FLV解复用实战-FlvParser源码阅读

总体流程

了解了FLV的封装原理，我们通过一个简单的FLV解析器的例子来看看FLV到底是怎么样封装/解封装的

该FLV的地址：功能强大的 FLV 文件分析和解析器

这个程序能够做什么：

（1）不能直接商用，因为每个tag解析后都存放到内存；

（2）audio parse

（3）video parse

（4）script parse

main函数

流程：

1、读取输入文件（flv类型的视频文件）

2、调用Process进行处理

3、退出

```
1 int main(int argc, char *argv[])
```

```
{
    cout << "Hi, this is FLV parser test program!\n";

    if (argc != 3)
    {
        cout << "FlvParser.exe [input flv] [output flv]" << endl;
        return 0;
    }

    fstream fin;
    fin.open(argv[1], ios_base::in | ios_base::binary);
    if (!fin)
        return 0;

    Process(fin, argv[2]);

    fin.close();

    return 1;
}
```

处理函数Process

流程：
1、读取文件
2、开始解析
3、打印解析信息
4、把解析之后的数据输出到另外一个文件中

```
void Process(fstream &fin, const char *filename)
{
    CFlvParser parser;

    int nBufSize = 2000 * 1024;
    int nFlvPos = 0;
    uint8_t *pBuf, *pBak;
```

```
    pBuf = new uint8_t[nBufSize];
    pBak = new uint8_t[nBufSize];

    while (1)
    {
        int nReadNum = 0;
        int nUsedLen = 0;
        fin.read((char *)pBuf + nFlvPos, nBufSize - nFlvPos);
        nReadNum = fin.gcount();
        if (nReadNum == 0)
            break;
        nFlvPos += nReadNum;

        parser.Parse(pBuf, nFlvPos, nUsedLen);
        if (nFlvPos != nUsedLen)
        {
            memcpy(pBak, pBuf + nUsedLen, nFlvPos - nUsedLen);
            memcpy(pBuf, pBak, nFlvPos - nUsedLen);
        }
        nFlvPos -= nUsedLen;
    }
    parser.PrintInfo();
    parser.DumpH264("parser.264");
    parser.DumpAAC("parser.aac");

    //dump into flv
    parser.DumpFlv(filename);

    delete []pBak;
    delete []pBuf;
}
```

解析函数

流程：

1、解析flv的头部

2、解析flv的Tag

```
int CFlvParser::Parse(uint8_t *pBuf, int nBufSize, int &nUsedLen)
{
    int nOffset = 0;

    if (_pFlvHeader == 0)
    {
        CheckBuffer(9);
        _pFlvHeader = CreateFlvHeader(pBuf+nOffset);
        nOffset += _pFlvHeader->nHeadSize;
    }

    while (1)
    {
        CheckBuffer(15);
        int nPrevSize = ShowU32(pBuf + nOffset);
        nOffset += 4;

        Tag *pTag = CreateTag(pBuf + nOffset, nBufSize-nOffset);
        if (pTag == NULL)
        {
            nOffset -= 4;
            break;
        }
        nOffset += (11 + pTag->_header.nDataSize);

        _vpTag.push_back(pTag);
    }

    nUsedLen = nOffset;
    return 0;
}
```

FLV相关的数据结构

CFlvParser表示FLV解析器

FLV由FLV头部和FLV体构成，其中FLV体是由一系列的FLV tag构成的

```
class CFlvParser
{
public:
    CFlvParser();
    virtual ~CFlvParser();

    int Parse(uint8_t *pBuf, int nBufSize, int &nUsedLen);
    int PrintInfo();
    int DumpH264(const std::string &path);
    int DumpAAC(const std::string &path);
    int DumpFlv(const std::string &path);

private:
    typedef struct FlvHeader_s
    {
        int nVersion;
        int bHaveVideo;
        int bHaveAudio;
        int nHeadSize;

        uint8_t *pFlvHeader;
    } FlvHeader;
    struct TagHeader
    {
        int nType;
        int nDataSize;
        int nTimeStamp;
        int nTSEx;
        int nStreamID;

        uint32_t nTotalTS;

        TagHeader() : nType(0), nDataSize(0), nTimeStamp(0), nTSEx(0), nStreamID(0), nTotalTS(0) {}
        ~TagHeader() {}
    };

    class Tag
```

```
38      {
39      public:
40          Tag() : _pTagHeader(NULL), _pTagData(NULL), _pMedia(NULL),
    _nMediaLen(0) {}
41          void Init(TagHeader *pHeader, uint8_t *pBuf, int nLeftLen);
42
43          TagHeader _header;
44          uint8_t *_pTagHeader;
45          uint8_t *_pTagData;
46          uint8_t *_pMedia;
47          int _nMediaLen;
48      };
49
50      class CVideoTag : public Tag
51      {
52      public:
53          CVideoTag(TagHeader *pHeader, uint8_t *pBuf, int nLeftLen,
    CFlvParser *pParser);
54
55          int _nFrameType;
56          int _nCodecID;
57          int ParseH264Tag(CFlvParser *pParser);
58          int ParseH264Configuration(CFlvParser *pParser, uint8_t *pT
    agData);
59          int ParseNalu(CFlvParser *pParser, uint8_t *pTagData);
60      };
61
62      class CAudioTag : public Tag
63      {
64      public:
65          CAudioTag(TagHeader *pHeader, uint8_t *pBuf, int nLeftLen,
    CFlvParser *pParser);
66
67          int _nSoundFormat;
68          int _nSoundRate;
69          int _nSoundSize;
70          int _nSoundType;
71
72          // aac
73          static int _aacProfile;
```

```
74          static int _sampleRateIndex;
75          static int _channelConfig;
76
77          int ParseAACTag(CFlvParser *pParser);
78          int ParseAudioSpecificConfig(CFlvParser *pParser, uint8_t *
    pTagData);
79          int ParseRawAAC(CFlvParser *pParser, uint8_t *pTagData);
80      };
81
82      struct FlvStat
83      {
84          int nMetaNum, nVideoNum, nAudioNum;
85          int nMaxTimeStamp;
86          int nLengthSize;
87
88          FlvStat() : nMetaNum(0), nVideoNum(0), nAudioNum(0), nMaxTi
    meStamp(0), nLengthSize(0){}
89          ~FlvStat() {}
90      };
91
92
93
94      static uint32_t ShowU32(uint8_t *pBuf) { return (pBuf[0] << 24)
    | (pBuf[1] << 16) | (pBuf[2] << 8) | pBuf[3]; }
95      static uint32_t ShowU24(uint8_t *pBuf) { return (pBuf[0] << 16)
    | (pBuf[1] << 8) | (pBuf[2]); }
96      static uint32_t ShowU16(uint8_t *pBuf) { return (pBuf[0] << 8)
    | (pBuf[1]); }
97      static uint32_t ShowU8(uint8_t *pBuf) { return (pBuf[0]); }
98      static void WriteU64(uint64_t & x, int length, int value)
99      {
100         uint64_t mask = 0xFFFFFFFFFFFFFFFF >> (64 - length);
101         x = (x << length) | ((uint64_t)value & mask);
102     }
103     static uint32_t WriteU32(uint32_t n)
104     {
105         uint32_t nn = 0;
106         uint8_t *p = (uint8_t *)&n;
107         uint8_t *pp = (uint8_t *)&nn;
108         pp[0] = p[3];
```

```
        pp[1] = p[2];
        pp[2] = p[1];
        pp[3] = p[0];
        return nn;
    }

    friend class Tag;

private:

    FlvHeader *CreateFlvHeader(uint8_t *pBuf);
    int DestroyFlvHeader(FlvHeader *pHeader);
    Tag *CreateTag(uint8_t *pBuf, int nLeftLen);
    int DestroyTag(Tag *pTag);
    int Stat();
    int StatVideo(Tag *pTag);
    int IsUserDataTag(Tag *pTag);

private:

    FlvHeader* _pFlvHeader;
    vector<Tag *> _vpTag;
    FlvStat _sStat;
    CVideojj *_vjj;

    // H.264
    int _nNalUnitLength;

};
```

FlvHeader表示FLV的头部

```
// FLV头
    typedef struct FlvHeader_s
    {
        int nVersion; // 版本
        int bHaveVideo; // 是否包含视频
```

```
        int bHaveAudio; // 是否包含音频
        int nHeadSize;  // FLV头部长度
        /*
        ** 指向存放FLV头部的buffer
        ** 上面的三个成员指明了FLV头部的信息，是从FLV的头部中“翻译”得到的，
        ** 真实的FLV头部是一个二进制比特串，放在一个buffer中，由pFlvHeader成员指明
        */
        uint8_t *pFlvHeader;
    } FlvHeader;
```

标签

标签包括标签头部和标签体，根据类型的不同，标签体可以分成三种：script类型的标签，音频标签、视频标签

标签头部

```
// Tag头部
struct TagHeader
{
    int nType;      // 类型
    int nDataSize;  // 标签body的大小
    int nTimeStamp; // 时间戳
    int nTSEx;      // 时间戳的扩展字节
    int nStreamID;  // 流的ID，总是0

    uint32_t nTotalTS;  // 完整的时间戳nTimeStamp和nTSEx拼装

    TagHeader() : nType(0), nDataSize(0), nTimeStamp(0), nTSEx(0), nStreamID(0), nTotalTS(0) {}
    ~TagHeader() {}
};
```

标签数据

script类型的标签

```
class Tag
{
    public:
    Tag() : _pTagHeader(NULL), _pTagData(NULL), _pMedia(NULL), _nMed
iaLen(0) {}
    void Init(TagHeader *pHeader, uint8_t *pBuf, int nLeftLen);

    TagHeader _header;
    uint8_t *_pTagHeader;   // 指向标签头部
    uint8_t *_pTagData;     // 指向标签body
    uint8_t *_pMedia;       // 指向标签的元数据
    int _nMediaLen;         // 数据长度
};
```

音频标签

```
class CAudioTag : public Tag
{
public:
    CAudioTag(TagHeader *pHeader, uint8_t *pBuf, int nLeftLen, CFlvP
arser *pParser);

    int _nSoundFormat;  // 音频编码类型
    int _nSoundRate;    // 采样率
    int _nSoundSize;    // 精度
    int _nSoundType;    // 类型

    // aac
    static int _aacProfile;     // 对应AAC profile
    static int _sampleRateIndex;    // 采样率索引
    static int _channelConfig;      // 通道设置

    int ParseAACTag(CFlvParser *pParser);
    int ParseAudioSpecificConfig(CFlvParser *pParser, uint8_t *pTagD
```

```
ata);
    int ParseRawAAC(CFlvParser *pParser, uint8_t *pTagData);
};
```

视频标签

```
class CVideoTag : public Tag
{
public:
    CVideoTag(TagHeader *pHeader, uint8_t *pBuf, int nLeftLen, CFlvP
arser *pParser);

    int _nFrameType;    // 帧类型
    int _nCodecID;      // 视频编解码类型
    int ParseH264Tag(CFlvParser *pParser);
    int ParseH264Configuration(CFlvParser *pParser, uint8_t *pTagDat
a);
    int ParseNalu(CFlvParser *pParser, uint8_t *pTagData);
};
```

解析FLV头部

入口函数

```
int CFlvParser::Parse(uint8_t *pBuf, int nBufSize, int &nUsedLen)
{
    int nOffset = 0;

    if (_pFlvHeader == 0)
    {
        CheckBuffer(9);
        _pFlvHeader = CreateFlvHeader(pBuf+nOffset);
```

```
        nOffset += _pFlvHeader->nHeadSize;
    }

    while (1)
    {
        CheckBuffer(15);
        int nPrevSize = ShowU32(pBuf + nOffset);
        nOffset += 4;

        Tag *pTag = CreateTag(pBuf + nOffset, nBufSize-nOffset);
        if (pTag == NULL)
        {
            nOffset -= 4;
            break;
        }
        nOffset += (11 + pTag->_header.nDataSize);

        _vpTag.push_back(pTag);
    }

    nUsedLen = nOffset;
    return 0;
}
```

FLV头部解析函数

```
CFlvParser::FlvHeader *CFlvParser::CreateFlvHeader(uint8_t *pBuf)
{
    FlvHeader *pHeader = new FlvHeader;
    pHeader->nVersion = pBuf[3];        // 版本号
    pHeader->bHaveAudio = (pBuf[4] >> 2) & 0x01;    // 是否有音频
    pHeader->bHaveVideo = (pBuf[4] >> 0) & 0x01;    // 是否有视频
    pHeader->nHeadSize = ShowU32(pBuf + 5);         // 头部长度

    pHeader->pFlvHeader = new uint8_t[pHeader->nHeadSize];
```

```
    memcpy(pHeader->pFlvHeader, pBuf, pHeader->nHeadSize);

    return pHeader;
}
```

解析标签头部

标签的解析过程

1、CFlvParser::Parse调用CreateTag解析标签
2、CFlvParser::CreateTag首先解析标签头部
3、根据标签头部的类型字段，判断标签的类型
4、如果是视频标签，那么解析视频标签
5、如果是音频标签，那么解析音频标签
6、如果是其他的标签，那么调用Tag::Init进行解析

解析标签头部的函数

```
CFlvParser::Tag *CFlvParser::CreateTag(uint8_t *pBuf, int nLeftLen)
{
    // 开始解析标签头部
    TagHeader header;
    header.nType = ShowU8(pBuf+0);  // 类型
    header.nDataSize = ShowU24(pBuf + 1);   // 标签body的长度
    header.nTimeStamp = ShowU24(pBuf + 4);  // 时间戳
    header.nTSEx = ShowU8(pBuf + 7);        // 时间戳的扩展字段
    header.nStreamID = ShowU24(pBuf + 8);   // 流的id
    header.nTotalTS = (uint32_t)((header.nTSEx << 24)) + header.nTimeStamp;
    // 标签头部解析结束

    cout << "total TS : " << header.nTotalTS << endl;
    cout << "nLeftLen : " << nLeftLen << " , nDataSize : " << header.nDataSize << endl;
```

```
    if ((header.nDataSize + 11) > nLeftLen)
    {
        return NULL;
    }

    Tag *pTag;
    switch (header.nType) {
    case 0x09:  // 视频类型的Tag
        pTag = new CVideoTag(&header, pBuf, nLeftLen, this);
        break;
    case 0x08:  // 音频类型的Tag
        pTag = new CAudioTag(&header, pBuf, nLeftLen, this);
        break;
    default:    // script类型的Tag
        pTag = new Tag();
        pTag->Init(&header, pBuf, nLeftLen);
    }

    return pTag;
}
```

解析视频标签

入口函数CreateTag

流程如下：

1. 解析标签头部
2. 判断标签头部的类型
3. 根据标签头部的类型，解析不同的标签
4. 如果是视频类型的标签，那么就创建并解析视频标签

```
CFlvParser::Tag *CFlvParser::CreateTag(uint8_t *pBuf, int nLeftLen)
{
    // 开始解析标签头部
    TagHeader header;
```

```
    header.nType = ShowU8(pBuf+0);  // 类型
    header.nDataSize = ShowU24(pBuf + 1);   // 标签body的长度
    header.nTimeStamp = ShowU24(pBuf + 4);  // 时间戳
    header.nTSEx = ShowU8(pBuf + 7);        // 时间戳的扩展字段
    header.nStreamID = ShowU24(pBuf + 8);   // 流的id
    header.nTotalTS = (uint32_t)((header.nTSEx << 24)) + header.nTimeStamp;
    // 标签头部解析结束

    cout << "total TS : " << header.nTotalTS << endl;
    cout << "nLeftLen : " << nLeftLen << " , nDataSize : " << header.nDataSize << endl;
    if ((header.nDataSize + 11) > nLeftLen)
    {
        return NULL;
    }

    Tag *pTag;
    switch (header.nType) {
    case 0x09:  // 视频类型的Tag
        pTag = new CVideoTag(&header, pBuf, nLeftLen, this);
        break;
    case 0x08:  // 音频类型的Tag
        pTag = new CAudioTag(&header, pBuf, nLeftLen, this);
        break;
    default:    // script类型的Tag
        pTag = new Tag();
        pTag->Init(&header, pBuf, nLeftLen);
    }

    return pTag;
}
```

创建视频标签

流程如下：

1. 初始化

2. 解析帧类型
3. 解析视频编码类型
4. 解析视频标签

```
CFlvParser::CVideoTag::CVideoTag(TagHeader *pHeader, uint8_t *pBuf,
int nLeftLen, CFlvParser *pParser)
{
    // 初始化
    Init(pHeader, pBuf, nLeftLen);

    uint8_t *pd = _pTagData;
    _nFrameType = (pd[0] & 0xf0) >> 4;  // 帧类型
    _nCodecID = pd[0] & 0x0f;           // 视频编码类型
    // 开始解析
    if (_header.nType == 0x09 && _nCodecID == 7)
    {
        ParseH264Tag(pParser);
    }
}
```

解析视频标签

流程如下：

1. 解析数据包类型
2. 如果数据包是配置信息，那么就解析配置信息
3. 如果数据包是视频数据，那么就解析视频数据

```
int CFlvParser::CVideoTag::ParseH264Tag(CFlvParser *pParser)
{
    uint8_t *pd = _pTagData;
    /*
    ** 数据包的类型
    ** 视频数据被压缩之后被打包成数据包在网上传输
    ** 有两种类型的数据包：视频信息包（sps、pps等）和视频数据包（视频的压缩数据）
    */
    int nAVCPacketType = pd[1];
    int nCompositionTime = CFlvParser::ShowU24(pd + 2);

```

```
    // 如果是视频配置信息
    if (nAVCPacketType == 0)    // AVC sequence header
    {
        ParseH264Configuration(pParser, pd);
    }
    // 如果是视频数据
    else if (nAVCPacketType == 1) // AVC NALU
    {
        ParseNalu(pParser, pd);
    }
    else
    {

    }
    return 1;
}
```

解析视频配置信息

流程如下：

1. 解析配置信息的长度
2. 解析sps、pps的长度
3. 保存元数据，元数据即sps、pps等

```
int CFlvParser::CVideoTag::ParseH264Configuration(CFlvParser *pParser, uint8_t *pTagData)
{
    uint8_t *pd = pTagData;
    // 跨过 Tag Data的VIDEODATA(1字节) AVCVIDEOPACKET(AVCPacketType和CompositionTime 4字节)
    // 总共跨过5个字节

    // NalUnit长度表示占用的字节
    pParser->_nNalUnitLength = (pd[9] & 0x03) + 1;  // lengthSizeMinusOne

    int sps_size, pps_size;
    // sps（序列参数集）的长度
    sps_size = CFlvParser::ShowU16(pd + 11);        // sequenceParam
```

```
eterSetLength
    // pps（图像参数集）的长度
    pps_size = CFlvParser::ShowU16(pd + 11 + (2 + sps_size) + 1);
// pictureParameterSetLength

    // 元数据的长度
    _nMediaLen = 4 + sps_size + 4 + pps_size;   // 添加start code
    _pMedia = new uint8_t[_nMediaLen];
    // 保存元数据
    memcpy(_pMedia, &nH264StartCode, 4);
    memcpy(_pMedia + 4, pd + 11 + 2, sps_size);
    memcpy(_pMedia + 4 + sps_size, &nH264StartCode, 4);
    memcpy(_pMedia + 4 + sps_size + 4, pd + 11 + 2 + sps_size + 2 +
1, pps_size);

    return 1;
}
```

解析视频数据

流程如下：

1、如果一个Tag还没解析完成，那么执行下面步骤
2、计算NALU的长度
3、获取NALU的起始码
4、保存NALU的数据
5、调用自定义的处理函数对NALU数据进行处理

```
int CFlvParser::CVideoTag::ParseNalu(CFlvParser *pParser, uint8_t *p
TagData)
{
    uint8_t *pd = pTagData;
    int nOffset = 0;

    _pMedia = new uint8_t[_header.nDataSize+10];
    _nMediaLen = 0;
    // 跨过 Tag Data的VIDEODATA(1字节) AVCVIDEOPACKET(AVCPacketType和Co
mpositionTime 4字节)
```

```
    nOffset = 5; // 总共跨过5个字节
    while (1)
    {
        // 如果解析完了一个Tag，那么就跳出循环
        if (nOffset >= _header.nDataSize)
            break;
        // 计算NALU（视频数据被包装成NALU在网上传输）的长度，
        // 一个tag可能包含多个nalu，所以每个nalu前面有NalUnitLength字节表示每个nalu的长度
        int nNaluLen;
        switch (pParser->_nNalUnitLength)
        {
        case 4:
            nNaluLen = CFlvParser::ShowU32(pd + nOffset);
            break;
        case 3:
            nNaluLen = CFlvParser::ShowU24(pd + nOffset);
            break;
        case 2:
            nNaluLen = CFlvParser::ShowU16(pd + nOffset);
            break;
        default:
            nNaluLen = CFlvParser::ShowU8(pd + nOffset);
        }
        // 获取NALU的起始码
        memcpy(_pMedia + _nMediaLen, &nH264StartCode, 4);
        // 复制NALU的数据
        memcpy(_pMedia + _nMediaLen + 4, pd + nOffset + pParser->_nNalUnitLength, nNaluLen);
        // 解析NALU
        pParser->_vjj->Process(_pMedia+_nMediaLen, 4+nNaluLen, _header.nTotalTS);
        _nMediaLen += (4 + nNaluLen);
        nOffset += (pParser->_nNalUnitLength + nNaluLen);
    }

    return 1;
}
```

自定义的视频处理

把视频的NALU解析出来之后，可以根据自己的需要往视频中添加内容

```
// 用户可以根据自己的需要，对该函数进行修改或者扩展
// 下面这个函数的功能大致就是往视频中写入SEI信息
int CVideojj::Process(uint8_t *pNalu, int nNaluLen, int nTimeStamp)
{
    // 如果起始码后面的两个字节是0x05或者0x06，那么表示IDR图像或者SEI信息
    if (pNalu[4] != 0x06 || pNalu[5] != 0x05)
        return 0;
    uint8_t *p = pNalu + 4 + 2;
    while (*p++ == 0xff);
    const char *szVideojjUUID = "VideojjLeonUUID";
    char *pp = (char *)p;
    for (int i = 0; i < strlen(szVideojjUUID); i++)
    {
        if (pp[i] != szVideojjUUID[i])
            return 0;
    }

    VjjSEI sei;
    sei.nTimeStamp = nTimeStamp;
    sei.nLen = nNaluLen - (pp - (char *)pNalu) - 16 - 1;
    sei.szUD = new char[sei.nLen];
    memcpy(sei.szUD, pp + 16, sei.nLen);
    _vVjjSEI.push_back(sei);

    return 1;
}
```

解析音频标签

入口函数CreateTag

流程如下：

1、解析标签头部

2、判断标签头部的类型

3、根据标签头部的类型，解析不同的标签

4、如果是音频类型的标签，那么就创建并解析音频标签

```
CFlvParser::Tag *CFlvParser::CreateTag(uint8_t *pBuf, int nLeftLen)
{
    // 开始解析标签头部
    TagHeader header;
    header.nType = ShowU8(pBuf+0);  // 类型
    header.nDataSize = ShowU24(pBuf + 1);   // 标签body的长度
    header.nTimeStamp = ShowU24(pBuf + 4);  // 时间戳
    header.nTSEx = ShowU8(pBuf + 7);        // 时间戳的扩展字段
    header.nStreamID = ShowU24(pBuf + 8);   // 流的id
    header.nTotalTS = (uint32_t)((header.nTSEx << 24)) + header.nTimeStamp;
    // 标签头部解析结束

    cout << "total TS : " << header.nTotalTS << endl;
    cout << "nLeftLen : " << nLeftLen << " , nDataSize : " << header.nDataSize << endl;
    if ((header.nDataSize + 11) > nLeftLen)
    {
        return NULL;
    }

    Tag *pTag;
    switch (header.nType) {
    case 0x09:  // 视频类型的Tag
        pTag = new CVideoTag(&header, pBuf, nLeftLen, this);
        break;
    case 0x08:  // 音频类型的Tag
        pTag = new CAudioTag(&header, pBuf, nLeftLen, this);
        break;
    default:    // script类型的Tag
        pTag = new Tag();
```

```
        pTag->Init(&header, pBuf, nLeftLen);
    }

    return pTag;
}
```

创建音频标签

流程如下：

1、初始化
2、解析音频编码类型
3、解析采样率
4、解析精度和类型
5、解析音频标签

```
CFlvParser::CAudioTag::CAudioTag(TagHeader *pHeader, uint8_t *pBuf,
int nLeftLen, CFlvParser *pParser)
{
    Init(pHeader, pBuf, nLeftLen);

    uint8_t *pd = _pTagData;
    _nSoundFormat = (pd[0] & 0xf0) >> 4;     // 音频格式
    _nSoundRate = (pd[0] & 0x0c) >> 2;       // 采样率
    _nSoundSize = (pd[0] & 0x02) >> 1;       // 采样精度
    _nSoundType = (pd[0] & 0x01);            // 是否立体声
    if (_nSoundFormat == 10)                 // AAC
    {
        ParseAACTag(pParser);
    }
}
```

解析音频标签

流程如下：

1、获取数据包的类型

2、判断数据包的类型

3、如果数据包是音频配置信息，那么解析有音频配置信息

4、如果是原始音频数据，那么对原始音频数据进行处理

```
int CFlvParser::CAudioTag::ParseAACTag(CFlvParser *pParser)
{
    uint8_t *pd = _pTagData;

    // 数据包的类型：音频配置信息，音频数据
    int nAACPacketType = pd[1];

    // 如果是音频配置信息
    if (nAACPacketType == 0)    // AAC sequence header
    {
        // 解析配置信息
        ParseAudioSpecificConfig(pParser, pd); // 解析AudioSpecificConfig
    }
    // 如果是音频数据
    else if (nAACPacketType == 1)   // AAC RAW
    {
        // 解析音频数据
        ParseRawAAC(pParser, pd);
    }
    else
    {

    }

    return 1;
}
```

处理始音频配置

流程如下：
1、解析AAC的采样率
2、解析采样率索引
3、解析声道

```
int CFlvParser::CAudioTag::ParseAudioSpecificConfig(CFlvParser *pParser, uint8_t *pTagData)
{
    uint8_t *pd = _pTagData;

    _aacProfile = ((pd[2]&0xf8)>>3);    // 5bit AAC编码级别
    _sampleRateIndex = ((pd[2]&0x07)<<1) | (pd[3]>>7);  // 4bit 真正的采样率索引
    _channelConfig = (pd[3]>>3) & 0x0f;                 // 4bit 通道数量
    printf("----- AAC ------\n");
    printf("profile:%d\n", _aacProfile);
    printf("sample rate index:%d\n", _sampleRateIndex);
    printf("channel config:%d\n", _channelConfig);

    _pMedia = NULL;
    _nMediaLen = 0;

    return 1;
}
```

处理原始音频数据

主要的功能是为原始的音频数据添加元数据，可以根据自己的需要进行改写

```
int CFlvParser::CAudioTag::ParseRawAAC(CFlvParser *pParser, uint8_t *pTagData)
{
    uint64_t bits = 0;  // 占用8字节
```

```
    // 数据长度
    int dataSize = _header.nDataSize - 2;   // 减去两字节的 audio tag data信息部分

    // 制作元数据
    WriteU64(bits, 12, 0xFFF);
    WriteU64(bits, 1, 0);
    WriteU64(bits, 2, 0);
    WriteU64(bits, 1, 1);
    WriteU64(bits, 2, _aacProfile - 1);
    WriteU64(bits, 4, _sampleRateIndex);
    WriteU64(bits, 1, 0);
    WriteU64(bits, 3, _channelConfig);
    WriteU64(bits, 1, 0);
    WriteU64(bits, 1, 0);
    WriteU64(bits, 1, 0);
    WriteU64(bits, 1, 0);
    WriteU64(bits, 13, 7 + dataSize);
    WriteU64(bits, 11, 0x7FF);
    WriteU64(bits, 2, 0);
    // WriteU64执行为上述的操作，最高的8bit还没有被移位到，实际是使用7个字节
    _nMediaLen = 7 + dataSize;
    _pMedia = new uint8_t[_nMediaLen];
    uint8_t p64[8];
    p64[0] = (uint8_t)(bits >> 56); // 是bits的最高8bit，实际为0
    p64[1] = (uint8_t)(bits >> 48); // 才是ADTS起始头 0xfff的高8bit
    p64[2] = (uint8_t)(bits >> 40);
    p64[3] = (uint8_t)(bits >> 32);
    p64[4] = (uint8_t)(bits >> 24);
    p64[5] = (uint8_t)(bits >> 16);
    p64[6] = (uint8_t)(bits >> 8);
    p64[7] = (uint8_t)(bits);

    memcpy(_pMedia, p64+1, 7);  // ADTS header, p64+1是从ADTS起始头开始
    memcpy(_pMedia + 7, pTagData + 2, dataSize); // AAC body

    return 1;
}
```

解析其他标签

入口函数CreateTag

流程如下：
1、解析标签头部
2、判断标签头部的类型
3、根据标签头部的类型，解析不同的标签
4、如果是其他类型的标签，那么就创建并解析其他类型标签

```
CFlvParser::Tag *CFlvParser::CreateTag(uint8_t *pBuf, int nLeftLen)
{
    // 开始解析标签头部
    TagHeader header;
    header.nType = ShowU8(pBuf+0);  // 类型
    header.nDataSize = ShowU24(pBuf + 1);   // 标签body的长度
    header.nTimeStamp = ShowU24(pBuf + 4);  // 时间戳
    header.nTSEx = ShowU8(pBuf + 7);        // 时间戳的扩展字段
    header.nStreamID = ShowU24(pBuf + 8);   // 流的id
    header.nTotalTS = (uint32_t)((header.nTSEx << 24)) + header.nTimeStamp;
    // 标签头部解析结束

    cout << "total TS : " << header.nTotalTS << endl;
    cout << "nLeftLen : " << nLeftLen << " , nDataSize : " << header.nDataSize << endl;
    if ((header.nDataSize + 11) > nLeftLen)
    {
        return NULL;
    }

    Tag *pTag;
    switch (header.nType) {
    case 0x09:  // 视频类型的Tag
        pTag = new CVideoTag(&header, pBuf, nLeftLen, this);
        break;
```

```
    case 0x08:  // 音频类型的Tag
        pTag = new CAudioTag(&header, pBuf, nLeftLen, this);
        break;
    default:    // script类型的Tag
        pTag = new Tag();
        pTag->Init(&header, pBuf, nLeftLen);
    }

    return pTag;
}
```

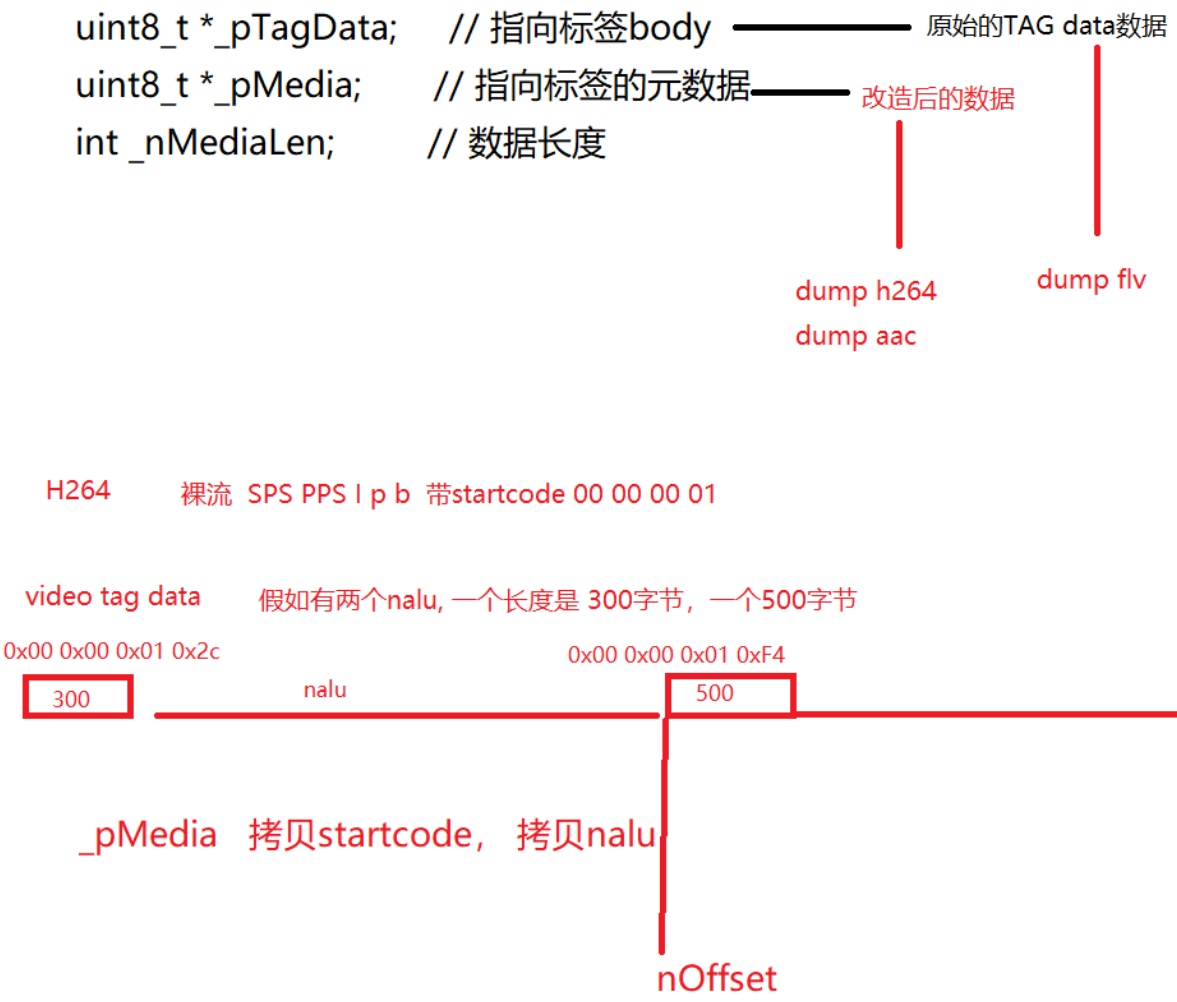
解析普通标签

没有太大的功能，就是数据的复制

```
void CFlvParser::Tag::Init(TagHeader *pHeader, uint8_t *pBuf, int nLeftLen)
{
    memcpy(&_header, pHeader, sizeof(TagHeader));
    // 复制标签头部信息
    _pTagHeader = new uint8_t[11];
    memcpy(_pTagHeader, pBuf, 11);      // 头部
    // 复制标签body
    _pTagData = new uint8_t[_header.nDataSize];
    memcpy(_pTagData, pBuf + 11, _header.nDataSize);
}
```

附录

上课画图



AMF格式

AMF数据第一个byte为此数据的类型，类型有：

Number	0×00	double类型
Boolean	0×01	bool类型
String	0×02	string类型
Object	0×03	object类型
MovieClip	0×04	Not available in Remoting
Null	0×05	null类型，空
Undefined	0×06	
Reference	0×07	

MixedArray	0×08	
EndOfObject	0×09	See Object ， 表示object结束
Array	0x0a	
Date	0x0b	
LongString	0x0c	
Unsupported	0x0d	
Recordset	0x0e	Remoting, server–to–client only
XML	0x0f	
TypedObject (Class instance)	0×10	
AMF3 data	0×11	Sent by Flash player 9+