原 RTMPdump（libRTMP） 源代码分析 3： AMF编码

2013年10月22日 21:18:47　阅读数：12199

==========

RTMPdump(libRTMP) 源代码分析系列文章：

RTMPdump 源代码分析 1： main()函数

RTMPDump (libRTMP) 源代码分析2：解析RTMP地址——RTMP_ParseURL()

RTMPdump (libRTMP) 源代码分析3： AMF编码

RTMPdump (libRTMP) 源代码分析4： 连接第一步——握手 (HandShake)

RTMPdump (libRTMP) 源代码分析5： 建立一个流媒体连接 (NetConnection部分)

RTMPdump (libRTMP) 源代码分析6： 建立一个流媒体连接 (NetStream部分 1)

RTMPdump (libRTMP) 源代码分析7： 建立一个流媒体连接 (NetStream部分 2)

RTMPdump (libRTMP) 源代码分析8： 发送消息 (Message)

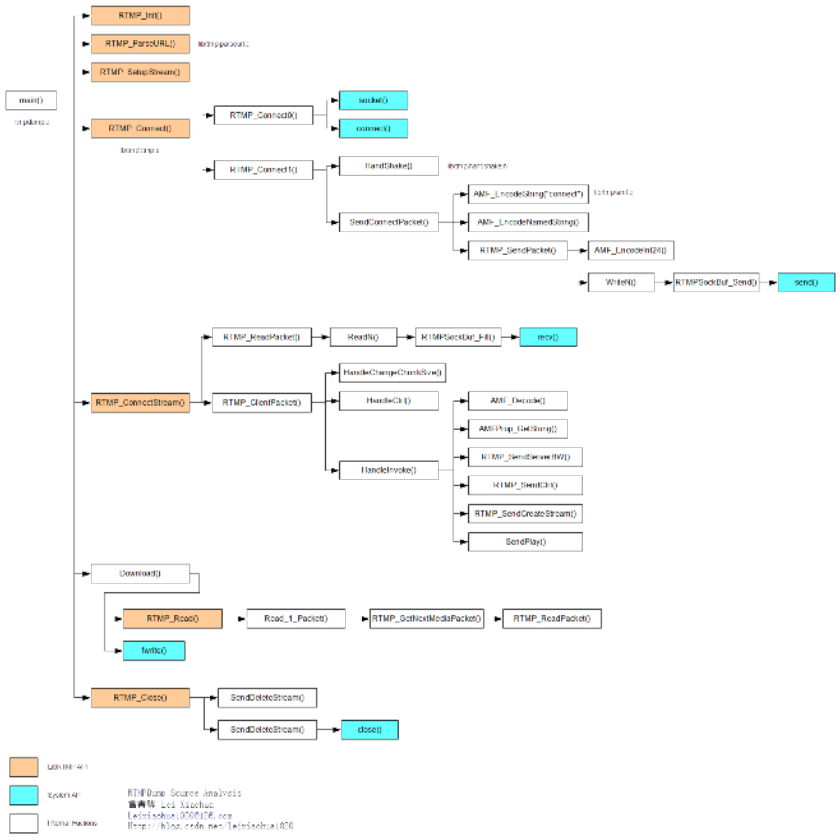
RTMPdump (libRTMP) 源代码分析9： 接收消息 (Message) (接收视音频数据)

RTMPdump (libRTMP) 源代码分析10： 处理各种消息 (Message)

==========

函数调用结构图

RTMPDump (libRTMP)的整体的函数调用结构图如下图所示。



单击查看大图

详细分析

之前分析了RTMPDump（libRTMP）解析RTMP的URL的源代码，在这里简单分析一下其AMF编码方面的源码。

AMF编码广泛用于Adobe公司的Flash以及Flex系统中。由于RTMP协议也是Adobe公司的，所以它也使用AMF进行通信。具体AMF是怎么使用的在这里就不做详细讨论了。RTMPDump如果想实现RTMP协议的流媒体的下载保存，就必须可以编码和解码AMF格式的数据。

amf.c是RTMPDump解析RTMP协议的函数存放的地方，在这里贴上其源代码。先不做详细解释了，以后有机会再补充。

```cpp
#include "stdafx.h"
/*  本文件主要包含了对AMF对象的操作
 *------------------------------------
 *AMF数据类型：
 *Type      Byte code
 *Number    0x00
 *Boolean   0x01
 *String    0x02
 *Object    0x03
 *MovieClip 0x04
 *Null      0x05
 *Undefined 0x06
 *Reference 0x07
 *MixedArray    0x08
 *EndOfObject   0x09
 *Array         0x0a
 *Date          0x0b
 *LongString    0x0c
 *Unsupported   0x0d
 *Recordset     0x0e
 *XML           0x0f
 *TypedObject (Class instance)  0x10
 *AMF3 data 0x11
 *------------------------------------
 *应用举例：
 *0.Number这里指的是double类型，数据用8字节表示，比如十六进制00 40 10 00 00 00 00 00就表示的是一个double数4.0
 *1.Boolean对应的是.net中的bool类型，数据使用1字节表示，和C语言差不多，使用00表示false，使用01表示true。比如十六进制01 01就表示true。
 *2.String相当于.net中的string类型，String所占用的空间有1个类型标识字节和2个表示字符串UTF8长度的字节加上字符串UTF8格式的内容组成。
 *  比如十六进制03 00 08 73 68 61 6E 67 67 75 61表示的就是字符串，该字符串长8字节，字符串内容为73 68 61 6E 67 67 75 61，对应的就是“shanggua”。
 *3.Object在对应的就是Hashtable，内容由UTF8字符串作为Key，其他AMF类型作为Value，该对象由3个字节：00 00 09来表示结束。
 *5.Null就是空对象，该对象只占用一个字节，那就是Null对象标识0x05。
 *6.Undefined 也是只占用一个字节0x06。
 *8.MixedArray相当于Hashtable，与3不同的是该对象定义了Hashtable的大小。
 */



#include <string.h>
#include <assert.h>
#include <stdlib.h>

#include "rtmp_sys.h"
#include "amf.h"
#include "log.h"
#include "bytes.h"

static const AMFObjectProperty AMFProp_Invalid = { {0, 0}, AMF_INVALID };
static const AVal AV_empty = { 0, 0 };

//大端Big-Endian
//低地址存放最高有效位（MSB），既高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。
//符合人脑逻辑，与计算机逻辑不同
//网络字节序 Network Order:TCP/IP各层协议将字节序定义为Big-Endian，因此TCP/IP协议中使
//用的字节序通常称之为网络字节序。
//主机序 Host Orader:它遵循Little-Endian规则。所以当两台主机之间要通过TCP/IP协议进行通
//信的时候就需要调用相应的函数进行主机序（Little-Endian）和网络序（Big-Endian）的转换。


/*AMF数据采用 Big-Endian（大端模式），主机采用Little-Endian（小端模式） */

unsigned short
AMF_DecodeInt16(const char *data)
{
  unsigned char *c = (unsigned char *) data;
  unsigned short val;
  val = (c[0] << 8) | c[1];//转换
  return val;
}

unsigned int
AMF_DecodeInt24(const char *data)
{
  unsigned char *c = (unsigned char *) data;
  unsigned int val;
  val = (c[0] << 16) | (c[1] << 8) | c[2];
  return val;
}

unsigned int
AMF_DecodeInt32(const char *data)
{
  unsigned char *c = (unsigned char *)data;
  unsigned int val;
```

```
  val = (c[0] << 24) | (c[1] << 16) | (c[2] << 8) | c[3];
  return val;
}

void
AMF_DecodeString(const char *data, AVal *bv)
{
  bv->av_len = AMF_DecodeInt16(data);
  bv->av_val = (bv->av_len > 0) ? (char *)data + 2 : NULL;
}

void
AMF_DecodeLongString(const char *data, AVal *bv)
{
  bv->av_len = AMF_DecodeInt32(data);
  bv->av_val = (bv->av_len > 0) ? (char *)data + 4 : NULL;
}

double
AMF_DecodeNumber(const char *data)
{
  double dVal;
#if __FLOAT_WORD_ORDER == __BYTE_ORDER
#if __BYTE_ORDER == __BIG_ENDIAN
  memcpy(&dVal, data, 8);
#elif __BYTE_ORDER == __LITTLE_ENDIAN
  unsigned char *ci, *co;
  ci = (unsigned char *)data;
  co = (unsigned char *)&dVal;
  co[0] = ci[7];
  co[1] = ci[6];
  co[2] = ci[5];
  co[3] = ci[4];
  co[4] = ci[3];
  co[5] = ci[2];
  co[6] = ci[1];
  co[7] = ci[0];
#endif
#else
#if __BYTE_ORDER == __LITTLE_ENDIAN /* __FLOAT_WORD_ORER == __BIG_ENDIAN */
  unsigned char *ci, *co;
  ci = (unsigned char *)data;
  co = (unsigned char *)&dVal;
  co[0] = ci[3];
  co[1] = ci[2];
  co[2] = ci[1];
  co[3] = ci[0];
  co[4] = ci[7];
  co[5] = ci[6];
  co[6] = ci[5];
  co[7] = ci[4];
#else /* __BYTE_ORDER == __BIG_ENDIAN && __FLOAT_WORD_ORER == __LITTLE_ENDIAN */
  unsigned char *ci, *co;
  ci = (unsigned char *)data;
  co = (unsigned char *)&dVal;
  co[0] = ci[4];
  co[1] = ci[5];
  co[2] = ci[6];
  co[3] = ci[7];
  co[4] = ci[0];
  co[5] = ci[1];
  co[6] = ci[2];
  co[7] = ci[3];
#endif
#endif
  return dVal;
}

int
AMF_DecodeBoolean(const char *data)
{
  return *data != 0;
}

char *
AMF_EncodeInt16(char *output, char *outend, short nVal)
{
  if (output+2 > outend)
    return NULL;

  output[1] = nVal & 0xff;
  output[0] = nVal >> 8;
  return output+2;
}
//3字节的int数据进行AMF编码，AMF采用大端模式
char *
AMF_EncodeInt24(char *output, char *outend, int nVal)
{
  if (output+3 > outend)
    return NULL;
  //倒过来
  output[2] = nVal & 0xff;
```

```
  output[2] = nVal & 0xff;
  output[1] = nVal >> 8;
  output[0] = nVal >> 16;
  //返回指针指向编码后数据的尾部
  return output+3;
}

char *
AMF_EncodeInt32(char *output, char *outend, int nVal)
{
  if (output+4 > outend)
    return NULL;

  output[3] = nVal & 0xff;
  output[2] = nVal >> 8;
  output[1] = nVal >> 16;
  output[0] = nVal >> 24;
  return output+4;
}

char *
AMF_EncodeString(char *output, char *outend, const AVal *bv)
{
  if ((bv->av_len < 65536 && output + 1 + 2 + bv->av_len > outend) ||
    output + 1 + 4 + bv->av_len > outend)
    return NULL;

  if (bv->av_len < 65536)
    {
      *output++ = AMF_STRING;

      output = AMF_EncodeInt16(output, outend, bv->av_len);
    }
  else
    {
      *output++ = AMF_LONG_STRING;

      output = AMF_EncodeInt32(output, outend, bv->av_len);
    }
  memcpy(output, bv->av_val, bv->av_len);
  output += bv->av_len;

  return output;
}

char *
AMF_EncodeNumber(char *output, char *outend, double dVal)
{
  if (output+1+8 > outend)
    return NULL;

  *output++ = AMF_NUMBER;   /* type: Number */

#if __FLOAT_WORD_ORDER == __BYTE_ORDER
#if __BYTE_ORDER == __BIG_ENDIAN
  memcpy(output, &dVal, 8);
#elif __BYTE_ORDER == __LITTLE_ENDIAN
  {
    unsigned char *ci, *co;
    ci = (unsigned char *)&dVal;
    co = (unsigned char *)output;
    co[0] = ci[7];
    co[1] = ci[6];
    co[2] = ci[5];
    co[3] = ci[4];
    co[4] = ci[3];
    co[5] = ci[2];
    co[6] = ci[1];
    co[7] = ci[0];
  }
#endif
#else
#if __BYTE_ORDER == __LITTLE_ENDIAN /* __FLOAT_WORD_ORER == __BIG_ENDIAN */
  {
    unsigned char *ci, *co;
    ci = (unsigned char *)&dVal;
    co = (unsigned char *)output;
    co[0] = ci[3];
    co[1] = ci[2];
    co[2] = ci[1];
    co[3] = ci[0];
    co[4] = ci[7];
    co[5] = ci[6];
    co[6] = ci[5];
    co[7] = ci[4];
  }
#else /* __BYTE_ORDER == __BIG_ENDIAN && __FLOAT_WORD_ORER == __LITTLE_ENDIAN */
  {
    unsigned char *ci, *co;
    ci = (unsigned char *)&dVal;
    co = (unsigned char *)output;
    co[0] = ci[4];
```

```
    co[1] = ci[5];
    co[2] = ci[6];
    co[3] = ci[7];
    co[4] = ci[0];
    co[5] = ci[1];
    co[6] = ci[2];
    co[7] = ci[3];
  }
#endif
#endif

  return output+8;
}

char *
AMF_EncodeBoolean(char *output, char *outend, int bVal)
{
  if (output+2 > outend)
    return NULL;

  *output++ = AMF_BOOLEAN;

  *output++ = bVal ? 0x01 : 0x00;

  return output;
}

char *
AMF_EncodeNamedString(char *output, char *outend, const AVal *strName, const AVal *strValue)
{
  if (output+2+strName->av_len > outend)
    return NULL;
  output = AMF_EncodeInt16(output, outend, strName->av_len);

  memcpy(output, strName->av_val, strName->av_len);
  output += strName->av_len;

  return AMF_EncodeString(output, outend, strValue);
}

char *
AMF_EncodeNamedNumber(char *output, char *outend, const AVal *strName, double dVal)
{
  if (output+2+strName->av_len > outend)
    return NULL;
  output = AMF_EncodeInt16(output, outend, strName->av_len);

  memcpy(output, strName->av_val, strName->av_len);
  output += strName->av_len;

  return AMF_EncodeNumber(output, outend, dVal);
}

char *
AMF_EncodeNamedBoolean(char *output, char *outend, const AVal *strName, int bVal)
{
  if (output+2+strName->av_len > outend)
    return NULL;
  output = AMF_EncodeInt16(output, outend, strName->av_len);

  memcpy(output, strName->av_val, strName->av_len);
  output += strName->av_len;

  return AMF_EncodeBoolean(output, outend, bVal);
}

void
AMFProp_GetName(AMFObjectProperty *prop, AVal *name)
{
  *name = prop->p_name;
}

void
AMFProp_SetName(AMFObjectProperty *prop, AVal *name)
{
  prop->p_name = *name;
}

AMFDataType
AMFProp_GetType(AMFObjectProperty *prop)
{
  return prop->p_type;
}

double
AMFProp_GetNumber(AMFObjectProperty *prop)
{
  return prop->p_vu.p_number;
}

int
```

```
AMFProp_GetBoolean(AMFObjectProperty *prop)
{
  return prop->p_vu.p_number != 0;
}

void
AMFProp_GetString(AMFObjectProperty *prop, AVal *str)
{
  *str = prop->p_vu.p_aval;
}

void
AMFProp_GetObject(AMFObjectProperty *prop, AMFObject *obj)
{
  *obj = prop->p_vu.p_object;
}

int
AMFProp_IsValid(AMFObjectProperty *prop)
{
  return prop->p_type != AMF_INVALID;
}

char *
AMFProp_Encode(AMFObjectProperty *prop, char *pBuffer, char *pBufEnd)
{
  if (prop->p_type == AMF_INVALID)
    return NULL;

  if (prop->p_type != AMF_NULL && pBuffer + prop->p_name.av_len + 2 + 1 >= pBufEnd)
    return NULL;

  if (prop->p_type != AMF_NULL && prop->p_name.av_len)
    {
      *pBuffer++ = prop->p_name.av_len >> 8;
      *pBuffer++ = prop->p_name.av_len & 0xff;
      memcpy(pBuffer, prop->p_name.av_val, prop->p_name.av_len);
      pBuffer += prop->p_name.av_len;
    }

  switch (prop->p_type)
    {
    case AMF_NUMBER:
      pBuffer = AMF_EncodeNumber(pBuffer, pBufEnd, prop->p_vu.p_number);
      break;

    case AMF_BOOLEAN:
      pBuffer = AMF_EncodeBoolean(pBuffer, pBufEnd, prop->p_vu.p_number != 0);
      break;

    case AMF_STRING:
      pBuffer = AMF_EncodeString(pBuffer, pBufEnd, &prop->p_vu.p_aval);
      break;

    case AMF_NULL:
      if (pBuffer+1 >= pBufEnd)
        return NULL;
      *pBuffer++ = AMF_NULL;
      break;

    case AMF_OBJECT:
      pBuffer = AMF_Encode(&prop->p_vu.p_object, pBuffer, pBufEnd);
      break;

    default:
      RTMP_Log(RTMP_LOGERROR, "%s, invalid type. %d", __FUNCTION__, prop->p_type);
      pBuffer = NULL;
    };

  return pBuffer;
}

#define AMF3_INTEGER_MAX    268435455
#define AMF3_INTEGER_MIN    -268435456

int
AMF3ReadInteger(const char *data, int32_t *valp)
{
  int i = 0;
  int32_t val = 0;

  while (i <= 2)
    {               /* handle first 3 bytes */
      if (data[i] & 0x80)
    {           /* byte used */
      val <<= 7;      /* shift up */
      val |= (data[i] & 0x7f);  /* add bits */
      i++;
    }
      else
    {
```

```
449.       break;
450.     }
451.     }
452.
453.   if (i > 2)
454.     {               /* use 4th byte, all 8bits */
455.       val <<= 8;
456.       val |= data[3];
457.
458.       /* range check */
459.       if (val > AMF3_INTEGER_MAX)
460.     val -= (1 << 29);
461.     }
462.   else
463.     {               /* use 7bits of last unparsed byte (0xxxxxxx) */
464.       val <<= 7;
465.       val |= data[i];
466.     }
467.
468.   *valp = val;
469.
470.   return i > 2 ? 4 : i + 1;
471. }
472.
473. int
474. AMF3ReadString(const char *data, AVal *str)
475. {
476.   int32_t ref = 0;
477.   int len;
478.   assert(str != 0);
479.
480.   len = AMF3ReadInteger(data, &ref);
481.   data += len;
482.
483.   if ((ref & 0x1) == 0)
484.     {               /* reference: 0xxx */
485.       uint32_t refIndex = (ref >> 1);
486.       RTMP_Log(RTMP_LOGDEBUG,
487.       "%s, string reference, index: %d, not supported, ignoring!",
488.       __FUNCTION__, refIndex);
489.       return len;
490.     }
491.   else
492.     {
493.       uint32_t nSize = (ref >> 1);
494.
495.       str->av_val = (char *)data;
496.       str->av_len = nSize;
497.
498.       return len + nSize;
499.     }
500.   return len;
501. }
502.
503. int
504. AMF3Prop_Decode(AMFObjectProperty *prop, const char *pBuffer, int nSize,
505.         int bDecodeName)
506. {
507.   int nOriginalSize = nSize;
508.   AMF3DataType type;
509.
510.   prop->p_name.av_len = 0;
511.   prop->p_name.av_val = NULL;
512.
513.   if (nSize == 0 || !pBuffer)
514.     {
515.       RTMP_Log(RTMP_LOGDEBUG, "empty buffer/no buffer pointer!");
516.       return -1;
517.     }
518.
519.   /* decode name */
520.   if (bDecodeName)
521.     {
522.       AVal name;
523.       int nRes = AMF3ReadString(pBuffer, &name);
524.
525.       if (name.av_len <= 0)
526.     return nRes;
527.
528.       prop->p_name = name;
529.       pBuffer += nRes;
530.       nSize -= nRes;
531.     }
532.
533.   /* decode */
534.   type = (AMF3DataType) *pBuffer++;
535.   nSize--;
536.
537.   switch (type)
538.     {
539.     case AMF3_UNDEFINED:
```

```
    case AMF3_NULL:
      prop->p_type = AMF_NULL;
      break;
    case AMF3_FALSE:
      prop->p_type = AMF_BOOLEAN;
      prop->p_vu.p_number = 0.0;
      break;
    case AMF3_TRUE:
      prop->p_type = AMF_BOOLEAN;
      prop->p_vu.p_number = 1.0;
      break;
    case AMF3_INTEGER:
      {
    int32_t res = 0;
    int len = AMF3ReadInteger(pBuffer, &res);
    prop->p_vu.p_number = (double)res;
    prop->p_type = AMF_NUMBER;
    nSize -= len;
    break;
      }
    case AMF3_DOUBLE:
      if (nSize < 8)
    return -1;
      prop->p_vu.p_number = AMF_DecodeNumber(pBuffer);
      prop->p_type = AMF_NUMBER;
      nSize -= 8;
      break;
    case AMF3_STRING:
    case AMF3_XML_DOC:
    case AMF3_XML:
      {
    int len = AMF3ReadString(pBuffer, &prop->p_vu.p_aval);
    prop->p_type = AMF_STRING;
    nSize -= len;
    break;
      }
    case AMF3_DATE:
      {
    int32_t res = 0;
    int len = AMF3ReadInteger(pBuffer, &res);

    nSize -= len;
    pBuffer += len;

    if ((res & 0x1) == 0)
      {         /* reference */
        uint32_t nIndex = (res >> 1);
        RTMP_Log(RTMP_LOGDEBUG, "AMF3_DATE reference: %d, not supported!", nIndex);
      }
    else
      {
        if (nSize < 8)
          return -1;

        prop->p_vu.p_number = AMF_DecodeNumber(pBuffer);
        nSize -= 8;
        prop->p_type = AMF_NUMBER;
      }
    break;
      }
    case AMF3_OBJECT:
      {
    int nRes = AMF3_Decode(&prop->p_vu.p_object, pBuffer, nSize, TRUE);
    if (nRes == -1)
      return -1;
    nSize -= nRes;
    prop->p_type = AMF_OBJECT;
    break;
      }
    case AMF3_ARRAY:
    case AMF3_BYTE_ARRAY:
    default:
      RTMP_Log(RTMP_LOGDEBUG, "%s - AMF3 unknown/unsupported datatype 0x%02x, @0x%08X",
      __FUNCTION__, (unsigned char)(*pBuffer), pBuffer);
      return -1;
    }

  return nOriginalSize - nSize;
}
//对AMF数据类型解析
int
AMFProp_Decode(AMFObjectProperty *prop, const char *pBuffer, int nSize,
           int bDecodeName)
{
  int nOriginalSize = nSize;
  int nRes;

  prop->p_name.av_len = 0;
  prop->p_name.av_val = NULL;

  if (nSize == 0 || !pBuffer)
```

```
    {
      RTMP_Log(RTMP_LOGDEBUG, "%s: Empty buffer/no buffer pointer!", __FUNCTION__);
      return -1;
    }

  if (bDecodeName && nSize < 4)
    {               /* at least name (length + at least 1 byte) and 1 byte of data */
      RTMP_Log(RTMP_LOGDEBUG,
      "%s: Not enough data for decoding with name, less than 4 bytes!",
      __FUNCTION__);
      return -1;
    }

  if (bDecodeName)
    {
      unsigned short nNameSize = AMF_DecodeInt16(pBuffer);
      if (nNameSize > nSize - 2)
    {
      RTMP_Log(RTMP_LOGDEBUG,
          "%s: Name size out of range: namesize (%d) > len (%d) - 2",
          __FUNCTION__, nNameSize, nSize);
      return -1;
    }

      AMF_DecodeString(pBuffer, &prop->p_name);
      nSize -= 2 + nNameSize;
      pBuffer += 2 + nNameSize;
    }

  if (nSize == 0)
    {
      return -1;
    }

  nSize--;

  prop->p_type = (AMFDataType) *pBuffer++;
  switch (prop->p_type)
    {
    //Number数据类型
    case AMF_NUMBER:
      if (nSize < 8)
    return -1;
      prop->p_vu.p_number = AMF_DecodeNumber(pBuffer);
      nSize -= 8;
      break;
     //Boolean数据类型
    case AMF_BOOLEAN:
      if (nSize < 1)
    return -1;
      prop->p_vu.p_number = (double)AMF_DecodeBoolean(pBuffer);
      nSize--;
      break;
      //String数据类型
    case AMF_STRING:
      {
    unsigned short nStringSize = AMF_DecodeInt16(pBuffer);

    if (nSize < (long)nStringSize + 2)
      return -1;
    AMF_DecodeString(pBuffer, &prop->p_vu.p_aval);
    nSize -= (2 + nStringSize);
    break;
      }
      //Object数据类型
    case AMF_OBJECT:
      {
    int nRes = AMF_Decode(&prop->p_vu.p_object, pBuffer, nSize, TRUE);
    if (nRes == -1)
      return -1;
    nSize -= nRes;
    break;
      }
    case AMF_MOVIECLIP:
      {
    RTMP_Log(RTMP_LOGERROR, "AMF_MOVIECLIP reserved!");
    return -1;
    break;
      }
    case AMF_NULL:
    case AMF_UNDEFINED:
    case AMF_UNSUPPORTED:
      prop->p_type = AMF_NULL;
      break;
    case AMF_REFERENCE:
      {
    RTMP_Log(RTMP_LOGERROR, "AMF_REFERENCE not supported!");
    return -1;
    break;
      }
    case AMF_ECMA_ARRAY:
      {
```

```
      {
    nSize -= 4;

    /* next comes the rest, mixed array has a final 0x000009 mark and names, so its an object */
    nRes = AMF_Decode(&prop->p_vu.p_object, pBuffer + 4, nSize, TRUE);
    if (nRes == -1)
      return -1;
    nSize -= nRes;
    prop->p_type = AMF_OBJECT;
    break;
      }
    case AMF_OBJECT_END:
      {
    return -1;
    break;
      }
    case AMF_STRICT_ARRAY:
      {
    unsigned int nArrayLen = AMF_DecodeInt32(pBuffer);
    nSize -= 4;

    nRes = AMF_DecodeArray(&prop->p_vu.p_object, pBuffer + 4, nSize,
                   nArrayLen, FALSE);
    if (nRes == -1)
      return -1;
    nSize -= nRes;
    prop->p_type = AMF_OBJECT;
    break;
      }
    case AMF_DATE:
      {
    RTMP_Log(RTMP_LOGDEBUG, "AMF_DATE");

    if (nSize < 10)
      return -1;

    prop->p_vu.p_number = AMF_DecodeNumber(pBuffer);
    prop->p_UTCoffset = AMF_DecodeInt16(pBuffer + 8);

    nSize -= 10;
    break;
      }
    case AMF_LONG_STRING:
      {
    unsigned int nStringSize = AMF_DecodeInt32(pBuffer);
    if (nSize < (long)nStringSize + 4)
      return -1;
    AMF_DecodeLongString(pBuffer, &prop->p_vu.p_aval);
    nSize -= (4 + nStringSize);
    prop->p_type = AMF_STRING;
    break;
      }
    case AMF_RECORDSET:
      {
    RTMP_Log(RTMP_LOGERROR, "AMF_RECORDSET reserved!");
    return -1;
    break;
      }
    case AMF_XML_DOC:
      {
    RTMP_Log(RTMP_LOGERROR, "AMF_XML_DOC not supported!");
    return -1;
    break;
      }
    case AMF_TYPED_OBJECT:
      {
    RTMP_Log(RTMP_LOGERROR, "AMF_TYPED_OBJECT not supported!");
    return -1;
    break;
      }
    case AMF_AVMPLUS:
      {
    int nRes = AMF3_Decode(&prop->p_vu.p_object, pBuffer, nSize, TRUE);
    if (nRes == -1)
      return -1;
    nSize -= nRes;
    prop->p_type = AMF_OBJECT;
    break;
      }
    default:
      RTMP_Log(RTMP_LOGDEBUG, "%s - unknown datatype 0x%02x, @0x%08X", __FUNCTION__,
      prop->p_type, pBuffer - 1);
      return -1;
    }

  return nOriginalSize - nSize;
}

void
AMFProp_Dump(AMFObjectProperty *prop)
{
  char strRes[256];
```

```
  char str[256];
  AVal name;

  if (prop->p_type == AMF_INVALID)
    {
      RTMP_Log(RTMP_LOGDEBUG, "Property: INVALID");
      return;
    }

  if (prop->p_type == AMF_NULL)
    {
      RTMP_Log(RTMP_LOGDEBUG, "Property: NULL");
      return;
    }

  if (prop->p_name.av_len)
    {
      name = prop->p_name;
    }
  else
    {
      name.av_val = "no-name.";
      name.av_len = sizeof("no-name.") - 1;
    }
  if (name.av_len > 18)
    name.av_len = 18;

  snprintf(strRes, 255, "Name: %18.*s, ", name.av_len, name.av_val);

  if (prop->p_type == AMF_OBJECT)
    {
      RTMP_Log(RTMP_LOGDEBUG, "Property: <%sOBJECT>", strRes);
      AMF_Dump(&prop->p_vu.p_object);
      return;
    }

  switch (prop->p_type)
    {
    case AMF_NUMBER:
      snprintf(str, 255, "NUMBER:\t%.2f", prop->p_vu.p_number);
      break;
    case AMF_BOOLEAN:
      snprintf(str, 255, "BOOLEAN:\t%s",
          prop->p_vu.p_number != 0.0 ? "TRUE" : "FALSE");
      break;
    case AMF_STRING:
      snprintf(str, 255, "STRING:\t%.*s", prop->p_vu.p_aval.av_len,
          prop->p_vu.p_aval.av_val);
      break;
    case AMF_DATE:
      snprintf(str, 255, "DATE:\ttimestamp: %.2f, UTC offset: %d",
          prop->p_vu.p_number, prop->p_UTCoffset);
      break;
    default:
      snprintf(str, 255, "INVALID TYPE 0x%02x", (unsigned char)prop->p_type);
    }

  RTMP_Log(RTMP_LOGDEBUG, "Property: <%s%s>", strRes, str);
}

void
AMFProp_Reset(AMFObjectProperty *prop)
{
  if (prop->p_type == AMF_OBJECT)
    AMF_Reset(&prop->p_vu.p_object);
  else
    {
      prop->p_vu.p_aval.av_len = 0;
      prop->p_vu.p_aval.av_val = NULL;
    }
  prop->p_type = AMF_INVALID;
}

/* AMFObject */

char *
AMF_Encode(AMFObject *obj, char *pBuffer, char *pBufEnd)
{
  int i;

  if (pBuffer+4 >= pBufEnd)
    return NULL;

  *pBuffer++ = AMF_OBJECT;

  for (i = 0; i < obj->o_num; i++)
    {
      char *res = AMFProp_Encode(&obj->o_props[i], pBuffer, pBufEnd);
      if (res == NULL)
    {
      RTMP_Log(RTMP_LOGERROR, "AMF Encode - failed to encode property in index %d",
```

```
905.            i);
906.       break;
907.     }
908.       else
909.     {
910.       pBuffer = res;
911.     }
912.     }
913.
914.   if (pBuffer + 3 >= pBufEnd)
915.     return NULL;            /* no room for the end marker */
916.
917.   pBuffer = AMF_EncodeInt24(pBuffer, pBufEnd, AMF_OBJECT_END);
918.
919.   return pBuffer;
920. }
921.
922. int
923. AMF_DecodeArray(AMFObject *obj, const char *pBuffer, int nSize,
924.         int nArrayLen, int bDecodeName)
925. {
926.   int nOriginalSize = nSize;
927.   int bError = FALSE;
928.
929.   obj->o_num = 0;
930.   obj->o_props = NULL;
931.   while (nArrayLen > 0)
932.     {
933.       AMFObjectProperty prop;
934.       int nRes;
935.       nArrayLen--;
936.
937.       nRes = AMFProp_Decode(&prop, pBuffer, nSize, bDecodeName);
938.       if (nRes == -1)
939.     bError = TRUE;
940.       else
941.     {
942.       nSize -= nRes;
943.       pBuffer += nRes;
944.       AMF_AddProp(obj, &prop);
945.     }
946.     }
947.   if (bError)
948.     return -1;
949.
950.   return nOriginalSize - nSize;
951. }
952.
953. int
954. AMF3_Decode(AMFObject *obj, const char *pBuffer, int nSize, int bAMFData)
955. {
956.   int nOriginalSize = nSize;
957.   int32_t ref;
958.   int len;
959.
960.   obj->o_num = 0;
961.   obj->o_props = NULL;
962.   if (bAMFData)
963.     {
964.       if (*pBuffer != AMF3_OBJECT)
965.     RTMP_Log(RTMP_LOGERROR,
966.         "AMF3 Object encapsulated in AMF stream does not start with AMF3_OBJECT!");
967.       pBuffer++;
968.       nSize--;
969.     }
970.
971.   ref = 0;
972.   len = AMF3ReadInteger(pBuffer, &ref);
973.   pBuffer += len;
974.   nSize -= len;
975.
976.   if ((ref & 1) == 0)
977.     {               /* object reference, 0xxx */
978.       uint32_t objectIndex = (ref >> 1);
979.
980.       RTMP_Log(RTMP_LOGDEBUG, "Object reference, index: %d", objectIndex);
981.     }
982.   else              /* object instance */
983.     {
984.       int32_t classRef = (ref >> 1);
985.
986.       AMF3ClassDef cd = { {0, 0}
987.       };
988.       AMFObjectProperty prop;
989.
990.       if ((classRef & 0x1) == 0)
991.     {           /* class reference */
992.       uint32_t classIndex = (classRef >> 1);
993.       RTMP_Log(RTMP_LOGDEBUG, "Class reference: %d", classIndex);
994.     }
995.       else
```

```
    {
      int32_t classExtRef = (classRef >> 1);
      int i;

      cd.cd_externalizable = (classExtRef & 0x1) == 1;
      cd.cd_dynamic = ((classExtRef >> 1) & 0x1) == 1;

      cd.cd_num = classExtRef >> 2;

      /* class name */

      len = AMF3ReadString(pBuffer, &cd.cd_name);
      nSize -= len;
      pBuffer += len;

      /*std::string str = className; */

      RTMP_Log(RTMP_LOGDEBUG,
          "Class name: %s, externalizable: %d, dynamic: %d, classMembers: %d",
          cd.cd_name.av_val, cd.cd_externalizable, cd.cd_dynamic,
          cd.cd_num);

      for (i = 0; i < cd.cd_num; i++)
        {
          AVal memberName;
          len = AMF3ReadString(pBuffer, &memberName);
          RTMP_Log(RTMP_LOGDEBUG, "Member: %s", memberName.av_val);
          AMF3CD_AddProp(&cd, &memberName);
          nSize -= len;
          pBuffer += len;
        }
    }

      /* add as referencable object */

      if (cd.cd_externalizable)
    {
      int nRes;
      AVal name = AVC("DEFAULT_ATTRIBUTE");

      RTMP_Log(RTMP_LOGDEBUG, "Externalizable, TODO check");

      nRes = AMF3Prop_Decode(&prop, pBuffer, nSize, FALSE);
      if (nRes == -1)
        RTMP_Log(RTMP_LOGDEBUG, "%s, failed to decode AMF3 property!",
          __FUNCTION__);
      else
        {
          nSize -= nRes;
          pBuffer += nRes;
        }

      AMFProp_SetName(&prop, &name);
      AMF_AddProp(obj, &prop);
    }
      else
    {
      int nRes, i;
      for (i = 0; i < cd.cd_num; i++)    /* non-dynamic */
        {
          nRes = AMF3Prop_Decode(&prop, pBuffer, nSize, FALSE);
          if (nRes == -1)
        RTMP_Log(RTMP_LOGDEBUG, "%s, failed to decode AMF3 property!",
              __FUNCTION__);

          AMFProp_SetName(&prop, AMF3CD_GetProp(&cd, i));
          AMF_AddProp(obj, &prop);

          pBuffer += nRes;
          nSize -= nRes;
        }
      if (cd.cd_dynamic)
        {
          int len = 0;

          do
        {
          nRes = AMF3Prop_Decode(&prop, pBuffer, nSize, TRUE);
          AMF_AddProp(obj, &prop);

          pBuffer += nRes;
          nSize -= nRes;

          len = prop.p_name.av_len;
        }
          while (len > 0);
        }
    }
      RTMP_Log(RTMP_LOGDEBUG, "class object!");
    }
  return nOriginalSize - nSize;
```

```
}
//解AMF编码的Object数据类型
int
AMF_Decode(AMFObject *obj, const char *pBuffer, int nSize, int bDecodeName)
{
  int nOriginalSize = nSize;
  int bError = FALSE;       /* if there is an error while decoding - try to at least find the end mark AMF_OBJECT_END */

  obj->o_num = 0;
  obj->o_props = NULL;
  while (nSize > 0)
    {
      AMFObjectProperty prop;
      int nRes;

      if (nSize >=3 && AMF_DecodeInt24(pBuffer) == AMF_OBJECT_END)
    {
      nSize -= 3;
      bError = FALSE;
      break;
    }

      if (bError)
    {
      RTMP_Log(RTMP_LOGERROR,
          "DECODING ERROR, IGNORING BYTES UNTIL NEXT KNOWN PATTERN!");
      nSize--;
      pBuffer++;
      continue;
    }
      //解Object里的Property
      nRes = AMFProp_Decode(&prop, pBuffer, nSize, bDecodeName);
      if (nRes == -1)
    bError = TRUE;
      else
    {
      nSize -= nRes;
      pBuffer += nRes;
      AMF_AddProp(obj, &prop);
    }
    }

  if (bError)
    return -1;

  return nOriginalSize - nSize;
}

void
AMF_AddProp(AMFObject *obj, const AMFObjectProperty *prop)
{
  if (!(obj->o_num & 0x0f))
    obj->o_props = (AMFObjectProperty *)
      realloc(obj->o_props, (obj->o_num + 16) * sizeof(AMFObjectProperty));
  obj->o_props[obj->o_num++] = *prop;
}

int
AMF_CountProp(AMFObject *obj)
{
  return obj->o_num;
}

AMFObjectProperty *
AMF_GetProp(AMFObject *obj, const AVal *name, int nIndex)
{
  if (nIndex >= 0)
    {
      if (nIndex <= obj->o_num)
    return &obj->o_props[nIndex];
    }
  else
    {
      int n;
      for (n = 0; n < obj->o_num; n++)
    {
      if (AVMATCH(&obj->o_props[n].p_name, name))
        return &obj->o_props[n];
    }
    }

  return (AMFObjectProperty *)&AMFProp_Invalid;
}

void
AMF_Dump(AMFObject *obj)
{
  int n;
  RTMP_Log(RTMP_LOGDEBUG, "(object begin)");
  for (n = 0; n < obj->o_num; n++)
    {
```

```
1178.       AMFProp_Dump(&obj->o_props[n]);
1179.     }
1180.   RTMP_Log(RTMP_LOGDEBUG, "(object end)");
1181. }
1182. 
1183. void
1184. AMF_Reset(AMFObject *obj)
1185. {
1186.   int n;
1187.   for (n = 0; n < obj->o_num; n++)
1188.     {
1189.       AMFProp_Reset(&obj->o_props[n]);
1190.     }
1191.   free(obj->o_props);
1192.   obj->o_props = NULL;
1193.   obj->o_num = 0;
1194. }
1195. 
1196. 
1197. /* AMF3ClassDefinition */
1198. 
1199. void
1200. AMF3CD_AddProp(AMF3ClassDef *cd, AVal *prop)
1201. {
1202.   if (!(cd->cd_num & 0x0f))
1203.     cd->cd_props = (AVal *)realloc(cd->cd_props, (cd->cd_num + 16) * sizeof(AVal));
1204.   cd->cd_props[cd->cd_num++] = *prop;
1205. }
1206. 
1207. AVal *
1208. AMF3CD_GetProp(AMF3ClassDef *cd, int nIndex)
1209. {
1210.   if (nIndex >= cd->cd_num)
1211.     return (AVal *)&AV_empty;
1212.   return &cd->cd_props[nIndex];
1213. }
```

可参考文件：

AMF3 中文版介绍： http://download.csdn.net/detail/leixiaohua1020/6389977

rtmpdump源代码（Linux）： http://download.csdn.net/detail/leixiaohua1020/6376561

rtmpdump源代码（VC 2005 工程）： http://download.csdn.net/detail/leixiaohua1020/6563163

文章标签： RTMPdump rtmp 源代码 AMF

个人分类： libRTMP

所属专栏： 开源多媒体项目源代码分析