

原 RTMPdump (libRTMP) 源代码分析 3：AMF编码

2013年10月22日 21:18:47 阅读数：12218

=====

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]  
1. #include "stdafx.h"
2. /*  本文件主要包含了对AMF对象的操作
3.  *-----
4.  *AMF数据类型：
5.  *Type      Byte code
6.  *Number     0x00
7.  *Boolean    0x01
8.  *String     0x02
9.  *Object     0x03
10. *MovieClip 0x04
11. *Null       0x05
12. *Undefined  0x06
13. *Reference  0x07
14. *MixedArray 0x08
15. *EndOfObject 0x09
16. *Array      0x0a
17. *Date       0x0b
18. *LongString 0x0c
19. *Unsupported 0x0d
20. *Recordset  0x0e
21. *XML        0x0f
22. *TypedObject (Class instance) 0x10
23. *AMF3 data  0x11
24. *-----
25. *应用举例：
26. *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
27. *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。
28. *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。
```

```

29.  * 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”。
30.  *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。
31.  *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。
32.  *6.Undefined 也是只占用一个字节0x06。
33.  *8.MixedArray相当于Hashtable，与3不同的是该对象定义了Hashtable的大小。
34.  */
35.
36.
37.
38. #include <string.h>
39. #include <assert.h>
40. #include <stdlib.h>
41.
42. #include "rtmp_sys.h"
43. #include "amf.h"
44. #include "log.h"
45. #include "bytes.h"
46.
47. static const AMFObjectProperty AMFProp_Invalid = { {0, 0}, AMF_INVALID };
48. static const AVVal AV_empty = { 0, 0 };
49.
50. //大端Big-Endian
51. //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。
52. //符合人脑逻辑，与计算机逻辑不同
53. //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
54. //用的字节序通常称之为网络字节序。
55. //主机序 Host Order: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
56. //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。
57.
58.
59. /*AMF数据采用 Big-Endian（大端模式），主机采用Little-Endian（小端模式） */
60.
61. unsigned short
62. AMF_DecodeInt16(const char *data)
63. {
64.     unsigned char *c = (unsigned char *) data;
65.     unsigned short val;
66.     val = (c[0] << 8) | c[1]; //转换
67.     return val;
68. }
69.
70. unsigned int
71. AMF_DecodeInt24(const char *data)
72. {
73.     unsigned char *c = (unsigned char *) data;
74.     unsigned int val;
75.     val = (c[0] << 16) | (c[1] << 8) | c[2];
76.     return val;
77. }
78.
79. unsigned int
80. AMF_DecodeInt32(const char *data)
81. {
82.     unsigned char *c = (unsigned char *)data;
83.     unsigned int val;
84.     val = (c[0] << 24) | (c[1] << 16) | (c[2] << 8) | c[3];
85.     return val;
86. }
87.
88. void
89. AMF_DecodeString(const char *data, AVVal *bv)
90. {
91.     bv->av_len = AMF_DecodeInt16(data);
92.     bv->av_val = (bv->av_len > 0) ? (char *)data + 2 : NULL;
93. }
94.
95. void
96. AMF_DecodeLongString(const char *data, AVVal *bv)
97. {
98.     bv->av_len = AMF_DecodeInt32(data);
99.     bv->av_val = (bv->av_len > 0) ? (char *)data + 4 : NULL;
100. }
101.
102. double
103. AMF_DecodeNumber(const char *data)
104. {
105.     double dVal;
106.     #if __FLOAT_WORD_ORDER == __BYTE_ORDER
107.     #if __BYTE_ORDER == __BIG_ENDIAN
108.         memcpy(&dVal, data, 8);
109.     #elif __BYTE_ORDER == __LITTLE_ENDIAN
110.         unsigned char *ci, *co;
111.         ci = (unsigned char *)data;
112.         co = (unsigned char *)&dVal;
113.         co[0] = ci[7];
114.         co[1] = ci[6];
115.         co[2] = ci[5];
116.         co[3] = ci[4];
117.         co[4] = ci[3];
118.         co[5] = ci[2];
119.         co[6] = ci[1];

```

```

120.     co[7] = ci[0];
121. #endif
122. #else
123. #if __BYTE_ORDER == __LITTLE_ENDIAN /* __FLOAT_WORD_ORDER == __BIG_ENDIAN */
124.     unsigned char *ci, *co;
125.     ci = (unsigned char *)data;
126.     co = (unsigned char *)&dVal;
127.     co[0] = ci[3];
128.     co[1] = ci[2];
129.     co[2] = ci[1];
130.     co[3] = ci[0];
131.     co[4] = ci[7];
132.     co[5] = ci[6];
133.     co[6] = ci[5];
134.     co[7] = ci[4];
135. #else /* __BYTE_ORDER == __BIG_ENDIAN && __FLOAT_WORD_ORDER == __LITTLE_ENDIAN */
136.     unsigned char *ci, *co;
137.     ci = (unsigned char *)data;
138.     co = (unsigned char *)&dVal;
139.     co[0] = ci[4];
140.     co[1] = ci[5];
141.     co[2] = ci[6];
142.     co[3] = ci[7];
143.     co[4] = ci[0];
144.     co[5] = ci[1];
145.     co[6] = ci[2];
146.     co[7] = ci[3];
147. #endif
148. #endif
149.     return dVal;
150. }
151.
152. int
153. AMF_DecodeBoolean(const char *data)
154. {
155.     return *data != 0;
156. }
157.
158. char *
159. AMF_EncodeInt16(char *output, char *outend, short nVal)
160. {
161.     if (output+2 > outend)
162.         return NULL;
163.
164.     output[1] = nVal & 0xff;
165.     output[0] = nVal >> 8;
166.     return output+2;
167. }
168. //3字节的int数据进行AMF编码, AMF采用大端模式
169. char *
170. AMF_EncodeInt24(char *output, char *outend, int nVal)
171. {
172.     if (output+3 > outend)
173.         return NULL;
174.     //倒过来
175.     output[2] = nVal & 0xff;
176.     output[1] = nVal >> 8;
177.     output[0] = nVal >> 16;
178.     //返回指针指向编码后数据的尾部
179.     return output+3;
180. }
181.
182. char *
183. AMF_EncodeInt32(char *output, char *outend, int nVal)
184. {
185.     if (output+4 > outend)
186.         return NULL;
187.
188.     output[3] = nVal & 0xff;
189.     output[2] = nVal >> 8;
190.     output[1] = nVal >> 16;
191.     output[0] = nVal >> 24;
192.     return output+4;
193. }
194.
195. char *
196. AMF_EncodeString(char *output, char *outend, const AVal *bv)
197. {
198.     if ((bv->av_len < 65536 && output + 1 + 2 + bv->av_len > outend) ||
199.         output + 1 + 4 + bv->av_len > outend)
200.         return NULL;
201.
202.     if (bv->av_len < 65536)
203.     {
204.         *output++ = AMF_STRING;
205.
206.         output = AMF_EncodeInt16(output, outend, bv->av_len);
207.     }
208.     else
209.     {
210.         *output++ = AMF_LONG_STRING;

```

```

211.     output = AMF_EncodeInt32(output, outend, bv->av_len);
212. }
213. memcpy(output, bv->av_val, bv->av_len);
214. output += bv->av_len;
215.
216.
217.     return output;
218. }
219.
220. char *
221. AMF_EncodeNumber(char *output, char *outend, double dVal)
222. {
223.     if (output+1+8 > outend)
224.         return NULL;
225.
226.     *output++ = AMF_NUMBER;    /* type: Number */
227.
228.     #if __FLOAT_WORD_ORDER == __BYTE_ORDER
229.     #if __BYTE_ORDER == __BIG_ENDIAN
230.         memcpy(output, &dVal, 8);
231.     #elif __BYTE_ORDER == __LITTLE_ENDIAN
232.     {
233.         unsigned char *ci, *co;
234.         ci = (unsigned char *)&dVal;
235.         co = (unsigned char *)output;
236.         co[0] = ci[7];
237.         co[1] = ci[6];
238.         co[2] = ci[5];
239.         co[3] = ci[4];
240.         co[4] = ci[3];
241.         co[5] = ci[2];
242.         co[6] = ci[1];
243.         co[7] = ci[0];
244.     }
245.     #endif
246.     #else
247.     #if __BYTE_ORDER == __LITTLE_ENDIAN /* __FLOAT_WORD_ORER == __BIG_ENDIAN */
248.     {
249.         unsigned char *ci, *co;
250.         ci = (unsigned char *)&dVal;
251.         co = (unsigned char *)output;
252.         co[0] = ci[3];
253.         co[1] = ci[2];
254.         co[2] = ci[1];
255.         co[3] = ci[0];
256.         co[4] = ci[7];
257.         co[5] = ci[6];
258.         co[6] = ci[5];
259.         co[7] = ci[4];
260.     }
261.     #else /* __BYTE_ORDER == __BIG_ENDIAN && __FLOAT_WORD_ORER == __LITTLE_ENDIAN */
262.     {
263.         unsigned char *ci, *co;
264.         ci = (unsigned char *)&dVal;
265.         co = (unsigned char *)output;
266.         co[0] = ci[4];
267.         co[1] = ci[5];
268.         co[2] = ci[6];
269.         co[3] = ci[7];
270.         co[4] = ci[0];
271.         co[5] = ci[1];
272.         co[6] = ci[2];
273.         co[7] = ci[3];
274.     }
275.     #endif
276.     #endif
277.
278.     return output+8;
279. }
280.
281. char *
282. AMF_EncodeBoolean(char *output, char *outend, int bVal)
283. {
284.     if (output+2 > outend)
285.         return NULL;
286.
287.     *output++ = AMF_BOOLEAN;
288.
289.     *output++ = bVal ? 0x01 : 0x00;
290.
291.     return output;
292. }
293.
294. char *
295. AMF_EncodeNamedString(char *output, char *outend, const AVal *strName, const AVal *strValue)
296. {
297.     if (output+2+strName->av_len > outend)
298.         return NULL;
299.     output = AMF_EncodeInt16(output, outend, strName->av_len);
300.
301.     memcpy(output, strName->av_val, strName->av_len);
302.
303.     output += strName->av_len;
304.     *output++ = 0;
305.     return output;
306. }

```

```

302.     output += strName->av_len;
303.
304.     return AMF_EncodeString(output, outend, strValue);
305. }
306.
307. char *
308. AMF_EncodeNamedNumber(char *output, char *outend, const AVVal *strName, double dVal)
309. {
310.     if (output+2+strName->av_len > outend)
311.         return NULL;
312.     output = AMF_EncodeInt16(output, outend, strName->av_len);
313.
314.     memcpy(output, strName->av_val, strName->av_len);
315.     output += strName->av_len;
316.
317.     return AMF_EncodeNumber(output, outend, dVal);
318. }
319.
320. char *
321. AMF_EncodeNamedBoolean(char *output, char *outend, const AVVal *strName, int bVal)
322. {
323.     if (output+2+strName->av_len > outend)
324.         return NULL;
325.     output = AMF_EncodeInt16(output, outend, strName->av_len);
326.
327.     memcpy(output, strName->av_val, strName->av_len);
328.     output += strName->av_len;
329.
330.     return AMF_EncodeBoolean(output, outend, bVal);
331. }
332.
333. void
334. AMFProp_GetName(AMFObjectProperty *prop, AVVal *name)
335. {
336.     *name = prop->p_name;
337. }
338.
339. void
340. AMFProp_SetName(AMFObjectProperty *prop, AVVal *name)
341. {
342.     prop->p_name = *name;
343. }
344.
345. AMFDataType
346. AMFProp_GetType(AMFObjectProperty *prop)
347. {
348.     return prop->p_type;
349. }
350.
351. double
352. AMFProp_GetNumber(AMFObjectProperty *prop)
353. {
354.     return prop->p_vu.p_number;
355. }
356.
357. int
358. AMFProp_GetBoolean(AMFObjectProperty *prop)
359. {
360.     return prop->p_vu.p_number != 0;
361. }
362.
363. void
364. AMFProp_GetString(AMFObjectProperty *prop, AVVal *str)
365. {
366.     *str = prop->p_vu.p_aval;
367. }
368.
369. void
370. AMFProp_GetObject(AMFObjectProperty *prop, AMFObject *obj)
371. {
372.     *obj = prop->p_vu.p_object;
373. }
374.
375. int
376. AMFProp_IsValid(AMFObjectProperty *prop)
377. {
378.     return prop->p_type != AMF_INVALID;
379. }
380.
381. char *
382. AMFProp_Encode(AMFObjectProperty *prop, char *pBuffer, char *pBufEnd)
383. {
384.     if (prop->p_type == AMF_INVALID)
385.         return NULL;
386.
387.     if (prop->p_type != AMF_NULL && pBuffer + prop->p_name.av_len + 2 + 1 >= pBufEnd)
388.         return NULL;
389.
390.     if (prop->p_type != AMF_NULL && prop->p_name.av_len)
391.     {
392.         *pBuffer++ = prop->p_name.av_len >> 8;
393.         *pBuffer++ = prop->p_name.av_len & 0xff;

```

```

393.     pBuffer += prop->p_name.av_len & 0xFF;
394.     memcpy(pBuffer, prop->p_name.av_val, prop->p_name.av_len);
395.     pBuffer += prop->p_name.av_len;
396. }
397.
398. switch (prop->p_type)
399. {
400.     case AMF_NUMBER:
401.         pBuffer = AMF_EncodeNumber(pBuffer, pBufEnd, prop->p_vu.p_number);
402.         break;
403.
404.     case AMF_BOOLEAN:
405.         pBuffer = AMF_EncodeBoolean(pBuffer, pBufEnd, prop->p_vu.p_number != 0);
406.         break;
407.
408.     case AMF_STRING:
409.         pBuffer = AMF_EncodeString(pBuffer, pBufEnd, &prop->p_vu.p_aval);
410.         break;
411.
412.     case AMF_NULL:
413.         if (pBuffer+1 >= pBufEnd)
414.             return NULL;
415.         *pBuffer++ = AMF_NULL;
416.         break;
417.
418.     case AMF_OBJECT:
419.         pBuffer = AMF_Encode(&prop->p_vu.p_object, pBuffer, pBufEnd);
420.         break;
421.
422.     default:
423.         RTMP_Log(RTMP_LOGERROR, "%s, invalid type. %d", __FUNCTION__, prop->p_type);
424.         pBuffer = NULL;
425.     };
426.
427.     return pBuffer;
428. }
429.
430. #define AMF3_INTEGER_MAX    268435455
431. #define AMF3_INTEGER_MIN    -268435456
432.
433. int
434. AMF3ReadInteger(const char *data, int32_t *valp)
435. {
436.     int i = 0;
437.     int32_t val = 0;
438.
439.     while (i <= 2)
440.     {
441.         /* handle first 3 bytes */
442.         if (data[i] & 0x80)
443.         {
444.             /* byte used */
445.             val <<= 7; /* shift up */
446.             val |= (data[i] & 0x7f); /* add bits */
447.             i++;
448.         }
449.         else
450.         {
451.             break;
452.         }
453.
454.         if (i > 2)
455.         {
456.             /* use 4th byte, all 8bits */
457.             val <<= 8;
458.             val |= data[3];
459.
460.             /* range check */
461.             if (val > AMF3_INTEGER_MAX)
462.                 val -= (1 << 29);
463.         }
464.         else
465.         {
466.             /* use 7bits of last unparsed byte (0xxxxxxx) */
467.             val <<= 7;
468.             val |= data[i];
469.         }
470.
471.         *valp = val;
472.
473.         return i > 2 ? 4 : i + 1;
474.     }
475. }
476.
477. int
478. AMF3ReadString(const char *data, AVal *str)
479. {
480.     int32_t ref = 0;
481.     int len;
482.     assert(str != 0);
483.
484.     len = AMF3ReadInteger(data, &ref);
485.     data += len;
486.
487.     if ((ref & 0x1) == 0)
488.     {
489.         /* 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:
540.     case AMF3_NULL:
541.         prop->p_type = AMF_NULL;
542.         break;
543.     case AMF3_FALSE:
544.         prop->p_type = AMF_BOOLEAN;
545.         prop->p_vu.p_number = 0.0;
546.         break;
547.     case AMF3_TRUE:
548.         prop->p_type = AMF_BOOLEAN;
549.         prop->p_vu.p_number = 1.0;
550.         break;
551.     case AMF3_INTEGER:
552.     {
553.         int32_t res = 0;
554.         int len = AMF3ReadInteger(pBuffer, &res);
555.         prop->p_vu.p_number = (double)res;
556.         prop->p_type = AMF_NUMBER;
557.         nSize -= len;
558.         break;
559.     }
560.     case AMF3_DOUBLE:
561.         if (nSize < 8)
562.             return -1;
563.         prop->p_vu.p_number = AMF_DecodeNumber(pBuffer);
564.         prop->p_type = AMF_NUMBER;
565.         nSize -= 8;
566.         break;
567.     case AMF3_STRING:
568.     case AMF3_XML_DOC:
569.     case AMF3_XML:
570.     {
571.         int len = AMF3ReadString(pBuffer, &prop->p_vu.p_aval);
572.         prop->p_type = AMF_STRING;
573.         nSize -= len;
574.         break;
575.     }

```

```

576.     case AMF3_DATE:
577.     {
578.         int32_t res = 0;
579.         int len = AMF3ReadInteger(pBuffer, &res);
580.
581.         nSize -= len;
582.         pBuffer += len;
583.
584.         if ((res & 0x1) == 0)
585.         {
586.             /* reference */
587.             uint32_t nIndex = (res >> 1);
588.             RTMP_Log(RTMP_LOGDEBUG, "AMF3_DATE reference: %d, not supported!", nIndex);
589.         }
590.         else
591.         {
592.             if (nSize < 8)
593.                 return -1;
594.
595.             prop->p_vu.p_number = AMF_DecodeNumber(pBuffer);
596.             nSize -= 8;
597.             prop->p_type = AMF_NUMBER;
598.         }
599.         break;
600.     case AMF3_OBJECT:
601.     {
602.         int nRes = AMF3_Decode(&prop->p_vu.p_object, pBuffer, nSize, TRUE);
603.         if (nRes == -1)
604.             return -1;
605.         nSize -= nRes;
606.         prop->p_type = AMF_OBJECT;
607.         break;
608.     }
609.     case AMF3_ARRAY:
610.     case AMF3_BYTE_ARRAY:
611.     default:
612.         RTMP_Log(RTMP_LOGDEBUG, "%s - AMF3 unknown/unsupported datatype 0x%02x, @0x%08X",
613.             __FUNCTION__, (unsigned char)(*pBuffer), pBuffer);
614.         return -1;
615.     }
616.
617.     return nOriginalSize - nSize;
618. }
619. //对AMF数据类型解析
620. int
621. AMFProp_Decode(AMFObjectProperty *prop, const char *pBuffer, int nSize,
622.     int bDecodeName)
623. {
624.     int nOriginalSize = nSize;
625.     int nRes;
626.
627.     prop->p_name.av_len = 0;
628.     prop->p_name.av_val = NULL;
629.
630.     if (nSize == 0 || !pBuffer)
631.     {
632.         RTMP_Log(RTMP_LOGDEBUG, "%s: Empty buffer/no buffer pointer!", __FUNCTION__);
633.         return -1;
634.     }
635.
636.     if (bDecodeName && nSize < 4)
637.     {
638.         /* at least name (length + at least 1 byte) and 1 byte of data */
639.         RTMP_Log(RTMP_LOGDEBUG,
640.             "%s: Not enough data for decoding with name, less than 4 bytes!",
641.             __FUNCTION__);
642.         return -1;
643.     }
644.
645.     if (bDecodeName)
646.     {
647.         unsigned short nNameSize = AMF_DecodeInt16(pBuffer);
648.         if (nNameSize > nSize - 2)
649.         {
650.             RTMP_Log(RTMP_LOGDEBUG,
651.                 "%s: Name size out of range: namesize (%d) > len (%d) - 2",
652.                 __FUNCTION__, nNameSize, nSize);
653.             return -1;
654.         }
655.
656.         AMF_DecodeString(pBuffer, &prop->p_name);
657.         nSize -= 2 + nNameSize;
658.         pBuffer += 2 + nNameSize;
659.     }
660.
661.     if (nSize == 0)
662.     {
663.         return -1;
664.     }
665.
666.     nSize--;

```



```

667. prop->p_type = (AMFDataType) *pBuffer++;
668. switch (prop->p_type)
669. {
670.     //Number数据类型
671.     case AMF_NUMBER:
672.         if (nSize < 8)
673.             return -1;
674.         prop->p_vu.p_number = AMF_DecodeNumber(pBuffer);
675.         nSize -= 8;
676.         break;
677.     //Boolean数据类型
678.     case AMF_BOOLEAN:
679.         if (nSize < 1)
680.             return -1;
681.         prop->p_vu.p_number = (double)AMF_DecodeBoolean(pBuffer);
682.         nSize--;
683.         break;
684.     //String数据类型
685.     case AMF_STRING:
686.         {
687.             unsigned short nStringSize = AMF_DecodeInt16(pBuffer);
688.
689.             if (nSize < (long)nStringSize + 2)
690.                 return -1;
691.             AMF_DecodeString(pBuffer, &prop->p_vu.p_aval);
692.             nSize -= (2 + nStringSize);
693.             break;
694.         }
695.     //Object数据类型
696.     case AMF_OBJECT:
697.         {
698.             int nRes = AMF_Decode(&prop->p_vu.p_object, pBuffer, nSize, TRUE);
699.             if (nRes == -1)
700.                 return -1;
701.             nSize -= nRes;
702.             break;
703.         }
704.     case AMF_MOVIECLIP:
705.         {
706.             RTMP_Log(RTMP_LOGERROR, "AMF_MOVIECLIP reserved!");
707.             return -1;
708.             break;
709.         }
710.     case AMF_NULL:
711.     case AMF_UNDEFINED:
712.     case AMF_UNSUPPORTED:
713.         prop->p_type = AMF_NULL;
714.         break;
715.     case AMF_REFERENCE:
716.         {
717.             RTMP_Log(RTMP_LOGERROR, "AMF_REFERENCE not supported!");
718.             return -1;
719.             break;
720.         }
721.     case AMF_ECMA_ARRAY:
722.         {
723.             nSize -= 4;
724.
725.             /* next comes the rest, mixed array has a final 0x000009 mark and names, so its an object */
726.             nRes = AMF_Decode(&prop->p_vu.p_object, pBuffer + 4, nSize, TRUE);
727.             if (nRes == -1)
728.                 return -1;
729.             nSize -= nRes;
730.             prop->p_type = AMF_OBJECT;
731.             break;
732.         }
733.     case AMF_OBJECT_END:
734.         {
735.             return -1;
736.             break;
737.         }
738.     case AMF_STRICT_ARRAY:
739.         {
740.             unsigned int nArrayLen = AMF_DecodeInt32(pBuffer);
741.             nSize -= 4;
742.
743.             nRes = AMF_DecodeArray(&prop->p_vu.p_object, pBuffer + 4, nSize,
744.                                   nArrayLen, FALSE);
745.             if (nRes == -1)
746.                 return -1;
747.             nSize -= nRes;
748.             prop->p_type = AMF_OBJECT;
749.             break;
750.         }
751.     case AMF_DATE:
752.         {
753.             RTMP_Log(RTMP_LOGDEBUG, "AMF_DATE");
754.
755.             if (nSize < 10)
756.                 return -1;
757.

```

```

758.     prop->p_vu.p_number = AMF_DecodeNumber(pBuffer);
759.     prop->p_UTCOffset = AMF_DecodeInt16(pBuffer + 8);
760.
761.     nSize -= 10;
762.     break;
763. }
764. case AMF_LONG_STRING:
765. {
766.     unsigned int nStringSize = AMF_DecodeInt32(pBuffer);
767.     if (nSize < (long)nStringSize + 4)
768.         return -1;
769.     AMF_DecodeLongString(pBuffer, &prop->p_vu.p_aval);
770.     nSize -= (4 + nStringSize);
771.     prop->p_type = AMF_STRING;
772.     break;
773. }
774. case AMF_RECORDSET:
775. {
776.     RTMP_Log(RTMP_LOGERROR, "AMF_RECORDSET reserved!");
777.     return -1;
778.     break;
779. }
780. case AMF_XML_DOC:
781. {
782.     RTMP_Log(RTMP_LOGERROR, "AMF_XML_DOC not supported!");
783.     return -1;
784.     break;
785. }
786. case AMF_TYPED_OBJECT:
787. {
788.     RTMP_Log(RTMP_LOGERROR, "AMF_TYPED_OBJECT not supported!");
789.     return -1;
790.     break;
791. }
792. case AMF_AVMPLUS:
793. {
794.     int nRes = AMF3_Decode(&prop->p_vu.p_object, pBuffer, nSize, TRUE);
795.     if (nRes == -1)
796.         return -1;
797.     nSize -= nRes;
798.     prop->p_type = AMF_OBJECT;
799.     break;
800. }
801. default:
802.     RTMP_Log(RTMP_LOGDEBUG, "%s - unknown datatype 0x%02x, @0x%08x", __FUNCTION__,
803.         prop->p_type, pBuffer - 1);
804.     return -1;
805. }
806.
807. return nOriginalSize - nSize;
808. }
809.
810. void
811. AMFProp_Dump(AMFObjectProperty *prop)
812. {
813.     char strRes[256];
814.     char str[256];
815.     AVal name;
816.
817.     if (prop->p_type == AMF_INVALID)
818.     {
819.         RTMP_Log(RTMP_LOGDEBUG, "Property: INVALID");
820.         return;
821.     }
822.
823.     if (prop->p_type == AMF_NULL)
824.     {
825.         RTMP_Log(RTMP_LOGDEBUG, "Property: NULL");
826.         return;
827.     }
828.
829.     if (prop->p_name.av_len)
830.     {
831.         name = prop->p_name;
832.     }
833.     else
834.     {
835.         name.av_val = "no-name.";
836.         name.av_len = sizeof("no-name.") - 1;
837.     }
838.     if (name.av_len > 18)
839.         name.av_len = 18;
840.
841.     snprintf(strRes, 255, "Name: %18.*s, ", name.av_len, name.av_val);
842.
843.     if (prop->p_type == AMF_OBJECT)
844.     {
845.         RTMP_Log(RTMP_LOGDEBUG, "Property: <%sOBJECT>", strRes);
846.         AMF_Dump(&prop->p_vu.p_object);
847.         return;
848.     }

```

```

849.
850.     switch (prop->p_type)
851.     {
852.     case AMF_NUMBER:
853.         snprintf(str, 255, "NUMBER:\t%.2f", prop->p_vu.p_number);
854.         break;
855.     case AMF_BOOLEAN:
856.         snprintf(str, 255, "BOOLEAN:\t%s",
857.             prop->p_vu.p_number != 0.0 ? "TRUE" : "FALSE");
858.         break;
859.     case AMF_STRING:
860.         snprintf(str, 255, "STRING:\t%.s", prop->p_vu.p_aval.av_len,
861.             prop->p_vu.p_aval.av_val);
862.         break;
863.     case AMF_DATE:
864.         snprintf(str, 255, "DATE:\ttimestamp: %.2f, UTC offset: %d",
865.             prop->p_vu.p_number, prop->p_UTCOffset);
866.         break;
867.     default:
868.         snprintf(str, 255, "INVALID TYPE 0x%02x", (unsigned char)prop->p_type);
869.     }
870.
871.     RTMP_Log(RTMP_LOGDEBUG, "Property: <%s%>", strRes, str);
872. }
873.
874. void
875. AMFProp_Reset(AMFObjectProperty *prop)
876. {
877.     if (prop->p_type == AMF_OBJECT)
878.         AMF_Reset(&prop->p_vu.p_object);
879.     else
880.     {
881.         prop->p_vu.p_aval.av_len = 0;
882.         prop->p_vu.p_aval.av_val = NULL;
883.     }
884.     prop->p_type = AMF_INVALID;
885. }
886.
887. /* AMFObject */
888.
889. char *
890. AMF_Encode(AMFObject *obj, char *pBuffer, char *pBufEnd)
891. {
892.     int i;
893.
894.     if (pBuffer+4 >= pBufEnd)
895.         return NULL;
896.
897.     *pBuffer++ = AMF_OBJECT;
898.
899.     for (i = 0; i < obj->o_num; i++)
900.     {
901.         char *res = AMFProp_Encode(&obj->o_props[i], pBuffer, pBufEnd);
902.         if (res == NULL)
903.         {
904.             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

```

```

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.     {
978.         /* object reference, 0xxx */
979.         uint32_t objectIndex = (ref >> 1);
980.
981.         RTMP_Log(RTMP_LOGDEBUG, "Object reference, index: %d", objectIndex);
982.     }
983.     else /* object instance */
984.     {
985.         int32_t classRef = (ref >> 1);
986.
987.         AMF3ClassDef cd = { {0, 0}
988.         };
989.         AMFObjectProperty prop;
990.
991.         if ((classRef & 0x1) == 0)
992.         {
993.             /* class reference */
994.             uint32_t classIndex = (classRef >> 1);
995.             RTMP_Log(RTMP_LOGDEBUG, "Class reference: %d", classIndex);
996.         }
997.         else
998.         {
999.             int32_t classExtRef = (classRef >> 1);
1000.             int i;
1001.
1002.             cd.cd_externalizable = (classExtRef & 0x1) == 1;
1003.             cd.cd_dynamic = ((classExtRef >> 1) & 0x1) == 1;
1004.
1005.             cd.cd_num = classExtRef >> 2;
1006.
1007.             /* class name */
1008.             len = AMF3ReadString(pBuffer, &cd.cd_name);
1009.             nSize -= len;
1010.             pBuffer += len;
1011.
1012.             /*std::string str = className; */
1013.
1014.             RTMP_Log(RTMP_LOGDEBUG,
1015.                 "Class name: %s, externalizable: %d, dynamic: %d, classMembers: %d",
1016.                 cd.cd_name.av_val, cd.cd_externalizable, cd.cd_dynamic,
1017.                 cd.cd_num);
1018.
1019.             for (i = 0; i < cd.cd_num; i++)
1020.             {
1021.                 AVal memberName;
1022.                 len = AMF3ReadString(pBuffer, &memberName);
1023.                 RTMP_Log(RTMP_LOGDEBUG, "Member: %s", memberName.av_val);
1024.                 AMF3CD_AddProp(&cd, &memberName);
1025.                 nSize -= len;
1026.                 pBuffer += len;
1027.             }
1028.         }
1029.
1030.         /* add as referencable object */
1031.         if (cd.cd_externalizable)

```

```

1032. {
1033.     int nRes;
1034.     AVa1 name = AVC("DEFAULT_ATTRIBUTE");
1035.
1036.     RTMP_Log(RTMP_LOGDEBUG, "Externalizable, TODO check");
1037.
1038.     nRes = AMF3Prop_Decode(&prop, pBuffer, nSize, FALSE);
1039.     if (nRes == -1)
1040.         RTMP_Log(RTMP_LOGDEBUG, "%s, failed to decode AMF3 property!",
1041.             __FUNCTION__);
1042.     else
1043.     {
1044.         nSize -= nRes;
1045.         pBuffer += nRes;
1046.     }
1047.
1048.     AMFProp_SetName(&prop, &name);
1049.     AMF_AddProp(obj, &prop);
1050. }
1051. else
1052. {
1053.     int nRes, i;
1054.     for (i = 0; i < cd.cd_num; i++) /* non-dynamic */
1055.     {
1056.         nRes = AMF3Prop_Decode(&prop, pBuffer, nSize, FALSE);
1057.         if (nRes == -1)
1058.             RTMP_Log(RTMP_LOGDEBUG, "%s, failed to decode AMF3 property!",
1059.                 __FUNCTION__);
1060.
1061.         AMFProp_SetName(&prop, AMF3CD_GetProp(&cd, i));
1062.         AMF_AddProp(obj, &prop);
1063.
1064.         pBuffer += nRes;
1065.         nSize -= nRes;
1066.     }
1067.     if (cd.cd_dynamic)
1068.     {
1069.         int len = 0;
1070.
1071.         do
1072.         {
1073.             nRes = AMF3Prop_Decode(&prop, pBuffer, nSize, TRUE);
1074.             AMF_AddProp(obj, &prop);
1075.
1076.             pBuffer += nRes;
1077.             nSize -= nRes;
1078.
1079.             len = prop.p_name.av_len;
1080.         }
1081.         while (len > 0);
1082.     }
1083. }
1084. RTMP_Log(RTMP_LOGDEBUG, "class object!");
1085. }
1086. return nOriginalSize - nSize;
1087. }
1088. //解AMF编码的Object数据类型
1089. int
1090. AMF_Decode(AMFObject *obj, const char *pBuffer, int nSize, int bDecodeName)
1091. {
1092.     int nOriginalSize = nSize;
1093.     int bError = FALSE; /* if there is an error while decoding - try to at least find the end mark AMF_OBJECT_END */
1094.
1095.     obj->o_num = 0;
1096.     obj->o_props = NULL;
1097.     while (nSize > 0)
1098.     {
1099.         AMFObjectProperty prop;
1100.         int nRes;
1101.
1102.         if (nSize >= 3 && AMF_DecodeInt24(pBuffer) == AMF_OBJECT_END)
1103.         {
1104.             nSize -= 3;
1105.             bError = FALSE;
1106.             break;
1107.         }
1108.
1109.         if (bError)
1110.         {
1111.             RTMP_Log(RTMP_LOGERROR,
1112.                 "DECODING ERROR, IGNORING BYTES UNTIL NEXT KNOWN PATTERN!");
1113.             nSize--;
1114.             pBuffer++;
1115.             continue;
1116.         }
1117.         //解Object里的Property
1118.         nRes = AMFProp_Decode(&prop, pBuffer, nSize, bDecodeName);
1119.         if (nRes == -1)
1120.             bError = TRUE;
1121.         else
1122.         {

```

```

1123.     nSize -= nRes;
1124.     pBuffer += nRes;
1125.     AMF_AddProp(obj, &prop);
1126. }
1127. }
1128.
1129. if (bError)
1130.     return -1;
1131.
1132. return nOriginalSize - nSize;
1133. }
1134.
1135. void
1136. AMF_AddProp(AMFObject *obj, const AMFObjectProperty *prop)
1137. {
1138.     if (!(obj->o_num & 0x0f))
1139.         obj->o_props = (AMFObjectProperty *)
1140.             realloc(obj->o_props, (obj->o_num + 16) * sizeof(AMFObjectProperty));
1141.     obj->o_props[obj->o_num++] = *prop;
1142. }
1143.
1144. int
1145. AMF_CountProp(AMFObject *obj)
1146. {
1147.     return obj->o_num;
1148. }
1149.
1150. AMFObjectProperty *
1151. AMF_GetProp(AMFObject *obj, const AVal *name, int nIndex)
1152. {
1153.     if (nIndex >= 0)
1154.     {
1155.         if (nIndex <= obj->o_num)
1156.             return &obj->o_props[nIndex];
1157.     }
1158.     else
1159.     {
1160.         int n;
1161.         for (n = 0; n < obj->o_num; n++)
1162.         {
1163.             if (AVMATCH(&obj->o_props[n].p_name, name))
1164.                 return &obj->o_props[n];
1165.         }
1166.     }
1167.
1168.     return (AMFObjectProperty *)&AMFProp_Invalid;
1169. }
1170.
1171. void
1172. AMF_Dump(AMFObject *obj)
1173. {
1174.     int n;
1175.     RTMP_Log(RTMP_LOGDEBUG, "(object begin)");
1176.     for (n = 0; n < obj->o_num; n++)
1177.     {
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>

版权声明：本文为博主原创文章，未经博主允许不得转载。<https://blog.csdn.net/leixiaohua1020/article/details/12954145>

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](#)

此PDF由spygg生成,请尊重原作者版权!!!

我的邮箱:liushidc@163.com