原 LAV Filter 源代码分析 4： LAV Video （2）

2013年10月27日 21:47:43 阅读数：6112

上一篇文章分析了LAV Filter 中的LAV Video的两个主要的类： CLAVVideo和CDecodeThread 。文章： LAV Filter 源代码分析 3： LAV Video （1）

在这里继续上篇文章的内容。文章中提到LAVVideo主要通过 CDecodeThread这个类进行解码线程的管理，其中有一个关键的管理函数： ThreadProc()，包含了对解码线程的各种操作。函数如下所示：

```cpp
//包含了对进程的各种操作
DWORD CDecodeThread::ThreadProc()
{
  HRESULT hr;
  DWORD cmd;

  BOOL bEOS = FALSE;
  BOOL bReinit = FALSE;

  SetThreadName(-1, "LAVVideo Decode Thread");

  HANDLE hWaitEvents[2] = { GetRequestHandle(), m_evInput };
  //不停转圈，永不休止
  while(1) {
    if (!bEOS && !bReinit) {
      // Wait for either an input sample, or an request
      WaitForMultipleObjects(2, hWaitEvents, FALSE, INFINITE);
    }
    //根据操作命令的不同
    if (CheckRequest(&cmd)) {
      switch (cmd) {
          //创建解码器
      case CMD_CREATE_DECODER:
        {
          CAutoLock lock(&m_ThreadCritSec);
          //创建
          hr = CreateDecoderInternal(m_ThreadCallContext.pmt, m_ThreadCallContext.codec);
          Reply(hr);

          m_ThreadCallContext.pmt = NULL;
        }
        break;
      case CMD_CLOSE_DECODER:
        {
            //关闭
          ClearQueues();
          SAFE_DELETE(m_pDecoder);
          Reply(S_OK);
        }
        break;
      case CMD_FLUSH:
        {
            //清楚
          ClearQueues();
          m_pDecoder->Flush();
          Reply(S_OK);
        }
        break;
      case CMD_EOS:
        {
          bEOS = TRUE;
          m_evEOSDone.Reset();
          Reply(S_OK);
        }
        break;
      case CMD_EXIT:
        {
            //退出
          Reply(S_OK);
          return 0;
        }
        break;
      case CMD_INIT_ALLOCATOR:
        {
          CAutoLock lock(&m_ThreadCritSec);
          hr = m_pDecoder->InitAllocator(m_ThreadCallContext.allocator);
          Reply(hr);

          m_ThreadCallContext.allocator = NULL;
        }
        break;
      case CMD_POST_CONNECT:
        {
          CAutoLock lock(&m_ThreadCritSec);
          hr = PostConnectInternal(m_ThreadCallContext.pin);
          Reply(hr);
```

```
77.
78.           m_ThreadCallContext.pin = NULL;
79.         }
80.         break;
81.       case CMD_REINIT:
82.         {
83.             //重启
84.           CMediaType &mt = m_pLAVVideo->GetInputMediaType();
85.           CreateDecoderInternal(&mt, m_Codec);
86.           m_TempSample[1] = m_NextSample;
87.           m_NextSample = m_FailedSample;
88.           m_FailedSample = NULL;
89.           bReinit = TRUE;
90.           m_evEOSDone.Reset();
91.           Reply(S_OK);
92.           m_bDecoderNeedsReInit = FALSE;
93.         }
94.         break;
95.       default:
96.         ASSERT(0);
97.       }
98.     }
99.
100.    if (m_bDecoderNeedsReInit) {
101.      m_evInput.Reset();
102.      continue;
103.    }
104.
105.    if (bReinit && !m_NextSample) {
106.      if (m_TempSample[0]) {
107.        m_NextSample = m_TempSample[0];
108.        m_TempSample[0] = NULL;
109.      } else if (m_TempSample[1]) {
110.        m_NextSample = m_TempSample[1];
111.        m_TempSample[1] = NULL;
112.      } else {
113.        bReinit = FALSE;
114.        m_evEOSDone.Set();
115.        m_evSample.Set();
116.        continue;
117.      }
118.    }
119.    //获得一份数据
120.    IMediaSample *pSample = GetSample();
121.    if (!pSample) {
122.      // Process the EOS now that the sample queue is empty
123.      if (bEOS) {
124.        bEOS = FALSE;
125.        m_pDecoder->EndOfStream();
126.        m_evEOSDone.Set();
127.        m_evSample.Set();
128.      }
129.      continue;
130.    }
131.    //解码
132.    DecodeInternal(pSample);
133.
134.    // Release the sample
135.    //释放
136.    SafeRelease(&pSample);
137.
138.    // Indicates we're done decoding this sample
139.    m_evDecodeDone.Set();
140.
141.    // Set the Sample Event to unblock any waiting threads
142.    m_evSample.Set();
143.  }
144.
145.  return 0;
146. }
```

该函数中，DecodeInternal(pSample)为实际上真正具有解码功能的函数，来看看它的源代码吧：

```cpp
STDMETHODIMP CDecodeThread::DecodeInternal(IMediaSample *pSample)
{
  HRESULT hr = S_OK;

  if (!m_pDecoder)
    return E_UNEXPECTED;
  //调用接口进行解码
  hr = m_pDecoder->Decode(pSample);

  // If a hardware decoder indicates a hard failure, we switch back to software
  // This is used to indicate incompatible media
  if (FAILED(hr) && m_bHWDecoder) {
    DbgLog((LOG_TRACE, 10, L"::Receive(): Hardware decoder indicates failure, switching back to software"));
    m_bHWDecoderFailed = TRUE;

    // Store the failed sample for re-try in a moment
    m_FailedSample = pSample;
    m_FailedSample->AddRef();

    // Schedule a re-init when the main thread goes there the next time
    m_bDecoderNeedsReInit = TRUE;

    // Make room in the sample buffer, to ensure the main thread can get in
    m_TempSample[0] = GetSample();
  }

  return S_OK;
}
```

该函数比较简短，从源代码中可以看出，调用了 m_pDecoder的Decode()方法 。其中 m_pDecoder为 ILAVDecoder类型的指针，而 ILAVDecoder是一个接口，并不包含实际的方法，如下所示 。注意，从程序注释中可以看出，每一个解码器都需要实现该接口规定的函数。

```cpp
/**
 * Decoder interface
 *
 * Every decoder needs to implement this to interface with the LAV Video core
 */
//接口
interface ILAVDecoder
{
  /**
   * Virtual destructor
   */
  virtual ~ILAVDecoder(void) {};

  /**
   * Initialize interfaces with the LAV Video core
   * This function should also be used to create all interfaces with external DLLs
   *
   * @param pSettings reference to the settings interface
   * @param pCallback reference to the callback interface
   * @return S_OK on success, error code if this decoder is lacking an external support dll
   */
  STDMETHOD(InitInterfaces)(ILAVVideoSettings *pSettings, ILAVVideoCallback *pCallback) PURE;

  /**
   * Check if the decoder is functional
   */
  STDMETHOD(Check)() PURE;

  /**
   * Initialize the codec to decode a stream specified by codec and pmt.
   *
   * @param codec Codec Id
   * @param pmt DirectShow Media Type
   * @return S_OK on success, an error code otherwise
   */
  STDMETHOD(InitDecoder)(AVCodecID codec, const CMediaType *pmt) PURE;

  /**
   * Decode a frame.
   *
   * @param pSample Media Sample to decode
   * @return S_OK if decoding was successfull, S_FALSE if no frame could be extracted, an error code if the decoder is not compatible
th the bitstream
   *
   * Note: When returning an actual error code, the filter will switch to the fallback software decoder! This should only be used for
tastrophic failures,
   * like trying to decode a unsupported format on a hardware decoder.
   */
  STDMETHOD(Decode)(IMediaSample *pSample) PURE;

  /**
   * Flush the decoder after a seek.
   * The decoder should discard any remaining data.
   *
```

```
    * @return unused
    */
   STDMETHOD(Flush)() PURE;

   /**
    * End of Stream
    * The decoder is asked to output any buffered frames for immediate delivery
    *
    * @return unused
    */
   STDMETHOD(EndOfStream)() PURE;

   /**
    * Query the decoder for the current pixel format
    * Mostly used by the media type creation logic before playback starts
    *
    * @return the pixel format used in the decoding process
    */
   STDMETHOD(GetPixelFormat)(LAVPixelFormat *pPix, int *pBpp) PURE;

   /**
    * Get the frame duration.
    *
    * This function is not mandatory, and if you cannot provide any specific duration, return 0.
    */
   STDMETHOD_(REFERENCE_TIME, GetFrameDuration)() PURE;

   /**
    * Query whether the format can potentially be interlaced.
    * This function should return false if the format can 100% not be interlaced, and true if it can be interlaced (but also progressi
.
    */
   STDMETHOD_(BOOL, IsInterlaced)() PURE;

   /**
    * Allows the decoder to handle an allocator.
    * Used by DXVA2 decoding
    */
   STDMETHOD(InitAllocator)(IMemAllocator **ppAlloc) PURE;

   /**
    * Function called after connection is established, with the pin as argument
    */
   STDMETHOD(PostConnect)(IPin *pPin) PURE;

   /**
    * Get the number of sample buffers optimal for this decoder
    */
   STDMETHOD_(long, GetBufferCount)() PURE;

   /**
    * Get the name of the decoder
    */
   STDMETHOD_(const WCHAR*, GetDecoderName)() PURE;

   /**
    * Get whether the decoder outputs thread-safe buffers
    */
   STDMETHOD(HasThreadSafeBuffers)() PURE;

   /**
    * Get whether the decoder should sync to the main thread
    */
   STDMETHOD(SyncToProcessThread)() PURE;
};
```

下面来看看封装libavcodec库的类吧，该类的定义位于decoders文件夹下，名为avcodec.h，如图所示：

该类名字叫CDecAvcodec，其继承了CDecBase。而CDecBase继承了ILAVDecoder。

```cpp
/* 雷霄骅
 * 中国传媒大学/数字电视技术
 * leixiaohua1020@126.com
 *
 */
/*
 *      Copyright (C) 2010-2013 Hendrik Leppkes
 *      http://www.1f0.de
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
```

```
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License along
 *  with this program; if not, write to the Free Software Foundation, Inc.,
 *  51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
 */

#pragma once

#include "DecBase.h"
#include "H264RandomAccess.h"

#include <map>

#define AVCODEC_MAX_THREADS 16

typedef struct {
  REFERENCE_TIME rtStart;
  REFERENCE_TIME rtStop;
} TimingCache;
//解码器（AVCODEC）（其实还有WMV9，CUVID等）
class CDecAvcodec : public CDecBase
{
public:
  CDecAvcodec(void);
  virtual ~CDecAvcodec(void);

  // ILAVDecoder
  STDMETHODIMP InitDecoder(AVCodecID codec, const CMediaType *pmt);
  //解码
  STDMETHODIMP Decode(const BYTE *buffer, int buflen, REFERENCE_TIME rtStart, REFERENCE_TIME rtStop, BOOL bSyncPoint, BOOL bDisconti
nuity);
  STDMETHODIMP Flush();
  STDMETHODIMP EndOfStream();
  STDMETHODIMP GetPixelFormat(LAVPixelFormat *pPix, int *pBpp);
  STDMETHODIMP_(REFERENCE_TIME) GetFrameDuration();
  STDMETHODIMP_(BOOL) IsInterlaced();
  STDMETHODIMP_(const WCHAR*) GetDecoderName() { return L"avcodec"; }
  STDMETHODIMP HasThreadSafeBuffers() { return S_OK; }
  STDMETHODIMP SyncToProcessThread() { return m_pAVCtx && m_pAVCtx->thread_count > 1 ? S_OK : S_FALSE; }

  // CDecBase
  STDMETHODIMP Init();

protected:
  virtual HRESULT AdditionaDecoderInit() { return S_FALSE; }
  virtual HRESULT PostDecode() { return S_FALSE; }
  virtual HRESULT HandleDXVA2Frame(LAVFrame *pFrame) { return S_FALSE; }
  //销毁解码器，各种Free
  STDMETHODIMP DestroyDecoder();

private:
  STDMETHODIMP ConvertPixFmt(AVFrame *pFrame, LAVFrame *pOutFrame);

protected:
  AVCodecContext       *m_pAVCtx;
  AVFrame              *m_pFrame;
  AVCodecID            m_nCodecId;
  BOOL                 m_bDXVA;

private:
  AVCodec              *m_pAVCodec;
  AVCodecParserContext *m_pParser;

  BYTE                 *m_pFFBuffer;
  BYTE                 *m_pFFBuffer2;
  int                  m_nFFBufferSize;
  int                  m_nFFBufferSize2;

  SwsContext           *m_pSwsContext;

  CH264RandomAccess    m_h264RandomAccess;

  BOOL                 m_bNoBufferConsumption;
  BOOL                 m_bHasPalette;

  // Timing settings
  BOOL                 m_bFFReordering;
  BOOL                 m_bCalculateStopTime;
  BOOL                 m_bRVDropBFrameTimings;
  BOOL                 m_bInputPadded;

  BOOL                 m_bBFrameDelay;
  TimingCache          m_tcBFrameDelay[2];
  int                  m_nBFramePos;

  TimingCache          m_tcThreadBuffer[AVCODEC_MAX_THREADS];
  int                  m_CurrentThread;
```

```
  REFERENCE_TIME       m_rtStartCache;
  BOOL                 m_bResumeAtKeyFrame;
  BOOL                 m_bWaitingForKeyFrame;
  int                  m_iInterlaced;
};
```

从 CDecAvcodec 类的定义可以看出，包含了各种功能的函数。首先我们看看初始化函数Init()

```cpp
// ILAVDecoder
STDMETHODIMP CDecAvcodec::Init()
{
#ifdef DEBUG
  DbgSetModuleLevel (LOG_CUSTOM1, DWORD_MAX); // FFMPEG messages use custom1
  av_log_set_callback(lavf_log_callback);
#else
  av_log_set_callback(NULL);
#endif
  //注册
  avcodec_register_all();
  return S_OK;
}
```

可见其调用了ffmpeg的API函数avcodec_register_all()进行了解码器的注册。

我们再来看看其解码函数Decode()：

```cpp
//解码
STDMETHODIMP CDecAvcodec::Decode(const BYTE *buffer, int buflen, REFERENCE_TIME rtStartIn, REFERENCE_TIME rtStopIn, BOOL bSyncPoint,
 BOOL bDiscontinuity)
{
  int     got_picture = 0;
  int     used_bytes  = 0;
  BOOL    bParserFrame = FALSE;
  BOOL    bFlush = (buffer == NULL);
  BOOL    bEndOfSequence = FALSE;
  //初始化Packet
  AVPacket avpkt;
  av_init_packet(&avpkt);

  if (m_pAVCtx->active_thread_type & FF_THREAD_FRAME) {
    if (!m_bFFReordering) {
      m_tcThreadBuffer[m_CurrentThread].rtStart = rtStartIn;
      m_tcThreadBuffer[m_CurrentThread].rtStop  = rtStopIn;
    }

    m_CurrentThread = (m_CurrentThread + 1) % m_pAVCtx->thread_count;
  } else if (m_bBFrameDelay) {
    m_tcBFrameDelay[m_nBFramePos].rtStart = rtStartIn;
    m_tcBFrameDelay[m_nBFramePos].rtStop = rtStopIn;
    m_nBFramePos = !m_nBFramePos;
  }

  uint8_t *pDataBuffer = NULL;
  if (!bFlush && buflen > 0) {
    if (!m_bInputPadded && (!(m_pAVCtx->active_thread_type & FF_THREAD_FRAME) || m_pParser)) {
      // Copy bitstream into temporary buffer to ensure overread protection
      // Verify buffer size
      if (buflen > m_nFFBufferSize) {
        m_nFFBufferSize = buflen;
        m_pFFBuffer = (BYTE *)av_realloc_f(m_pFFBuffer, m_nFFBufferSize + FF_INPUT_BUFFER_PADDING_SIZE, 1);
        if (!m_pFFBuffer) {
          m_nFFBufferSize = 0;
          return E_OUTOFMEMORY;
        }
      }

      memcpy(m_pFFBuffer, buffer, buflen);
      memset(m_pFFBuffer+buflen, 0, FF_INPUT_BUFFER_PADDING_SIZE);
      pDataBuffer = m_pFFBuffer;
    } else {
      pDataBuffer = (uint8_t *)buffer;
    }

    if (m_nCodecId == AV_CODEC_ID_H264) {
      BOOL bRecovered = m_h264RandomAccess.searchRecoveryPoint(pDataBuffer, buflen);
      if (!bRecovered) {
        return S_OK;
      }
    } else if (m_nCodecId == AV_CODEC_ID_VP8 && m_bWaitingForKeyFrame) {
      if (!(pDataBuffer[0] & 1)) {
        DbgLog((LOG_TRACE, 10, L"::Decode(): Found VP8 key-frame, resuming decoding"));
        m_bWaitingForKeyFrame = FALSE;
      } else {
        return S_OK;
```

```
        return S_OK;
      }
    }
  }

  while (buflen > 0 || bFlush) {
    REFERENCE_TIME rtStart = rtStartIn, rtStop = rtStopIn;

    if (!bFlush) {
        //设置AVPacket中的数据
      avpkt.data = pDataBuffer;
      avpkt.size = buflen;
      avpkt.pts = rtStartIn;
      if (rtStartIn != AV_NOPTS_VALUE && rtStopIn != AV_NOPTS_VALUE)
        avpkt.duration = (int)(rtStopIn - rtStartIn);
      else
        avpkt.duration = 0;
      avpkt.flags = AV_PKT_FLAG_KEY;

      if (m_bHasPalette) {
        m_bHasPalette = FALSE;
        uint32_t *pal = (uint32_t *)av_packet_new_side_data(&avpkt, AV_PKT_DATA_PALETTE, AVPALETTE_SIZE);
        int pal_size = FFMIN((1 << m_pAVCtx->bits_per_coded_sample) << 2, m_pAVCtx->extradata_size);
        uint8_t *pal_src = m_pAVCtx->extradata + m_pAVCtx->extradata_size - pal_size;

        for (int i = 0; i < pal_size/4; i++)
          pal[i] = 0xFF<<24 | AV_RL32(pal_src+4*i);
      }
    } else {
      avpkt.data = NULL;
      avpkt.size = 0;
    }

    // Parse the data if a parser is present
    // This is mandatory for MPEG-1/2
    // 不一定需要
    if (m_pParser) {
      BYTE *pOut = NULL;
      int pOut_size = 0;

      used_bytes = av_parser_parse2(m_pParser, m_pAVCtx, &pOut, &pOut_size, avpkt.data, avpkt.size, AV_NOPTS_VALUE, AV_NOPTS_VALUE, 0


      if (used_bytes == 0 && pOut_size == 0 && !bFlush) {
        DbgLog((LOG_TRACE, 50, L"::Decode() - could not process buffer, starving?"));
        break;
      }

      // Update start time cache
      // If more data was read then output, update the cache (incomplete frame)
      // If output is bigger, a frame was completed, update the actual rtStart with the cached value, and then overwrite the cache
      if (used_bytes > pOut_size) {
        if (rtStartIn != AV_NOPTS_VALUE)
          m_rtStartCache = rtStartIn;
      } else if (used_bytes == pOut_size || ((used_bytes + 9) == pOut_size)) {
        // Why +9 above?
        // Well, apparently there are some broken MKV muxers that like to mux the MPEG-
2 PICTURE_START_CODE block (which is 9 bytes) in the package with the previous frame
        // This would cause the frame timestamps to be delayed by one frame exactly, and cause timestamp reordering to go wrong.
        // So instead of failing on those samples, lets just assume that 9 bytes are that case exactly.
        m_rtStartCache = rtStartIn = AV_NOPTS_VALUE;
      } else if (pOut_size > used_bytes) {
        rtStart = m_rtStartCache;
        m_rtStartCache = rtStartIn;
        // The value was used once, don't use it for multiple frames, that ends up in weird timings
        rtStartIn = AV_NOPTS_VALUE;
      }

       bParserFrame = (pOut_size > 0);

      if (pOut_size > 0 || bFlush) {

        if (pOut && pOut_size > 0) {
          if (pOut_size > m_nFFBufferSize2) {
            m_nFFBufferSize2    = pOut_size;
            m_pFFBuffer2 = (BYTE *)av_realloc_f(m_pFFBuffer2, m_nFFBufferSize2 + FF_INPUT_BUFFER_PADDING_SIZE, 1);
            if (!m_pFFBuffer2) {
              m_nFFBufferSize2 = 0;
              return E_OUTOFMEMORY;
            }
          }
          memcpy(m_pFFBuffer2, pOut, pOut_size);
          memset(m_pFFBuffer2+pOut_size, 0, FF_INPUT_BUFFER_PADDING_SIZE);

          avpkt.data = m_pFFBuffer2;
          avpkt.size = pOut_size;
          avpkt.pts = rtStart;
          avpkt.duration = 0;

          const uint8_t *eosmarker = CheckForEndOfSequence(m_nCodecId, avpkt.data, avpkt.size, &m_MpegParserState);
          if (eosmarker) {
            bEndOfSequence = TRUE;
```

```
          }
        } else {
          avpkt.data = NULL;
          avpkt.size = 0;
        }
        //真正的解码
        int ret2 = avcodec_decode_video2 (m_pAVCtx, m_pFrame, &got_picture, &avpkt);
        if (ret2 < 0) {
          DbgLog((LOG_TRACE, 50, L"::Decode() - decoding failed despite successfull parsing"));
          got_picture = 0;
        }
      } else {
        got_picture = 0;
      }
    } else {
      used_bytes = avcodec_decode_video2 (m_pAVCtx, m_pFrame, &got_picture, &avpkt);
    }

    if (FAILED(PostDecode())) {
      av_frame_unref(m_pFrame);
      return E_FAIL;
    }

    // Decoding of this frame failed ... oh well!
    if (used_bytes < 0) {
      av_frame_unref(m_pFrame);
      return S_OK;
    }

    // When Frame Threading, we won't know how much data has been consumed, so it by default eats everything.
    // In addition, if no data got consumed, and no picture was extracted, the frame probably isn't all that useufl.
    // The MJPEB decoder is somewhat buggy and doesn't let us know how much data was consumed really...
    if ((!m_pParser && (m_pAVCtx->active_thread_type & FF_THREAD_FRAME || (!got_picture && used_bytes == 0))) || m_bNoBufferConsumption || bFlush) {
      buflen = 0;
    } else {
      buflen -= used_bytes;
      pDataBuffer += used_bytes;
    }

    // Judge frame usability
    // This determines if a frame is artifact free and can be delivered
    // For H264 this does some wicked magic hidden away in the H264RandomAccess class
    // MPEG-2 and VC-1 just wait for a keyframe..
    if (m_nCodecId == AV_CODEC_ID_H264 && (bParserFrame || !m_pParser || got_picture)) {
      m_h264RandomAccess.judgeFrameUsability(m_pFrame, &got_picture);
    } else if (m_bResumeAtKeyFrame) {
      if (m_bWaitingForKeyFrame && got_picture) {
        if (m_pFrame->key_frame) {
          DbgLog((LOG_TRACE, 50, L"::Decode() - Found Key-Frame, resuming decoding at %I64d", m_pFrame->pkt_pts));
          m_bWaitingForKeyFrame = FALSE;
        } else {
          got_picture = 0;
        }
      }
    }

    // Handle B-frame delay for frame threading codecs
    if ((m_pAVCtx->active_thread_type & FF_THREAD_FRAME) && m_bBFrameDelay) {
      m_tcBFrameDelay[m_nBFramePos] = m_tcThreadBuffer[m_CurrentThread];
      m_nBFramePos = !m_nBFramePos;
    }

    if (!got_picture || !m_pFrame->data[0]) {
      if (!avpkt.size)
        bFlush = FALSE; // End flushing, no more frames
      av_frame_unref(m_pFrame);
      continue;
    }

    //////////////////////////////////////////////////////////////////////////////////////////
    // Determine the proper timestamps for the frame, based on different possible flags.
    //////////////////////////////////////////////////////////////////////////////////////////
    if (m_bFFReordering) {
      rtStart = m_pFrame->pkt_pts;
      if (m_pFrame->pkt_duration)
        rtStop = m_pFrame->pkt_pts + m_pFrame->pkt_duration;
      else
        rtStop = AV_NOPTS_VALUE;
    } else if (m_bBFrameDelay && m_pAVCtx->has_b_frames) {
      rtStart = m_tcBFrameDelay[m_nBFramePos].rtStart;
      rtStop  = m_tcBFrameDelay[m_nBFramePos].rtStop;
    } else if (m_pAVCtx->active_thread_type & FF_THREAD_FRAME) {
      unsigned index = m_CurrentThread;
      rtStart = m_tcThreadBuffer[index].rtStart;
      rtStop  = m_tcThreadBuffer[index].rtStop;
    }

    if (m_bRVDropBFrameTimings && m_pFrame->pict_type == AV_PICTURE_TYPE_B) {
      rtStart = AV_NOPTS_VALUE;
    }
```

```
237.
238.     if (m_bCalculateStopTime)
239.       rtStop = AV_NOPTS_VALUE;
240.
241.     ////////////////////////////////////////////////////////////////////////////////////
242.     // All required values collected, deliver the frame
243.     ////////////////////////////////////////////////////////////////////////////////////
244.     LAVFrame *pOutFrame = NULL;
245.     AllocateFrame(&pOutFrame);
246.
247.     AVRational display_aspect_ratio;
248.     int64_t num = (int64_t)m_pFrame->sample_aspect_ratio.num * m_pFrame->width;
249.     int64_t den = (int64_t)m_pFrame->sample_aspect_ratio.den * m_pFrame->height;
250.     av_reduce(&display_aspect_ratio.num, &display_aspect_ratio.den, num, den, 1 << 30);
251.
252.     pOutFrame->width        = m_pFrame->width;
253.     pOutFrame->height       = m_pFrame->height;
254.     pOutFrame->aspect_ratio = display_aspect_ratio;
255.     pOutFrame->repeat       = m_pFrame->repeat_pict;
256.     pOutFrame->key_frame    = m_pFrame->key_frame;
257.     pOutFrame->frame_type   = av_get_picture_type_char(m_pFrame->pict_type);
258.     pOutFrame->ext_format   = GetDXVA2ExtendedFlags(m_pAVCtx, m_pFrame);
259.
260.     if (m_pFrame->interlaced_frame || (!m_pAVCtx->progressive_sequence && (m_nCodecId == AV_CODEC_ID_H264 || m_nCodecId == AV_CODEC_ID_MPEG2VIDEO)))
261.       m_iInterlaced = 1;
262.     else if (m_pAVCtx->progressive_sequence)
263.       m_iInterlaced = 0;
264.
265.     pOutFrame->interlaced   = (m_pFrame->interlaced_frame || (m_iInterlaced == 1 && m_pSettings->GetDeinterlacingMode() == DeintMode_Aggressive) || m_pSettings->GetDeinterlacingMode() == DeintMode_Force) && !(m_pSettings->GetDeinterlacingMode() == DeintMode_Disable);
266.
267.     LAVDeintFieldOrder fo   = m_pSettings->GetDeintFieldOrder();
268.     pOutFrame->tff          = (fo == DeintFieldOrder_Auto) ? m_pFrame->top_field_first : (fo == DeintFieldOrder_TopFieldFirst);
269.
270.     pOutFrame->rtStart      = rtStart;
271.     pOutFrame->rtStop       = rtStop;
272.
273.     PixelFormatMapping map  = getPixFmtMapping((AVPixelFormat)m_pFrame->format);
274.     pOutFrame->format       = map.lavpixfmt;
275.     pOutFrame->bpp          = map.bpp;
276.
277.     if (m_nCodecId == AV_CODEC_ID_MPEG2VIDEO || m_nCodecId == AV_CODEC_ID_MPEG1VIDEO)
278.       pOutFrame->avgFrameDuration = GetFrameDuration();
279.
280.     if (map.conversion) {
281.       ConvertPixFmt(m_pFrame, pOutFrame);
282.     } else {
283.       for (int i = 0; i < 4; i++) {
284.         pOutFrame->data[i]   = m_pFrame->data[i];
285.         pOutFrame->stride[i] = m_pFrame->linesize[i];
286.       }
287.
288.       pOutFrame->priv_data = av_frame_alloc();
289.       av_frame_ref((AVFrame *)pOutFrame->priv_data, m_pFrame);
290.       pOutFrame->destruct  = lav_avframe_free;
291.     }
292.
293.     if (bEndOfSequence)
294.       pOutFrame->flags |= LAV_FRAME_FLAG_END_OF_SEQUENCE;
295.
296.     if (pOutFrame->format == LAVPixFmt_DXVA2) {
297.       pOutFrame->data[0] = m_pFrame->data[4];
298.       HandleDXVA2Frame(pOutFrame);
299.     } else {
300.       Deliver(pOutFrame);
301.     }
302.
303.     if (bEndOfSequence) {
304.       bEndOfSequence = FALSE;
305.       if (pOutFrame->format == LAVPixFmt_DXVA2) {
306.         HandleDXVA2Frame(m_pCallback->GetFlushFrame());
307.       } else {
308.         Deliver(m_pCallback->GetFlushFrame());
309.       }
310.     }
311.
312.     if (bFlush) {
313.       m_CurrentThread = (m_CurrentThread + 1) % m_pAVCtx->thread_count;
314.     }
315.     av_frame_unref(m_pFrame);
316.   }
317.
318.   return S_OK;
319. }
```

终于，我们从这个函数中看到了很多的ffmpeg的API，结构体，以及变量。比如解码视频的函数avcodec_decode_video2()。

解码器初始化函数：InitDecoder()

```cpp
//创建解码器
STDMETHODIMP CDecAvcodec::InitDecoder(AVCodecID codec, const CMediaType *pmt)
{
    //要是有，先销毁
  DestroyDecoder();
  DbgLog((LOG_TRACE, 10, L"Initializing ffmpeg for codec %S", avcodec_get_name(codec)));

  BITMAPINFOHEADER *pBMI = NULL;
  videoFormatTypeHandler((const BYTE *)pmt->Format(), pmt->FormatType(), &pBMI);
  //查找解码器
  m_pAVCodec = avcodec_find_decoder(codec);
  CheckPointer(m_pAVCodec, VFW_E_UNSUPPORTED_VIDEO);
  //初始化上下文环境
  m_pAVCtx = avcodec_alloc_context3(m_pAVCodec);
  CheckPointer(m_pAVCtx, E_POINTER);

  if(codec == AV_CODEC_ID_MPEG1VIDEO || codec == AV_CODEC_ID_MPEG2VIDEO || pmt->subtype == FOURCCMap(MKTAG('H','2','6','4')) || pmt->subtype == FOURCCMap(MKTAG('h','2','6','4'))) {
    m_pParser = av_parser_init(codec);
  }

  DWORD dwDecFlags = m_pCallback->GetDecodeFlags();

  LONG biRealWidth = pBMI->biWidth, biRealHeight = pBMI->biHeight;
  if (pmt->formattype == FORMAT_VideoInfo || pmt->formattype == FORMAT_MPEGVideo) {
    VIDEOINFOHEADER *vih = (VIDEOINFOHEADER *)pmt->Format();
    if (vih->rcTarget.right != 0 && vih->rcTarget.bottom != 0) {
      biRealWidth  = vih->rcTarget.right;
      biRealHeight = vih->rcTarget.bottom;
    }
  } else if (pmt->formattype == FORMAT_VideoInfo2 || pmt->formattype == FORMAT_MPEG2Video) {
    VIDEOINFOHEADER2 *vih2 = (VIDEOINFOHEADER2 *)pmt->Format();
    if (vih2->rcTarget.right != 0 && vih2->rcTarget.bottom != 0) {
      biRealWidth  = vih2->rcTarget.right;
      biRealHeight = vih2->rcTarget.bottom;
    }
  }
  //各种赋值
  m_pAVCtx->codec_id              = codec;
  m_pAVCtx->codec_tag             = pBMI->biCompression;
  m_pAVCtx->coded_width           = pBMI->biWidth;
  m_pAVCtx->coded_height          = abs(pBMI->biHeight);
  m_pAVCtx->bits_per_coded_sample = pBMI->biBitCount;
  m_pAVCtx->error_concealment     = FF_EC_GUESS_MVS | FF_EC_DEBLOCK;
  m_pAVCtx->err_recognition       = AV_EF_CAREFUL;
  m_pAVCtx->workaround_bugs       = FF_BUG_AUTODETECT;
  m_pAVCtx->refcounted_frames     = 1;

  if (codec == AV_CODEC_ID_H264)
    m_pAVCtx->flags2             |= CODEC_FLAG2_SHOW_ALL;

  // Setup threading
  int thread_type = getThreadFlags(codec);
  if (thread_type) {
    // Thread Count. 0 = auto detect
    int thread_count = m_pSettings->GetNumThreads();
    if (thread_count == 0) {
      thread_count = av_cpu_count() * 3 / 2;
    }

    m_pAVCtx->thread_count = max(1, min(thread_count, AVCODEC_MAX_THREADS));
    m_pAVCtx->thread_type = thread_type;
  } else {
    m_pAVCtx->thread_count = 1;
  }

  if (dwDecFlags & LAV_VIDEO_DEC_FLAG_NO_MT) {
    m_pAVCtx->thread_count = 1;
  }
  //初始化AVFrame
  m_pFrame = av_frame_alloc();
  CheckPointer(m_pFrame, E_POINTER);

  m_h264RandomAccess.SetAVCNALSize(0);

  // Process Extradata
  //处理ExtraData
  BYTE *extra = NULL;
  size_t extralen = 0;
  getExtraData(*pmt, NULL, &extralen);

  BOOL bH264avc = FALSE;
  if (extralen > 0) {
    DbgLog((LOG_TRACE, 10, L"-> Processing extradata of %d bytes", extralen));
    // Reconstruct AVC1 extradata format
    if (pmt->formattype == FORMAT_MPEG2Video && (m_pAVCtx->codec_tag == MAKEFOURCC('a','v','c','1') || m_pAVCtx->codec_tag == MAKEFOURCC('A','V','C','1') || m_pAVCtx->codec_tag == MAKEFOURCC('C','C','V','1'))) {
```

```
      MPEG2VIDEOINFO *mp2vi = (MPEG2VIDEOINFO *)pmt->Format();
      extralen += 7;
      extra = (uint8_t *)av_mallocz(extralen + FF_INPUT_BUFFER_PADDING_SIZE);
      extra[0] = 1;
      extra[1] = (BYTE)mp2vi->dwProfile;
      extra[2] = 0;
      extra[3] = (BYTE)mp2vi->dwLevel;
      extra[4] = (BYTE)(mp2vi->dwFlags ? mp2vi->dwFlags : 4) - 1;

      // Actually copy the metadata into our new buffer
      size_t actual_len;
      getExtraData(*pmt, extra+6, &actual_len);

      // Count the number of SPS/PPS in them and set the length
      // We'll put them all into one block and add a second block with 0 elements afterwards
      // The parsing logic does not care what type they are, it just expects 2 blocks.
      BYTE *p = extra+6, *end = extra+6+actual_len;
      BOOL bSPS = FALSE, bPPS = FALSE;
      int count = 0;
      while (p+1 < end) {
        unsigned len = (((unsigned)p[0] << 8) | p[1]) + 2;
        if (p + len > end) {
          break;
        }
        if ((p[2] & 0x1F) == 7)
          bSPS = TRUE;
        if ((p[2] & 0x1F) == 8)
          bPPS = TRUE;
        count++;
        p += len;
      }
      extra[5] = count;
      extra[extralen-1] = 0;

      bH264avc = TRUE;
      m_h264RandomAccess.SetAVCNALSize(mp2vi->dwFlags);
    } else if (pmt->subtype == MEDIASUBTYPE_LAV_RAWVIDEO) {
      if (extralen < sizeof(m_pAVCtx->pix_fmt)) {
        DbgLog((LOG_TRACE, 10, L"-> LAV RAW Video extradata is missing.."));
      } else {
        extra = (uint8_t *)av_mallocz(extralen + FF_INPUT_BUFFER_PADDING_SIZE);
        getExtraData(*pmt, extra, NULL);
        m_pAVCtx->pix_fmt = *(AVPixelFormat *)extra;
        extralen -= sizeof(AVPixelFormat);
        memmove(extra, extra+sizeof(AVPixelFormat), extralen);
      }
    } else {
      // Just copy extradata for other formats
      extra = (uint8_t *)av_mallocz(extralen + FF_INPUT_BUFFER_PADDING_SIZE);
      getExtraData(*pmt, extra, NULL);
    }
    // Hack to discard invalid MP4 metadata with AnnexB style video
    if (codec == AV_CODEC_ID_H264 && !bH264avc && extra[0] == 1) {
      av_freep(&extra);
      extralen = 0;
    }
    m_pAVCtx->extradata = extra;
    m_pAVCtx->extradata_size = (int)extralen;
  } else {
    if (codec == AV_CODEC_ID_VP6 || codec == AV_CODEC_ID_VP6A || codec == AV_CODEC_ID_VP6F) {
      int cropH = pBMI->biWidth - biRealWidth;
      int cropV = pBMI->biHeight - biRealHeight;
      if (cropH >= 0 && cropH <= 0x0f && cropV >= 0 && cropV <= 0x0f) {
        m_pAVCtx->extradata = (uint8_t *)av_mallocz(1 + FF_INPUT_BUFFER_PADDING_SIZE);
        m_pAVCtx->extradata_size = 1;
        m_pAVCtx->extradata[0] = (cropH << 4) | cropV;
      }
    }
  }

  m_h264RandomAccess.flush(m_pAVCtx->thread_count);
  m_CurrentThread = 0;
  m_rtStartCache = AV_NOPTS_VALUE;

  LAVPinInfo lavPinInfo = {0};
  BOOL bLAVInfoValid = SUCCEEDED(m_pCallback->GetLAVPinInfo(lavPinInfo));

  m_bInputPadded = dwDecFlags & LAV_VIDEO_DEC_FLAG_LAVSPLITTER;

  // Setup codec-specific timing logic
  BOOL bVC1IsPTS = (codec == AV_CODEC_ID_VC1 && !(dwDecFlags & LAV_VIDEO_DEC_FLAG_VC1_DTS));

  // Use ffmpegs logic to reorder timestamps
  // This is required for H264 content (except AVI), and generally all codecs that use frame threading
  // VC-1 is also a special case. Its required for splitters that deliver PTS timestamps (see bVC1IsPTS above)
  m_bFFReordering        =  ( codec == AV_CODEC_ID_H264 && !(dwDecFlags & LAV_VIDEO_DEC_FLAG_H264_AVI))
                          || codec == AV_CODEC_ID_VP8
                          || codec == AV_CODEC_ID_VP3
                          || codec == AV_CODEC_ID_THEORA
                          || codec == AV_CODEC_ID_HUFFYUV
                          || codec == AV_CODEC_ID_FFVHUFF
```

```
                          || codec == AV_CODEC_ID_MPEG2VIDEO
                          || codec == AV_CODEC_ID_MPEG1VIDEO
                          || codec == AV_CODEC_ID_DIRAC
                          || codec == AV_CODEC_ID_UTVIDEO
                          || codec == AV_CODEC_ID_DNXHD
                          || codec == AV_CODEC_ID_JPEG2000
                          || (codec == AV_CODEC_ID_MPEG4 && pmt->formattype == FORMAT_MPEG2Video)
                          || bVC1IsPTS;

  // Stop time is unreliable, drop it and calculate it
  m_bCalculateStopTime   = (codec == AV_CODEC_ID_H264 || codec == AV_CODEC_ID_DIRAC || (codec == AV_CODEC_ID_MPEG4 && pmt->formattype == FORMAT_MPEG2Video) || bVC1IsPTS);

  // Real Video content has some odd timestamps
  // LAV Splitter does them allright with RV30/RV40, everything else screws them up
  m_bRVDropBFrameTimings = (codec == AV_CODEC_ID_RV10 || codec == AV_CODEC_ID_RV20 || ((codec == AV_CODEC_ID_RV30 || codec == AV_CODEC_ID_RV40) && (!(dwDecFlags & LAV_VIDEO_DEC_FLAG_LAVSPLITTER) || (bLAVInfoValid && (lavPinInfo.flags & LAV_STREAM_FLAG_RV34_MKV)))));

  // Enable B-Frame delay handling
  m_bBFrameDelay = !m_bFFReordering && !m_bRVDropBFrameTimings;

  m_bWaitingForKeyFrame = TRUE;
  m_bResumeAtKeyFrame =    codec == AV_CODEC_ID_MPEG2VIDEO
                        || codec == AV_CODEC_ID_VC1
                        || codec == AV_CODEC_ID_RV30
                        || codec == AV_CODEC_ID_RV40
                        || codec == AV_CODEC_ID_VP3
                        || codec == AV_CODEC_ID_THEORA
                        || codec == AV_CODEC_ID_MPEG4;

  m_bNoBufferConsumption =    codec == AV_CODEC_ID_MJPEGB
                           || codec == AV_CODEC_ID_LOCO
                           || codec == AV_CODEC_ID_JPEG2000;

  m_bHasPalette = m_pAVCtx->bits_per_coded_sample <= 8 && m_pAVCtx->extradata_size && !(dwDecFlags & LAV_VIDEO_DEC_FLAG_LAVSPLITTER)
                  &&  (codec == AV_CODEC_ID_MSVIDEO1
                    || codec == AV_CODEC_ID_MSRLE
                    || codec == AV_CODEC_ID_CINEPAK
                    || codec == AV_CODEC_ID_8BPS
                    || codec == AV_CODEC_ID_QPEG
                    || codec == AV_CODEC_ID_QTRLE
                    || codec == AV_CODEC_ID_TSCC);

  if (FAILED(AdditionaDecoderInit())) {
    return E_FAIL;
  }

  if (bLAVInfoValid) {
    // Setting has_b_frames to a proper value will ensure smoother decoding of H264
    if (lavPinInfo.has_b_frames >= 0) {
      DbgLog((LOG_TRACE, 10, L"-> Setting has_b_frames to %d", lavPinInfo.has_b_frames));
      m_pAVCtx->has_b_frames = lavPinInfo.has_b_frames;
    }
  }

  // Open the decoder
  //打开解码器
  int ret = avcodec_open2(m_pAVCtx, m_pAVCodec, NULL);
  if (ret >= 0) {
    DbgLog((LOG_TRACE, 10, L"-> ffmpeg codec opened successfully (ret: %d)", ret));
    m_nCodecId = codec;
  } else {
    DbgLog((LOG_TRACE, 10, L"-> ffmpeg codec failed to open (ret: %d)", ret));
    DestroyDecoder();
    return VFW_E_UNSUPPORTED_VIDEO;
  }

  m_iInterlaced = 0;
  for (int i = 0; i < countof(ff_interlace_capable); i++) {
    if (codec == ff_interlace_capable[i]) {
      m_iInterlaced = -1;
      break;
    }
  }

  // Detect chroma and interlaced
  if (m_pAVCtx->extradata && m_pAVCtx->extradata_size) {
    if (codec == AV_CODEC_ID_MPEG2VIDEO) {
      CMPEG2HeaderParser mpeg2Parser(extra, extralen);
      if (mpeg2Parser.hdr.valid) {
        if (mpeg2Parser.hdr.chroma < 2) {
          m_pAVCtx->pix_fmt = AV_PIX_FMT_YUV420P;
        } else if (mpeg2Parser.hdr.chroma == 2) {
          m_pAVCtx->pix_fmt = AV_PIX_FMT_YUV422P;
        }
        m_iInterlaced = mpeg2Parser.hdr.interlaced;
      }
    } else if (codec == AV_CODEC_ID_H264) {
      CH264SequenceParser h264parser;
      if (bH264avc)
```

```
        h264parser.ParseNALs(extra+6, extralen-6, 2);
      else
        h264parser.ParseNALs(extra, extralen, 0);
      if (h264parser.sps.valid)
        m_iInterlaced = h264parser.sps.interlaced;
    } else if (codec == AV_CODEC_ID_VC1) {
      CVC1HeaderParser vc1parser(extra, extralen);
      if (vc1parser.hdr.valid)
        m_iInterlaced = (vc1parser.hdr.interlaced ? -1 : 0);
    }
  }

  if (codec == AV_CODEC_ID_DNXHD)
    m_pAVCtx->pix_fmt = AV_PIX_FMT_YUV422P10;
  else if (codec == AV_CODEC_ID_FRAPS)
    m_pAVCtx->pix_fmt = AV_PIX_FMT_BGR24;

  if (bLAVInfoValid && codec != AV_CODEC_ID_FRAPS && m_pAVCtx->pix_fmt != AV_PIX_FMT_DXVA2_VLD)
    m_pAVCtx->pix_fmt = lavPinInfo.pix_fmt;

  DbgLog((LOG_TRACE, 10, L"AVCodec init successfull. interlaced: %d", m_iInterlaced));

  return S_OK;
}
```

解码器销毁函数：DestroyDecoder()

```cpp
//销毁解码器，各种Free
STDMETHODIMP CDecAvcodec::DestroyDecoder()
{
  DbgLog((LOG_TRACE, 10, L"Shutting down ffmpeg..."));
  m_pAVCodec    = NULL;

  if (m_pParser) {
    av_parser_close(m_pParser);
    m_pParser = NULL;
  }

  if (m_pAVCtx) {
    avcodec_close(m_pAVCtx);
    av_freep(&m_pAVCtx->extradata);
    av_freep(&m_pAVCtx);
  }
  av_frame_free(&m_pFrame);

  av_freep(&m_pFFBuffer);
  m_nFFBufferSize = 0;

  av_freep(&m_pFFBuffer2);
  m_nFFBufferSize2 = 0;

  if (m_pSwsContext) {
    sws_freeContext(m_pSwsContext);
    m_pSwsContext = NULL;
  }

  m_nCodecId = AV_CODEC_ID_NONE;

  return S_OK;
}
```


文章标签： LAVFilter 源代码 解码 ffmpeg directshow

个人分类： LAV Filter

所属专栏： 开源多媒体项目源代码分析