystack, const char * needle);

//忽略大小写字母的字符串匹配
int av_stristart (const char * str, const char * pfx, const char ** ptr);

//标准库中 strstr 的替代版本,限制字符串长度
char * av_strnstr (const char * haystack, const char * needle, size_t hay_length);

//查找匹配子字符串,如果找到,则返回非 0,否则返回 0
//如果 pfx 是 str 的子串,则 ptr 将返回第一匹配字符串的地址
```

```
int av_strstart (const char * str, const char * pfx, const char ** ptr);

//ASCII 字符串比较,对大小写敏感
int av_strcasecmp (const char * a, const char * b);

//ASCII 字符串比较,对大小写不敏感
int av_strncasecmp (const char * a, const char * b, size_t n);

//与 BSD strlcpy 功能类似,复制最多 size - 1 个字符,并将 dst[size - 1]置为'\0'
size_t av_strlcpy (char * dst, const char * src, size_t size);
//字符串格式化输出
size_t av_strlcatf (char * dst, size_t size, const char * fmt,...);

//字符串拼接,该保证 dst 的总长度不超过 size - 1
size_t av_strlcat (char * dst, const char * src, size_t size);

//返回从字符串开始位置连续的非'\0'字符的数目
size_t av_strnlen (const char * s, size_t len);

//将字符串和参数格式化输出到一个临时缓冲区,正常返回的指针需要通过调用 av_free 释放
char * av_asprintf (const char * fmt,...)

//将 double 转换为字符串
char * av_d2str (double d);

//字符串分割
char * av_get_token (const char ** buf, const char * term);
//类似 POSIX.1 中的 strtok_r()函数
char * av_strtok (char * s, const char * delim, char ** saveptr);

//判断字符类型和转换
int av_isdigit (int c); //ASCII isdigit
int av_isgraph (int c); //ASCII isgraph
int av_isspace (int c); //ASCII isspace
int av_toupper (int c); //ASCII 字符转大写
int av_tolower (int c); //ASCII 字符转小写
int av_isxdigit (int c); //ASCII isxdigit

//查找盘符或根目录
const char * av_basename (const char * path);

//获得目录名,此函数会修改 path 变量的值
const char * av_dirname (char * path);

//在 names 中查找 name,如果找到,则返回 1;否则返回 0
int av_match_name (const char * name, const char * names);
```

```
//路径拼接,添加新的 folder
char * av_append_path_component (const char * path, const char * component);

//文本转译处理
int av_escape (char ** dst, const char * src, const char * special_chars, enum AVEscapeMode
mode, int flags);

//UTF-8 字符读取
int av_utf8_decode (int32_t * codep, const uint8_t ** bufp, const uint8_t * buf_end, unsigned
int flags);

//在 list 查找 name
int av_match_list (const char * name, const char * list, char separator);
```

3. 日志函数

日志模块主要提供日志输出的相关接口和宏,通过这些接口外部可以控制 FFmpeg 所有的日志输出,其中与日志级别相关的宏定义如表 5-3 所示。

表 5-3 AV_LOG 的日志级别

枚 举 名	值	描 述
AV_LOG_QUIET	−8	无任何日志输出
AV_LOG_PANIC	0	发生无法处理的错误,程序将崩溃
AV_LOG_FATAL	8	发生无法恢复的错误,例如特定格式的文件头没找到
AV_LOG_ERROR	16	发生错误,无法无损恢复,但是对后续运行无影响
AV_LOG_WARNING	24	警告信息,有些部分不太正确,可能会也可能不会导致问题
AV_LOG_INFO	32	标准信息输出,通常可以使用这个
AV_LOG_VERBOSE	40	详细信息,通常比较多,频繁输出那种
AV_LOG_Debug	48	仅用于 libav * 开发者使用的标志
AV_LOG_TRACE	56	特别烦琐的调试信息,仅用在 libav * 开发中

与日志操作相关的 API 及其他宏定义,代码如下:

```
//chapter5/5.1.help.txt
//最大 log 级别的跨度
#define AV_LOG_MAX_OFFSET (AV_LOG_TRACE - AV_LOG_QUIET)
#define AV_LOG_C(x) ((x) << 8) //设置日志输出的颜色,通常用法如下
av_log(ctx, AV_LOG_INFO|AV_LOG_C(255), "Message in red\n");
//跳过重复的日志
#define AV_LOG_SKIP_REPEATED   1
//输出日志对应的 level
#define AV_LOG_PRINT_LEVEL   2
```

```
//日志输出函数包括 log()和 vlog()
void av_log (void * avcl, int level, const char * fmt,...);
void av_vlog (void * avcl, int level, const char * fmt, va_list vl);

//设置和获取日志级别:av_log_level,av_set_level
int av_log_get_level (void);
void av_log_set_level (int level);

//设置日志输出回调函数及默认的日志回调函数
void av_log_set_callback (void( * callback)(void * , int, const char * , va_list));
void av_log_default_callback (void * avcl, int level, const char * fmt, va_list vl);

//获得 ctx 的名字
const char * av_default_item_name (void * ctx);

//获得 AVClassCategory 分类
AVClassCategory av_default_get_category (void * ptr)

//使用 FFmpeg 默认的 log 输出方式格式化参数
void av_log_format_line (void * ptr, int level, const char * fmt, va_list vl, char * line, int
line_size, int * print_prefix);
int av_log_format_line2 (void * ptr, int level, const char * fmt, va_list vl, char * line, int
line_size, int * print_prefix);

//设置与日志输出相关的标志值 flag
void av_log_set_flags (int arg);
int av_log_get_flags (void);
```

4. 内存管理函数

与内存管理相关的函数主要涉及堆管理,其中堆函数的代码如下:

```
//chapter5/5.1.help.txt
//分配指定大小的内存,类似 C 的 malloc
void * av_malloc (size_t size);

//分配内存并将该内存初始化为 0
void * av_mallocz (size_t size);

//作为 C 中 calloc 的替代
void * av_calloc (size_t nmemb, size_t size);

//作为 C 中 realloc 的替代
void * av_realloc (void * ptr, size_t size);
```

```
//在 buffer 不足的情况下,重新分配内存,否则不做处理
void * av_fast_realloc (void * ptr, unsigned int * size, size_t min_size);

//在 buffer 不够时重新分配,够用时直接使用默认的
void av_fast_malloc (void * ptr, unsigned int * size, size_t min_size);

//与 av_fast_malloc 功能类似,只是分配成功之后初始化为 0
void av_fast_mallocz (void * ptr, unsigned int * size, size_t min_size);

//释放内存,类似 C 中的 free
void av_free (void * ptr);

//释放内存,并将指针置为空
void av_freep (void * ptr);

//字符串复制,并返回一个内存指针,返回值需要通过 av_free 释放
char * av_strdup (const char * s);
char * av_strndup (const char * s, size_t len);

//内存区域复制,并返回一个动态申请的内存
void * av_memdup (const void * p, size_t size);

//支持重叠区域的 memcpy 实现
void av_memcpy_backptr (uint8_t * dst, int back, int cnt);

//检查 a * b 是否存在溢出,返回值 0 表示成功, AVERROR(EINVAL)表示存在溢出
int av_size_mult (size_t a, size_t b, size_t * r);
```

5. 错误码及错误处理函数

FFmpeg 提供了统一的与错误处理逻辑相关的 API 及宏定义,代码如下:

```
//chapter5/5.1.help.txt
#define AVERROR(e) (e)                                  //错误值
#define AVUNERROR(e) (e)                                //非错误值
#define FFERRTAG(a, b, c, d) (-(int)MKTAG(a, b, c, d))//生成四字节错误码
#define AV_ERROR_MAX_STRING_SIZE 64                     //错误字符的最大长度

//错误码转字符串,方便使用的宏
#define av_err2str(errnum) av_make_error_string((char[AV_ERROR_MAX_STRING_SIZE]){0}, AV_
ERROR_MAX_STRING_SIZE, errnum)

//根据错误码,将错误信息输到 errbuf 中,如果返回负值,则表示错误码未找到
int av_strerror (int errnum, char * errbuf, size_t errbuf_size);

//功能类似,只是返回值不同
static char * av_make_error_string (char * errbuf, size_t errbuf_size, int errnum);
```

6. 与数据结构操作系统相关的函数

1）AVBuffer

AVBuffer是一系列支持引用计数的数据缓冲的API集合，包括API及相关结构体，代码如下：

```
//chapter5/5.1.help.txt
//基于引用计数的 buffer 类型,外部不可见,通过 AVBufferRef 访问
typedef struct AVBuffer AVBuffer;

//对数据 buffer 的引用,该结构不应该被直接分配
struct AVBufferRef {
    AVBuffer * buffer;
    uint8_t * data; //数据缓冲和长度
    int      size;
};

//创建将长度指定为 size 的 buffer,一般调用 av_malloc
AVBufferRef * av_buffer_alloc (int size);
AVBufferRef * av_buffer_allocz (int size);
//使用 data 中长度为 size 的数据创建 AVBuffer,并注册释放函数及标志
AVBufferRef * av_buffer_create (uint8_t * data, int size, void( * free)(void * opaque, uint8_
t * data), void * opaque, int flags);

//使用默认方式释放 buffer
void av_buffer_default_free (void * opaque, uint8_t * data);

//复制和减少对 buffer 的引用计数
AVBufferRef * av_buffer_ref (AVBufferRef * buf);
void av_buffer_unref (AVBufferRef ** buf);

//buffer 是否可写
int av_buffer_is_writable (const AVBufferRef * buf);

//返回 av_buffer_create 设置的 opaque 参数
void * av_buffer_get_opaque (const AVBufferRef * buf);

//返回当前 buffer 的引用计数值
int av_buffer_get_ref_count (const AVBufferRef * buf);

//创建一个可写的 buffer
int av_buffer_make_writable (AVBufferRef ** buf);

//重新分配 buffer 大小
int av_buffer_realloc (AVBufferRef ** buf, int size);
```

2）AVBufferPool

AVBufferPool是一个由AVBuffer构成的、没有线程锁的、线程安全的缓冲池。频繁地分配和释放大量内存缓冲区效率会较低。AVBufferPool主要解决用户需要使用同样长度的缓冲区的情况(例如原始PCM格式的音频帧)。

开始时用户可以调用av_buffer_pool_init()函数来创建缓冲池,然后在任何时间都可以调用av_buffer_pool_get()函数来获得buffer,在该buffer的引用计数为0时,将会返给缓冲池,这样就可以被循环使用了。用户使用完缓冲池之后可以调用av_buffer_pool_uninit()函数来释放缓冲池。相关的API函数的代码如下:

```
//chapter5/5.1.help.txt
//创建缓冲池
AVBufferPool * av_buffer_pool_init (int size, AVBufferRef * ( * alloc)(int size));

//Allocate and initialize a buffer pool with a more complex allocator.
//使用更复杂的分配器分配和初始化缓冲池,可以使用用户自定义的分配和释放函数
AVBufferPool * av_buffer_pool_init2 (int size, void * opaque, AVBufferRef * ( * alloc)(void
 * opaque, int size), void( * pool_free)(void * opaque));

//释放缓冲池
void av_buffer_pool_uninit (AVBufferPool ** pool);

//获得 buffer
AVBufferRef * av_buffer_pool_get (AVBufferPool * pool);
```

3）AVDictionary

AVDictionary是一个简单的键-值对的字典集,其中公开的结构体代码如下:

```
//chapter5/5.1.help.txt
typedef struct AVDictionaryEntry {
    char * key;
    char * value;
} AVDictionaryEntry;

typedef struct AVDictionary AVDictionary;
```

支持的宏定义,如表5-4所示。

表5-4 AV_DICT相关的宏定义

枚 举 名	值	描 述
AV_DICT_MATCH_CASE	1	大小写完全匹配的key检索
AV_DICT_IGNORE_SUFFIX	2	忽略后缀
AV_DICT_DONT_STRDUP_KEY	4	不复制key

续表

枚 举 名	值	描 述
AV_DICT_DONT_STRDUP_VAL	8	不制 value
AV_DICT_DONT_OVERWRITE	16	不覆盖原有值
AV_DICT_MULTIKEY	64	支持多重键值

支持的相关 API 函数,代码如下:

```
//chapter5/5.1.help.txt
//从 prev 开始检索 m 中的键值为 key 的元素
AVDictionaryEntry * av_dict_get (const AVDictionary * m, const char * key, const
AVDictionaryEntry * prev, int flags);

//获得 m 中的元素个数
int av_dict_count (const AVDictionary * m);
//设置 * pm 中的 key:value 键 - 值对
int av_dict_set (AVDictionary ** pm, const char * key, const char * value, int flags);
//使用 int64_t 的 value 类型
int av_dict_set_int (AVDictionary ** pm, const char * key, int64_t value, int flags);
//从 str 中解析键 - 值对,并添加到 * pm 中
int av_dict_parse_string (AVDictionary ** pm, const char * str, const char * key_val_sep,
const char * pairs_sep, int flags);
//AVDictionary 复制
int av_dict_copy (AVDictionary ** dst, const AVDictionary * src, int flags);
//释放 AVDictionary
void av_dict_free (AVDictionary ** m);
//按照字符串格式输出 AVDictionary 的所有数据
int av_dict_get_string (const AVDictionary * m, char ** buffer, const char key_val_sep, const
char pairs_sep);
```

4) AVTree

AVTree 是一种树容器,支持的插入、删除、查找等常用操作都是 $O(\log n)$ 复杂度的实现版本,主要提供的 API 的代码如下:

```
//chapter5/5.1.help.txt
//创建
struct AVTreeNode * av_tree_node_alloc (void);
//元素查找
void * av_tree_find (const struct AVTreeNode * root, void * key, int( * cmp)(const void * key,
const void * b), void * next[2]);
//元素插入
void * av_tree_insert (struct AVTreeNode ** rootp, void * key, int( * cmp)(const void * key,
const void * b), struct AVTreeNode ** next);
//销毁
```

```
void av_tree_destroy (struct AVTreeNode * t);
//数元素枚举
void av_tree_enumerate (struct AVTreeNode * t, void * opaque, int( * cmp)(void * opaque, void
 * elem), int( * enu)(void * opaque, void * elem));
```

5）AVFrame

AVFrame 是对原始多媒体数据的一个基于引用计数的抽象，比较常用的是音频帧和视频帧（例如 PCM、YUV 等格式的原始音视频帧），完整代码可以参考 libavutil/frame.h 文件。

AVFrame 必须使用 av_frame_alloc()函数进行分配，需要注意的是该函数仅仅分配 AVFrame 本身，AVFrame 中的数据缓冲必须通过其他方式管理。AVFrame 必须使用 av_frame_free()函数释放。AVFrame 通常只分配一次，然后用于存放不同的数据，例如 AVFrame 可以保存从 decoder 中解码出来的数据。在这种情况下 av_frame_unref()函数将释放所有由 frame 添加的引用计数并将其重置为初始值。

AVFrame 中的数据通常基于 AVBuffer API 提供的引用计数机制，其中的引用计数保存在 AVFrame.buf/AVFrame.extended_buf 字段中。sizeof(AVFrame)并不是公开 API 的一部分，以保证 AVFrame 中新添加成员之后可以正常运行。AVFrame 中的成员可以通过 AVOptions 访问，使用其对应的名称字符串即可。AVFrame 的 AVClass 可以通过 avcodec_get_frame_class()函数获得。

它提供的主要函数的代码如下：

```
//chapter5/5.1.help.txt
//获取和设置 AVFrame.best_effort_timestamp 的值
int64_t av_frame_get_best_effort_timestamp (const AVFrame * frame);
void av_frame_set_best_effort_timestamp (AVFrame * frame, int64_t val);

//获取和设置 AVFrame.pkt_duration
int64_t av_frame_get_pkt_duration (const AVFrame * frame);
void av_frame_set_pkt_duration (AVFrame * frame, int64_t val);

//获取和设置 AVFrame.pkt_pos
int64_t av_frame_get_pkt_pos (const AVFrame * frame);
void av_frame_set_pkt_pos (AVFrame * frame, int64_t val);

//获取和设置 AVFrame.channel_layout
int64_t av_frame_get_channel_layout (const AVFrame * frame);
void    av_frame_set_channel_layout (AVFrame * frame, int64_t val);

//获取和设置 AVFrame.channels
int     av_frame_get_channels (const AVFrame * frame);
void    av_frame_set_channels (AVFrame * frame, int val);

//获取和设置 AVFrame.sample_rate
```

```
int     av_frame_get_sample_rate (const AVFrame * frame);
void    av_frame_set_sample_rate (AVFrame * frame, int val);

//获取和设置 AVFrame.metadata
AVDictionary *   av_frame_get_metadata (const AVFrame * frame);
void av_frame_set_metadata (AVFrame * frame, AVDictionary * val);

//获取和设置 AVFrame.decode_error_flags
int av_frame_get_decode_error_flags (const AVFrame * frame);
void av_frame_set_decode_error_flags (AVFrame * frame, int val);

//获取和设置 AVFrame.pkt_size
int av_frame_get_pkt_size (const AVFrame * frame);
void av_frame_set_pkt_size (AVFrame * frame, int val);
AVDictionary ** avpriv_frame_get_metadatap (AVFrame * frame)

//获取和设置 AVFrame.colorspace
enum AVColorSpace av_frame_get_colorspace (const AVFrame * frame);
void av_frame_set_colorspace (AVFrame * frame, enum AVColorSpace val);

//获取和设置 AVFrame.color_range
enum AVColorRange av_frame_get_color_range (const AVFrame * frame);
void av_frame_set_color_range (AVFrame * frame, enum AVColorRange val)

//获得 ColorSpace 的名称
const char * av_get_colorspace_name (enum AVColorSpace val);

//创建和释放 AVFrame
AVFrame * av_frame_alloc (void);
void av_frame_free (AVFrame ** frame);

//增加引用计数
int av_frame_ref (AVFrame * dst, const AVFrame * src);
//AVFrame 复制
AVFrame * av_frame_clone (const AVFrame * src);
//去除引用计数
void av_frame_unref (AVFrame * frame);
//引用计数转译
void av_frame_move_ref (AVFrame * dst, AVFrame * src);
//重新分配缓冲区
int av_frame_get_buffer (AVFrame * frame, int align);
//AVFrame 复制
int av_frame_copy (AVFrame * dst, const AVFrame * src);
int av_frame_copy_props (AVFrame * dst, const AVFrame * src);

//获得某个平面的数据
AVBufferRef * av_frame_get_plane_buffer (AVFrame * frame, int plane);
```

6）AVOptions

AVOptions 提供了通用的 option 设置和获取机制，可适用于任意 struct（通常要求该结构体的第 1 个成员必须是 AVClass 指针，该 AVClass.options 必须指向一个 AVOptions 的静态数组，以 NULL 作为结束），该结构体定义的代码如下：

```
//chapter5/5.1.help.txt
struct AVOption {
    const char * name;        //名称
    const char * help;        //说明信息
    int offset;               //相对于上下文的偏移量,对常量而言,必须是 0
    enum AVOptionType type;   //类型

    //实际存储数据的共用体
    union {
        int64_t i64;
        double dbl;
        const char * str;
        AVRational q;
    } default_val;
    double min;
    double max;
    int flags; //AV_OPT_FLAG_XXX
    const char * unit;
};

struct AVOptionRange {
    const char * str;
    double value_min, value_max; //值范围,对字符串表示长度,对分辨率表示最大最小像素个数
    double component_min, component_max;//实际数据取值区间,对字符串,表示 Unicode 的取值范
//围 ASCII 为[0,127]
    int is_range;             //是否是一个取值范围,1:是;0:单值
};

struct AVOptionRanges {
    AVOptionRange ** range;
    int nb_ranges;            //range 数目
    int nb_components;        //组件数目
} AVOptionRanges;
```

选项（Option）的设置及与获取相关的 API 函数，代码如下：

```
//chapter5/5.1.help.txt
//set 类函数
int av_opt_set (void * obj, const char * name, const char * val, int search_flags); //任意字符串
int av_opt_set_int (void * obj, const char * name, int64_t val, int search_flags);  //int
```

```
int av_opt_set_double (void * obj, const char * name, double val, int search_flags); //double
int av_opt_set_q (void * obj, const char * name, AVRational val, int search_flags);
//AVRational
int av_opt_set_bin (void * obj, const char * name, const uint8_t * val, int size, int search_
flags); //二进制
int av_opt_set_image_size (void * obj, const char * name, int w, int h, int search_flags);
//图像分辨率
int av_opt_set_pixel_fmt (void * obj, const char * name, enum AVPixelFormat fmt, int search_
flags); //PixelFormat
int av_opt_set_sample_fmt (void * obj, const char * name, enum AVSampleFormat fmt, int search_
flags); //SampleFormat
int av_opt_set_video_rate (void * obj, const char * name, AVRational val, int search_flags);
//视频帧率
int av_opt_set_channel_layout (void * obj, const char * name, int64_t ch_layout, int search_
flags); //channel_layou
int av_opt_set_dict_val (void * obj, const char * name, const AVDictionary * val, int search_
flags); //AVDictionary

//get 类函数
int av_opt_get (void * obj, const char * name, int search_flags, uint8_t ** out_val);
int av_opt_get_int (void * obj, const char * name, int search_flags, int64_t * out_val);
int av_opt_get_double (void * obj, const char * name, int search_flags, double * out_val);
int av_opt_get_q (void * obj, const char * name, int search_flags, AVRational * out_val);
int av_opt_get_image_size (void * obj, const char * name, int search_flags, int * w_out, int
* h_out);
int av_opt_get_pixel_fmt (void * obj, const char * name, int search_flags, enum AVPixelFormat
* out_fmt);
int av_opt_get_sample_fmt (void * obj, const char * name, int search_flags, enum
AVSampleFormat * out_fmt);
int av_opt_get_video_rate (void * obj, const char * name, int search_flags, AVRational * out_val);
int av_opt_get_channel_layout (void * obj, const char * name, int search_flags, int64_t * ch_
layout);
int av_opt_get_dict_val (void * obj, const char * name, int search_flags, AVDictionary ** out_val);
```

与 AVOptions 相关的 API 及结构体,代码如下:

```
//chapter5/5.1.help.txt
//优先检索给定对象的子对象
#define AV_OPT_SEARCH_CHILDREN   (1 << 0)
#define AV_OPT_SEARCH_FAKE_OBJ   (1 << 1)
//在 av_opt_get 中支持返回 NULL,而不是空字符串
#define AV_OPT_ALLOW_NULL   (1 << 2)
```

```
//支持多组件范围的 option
#define AV_OPT_MULTI_COMPONENT_RANGE    (1 << 12)
//只保存非默认值的 option
#define AV_OPT_SERIALIZE_SKIP_DEFAULTS   0x00000001
//只保存完全符合 opt_flags 的 option
#define AV_OPT_SERIALIZE_OPT_FLAGS_EXACT   0x00000002

//枚举值
enum AVOptionType {
  AV_OPT_TYPE_FLAGS, AV_OPT_TYPE_INT, AV_OPT_TYPE_INT64, AV_OPT_TYPE_DOUBLE,
  AV_OPT_TYPE_FLOAT, AV_OPT_TYPE_STRING, AV_OPT_TYPE_RATIONAL, AV_OPT_TYPE_BINARY,
  AV_OPT_TYPE_DICT, AV_OPT_TYPE_UINT64, AV_OPT_TYPE_CONST = 128, AV_OPT_TYPE_IMAGE_SIZE =
MKBETAG('S','I','Z','E'),
  AV_OPT_TYPE_PIXEL_FMT = MKBETAG('P','F','M','T'), AV_OPT_TYPE_SAMPLE_FMT = MKBETAG('S','F',
'M','T'),
  AV_OPT_TYPE_VIDEO_RATE = MKBETAG('V','R','A','T'), AV_OPT_TYPE_DURATION = MKBETAG('D','U',
'R',' '),
  AV_OPT_TYPE_COLOR = MKBETAG('C','O','L','R'), AV_OPT_TYPE_CHANNEL_LAYOUT = MKBETAG('C','H',
'L','A'),
  AV_OPT_TYPE_BOOL = MKBETAG('B','O','O','L')
}

enum  { AV_OPT_FLAG_IMPLICIT_KEY = 1 }

//支持的函数
//显示 obj 的所有 options
int av_opt_show2 (void * obj, void * av_log_obj, int req_flags, int rej_flags);
//将 s 的所有 options 设置为默认值
void av_opt_set_defaults (void * s);
void av_opt_set_defaults2 (void * s, int mask, int flags);

//解析 opts 中的键 - 值对,并设置(主要区别是对 ctx 要求不一样)
int av_set_options_string (void * ctx, const char * opts, const char * key_val_sep, const char
 * pairs_sep);
int av_opt_set_from_string (void * ctx, const char * opts, const char * const * shorthand,
const char * key_val_sep, const char * pairs_sep);
//释放 obj 所有分配的 options 资源
void av_opt_free (void * obj);
//检查 obj.flag_name 对应的 AVOption 属性名为 filed_name 时该值是否设置
int av_opt_flag_is_set (void * obj, const char * field_name, const char * flag_name);
//从 AVDictionary 读取 option
int av_opt_set_dict (void * obj, struct AVDictionary ** options);
int av_opt_set_dict2 (void * obj, struct AVDictionary ** options, int search_flags);

//从 ropts 开始提取键 - 值对
```

```
int av_opt_get_key_value (const char ** ropts, const char * key_val_sep, const char * pairs_
sep, unsigned flags, char ** rkey, char ** rval);
//在 obj 中查找名字为 name 的 AVOption
const AVOption * av_opt_find (void * obj, const char * name, const char * unit, int opt_flags,
int search_flags);
const AVOption * av_opt_find2 (void * obj, const char * name, const char * unit, int opt_
flags, int search_flags, void ** target_obj);

//遍历 obj 中所有的 AVOption
const AVOption * av_opt_next (const void * obj, const AVOption * prev);
//遍历 obj 中所有使能的子对象
void *   av_opt_child_next (void * obj, void * prev);
const AVClass * av_opt_child_class_next (const AVClass * parent, const AVClass * prev);

//获取 obj 中名字为 name 的属性的指针
void * av_opt_ptr (const AVClass * avclass, void * obj, const char * name);

//AVOption 复制
int av_opt_copy (void * dest, const void * src);

//释放 AVOptionRanges,并置为 NULL
void av_opt_freep_ranges (AVOptionRanges ** ranges);
//获取有效范围的取值
int av_opt_query_ranges (AVOptionRanges ** , void * obj, const char * key, int flags);
int av_opt_query_ranges_default (AVOptionRanges ** , void * obj, const char * key, int
flags);

//获取是否所有的 AVOption 都是默认值
int av_opt_is_set_to_default (void * obj, const AVOption * o);
int av_opt_is_set_to_default_by_name (void * obj, const char * name, int search_flags);
//序列化所有的 AVOption
int av_opt_serialize (void * obj, int opt_flags, int flags, char ** buffer, const char key_val
_sep, const char pairs_sep);
    Serialize object's options. More...
```

7. 其他辅助函数

其他辅助函数主要包括密钥、哈希值、宏、库版本、常量等。FFmpeg 支持 AES、Base64、Blowfish 等常用的密钥算法，哈希函数支持 CRC、MD5、SHA、SHA-512 等。

1）与库版本相关的 **API** 及宏

代码如下：

```
//chapter5/5.1.help.txt
//版本号的两种信息
#define AV_VERSION_INT(a, b, c)   ((a)<< 16 | (b)<< 8 | (c))
```

```
#define AV_VERSION_DOT(a, b, c)   a ##.##b ##.##c
#define AV_VERSION(a, b, c)   AV_VERSION_DOT(a, b, c)

//主版本、次版本、微版本号
#define AV_VERSION_MAJOR(a) ((a) >> 16)
#define AV_VERSION_MINOR(a) (((a) & 0x00FF00) >> 8)
#define AV_VERSION_MICRO(a) ((a) & 0xFF)

#define LIBAVUTIL_VERSION_MAJOR   55
#define LIBAVUTIL_VERSION_MINOR   74
#define LIBAVUTIL_VERSION_MICRO   100

#define LIBAVUTIL_VERSION_INT
#define LIBAVUTIL_VERSION

#define LIBAVUTIL_BUILD LIBAVUTIL_VERSION_INT

//与库版本相关的 API
unsigned avutil_version (void);              //获取版本
const char * av_version_info (void);         //获取版本信息
const char * avutil_configuration (void);    //配置字符串
const char * avutil_license (void);          //获取版本 license
```

2）与字符串处理相关的宏

代码如下：

```
//chapter5/5.1.help.txt
#define AV_STRINGIFY(s) AV_TOSTRING(s)       //转换为字符串
#define AV_TOSTRING(s) #s
#define AV_GLUE(a, b) a ##b                  //两个变量拼接
#define AV_JOIN(a, b) AV_GLUE(a, b)
```

3）与时间戳相关的宏

代码如下：

```
//chapter5/5.1.help.txt
//未定义的时间戳
#define AV_NOPTS_VALUE ((int64_t)UINT64_C(0x8000000000000000))

//内部时间基准,通常是微秒(μs)
#define AV_TIME_BASE   1000000
#define AV_TIME_BASE_Q   (AVRational){1, AV_TIME_BASE}
```

4）AVMediaType 及 FourCC

代码如下：

```
//chapter5/5.1.help.txt
#define AV_FOURCC_MAX_STRING_SIZE 32
#define av_fourcc2str(fourcc) av_fourcc_make_string((char[AV_FOURCC_MAX_STRING_SIZE]){0},
fourcc)
//将 fourcc 填充到字符串中
char * av_fourcc_make_string (char * buf, uint32_t fourcc);
//媒体类型:音频、视频、数据、未知
enum    AVMediaType {
  AVMEDIA_TYPE_UNKNOWN = -1, AVMEDIA_TYPE_VIDEO, AVMEDIA_TYPE_AUDIO, AVMEDIA_TYPE_DATA,
  AVMEDIA_TYPE_SUBTITLE, AVMEDIA_TYPE_ATTACHMENT, AVMEDIA_TYPE_NB
}
//获得 AVMediaType 对应的字符串
const char * av_get_media_type_string (enum AVMediaType media_type);

//使用 utf8 的文件名打开文件
FILE * av_fopen_utf8 (const char * path, const char * mode);

//返回表示内部时间戳的 time_base
AVRational av_get_time_base_q (void);
```

5）AVPicture 相关

关于 AVPicture 的用法说明，其中包括像素采样格式、基本图像平面操作等，相关的 API 及枚举量的定义，代码如下：

```
//chapter5/5.1.help.txt
//各种帧类型,I/P/B/S/SI/SP/BI
enum  AVPictureType {
  AV_PICTURE_TYPE_NONE = 0, AV_PICTURE_TYPE_I, AV_PICTURE_TYPE_P, AV_PICTURE_TYPE_B,
  AV_PICTURE_TYPE_S, AV_PICTURE_TYPE_SI, AV_PICTURE_TYPE_SP, AV_PICTURE_TYPE_BI
}

//使用单个字符表示图片类型,未知时返回'?'
char av_get_picture_type_char (enum AVPictureType pict_type);

//计算给定采样格式和宽度的图像的 linesize
int av_image_get_linesize (enum AVPixelFormat pix_fmt, int width, int plane);

//填充特定采样格式和宽度的 linesize 数组
int av_image_fill_linesizes (int linesizes[4], enum AVPixelFormat pix_fmt, int width);
```

```
//使用 ptr 中的数据填充 plane 中的数组(不申请内存)
int av_image_fill_pointers (uint8_t * data[4], enum AVPixelFormat pix_fmt, int height, uint8_
t * ptr, const int linesizes[4]);

//根据给定的格式分配 Image 数组,主要是 pointers 和 linesizes
int av_image_alloc (uint8_t * pointers[4], int linesizes[4], int w, int h, enum AVPixelFormat
pix_fmt, int align);

//复制特定 plane 的数据,即将 src 中的数据复制到 dst 中
void av_image_copy_plane (uint8_t * dst, int dst_linesize, const uint8_t * src, int src_
linesize, int Bytewidth, int height);

//复制 Image,将 src_data 中的数据复制到 dst_data 中
void av_image_copy (uint8_t * dst_data[4], int dst_linesizes[4], const uint8_t * src_data
[4], const int src_linesizes[4], enum AVPixelFormat pix_fmt, int width, int height);

//将 src 填充到 dst_data 中
int av_image_fill_arrays (uint8_t * dst_data[4], int dst_linesize[4], const uint8_t * src,
enum AVPixelFormat pix_fmt, int width, int height, int align);

//返回指定格式的图片需要的缓冲区长度
int av_image_get_buffer_size (enum AVPixelFormat pix_fmt, int width, int height, int align);

//将图像填充为黑色
int av_image_fill_black (uint8_t * dst_data[4], const ptrdiff_t dst_linesize[4], enum
AVPixelFormat pix_fmt, enum AVColorRange range, int width, int height);
```

5.2 AVLog 应用案例及剖析

FFmpeg 有专门的日志输出系统,核心函数只有一个:av_log(),下面通过案例来详细应用并解析相关知识点。

1. 创建 Qt 工程使用 FFmpeg 操作目录

1) 创建 Qt 工程

打开 Qt Creator,创建一个 Qt Console 工程,具体操作步骤可以参考“1.4 搭建 FFmpeg 的 Qt 开发环境”,工程名称为 QtFFmpeg5_Chapter5_001,如图 5-2 所示。由于使用的是 FFmpeg 5.0.1 的 64 位开发包,所以编译套件应选择 64 位的 MSVC 或 MinGW,如图 5-3 所示。

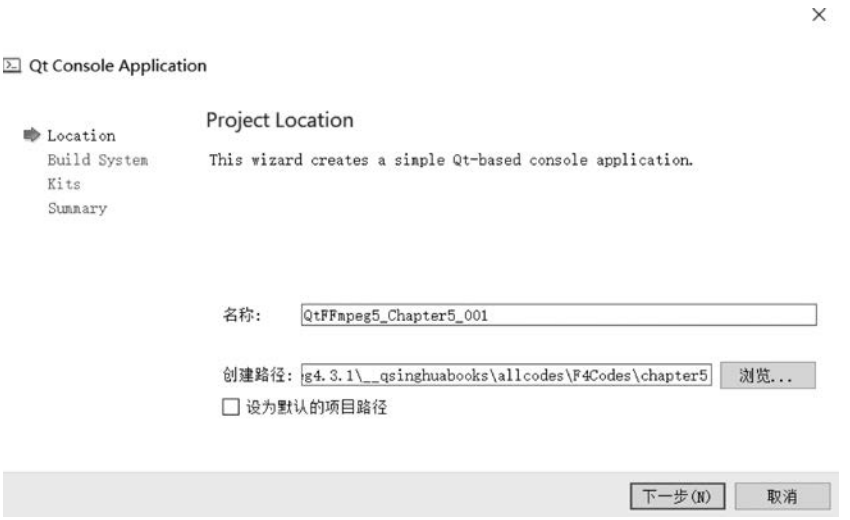

图 5-2　Qt 创建工程

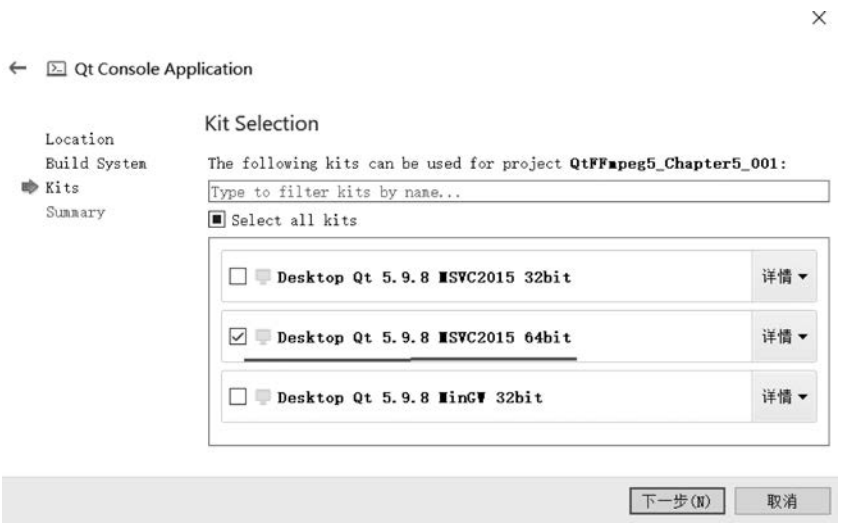

图 5-3　Qt 工程之 64 位编译器

2）引用 **FFmpeg** 的头文件和库文件

打开配置文件 QtFFmpeg5_Chapter5_001.pro，添加引用头文件及库文件的代码，如图 5-4 所示。由于笔者的工程目录 QtFFmpeg5_Chapter5_001 在 chapter5 目录下，而 chapter5 目录与 ffmpeg-n5.0-latest-win64-gpl-shared-5.0 目录是平级关系，所以项目配置文件里引用 FFmpeg 开发包目录的代码是 $$PWD/../../ffmpeg-n5.0-latest-win64-gpl-shared-5.0/，$$PWD 代表当前配置文件（QtFFmpeg5_Chapter5_001.pro）所在的目录，../../代表父目录的父目录。

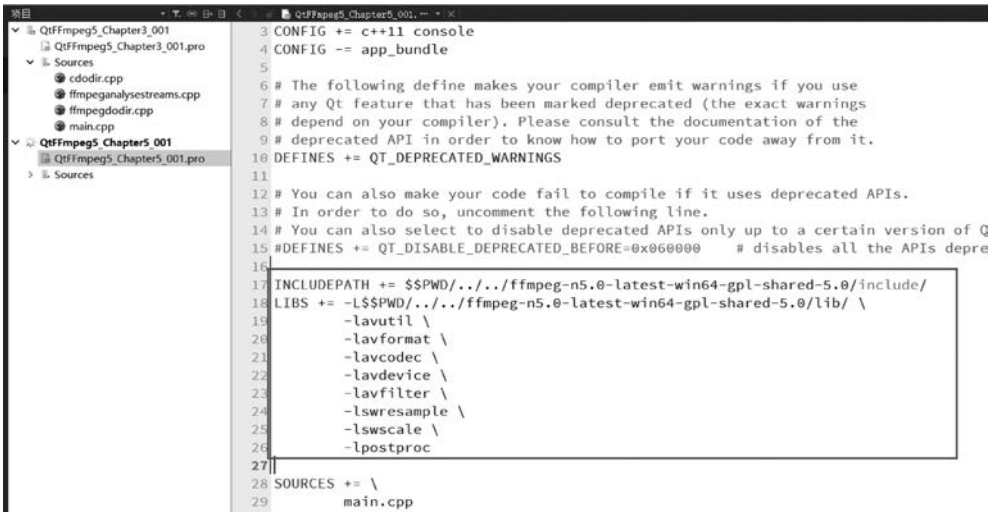

图 5-4　Qt 工程引用 FFmpeg 的头文件和库文件

在.pro 项目配置中，添加头文件和库文件的引用，代码如下：

```
//chapter5/QtFFmpeg5_Chapter5_001/QtFFmpeg5_Chapter5_001.pro
INCLUDEPATH += $$PWD/../../ffmpeg-n5.0-latest-win64-gpl-shared-5.0/include/
LIBS += -L$$PWD/../../ffmpeg-n5.0-latest-win64-gpl-shared-5.0/lib/ \
        -lavutil \
        -lavformat \
        -lavcodec \
        -lavdevice \
        -lavfilter \
        -lswresample \
        -lswscale \
        -lpostproc
```

2. FFmpeg 的日志输出 av_log()函数案例应用

FFmpeg 的日志信息从重到轻分为 Panic、Fatal、Error、Warning、Info、Verbose、Debug 几个级别，并且日志信息输到控制台的颜色也不同。下面的函数输出了几种不同级别的日志。默认情况下，所有日志信息都发送到 stderr，而不是 stdout，代码如下：

```
//chapter5/QtFFmpeg5_Chapter5_001/main.cpp
void test_log(){
    av_register_all();
    AVFormatContext *pobj = NULL;
    pobj = avformat_alloc_context();
    printf("====================================\n");
```

```
    av_log(pobj ,AV_LOG_PANIC,"Panic: Something went really wrong and we will crash now.\n");
    av_log(pobj ,AV_LOG_FATAL,"Fatal: Something went wrong and recovery is not possible.\n");
    av_log(pobj ,AV_LOG_ERROR,"Error: Something went wrong and cannot losslessly be
recovered.\n");
    av_log(pobj ,AV_LOG_WARNING,"Warning: This may or may not lead to problems.\n");
    av_log(pobj ,AV_LOG_INFO,"Info: Standard information.\n");
    av_log(pobj ,AV_LOG_VERBOSE,"Verbose: Detailed information.\n");
    av_log(pobj ,AV_LOG_Debug,"Debug: Stuff which is only useful for libav* developers.\n");
    printf("====================================\n");
    avformat_free_context(pobj );
}
```

将上述函数添加到 main.cpp 文件中，需要引用相关的头文件，代码如下：

```
//chapter5/QtFFmpeg5_Chapter5_001/main.cpp
extern "C"{   //C++调用 C 函数
    #include <libavutil/avutil.h>
    #include <libavformat/avformat.h>
}
```

注意：C++调用 C 函数，需要用 extern "C"{}将头文件括起来。

编译并运行该项目，会输出不同颜色的日志信息，如图 5-5 所示。

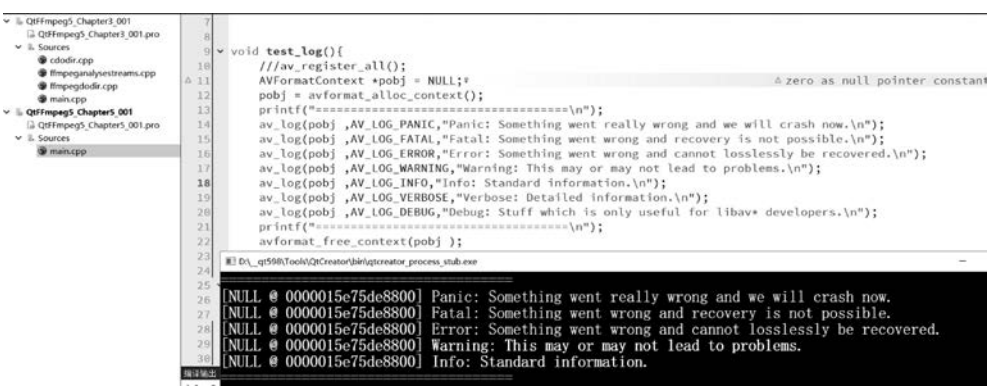

图 5-5　FFmpeg 的输出日志

观察日志输出信息，会发现没有输出 AV_LOG_VERBOSE 和 AV_LOG_Debug 信息。这是因为 FFmpeg 的默认日志级别是 AV_LOG_INFO。可以通过 av_log_set_level()函数设置日志级别，修改 test_log()函数，代码如下：

```
//chapter5/QtFFmpeg5_Chapter5_001/main.cpp
void test_log(){
```

```
    //av_register_all();
    AVFormatContext * pobj = NULL;
    pobj = avformat_alloc_context();

    //av_log_set_level(AV_LOG_VERBOSE);  //设置日志级别
    av_log_set_level(AV_LOG_Debug);
     ...
}
```

重新编译并运行该项目，会输出不同颜色的日志信息，此时可发现多了两条绿色的日志信息（AV_LOG_VERBOSE 和 AV_LOG_Debug），如图 5-6 所示。

图 5-6　FFmpeg 的日志级别控制及输出日志

3. FFmpeg 的日志级别

FFmpeg 的日志信息从重到轻分为 Panic、Fatal、Error、Warning、Info、Verbose、Debug 几个级别，可以查看官网的介绍，网址为 https://ffmpeg.org/ffmpeg.html#Generic-options。相关的几个宏定义在 libavutil/log.h 文件中，代码如下：

```
//chapter5/5.2.help.txt
/**
 * Print no output. :不输出日志
 */
#define AV_LOG_QUIET     -8

/** 出了点问题，现在要崩溃了
 * Something went really wrong and we will crash now.
 */
#define AV_LOG_PANIC     0

/** 出现问题，无法恢复
```

```
 * Something went wrong and recovery is not possible.
 * For example, no header was found for a format which depends
 * on headers or an illegal combination of parameters is used.
 * 例如,找不到需要依赖于头的相关格式的头信息,或者使用了非法的参数组合.
 */
#define AV_LOG_FATAL     8

/** 出了问题,无法无损地恢复. 然而,并非所有未来的数据都会受到影响.
 * Something went wrong and cannot losslessly be recovered.
 * However, not all future data is affected.
 */
#define AV_LOG_ERROR     16

/** 有些东西看起来不太对劲.这可能会也可能不会导致问题.例如使用'-vstrict -2'.
 * Something somehow does not look correct. This may or may not
 * lead to problems. An example would be the use of '-vstrict -2'.
 */
#define AV_LOG_WARNING   24

/**
 * Standard information. 标准信息,这个是默认级别
 */
#define AV_LOG_INFO      32

/**
 * Detailed information. 细节信息
 */
#define AV_LOG_VERBOSE   40

/** 只对 libav* 开发人员有用的东西.
 * Stuff which is only useful for libav* developers.
 */
#define AV_LOG_Debug     48

/** 非常详细的调试,对 libav* 开发非常有用.
 * Extremely verbose Debugging, useful for libav* development.
 */
#define AV_LOG_TRACE     56
```

注意：FFmpeg 的默认日志级别是 AV_LOG_INFO,此外,还有一个级别不输出任何信息,即 AV_LOG_QUIET。

4. FFmpeg 的日志输出 av_log()函数详解

av_log()是 FFmpeg 中输出日志的函数,在 FFmpeg 的源代码文件中到处遍布着 av_

log()函数。一般情况下 FFmpeg 类库的源代码中不允许使用 printf()这种函数,所有的输出一律使用 av_log()。av_log()函数声明在头文件 libavutil\log.h 中,代码如下:

```
//chapter5/5.2.help.txt
/**
 * Send the specified message to the log if the level is less than or equal
 * to the current av_log_level. By default, all logging messages are sent to
 * stderr. This behavior can be altered by setting a different logging callback
 * function.
 *如果级别小于或等于当前 av_log 级别,则将指定的消息发送到日志.
 *默认情况下,所有日志消息都发送到 stderr.注意是默认输出到 stderr.
 *可以通过设置不同的日志回调函数来更改此行为
 * @see av_log_set_callback
 *
 * @param avcl :A pointer to an arbitrary struct of which the first field is a  pointer to an
AVClass struct or NULL if general log.
 *指向任意结构的指针,其中第 1 个字段是指向 AVClass 结构的指针,如果是常规日志,则为 NULL
 * @param level :The importance level of the message expressed using a @ref
 *         lavu_log_constants "Logging Constant". 消息的重要性级别.
 * @param fmt The format string (printf-compatible) that specifies how
 *         subsequent arguments are converted to output.格式化字符串
 */
void av_log(void *avcl, int level, const char *fmt, ...) av_printf_format(3, 4);
```

这个函数的声明有以下两个地方比较特殊:

(1) 函数最后一个参数是"..."。在 C 语言中,在函数参数数量不确定的情况下使用"..."来代表参数。例如 printf()的函数原型定义,代码如下:

```
int printf (const char*, ...);
```

(2) 声明后面有一个 av_printf_format(3, 4),其作用是按照 printf()的格式检查 av_log()的格式。

av_log()函数每个字段的含义如下。

(1) avcl:指定一个包含 AVClass 的结构体。

(2) level:日志的级别。

(3) fmt:和 printf()一样,格式化的字符串。

由此可见,av_log()和 printf()的不同主要在于前面多了两个参数,其中第 1 个参数用于指定该 log 所属的结构体,例如 AVFormatContext、AVCodecContext 等。第 2 个参数用于指定 log 的级别,源代码中定义了几个级别,代码如下:

```
//chapter5/5.2.help.txt
#define AV_LOG_QUIET     -8
```

```
#define AV_LOG_PANIC     0
#define AV_LOG_FATAL     8
#define AV_LOG_ERROR    16
#define AV_LOG_WARNING  24
#define AV_LOG_INFO     32
#define AV_LOG_VERBOSE  40
#define AV_LOG_Debug    48
```

从定义中可知，随着严重程度的逐渐下降，一共包含如下级别：AV_LOG_PANIC、AV_LOG_FATAL、AV_LOG_ERROR、AV_LOG_WARNING、AV_LOG_INFO、AV_LOG_VERBOSE 和 AV_LOG_Debug。每个级别定义的数值代表了严重程度，数值越小代表越严重。默认的级别是 AV_LOG_INFO。此外，还有一个级别不输出任何信息，即 AV_LOG_QUIET。

当前项目中存在着一个“Log 级别”，所有严重程度高于该级别的 Log 信息都会输出。例如当前的 Log 级别是 AV_LOG_WARNING，则会输出 AV_LOG_PANIC、AV_LOG_FATAL、AV_LOG_ERROR 和 AV_LOG_WARNING 级别的信息，而不会输出 AV_LOG_INFO 级别的信息。可以通过 av_log_get_level()函数获得当前 Log 的级别，通过另一个函数 av_log_set_level()设置当前项目的 Log 级别。

av_log_get_level()和 av_log_set_level()的函数定义，代码如下：

```
//chapter5/5.2.help.txt
void av_log_set_level(int level)
{
    av_log_level = level;
}
int av_log_get_level(void)
{
    return av_log_level;
}
```

以上两个函数主要操作了一个静态全局变量 av_log_level，该变量用于存储当前系统 Log 的级别，代码如下：

```
static int av_log_level = AV_LOG_INFO;
```

av_log()函数的源代码位于 libavutil\log.c 文件中，代码如下：

```
//chapter5/5.2.help.txt
void av_log(void* avcl, int level, const char *fmt, ...)
{
    AVClass* avc = avcl ? *(AVClass **) avcl : NULL;
```

```
    va_list vl;
    va_start(vl, fmt);
    if (avc && avc->version >= (50 << 16 | 15 << 8 | 2) &&
        avc->log_level_offset_offset && level >= AV_LOG_FATAL)
        level += *(int *)(((uint8_t *)avcl) + avc->log_level_offset_offset);
    av_vlog(avcl, level, fmt, vl);
    va_end(vl);
}
```

下面看一下C语言函数中"..."参数的含义,与它相关的部分还涉及以下4部分,包括va_list变量、va_start()、va_arg()和va_end()。va_list是一个指向函数的参数的指针;va_start()用于初始化va_list变量;va_arg()用于返回可变参数;va_start()用于结束可变参数的获取。有关它们的用法可以参考一个小案例,代码如下:

```
//chapter5/QtFFmpeg5_Chapter5_001/valistdemo.cpp
#include <stdio.h>
#include <stdarg.h>
void fun2(int a,...){
    va_list pp;
    va_start(pp, a);
    do{
        printf("param = %d\n", a);
        a = va_arg(pp,int); //使 pp 指向下一个参数,将下一个参数的值赋给变量 a
    }
    while (a != 0); //直到参数为 0 时停止循环
}

int main(int argc, char * argv[]){
    fun2(20, 40, 60, 80, 0);
    return 0;
}
```

在av_log()的源代码中,在va_start()和va_end()之间,调用了另一个函数av_vlog(),它是一个FFmpeg的API函数,位于libavutil\log.h文件中,代码如下:

```
void av_vlog(void *avcl, int level, const char *fmt, va_list vl);
```

在项目中添加valistdemo.cpp文件,将上述代码复制进去,并将main.cpp文件中的main函数重命名为main3,然后运行程序,如图5-7所示。

5. FFmpeg的日志回调函数

在一些非命令行程序(如MFC程序和Android程序等)中,av_log()调用的fprintf(stderr,…)无法将日志内容显示出来。对于这种情况,FFmpeg提供了日志回调函数av_

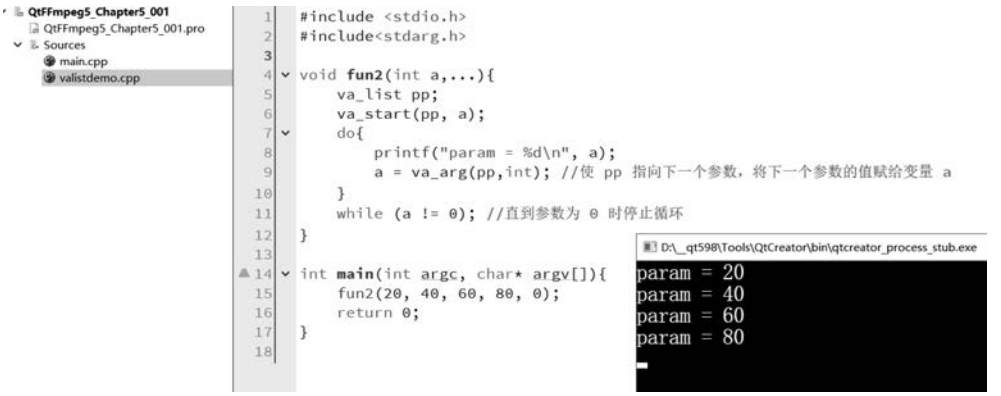

图 5-7　C 语言的"..."可变参数

log_set_callback()。该函数可以指定一个自定义的日志输出函数，以便将日志输出到指定的位置，函数声明位于 libavutil/log.h 文件中，代码如下：

```
//chapter5/QtFFmpeg5_Chapter5_001/main.cpp
/**
 * Set the logging callback:设置日志回调
 * 注意: 即使应用程序本身不使用线程,回调也必须是线程安全的,因为某些编解码器是多线程的
 * @note The callback must be thread safe, even if the application does not use     threads
itself as some codecs are multithreaded.
 *
 * @see av_log_default_callback
 *
 * @param callback A logging function with a compatible signature.
 */
void av_log_set_callback(void ( *callback)(void*, int, const char*, va_list));
```

下面的自定义函数 custom_output()将日志输出到当前路径下的 simplest_ffmpeg_log.txt 文本中，代码如下：

```
//chapter5/QtFFmpeg5_Chapter5_001/main.cpp
//需要注意:Qt 的当前目录"./"具体代表的路径
void custom_output(void* ptr, int level, const char* fmt,va_list vl){
    FILE *fp = fopen("./simplest_ffmpeg_log.txt","a+");
    if(fp){
        vfprintf(fp,fmt,vl);
        fflush(fp);
        fclose(fp);
    }
}
```

```
int main(int argc, char * argv[])
{
    QCoreApplication a(argc, argv);
    //设置日志回调
    av_log_set_callback(custom_output);  //注意:需要添加到 test_log()函数前
    test_log();

    return a.exec();
}
```

注意：av_log_set_callback()函数需要添加到 test_log()函数前。

运行程序，会发现控制台上没有输出任何日志信息，如图 5-8 所示，然后观察 Qt 的项目生成路径（笔者的目录名称是 build-QtFFmpeg5_Chapter5_001-Desktop_Qt_5_9_8_MSVC2015_64bit-Debug）下多了一个文件 simplest_ffmpeg_log.txt，打开文件后可以发现有相关的日志输出信息，如图 5-9 所示。

图 5-8　FFmpeg 的日志回调函数

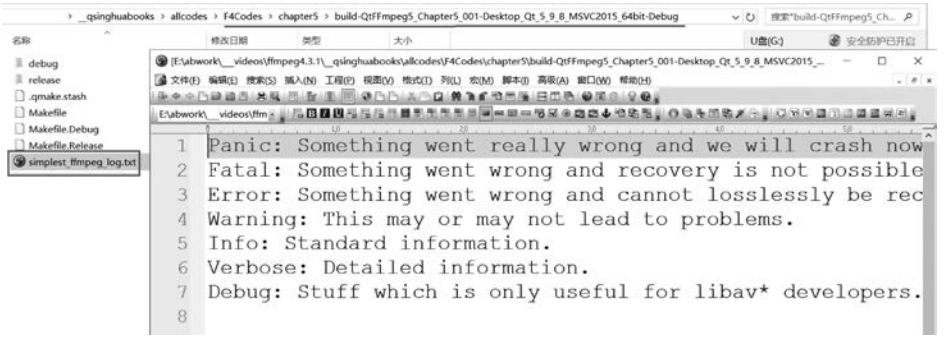

图 5-9　FFmpeg 的日志输出到文件中

5.3 AVParseUtil 应用案例及剖析

AVParseUtil 是 FFmpeg 的字符串解析工具，包括一系列 av_parse_xxx 开头的函数，例如分辨率解析函数 av_parse_video_size()可以从形如“1920x1080”的字符串中解析出图像宽为 1920、高为 1080；帧率函数 av_parse_video_rate()可以解析出帧率信息；时间解析函数则可以从形如“00:01:01”的字符串解析出时间的毫秒数。这些函数的声明位于 libavutil/parseutils.h 文件中，代码如下：

```
//chapter5/5.3.help.txt
//解析时间
int av_parse_time(int64_t *timeval, const char *timestr, int duration);
//解析颜色
int av_parse_color(uint8_t *rgba_color, const char *color_string, int slen,
                   void *log_ctx);
//解析帧率
int av_parse_video_rate(AVRational *rate, const char *str);
//解析分辨率
int av_parse_video_size(int *width_ptr, int *height_ptr, const char *str);
//解析 str 并将解析后的比率存储在 q 中
int av_parse_ratio(AVRational *q, const char *str, int max,
                   int log_offset, void *log_ctx);
```

1. AVParseUtil 案例

下面的案例显示了这几个函数的用法，代码如下：

```
//chapter5/QtFFmpeg5_Chapter5_001/main.cpp
//解析函数:avparseutils 相关的函数应用案例
void test_parseutil(){
    char input_strBuf[256] = {0};
    printf(" ========== Parse Video Size ==========\n");
    int output_w = 0;
    int output_h = 0;
    strcpy(input_strBuf, "1920x1080");

    av_parse_video_size(&output_w, &output_h, input_strBuf);
    printf("w:%4d | h:%4d\n",output_w,output_h);

    strcpy(input_strBuf,"vga");
    //strcpy(input_strBuf,"hd1080");
    //strcpy(input_strBuf,"ntsc");
    av_parse_video_size(&output_w,&output_h,input_strBuf);
```

```
    printf("w: %4d | h: %4d\n",output_w,output_h);


    printf(" ========== Parse Frame Rate ========== \n");
    AVRational output_rational = {0,0};
    strcpy(input_strBuf, "25/1");
    av_parse_video_rate(&output_rational,input_strBuf);
    printf("framerate: %d/ %d\n",output_rational.num,output_rational.den);


    strcpy(input_strBuf,"pal");
    av_parse_video_rate(&output_rational,input_strBuf);
    printf("framerate: %d/ %d\n",output_rational.num,output_rational.den);


    printf(" ============ Parse Time ============== \n");
    int64_t output_timeval;
    strcpy(input_strBuf, "00:01:03");
    av_parse_time(&output_timeval, input_strBuf,1);
    printf("microseconds: %lld\n", output_timeval);
    printf(" ====================================== \n");
}
```

注意：需要引入 AVParseUtil 的头文件：#include < libavutil/parseutils.h >。

将该函数复制到 QtFFmpeg5_Chapter5_001 项目的 main.cpp 文件中，运行程序，如图 5-10 所示。

图 5-10　FFmpeg 的字符串解析函数

可以看出，PAL 制的视频帧率是 25/1，VGA 格式的分辨率是 640×480 像素。

2. VGA 简介

1）VGA 相关的分辨率

与 VGA 相关的几个概念如下。

（1）QVGA（QuarterVGA）：标准 VGA 分辨率的 1/4 尺寸，亦即 320×240 像素，目前主要应用于手机及便携播放器上面；QQVGA 为 QVGA 的 1/4 屏，分辨率为 120×160 像素。

（2）VGA（Video Graphics Array）：分辨率为 640×480 像素，一些小的便携设备在使用这种屏幕。

（3）SVGA（Super Video Graphics Array）：属于 VGA 屏幕的替代品，最大支持 800×600 像素。

（4）XGA（Extended Graphics Array）：这是目前笔记本普遍采用的一种 LCD 屏幕，市面上将近 80%的笔记本采用了这种产品。它支持最大 1024×768 分辨率，屏幕大小从 10.4 英寸、12.1 英寸、13.3 英寸、14.1 英寸到 15.1 英寸都有。

（5）SXGA+（Super Extended Graphics Array）：作为 SXGA 的一种扩展 SXGA+是一种专门为笔记本设计的屏幕。其显示分辨率为 1400×1050 像素。由于笔记本 LCD 屏幕的水平与垂直点距不同于普通桌面 LCD，所以其显示的精度要比普通 17 英寸的桌面 LCD 高出不少。

（6）UVGA（Ultra Video Graphics Array）：这种屏幕应用在 15 英寸屏幕的笔记本上，支持最大 1600×1200 分辨率。由于对制造工艺要求较高所以价格也比较昂贵。目前只有少部分高端的移动工作站配备了这一类型的屏幕。

（7）WXGA（Wide Extended Graphics Array）：作为普通 XGA 屏幕的宽屏版本，WXGA 采用 16∶10 的横宽比例来扩大屏幕的尺寸。其最大显示分辨率为 1280×800 像素。由于其水平像素只有 800，所以除了一般 15 英寸的笔记本外，也有 12.1 英寸的笔记本采用了这种类型的屏幕。

（8）WXGA+（Wide Extended Graphics Array）：这是一种 WXGA 的扩展，其最大显示分辨率为 1280×854 像素。由于其横宽比例为 15∶10 而非标准宽屏的 16∶10，所以只有少部分屏幕尺寸在 15.2 英寸的笔记本采用这种产品。

（9）WSXGA+（Wide Super Extended Graphics Array）：其显示分辨率为 1680×1050 像素，除了大多数 15 英寸以上的宽屏笔记本以外，目前较为流行的大尺寸 LCD-TV 也采用了这种类型的产品。

（10）WUVGA（Wide Ultra Video Graphics Array）：和 4∶3 规格中的 UVGA 一样，WUVGA 屏幕是非常少见的，其显示分辨率可以达到 1920×1200 像素。

2）以 K 命名的分辨率

以 K 命名的分辨率比较模糊混乱，其中 2K 可以泛指长边有 2000 像素等级的分辨率。数字电影联盟（Digital Cinema Initiatives，DCI）将 2048×1080 分辨率定义为 2K，不过更多

厂商将2560×1440分辨率定义为2K。也有些厂商将稍大于2K的分辨率引申为2K+甚至是3K。根据定义4K、8K屏则是指长边分辨率分别达到4000和8000像素级别，宽达到2000和4000像素级别的分辨率，但现在手机屏幕做得越来越长，对K的定义也随之改变。全球首款4K HDR屏幕手机(Xperia XZ Premium)采用标准16∶9的4K屏幕，分辨率为3840×2160。后来的Xperia 1虽然也是4K屏，但比例改成了21∶9，所以分辨率相应地变成了3840×1644，总像素减少了大约23.89%。这几个分辨率如图5-11所示。

3）以字母简称命名方式的分辨率

该方式的命名系统基本没有规律可循，但是指向性比较明显，一看到名字就可以知道是什么类型的屏幕，不像2K、4K这么模糊。例如HD(High Definition)就是高清晰度的意思，指的是1280×720分辨率，现在大多数分辨率的简称就是在它的基础上命名的，例如FHD、QHD和UHD等，但有一个很特别，即qHD(quarter High Definition)，q的意思是四分之一，指FHD总像素的四分之一，所以长和宽都只有FHD分辨率的一半，即960×540，实际分辨率比QHD小得多，而SD(Standard Definition)的意思是分辨率不足以达到HD的标准，一般标准4∶3的比例是640×480(VGA)，16∶9是840×480(WVGA)。这几种分辨率如图5-12所示。

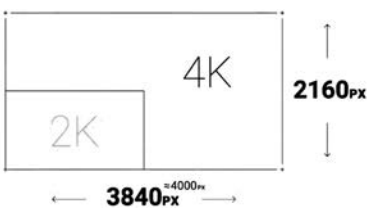

图5-11　2K和4K的分辨率

HD|高清晰度
High Definition 1280×720(720p)
FHD|全高清
Full High Definition 1920×1080(1080p)
QHD|四倍高清
Quad High Definition 2560×1440(2K)
UHD|超高清
Ultra High Definition 3840×2160或7680×4320(4K或8K)

图5-12　HD和UHD的分辨率

3. PAL与NTSC

电视信号的标准简称制式，可以简单地理解为用来实现电视图像或声音信号所采用的一种技术标准，即一个国家或地区播放节目时所采用的特定制度和技术标准。电视制式就是用来实现电视图像信号和伴音信号，或其他信号传输的方法和电视图像的显示格式，以及这种方法和电视图像显示格式所采用的技术标准。只有遵循相同的技术标准，才能实现电视机正常接收电视信号、播放电视节目。

PAL制又称为帕尔制，是英文Phase Alteration Line的缩写，意思是逐行倒相，属于同时制。PAL由德国人Walter Bruch在1967年提出，当时他为德律风根(Telefunken)工作。PAL有时亦被用来指625线、每秒25帧、隔行扫描、PAL色彩编码的电视制式。

NTSC是National Television Standards Committee(美国国家电视标准委员会)的英文缩写，是美国和日本的主流电视标准。NTSC每秒发送30个隔行扫描帧，分辨率为525线。

NTSC和PAL归根到底只是两种不同的视频格式，其主要差别在于NTSC每秒是60

场而 PAL 每秒是 50 场，由于现在的电视都采取隔行模式，所以 NTSC 每秒可以得到 30 个完整的视频帧，而 PAL 每秒可以得到 25 个完整的视频帧。

5.4 AVDictionary 应用案例及剖析

AVDictionary 是 FFmpeg 的键-值对存储工具，FFmpeg 经常使用 AVDictionary 设置/读取内部参数。经常被使用在 avformat_open_input()、avformat_init_output()、avcodec_open2()等函数中，用于设置 demux、muxer、codec 的 private options，即成员变量 priv_data。相关的数据结构和 API 定义在 libavutil/dict.h 文件中，代码如下：

```
//chapter5/5.4.help.txt
typedef struct AVDictionaryEntry {
    char *key;
    char *value;
} AVDictionaryEntry;
typedef struct AVDictionary AVDictionary;

AVDictionaryEntry *av_dict_get(const AVDictionary *m, const char *key,
                               const AVDictionaryEntry *prev, int flags);
int av_dict_count(const AVDictionary *m);
int av_dict_set(AVDictionary **pm, const char *key, const char *value, int flags);
void av_dict_free(AVDictionary **m);
```

注意：需要引入 AVDictionary 的头文件：#include <libavutil/dict.h>。

下面列举一个 AVDictionary 的应用案例，代码如下：

```
//chapter5/QtFFmpeg5_Chapter5_001/main.cpp
void test_avdictionary()
{
    AVDictionary *d = NULL;
    AVDictionaryEntry *t = NULL;

    av_dict_set(&d, "name", "zhangsan", 0);
    av_dict_set(&d, "gender", "man", 0);
    av_dict_set(&d, "website", "http://www.hellotongtong.com", 0);
    //av_strdup()
    char *k = av_strdup("location");
    char *v = av_strdup("Beijing-China");
    av_dict_set(&d, k, v, AV_DICT_DONT_STRDUP_KEY | AV_DICT_DONT_STRDUP_VAL);

    printf("====================================\n");
```

```
    int dict_cnt = av_dict_count(d); //该字典的总项数
    printf("dict_count: %d\n", dict_cnt);
    printf("dict_element:\n");
    //遍历该字典,通过 while 循环读取 av_dict_get
    while (t = av_dict_get(d, "", t, AV_DICT_IGNORE_SUFFIX)) {
        printf("key: %10s  |  value: %s\n",t->key,t->value);
    }

    t = av_dict_get(d, "website", t, AV_DICT_IGNORE_SUFFIX);
    printf("website is %s\n", t->value);
    printf("==========================================\n");
    av_dict_free(&d);
}
```

将该函数复制到 QtFFmpeg5_Chapter5_001 项目的 main.cpp 文件中,运行程序,如图 5-13 所示。

图 5-13　FFmepg 的 AVDictionary 操作

5.5　AVOption 应用案例及剖析

AVOption 是 FFmpeg 的选项设置工具,与 AVOption 最相关的选项设置函数就是 av_opt_set()。AVOption 的核心概念就是“根据字符串操作结构体的属性值”。有关 AVOption 函数的说明在 libavutil\opt.h 文件中,对于 libx264 提供的 codec 选项可以参见 libavcodec\libx264.c 文件。

1. 选项设置

对于 AVCodecContext 类型,可以使用成员方式访问后直接设置,也可以使用 av_opt_set_xxx 系列函数设置。例如下面代码块中的 #if 和 #else 之间代码的作用和 #else 和

#endif 之间代码的作用是一样的，代码如下：

```
//chapter5/QtFFmpeg5_Chapter5_001/main.cpp
#if TEST_OPT
    av_opt_set(pCodecCtx,"b","400000",0);            //bitrate
    //Another method
    //av_opt_set_int(pCodecCtx,"b",400000,0);        //bitrate
    av_opt_set(pCodecCtx,"time_base","1/25",0);      //time_base
    av_opt_set(pCodecCtx,"bf","5",0);                //max b frame
    av_opt_set(pCodecCtx,"g","25",0);                //gop
    av_opt_set(pCodecCtx,"qmin","10",0);             //qmin/qmax
    av_opt_set(pCodecCtx,"qmax","51",0);
#else
    pCodecCtx->time_base.num = 1;
    pCodecCtx->time_base.den = 25;
    pCodecCtx->max_b_frames = 5;
    pCodecCtx->bit_rate = 400000;
    pCodecCtx->gop_size = 25;
    pCodecCtx->qmin = 10;
    pCodecCtx->qmax = 51;
#endif
```

2. 选项获取

av_opt_get()可以将结构体的属性值以字符串的形式返回，代码如下：

```
//chapter5/QtFFmpeg5_Chapter5_001/main.cpp
//使用 char val_str[128]是无效的，必须用堆空间内存
char *val_str = (char *)av_malloc(128);

//preset: ultrafast, superfast, veryfast, faster, fast,
//medium, slow, slower, veryslow, placebo
av_opt_set(pCodecCtx->priv_data,"preset","slow",0);
//tune: film, animation, grain, stillimage, psnr,
//ssim, fastdecode, zerolatency
av_opt_set(pCodecCtx->priv_data,"tune","zerolatency",0);
//profile: baseline, main, high, high10, high422, high444
av_opt_set(pCodecCtx->priv_data,"profile","main",0);

//print
av_opt_get(pCodecCtx->priv_data,"preset",0,(uint8_t **)&val_str);
printf("preset val: %s\n",val_str);
av_opt_get(pCodecCtx->priv_data,"tune",0,(uint8_t **)&val_str);
printf("tune val: %s\n",val_str);
av_opt_get(pCodecCtx->priv_data,"profile",0,(uint8_t **)&val_str);
printf("profile val: %s\n",val_str);

av_free(val_str);
```

对于非字符串选项，如 int 类型，可以使用 av_opt_get_int()函数，其他类型与此类似，代码如下：

```
//chapter5/QtFFmpeg5_Chapter5_001/main.cpp
int64_t gop;
av_opt_get_int(pCodecCtx->priv_data,"g",0,&gop);
printf("gop val: %lld\n",gop);
```

3. 选项查找

可以通过 av_opt_find()函数获取结构体中任意选项的 AVOption 结构体，并打印 AVOption 的值，代码如下：

```
//chapter5/QtFFmpeg5_Chapter5_001/main.cpp
void print_opt(const AVOption *opt_test){//打印相关的选项内容
    printf("====================================\n");
    printf("Option Information:\n");
    printf("[name]%s\n",opt_test->name);
    printf("[help]%s\n",opt_test->help);
    printf("[offset]%d\n",opt_test->offset);

    switch(opt_test->type){
    case AV_OPT_TYPE_INT:{
        printf("[type]int\n[default]%d\n",opt_test->default_val.i64);
        break;
        }
    case AV_OPT_TYPE_INT64:{
        printf("[type]int64\n[default]%lld\n",opt_test->default_val.i64);
        break;
        }
    case AV_OPT_TYPE_FLOAT:{
        printf("[type]float\n[default]%f\n",opt_test->default_val.dbl);
        break;
        }
    case AV_OPT_TYPE_STRING:{
        printf("[type]string\n[default]%s\n",opt_test->default_val.str);
        break;
        }
    case AV_OPT_TYPE_RATIONAL:{

    printf("[type]rational\n[default]%d/%d\n",opt_test->default_val.q.num,opt_test->
default_val.q.den);
        break;
        }
    default:{
        printf("[type]others\n");
```

```
            break;
            }
        }

        printf("[max val] % f\n",opt_test -> max);
        printf("[min val] % f\n",opt_test -> min);

        if(opt_test -> flags&AV_OPT_FLAG_ENCODING_PARAM){
            printf("Encoding param.\n");
        }
        if(opt_test -> flags&AV_OPT_FLAG_DECODING_PARAM){
            printf("Decoding param.\n");
        }
        if(opt_test -> flags&AV_OPT_FLAG_AUDIO_PARAM){
            printf("Audio param.\n");
        }
        if(opt_test -> flags&AV_OPT_FLAG_VIDEO_PARAM){
            printf("Video param.\n");
        }
        if(opt_test -> unit!= NULL)
            printf("Unit belong to: % s\n",opt_test -> unit);

        printf(" ====================================\n");
    }

    # define TEST_OPT 1
    void test_avoptions()
    {
        AVCodecContext * pCodecCtx;
        //为一个"空编码器"申请上下文空间,仅仅用作测试
        pCodecCtx = avcodec_alloc_context3(NULL);
        if(pCodecCtx == NULL){
            printf("alloc failed\n");
            return;
        }

    # if TEST_OPT
        av_opt_set(pCodecCtx,"b","200000", 0);          //bitrate
        av_opt_set(pCodecCtx,"bf","3", 0);              //max b frame
        av_opt_set(pCodecCtx,"g","15", 0);              //gop
        av_opt_set(pCodecCtx,"qmin","10", 0);           //qmin/qmax
        av_opt_set(pCodecCtx,"qmax","31", 0);
    # else
        pCodecCtx -> time_base.num = 1;
        pCodecCtx -> time_base.den = 25;
        pCodecCtx -> max_b_frames = 5;
```

```
    pCodecCtx->bit_rate = 400000;
    pCodecCtx->gop_size = 25;
    pCodecCtx->qmin = 10;
    pCodecCtx->qmax = 51;
#endif

    const AVOption *opt = NULL;
    opt = av_opt_find(pCodecCtx, "b", NULL, 0, 0);
    print_opt(opt);

    opt = av_opt_find(pCodecCtx, "g", NULL, 0, 0);
    print_opt(opt);


    //释放这个"空编码器"的申请上下文空间,注意防止"野指针"
    if(pCodecCtx){
        avcodec_free_context(&pCodecCtx);
        pCodecCtx = NULL;
    }
}
```

将上述代码复制到 QtFFmpeg5_Chapter5_001 项目的 main.cpp 文件中，在 test_avoptions()函数中为一个“空编码器”申请上下文空间(AVCodecContext)，仅仅用作测试，然后使用 av_opt_set()函数设置各种编解码参数，再使用 av_opt_find()函数查询选项，最后记着释放这个“空编码器”的申请上下文空间，并要注意防止“野指针”。运行程序，如图 5-14 所示。

图 5-14　FFmepg 的 AVOption 操作