

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

2013年10月23日 15:46:12 阅读数：12019

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 \(NetStream部分 2\)](#)

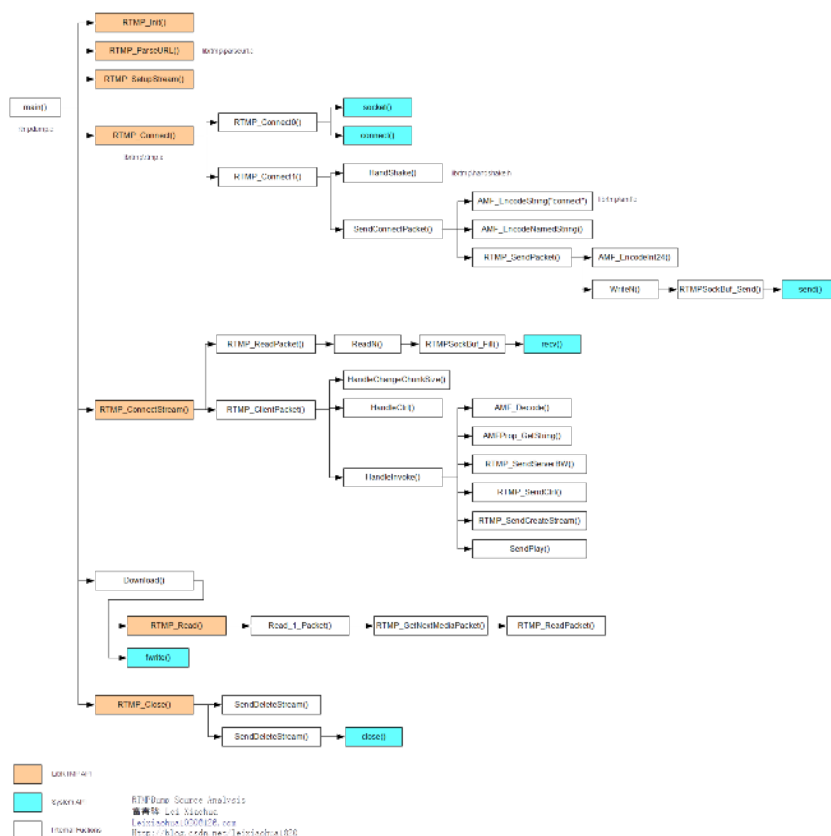
[RTMPdump \(libRTMP\) 源代码分析8：发送消息 \(Message\)](#)

[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

前一篇文章分析了RTMPdump (libRTMP) 的发送消息 (Message) 方面的源代码：[RTMPdump \(libRTMP\) 源代码分析 8：发送消息 \(Message\)](#)

ge)

在这里在研究接收消息（Message）的源代码，接收消息最典型的应用就是接收视音频数据了，因为视频和音频分别都属于RTMP协议规范中的一种消息。在这里主要分析接收视音频数据。

RTMPdump中完成视音频数据的接收（也可以说是视音频数据的下载）的函数是：RTMP_Read()。

RTMPdump主程序中的Download()函数就是通过调用RTMP_Read()完成数据接收，从而实现下载的。

那么我们马上开始吧，首先看看RTMP_Read()函数：

```
[cpp]    
1. //FLV文件头  
2. static const char flvHeader[] = { 'F', 'L', 'V', 0x01,  
3. 0x00, /* 0x04代表有音频, 0x01代表有视频 */  
4. 0x00, 0x00, 0x00, 0x09,  
5. 0x00, 0x00, 0x00, 0x00  
6. };  
7.  
8. #define HEADERBUF (128*1024)  
9. int  
10. RTMP_Read(RTMP *r, char *buf, int size)  
11. {  
12.     int nRead = 0, total = 0;  
13.  
14.     /* can't continue */  
15. fail:  
16.     switch (r->m_read.status) {  
17.     case RTMP_READ_EOF:  
18.     case RTMP_READ_COMPLETE:  
19.         return 0;  
20.     case RTMP_READ_ERROR: /* corrupted stream, resume failed */  
21.         SetSockError(EINVAL);  
22.         return -1;  
23.     default:  
24.         break;  
25.     }  
26.  
27.     /* first time thru */  
28.     if (!(r->m_read.flags & RTMP_READ_HEADER))  
29.     {  
30.         if (!(r->m_read.flags & RTMP_READ_RESUME))  
31.         {  
32.             //分配内存, 指向buf的首部和尾部  
33.             char *mybuf = (char *) malloc(HEADERBUF), *end = mybuf + HEADERBUF;  
34.             int cnt = 0;  
35.             //buf指向同一地址  
36.             r->m_read.buf = mybuf;  
37.             r->m_read buflen = HEADERBUF;  
38.  
39.             //把Flv的首部复制到mybuf指向的内存  
40.             //RTMP传递的多媒体数据是“砍头”的FLV文件  
41.             memcpy(mybuf, flvHeader, sizeof(flvHeader));  
42.             //m_read.buf指针后移flvHeader个单位  
43.             r->m_read.buf += sizeof(flvHeader);  
44.             //buf长度增加flvHeader长度  
45.             r->m_read buflen -= sizeof(flvHeader);  
46.             //timestamp=0, 不是多媒体数据  
47.             while (r->m_read.timestamp == 0)  
48.             {  
49.                 //读取一个Packet, 到r->m_read.buf  
50.                 //nRead为读取结果标记  
51.                 nRead = Read_1_Packet(r, r->m_read.buf, r->m_read buflen);  
52.                 //有错误  
53.                 if (nRead < 0)  
54.                 {  
55.                     free(mybuf);  
56.                     r->m_read.buf = NULL;  
57.                     r->m_read buflen = 0;  
58.                     r->m_read.status = nRead;  
59.                     goto fail;  
60.                 }  
61.                 /* buffer overflow, fix buffer and give up */  
62.                 if (r->m_read.buf < mybuf || r->m_read.buf > end) {  
63.                     mybuf = (char *) realloc(mybuf, cnt + nRead);  
64.                     memcpy(mybuf+cnt, r->m_read.buf, nRead);  
65.                     r->m_read.buf = mybuf+cnt+nRead;  
66.                     break;  
67.                 }  
68.                 //  
69.                 //记录读取的字节数  
70.                 cnt += nRead;  
71.                 //m_read.buf指针后移nRead个单位  
72.                 r->m_read.buf += nRead;  
73.                 r->m_read buflen -= nRead;  
74.                 //当dataType=00000101时, 即有视频和音频时  
75.                 //说明有多媒体数据了  
76.                 if (r->m_read.dataType == 5)  
77.                     break;
```

```

77.         break;
78.     }
79.     //读入数据类型
80.     //注意：mybuf指针位置一直没动
81.     //mybuf[4]中第 6 位表示是否存在音频Tag。第 8 位表示是否存在视频Tag。
82.     mybuf[4] = r->m_read.dataType;
83.     //两个指针之间的差
84.     r->m_read.buflen = r->m_read.buf - mybuf;
85.     r->m_read.buf = mybuf;
86.     //这句很重要！后面memcpy
87.     r->m_read.bufpos = mybuf;
88. }
89. //flags标明已经读完了文件头
90. r->m_read.flags |= RTMP_READ_HEADER;
91. }
92.
93. if ((r->m_read.flags & RTMP_READ_SEEKING) && r->m_read.buf)
94. {
95.     /* drop whatever's here */
96.     free(r->m_read.buf);
97.     r->m_read.buf = NULL;
98.     r->m_read.bufpos = NULL;
99.     r->m_read.buflen = 0;
100. }
101.
102. /* If there's leftover data buffered, use it up */
103. if (r->m_read.buf)
104. {
105.     nRead = r->m_read.buflen;
106.     if (nRead > size)
107.         nRead = size;
108.     //m_read.bufpos指向mybuf
109.     memcpy(buf, r->m_read.bufpos, nRead);
110.     r->m_read.buflen -= nRead;
111.     if (!r->m_read.buflen)
112.     {
113.         free(r->m_read.buf);
114.         r->m_read.buf = NULL;
115.         r->m_read.bufpos = NULL;
116.     }
117.     else
118.     {
119.         r->m_read.bufpos += nRead;
120.     }
121.     buf += nRead;
122.     total += nRead;
123.     size -= nRead;
124. }
125. //接着读
126. while (size > 0 && (nRead = Read_1_Packet(r, buf, size)) >= 0)
127. {
128.     if (!nRead) continue;
129.     buf += nRead;
130.     total += nRead;
131.     size -= nRead;
132.     break;
133. }
134. if (nRead < 0)
135.     r->m_read.status = nRead;
136.
137. if (size < 0)
138.     total += size;
139. return total;
140. }

```

程序关键的地方都已经注释上了代码，在此就不重复说明了。有一点要提一下：RTMP传送的视音频数据的格式和FLV（FLash Video）格式是一样的，把接收下来的数据直接存入文件就可以了。但是这些视音频数据没有文件头，是纯视音频数据，因此需要在其前面加上FLV格式的文件头，这样得到的数据存成文件后才能被一般的视频播放器所播放。FLV格式的文件头是13个字节，如代码中所示。

RTMP_Read()中实际读取数据的函数是Read_1_Packet()，它的功能是从网络上读取一个RTMPPacket的数据，来看看它的源代码吧：

```

[cpp]
1. /* 从流媒体中读取多媒体packet。
2.  * Returns -3 if Play.Close/Stop, -2 if fatal error, -1 if no more media
3.  * packets, 0 if ignorable error, >0 if there is a media packet
4.  */
5. static int
6. Read_1_Packet(RTMP *r, char *buf, unsigned int buflen)
7. {
8.     uint32_t prevTagSize = 0;
9.     int rtnGetNextMediaPacket = 0, ret = RTMP_READ_EOF;
10.    RTMPPacket packet = { 0 };
11.    int recopy = FALSE;
12.    unsigned int size;
13.    char *ptr, *pend;
14.    uint32_t nTimeStamp = 0;

```

```

15. unsigned int len;
16. //获取下一个packet
17. rtnGetNextMediaPacket = RTMP_GetNextMediaPacket(r, &packet);
18. while (rtnGetNextMediaPacket)
19. {
20.     char *packetBody = packet.m_body;
21.     unsigned int nPacketLen = packet.m_nBodySize;
22.
23.     /* Return -3 if this was completed nicely with invoke message
24.      * Play.Stop or Play.Complete
25.      */
26.     if (rtnGetNextMediaPacket == 2)
27.     {
28.         RTMP_Log(RTMP_LOGDEBUG,
29.             "Got Play.Complete or Play.Stop from server. "
30.             "Assuming stream is complete");
31.         ret = RTMP_READ_COMPLETE;
32.         break;
33.     }
34.     //设置dataType
35.     r->m_read.dataType |= (((packet.m_packetType == 0x08) << 2) |
36.         (packet.m_packetType == 0x09));
37.     //MessageID为9时, 为视频数据, 数据太小时。。。
38.     if (packet.m_packetType == 0x09 && nPacketLen <= 5)
39.     {
40.         RTMP_Log(RTMP_LOGDEBUG, "ignoring too small video packet: size: %d",
41.             nPacketLen);
42.         ret = RTMP_READ_IGNORE;
43.         break;
44.     }
45.     //MessageID为8时, 为音频数据, 数据太小时。。。
46.     if (packet.m_packetType == 0x08 && nPacketLen <= 1)
47.     {
48.         RTMP_Log(RTMP_LOGDEBUG, "ignoring too small audio packet: size: %d",
49.             nPacketLen);
50.         ret = RTMP_READ_IGNORE;
51.         break;
52.     }
53.
54.     if (r->m_read.flags & RTMP_READ_SEEKING)
55.     {
56.         ret = RTMP_READ_IGNORE;
57.         break;
58.     }
59. #ifdef _DEBUG
60.     RTMP_Log(RTMP_LOGDEBUG, "type: %02X, size: %d, TS: %d ms, abs TS: %d",
61.         packet.m_packetType, nPacketLen, packet.m_nTimeStamp,
62.         packet.m_hasAbsTimeStamp);
63.     if (packet.m_packetType == 0x09)
64.         RTMP_Log(RTMP_LOGDEBUG, "frametype: %02X", (*packetBody & 0xf0));
65. #endif
66.
67.     if (r->m_read.flags & RTMP_READ_RESUME)
68.     {
69.         /* check the header if we get one */
70.         //此类packet的timestamp都是0
71.         if (packet.m_nTimeStamp == 0)
72.         {
73.             //messageID=18, 数据消息 (AMF0)
74.             if (r->m_read.nMetaHeaderSize > 0
75.                 && packet.m_packetType == 0x12)
76.             {
77.                 //获取metadata
78.                 AMFObject metaObj;
79.                 int nRes =
80.                     AMF_Decode(&metaObj, packetBody, nPacketLen, FALSE);
81.                 if (nRes >= 0)
82.                 {
83.                     AVal metastring;
84.                     AMFProp_GetString(AMF_GetProp(&metaObj, NULL, 0),
85.                         &metastring);
86.
87.                     if (AVMATCH(&metastring, &av_onMetaData))
88.                     {
89.                         /* compare */
90.                         if ((r->m_read.nMetaHeaderSize != nPacketLen) ||
91.                             (memcmp
92.                                 (r->m_read.metaHeader, packetBody,
93.                                 r->m_read.nMetaHeaderSize) != 0))
94.                         {
95.                             ret = RTMP_READ_ERROR;
96.                         }
97.                     }
98.                     AMF_Reset(&metaObj);
99.                     if (ret == RTMP_READ_ERROR)
100.                         break;
101.                 }
102.             }
103.
104.             /* check first keyframe to make sure we got the right position
105.              * in the stream! (the first non ignored frame)
106.              */

```



```

198.         }
199.         else if (r->m_read.nResumeTS < ts)
200.         {
201.             /* the timestamp ts will only increase with
202.              * further packets, wait for seek
203.              */
204.             goto stopKeyframeSearch;
205.         }
206.     }
207.     pos += (11 + dataSize + 4);
208. }
209. if (ts < r->m_read.nResumeTS)
210. {
211.     RTMP_Log(RTMP_LOGERROR,
212.         "First packet does not contain keyframe, all "
213.         "timestamps are smaller than the keyframe "
214.         "timestamp; probably the resume seek failed?");
215. }
216. stopKeyframeSearch:
217. ;
218. if (!(r->m_read.flags & RTMP_READ_GOTFLVK))
219. {
220.     RTMP_Log(RTMP_LOGERROR,
221.         "Couldn't find the seeked keyframe in this chunk!");
222.     ret = RTMP_READ_IGNORE;
223.     break;
224. }
225. }
226. }
227. }
228.
229. if (packet.m_nTimeStamp > 0
230.     && (r->m_read.flags & (RTMP_READ_GOTKF|RTMP_READ_GOTFLVK)))
231. {
232.     /* another problem is that the server can actually change from
233.      * 09/08 video/audio packets to an FLV stream or vice versa and
234.      * our keyframe check will prevent us from going along with the
235.      * new stream if we resumed.
236.      *
237.      * in this case set the 'found keyframe' variables to true.
238.      * We assume that if we found one keyframe somewhere and were
239.      * already beyond TS > 0 we have written data to the output
240.      * which means we can accept all forthcoming data including the
241.      * change between 08/09 <-> FLV packets
242.      */
243.     r->m_read.flags |= (RTMP_READ_GOTKF|RTMP_READ_GOTFLVK);
244. }
245.
246. /* skip till we find our keyframe
247.  * (seeking might put us somewhere before it)
248.  */
249. if (!(r->m_read.flags & RTMP_READ_GOTKF) &&
250.     packet.m_packetType != 0x16)
251. {
252.     RTMP_Log(RTMP_LOGWARNING,
253.         "Stream does not start with requested frame, ignoring data... ");
254.     r->m_read.nIgnoredFrameCounter++;
255.     if (r->m_read.nIgnoredFrameCounter > MAX_IGNORED_FRAMES)
256.         ret = RTMP_READ_ERROR; /* fatal error, couldn't continue stream */
257.     else
258.         ret = RTMP_READ_IGNORE;
259.     break;
260. }
261. /* ok, do the same for FLV streams */
262. if (!(r->m_read.flags & RTMP_READ_GOTFLVK) &&
263.     packet.m_packetType == 0x16)
264. {
265.     RTMP_Log(RTMP_LOGWARNING,
266.         "Stream does not start with requested FLV frame, ignoring data... ");
267.     r->m_read.nIgnoredFlvFrameCounter++;
268.     if (r->m_read.nIgnoredFlvFrameCounter > MAX_IGNORED_FRAMES)
269.         ret = RTMP_READ_ERROR;
270.     else
271.         ret = RTMP_READ_IGNORE;
272.     break;
273. }
274.
275. /* we have to ignore the 0ms frames since these are the first
276.  * keyframes; we've got these so don't mess around with multiple
277.  * copies sent by the server to us! (if the keyframe is found at a
278.  * later position there is only one copy and it will be ignored by
279.  * the preceding if clause)
280.  */
281. if (!(r->m_read.flags & RTMP_READ_NO_IGNORE) &&
282.     packet.m_packetType != 0x16)
283. {
284.     /* exclude type 0x16 (FLV) since it can
285.      * contain several FLV packets */
286.     if (packet.m_nTimeStamp == 0)
287.     {
288.         ret = RTMP_READ_IGNORE;
289.         break;

```

```

289.     }
290.     else
291.     {
292.         /* stop ignoring packets */
293.         r->m_read.flags |= RTMP_READ_NO_IGNORE;
294.     }
295.     }
296. }
297.
298.     /* calculate packet size and allocate slop buffer if necessary */
299.     size = nPacketLen +
300.     (packet.m_packetType == 0x08 || packet.m_packetType == 0x09
301.     || packet.m_packetType == 0x12) ? 11 : 0) +
302.     (packet.m_packetType != 0x16 ? 4 : 0);
303.
304.     if (size + 4 > buflen)
305.     {
306.         /* the extra 4 is for the case of an FLV stream without a last
307.         * prevTagSize (we need extra 4 bytes to append it) */
308.         r->m_read.buf = (char *) malloc(size + 4);
309.         if (r->m_read.buf == 0)
310.         {
311.             RTMP_Log(RTMP_LOGERROR, "Couldn't allocate memory!");
312.             ret = RTMP_READ_ERROR;      /* fatal error */
313.             break;
314.         }
315.         recopy = TRUE;
316.         ptr = r->m_read.buf;
317.     }
318.     else
319.     {
320.         ptr = buf;
321.     }
322.     pend = ptr + size + 4;
323.
324.     /* use to return timestamp of last processed packet */
325.
326.     /* audio (0x08), video (0x09) or metadata (0x12) packets :
327.     * construct 11 byte header then add rtmp packet's data */
328.     if (packet.m_packetType == 0x08 || packet.m_packetType == 0x09
329.     || packet.m_packetType == 0x12)
330.     {
331.         nTimeStamp = r->m_read.nResumeTS + packet.m_nTimeStamp;
332.         prevTagSize = 11 + nPacketLen;
333.
334.         *ptr = packet.m_packetType;
335.         ptr++;
336.         ptr = AMF_EncodeInt24(ptr, pend, nPacketLen);
337.
338. #if 0
339.         if(packet.m_packetType == 0x09) { /* video */
340.
341.             /* H264 fix: */
342.             if((packetBody[0] & 0x0f) == 7) { /* CodecId = H264 */
343.                 uint8_t packetType = *(packetBody+1);
344.
345.                 uint32_t ts = AMF_DecodeInt24(packetBody+2); /* composition time */
346.                 int32_t cts = (ts+0xff800000)^0xff800000;
347.                 RTMP_Log(RTMP_LOGDEBUG, "cts : %d\n", cts);
348.
349.                 nTimeStamp -= cts;
350.                 /* get rid of the composition time */
351.                 CRTMP::EncodeInt24(packetBody+2, 0);
352.             }
353.             RTMP_Log(RTMP_LOGDEBUG, "VIDEO: nTimeStamp: 0x%08X (%d)\n", nTimeStamp, nTimeStamp);
354.         }
355. #endif
356.
357.         ptr = AMF_EncodeInt24(ptr, pend, nTimeStamp);
358.         *ptr = (char)((nTimeStamp & 0xFF000000) >> 24);
359.         ptr++;
360.
361.         /* stream id */
362.         ptr = AMF_EncodeInt24(ptr, pend, 0);
363.     }
364.
365.     memcpy(ptr, packetBody, nPacketLen);
366.     len = nPacketLen;
367.
368.     /* correct tagSize and obtain timestamp if we have an FLV stream */
369.     if (packet.m_packetType == 0x16)
370.     {
371.         unsigned int pos = 0;
372.         int delta;
373.
374.         /* grab first timestamp and see if it needs fixing */
375.         // nTimeStamp = AMF_DecodeInt24(packetBody + 4);
376.         // nTimeStamp |= (packetBody[7] << 24);
377.         // delta = packet.m_nTimeStamp - nTimeStamp;
378.
379.         while (pos + 11 < nPacketLen)

```

```

380.     {
381.         /* size without header (11) and without prevTagSize (4) */
382.         uint32_t dataSize = AMF_DecodeInt24(packetBody + pos + 1);
383.         nTimeStamp = AMF_DecodeInt24(packetBody + pos + 4);
384.         nTimeStamp |= (packetBody[pos + 7] << 24);
385.
386.         // if (delta)
387.         // {
388.         //     nTimeStamp += delta;
389.         //     AMF_EncodeInt24(ptr+pos+4, pend, nTimeStamp);
390.         //     ptr[pos+7] = nTimeStamp>>24;
391.         // }
392.
393.         /* set data type */
394.         r->m_read.dataType |= (((*(packetBody + pos) == 0x08) << 2) |
395.                                (*(packetBody + pos) == 0x09));
396.
397.         if (pos + 11 + dataSize + 4 > nPacketLen)
398.         {
399.             if (pos + 11 + dataSize > nPacketLen)
400.             {
401.                 RTMP_Log(RTMP_LOGERROR,
402.                     "Wrong data size (%lu), stream corrupted, aborting!",
403.                     dataSize);
404.                 ret = RTMP_READ_ERROR;
405.                 break;
406.             }
407.             RTMP_Log(RTMP_LOGWARNING, "No tagSize found, appending!");
408.
409.             /* we have to append a last tagSize! */
410.             prevTagSize = dataSize + 11;
411.             AMF_EncodeInt32(ptr + pos + 11 + dataSize, pend,
412.                 prevTagSize);
413.             size += 4;
414.             len += 4;
415.         }
416.         else
417.         {
418.             prevTagSize =
419.                 AMF_DecodeInt32(packetBody + pos + 11 + dataSize);
420.
421. #ifdef _DEBUG
422.             RTMP_Log(RTMP_LOGDEBUG,
423.                 "FLV Packet: type %02X, dataSize: %lu, tagSize: %lu, timeStamp: %lu ms",
424.                 (unsigned char)packetBody[pos], dataSize, prevTagSize,
425.                 nTimeStamp);
426. #endif
427.
428.             if (prevTagSize != (dataSize + 11))
429.             {
430. #ifdef _DEBUG
431.                 RTMP_Log(RTMP_LOGWARNING,
432.                     "Tag and data size are not consistent, writing tag size according to dataSize+11: %d",
433.                     dataSize + 11);
434. #endif
435.
436.                 prevTagSize = dataSize + 11;
437.                 AMF_EncodeInt32(ptr + pos + 11 + dataSize, pend,
438.                     prevTagSize);
439.             }
440.         }
441.
442.         pos += prevTagSize + 4; /*(11+dataSize+4); */
443.     }
444. }
445. ptr += len;
446.
447. if (packet.m_packetType != 0x16)
448. {
449.     /* FLV tag packets contain their own prevTagSize */
450.     AMF_EncodeInt32(ptr, pend, prevTagSize);
451. }
452.
453. /* In non-live this nTimeStamp can contain an absolute TS.
454.  * Update ext timestamp with this absolute offset in non-live mode
455.  * otherwise report the relative one
456.  */
457. /* RTMP_Log(RTMP_LOGDEBUG, "type: %02X, size: %d, pktTS: %dms, TS: %dms, bLiveStream: %d", packet.m_packetType, nPacketLen, packet.m_nTimeStamp, nTimeStamp, r->Link.lFlags & RTMP_LF_LIVE); */
458. r->m_read.timestamp = (r->Link.lFlags & RTMP_LF_LIVE) ? packet.m_nTimeStamp : nTimeStamp;
459.
460. ret = size;
461. break;
462. }
463.
464. if (rtnGetNextMediaPacket)
465.     RTMPPacket_Free(&packet);
466.
467. if (recopy)
468. {
469.     len = ret > buflen ? buflen : ret;

```



```

470.     memcpy(buf, r->m_read.buf, len);
471.     r->m_read.bufpos = r->m_read.buf + len;
472.     r->m_read buflen = ret - len;
473. }
474. return ret;
475. }

```

函数功能很多，重要的地方已经加上了注释，在此不再细分析。Read_1_Packet()里面实现从网络中读取视音频数据的函数是RTMP_GetNextMediaPacket()。下面我们来看看该函数的源代码：

```

1. int
2. RTMP_GetNextMediaPacket(RTMP *r, RTMPPacket *packet)
3. {
4.     int bHasMediaPacket = 0;
5.
6.     while (!bHasMediaPacket && RTMP_IsConnected(r)
7.           && RTMP_ReadPacket(r, packet))
8.     {
9.         if (!RTMPPacket_IsReady(packet))
10.        {
11.            continue;
12.        }
13.
14.        bHasMediaPacket = RTMP_ClientPacket(r, packet);
15.
16.        if (!bHasMediaPacket)
17.        {
18.            RTMPPacket_Free(packet);
19.        }
20.        else if (r->m_pausing == 3)
21.        {
22.            if (packet->m_nTimeStamp <= r->m_mediaStamp)
23.            {
24.                bHasMediaPacket = 0;
25.
26.                #ifdef _DEBUG
27.                RTMP_Log(RTMP_LOGDEBUG,
28.                    "Skipped type: %02X, size: %d, TS: %d ms, abs TS: %d, pause: %d ms",
29.                    packet->m_packetType, packet->m_nBodySize,
30.                    packet->m_nTimeStamp, packet->m_hasAbsTimestamp,
31.                    r->m_mediaStamp);
32.                #endif
33.                continue;
34.            }
35.            r->m_pausing = 0;
36.        }
37.
38.        if (bHasMediaPacket)
39.            r->m_bPlaying = TRUE;
40.        else if (r->m_sb.sb_timedout && !r->m_pausing)
41.            r->m_pauseStamp = r->m_channelTimestamp[r->m_mediaChannel];
42.
43.        return bHasMediaPacket;
44.    }

```

这里有两个函数比较重要：RTMP_ReadPacket()以及RTMP_ClientPacket()。这两个函数中，前一个函数负责从网络上读取数据，后一个负责处理数据。这部分与建立RTMP连接的网络流(NetStream)的时候很相似，参考：[RTMPdump \(libRTMP\) 源代码分析 6：建立一个流媒体连接 \(NetStream部分 1\)](#)

RTMP_ClientPacket()在前文中已经做过分析，在此不再重复叙述。在这里重点分析一下RTMP_ReadPacket()，来看看它的源代码。

```

1. //读取收下来的Chunk
2. int
3. RTMP_ReadPacket(RTMP *r, RTMPPacket *packet)
4. {
5.     //packet 存储读取完后的数据
6.     //Chunk Header最大值18
7.     uint8_t hbuf[RTMP_MAX_HEADER_SIZE] = { 0 };
8.     //header 指向的是从Socket中收下来的数据
9.     char *header = (char *)hbuf;
10.    int nSize, hSize, nToRead, nChunk;
11.    int didAlloc = FALSE;
12.
13.    RTMP_Log(RTMP_LOGDEBUG2, "%s: fd=%d", __FUNCTION__, r->m_sb.sb_socket);
14.    //收下来的数据存入hbuf
15.    if (ReadN(r, (char *)hbuf, 1) == 0)
16.    {
17.        RTMP_Log(RTMP_LOGERROR, "%s, failed to read RTMP packet header", __FUNCTION__);
18.        return FALSE;
19.    }
20.    //块类型fmt
21.    packet->m_headerType = (hbuf[0] & 0xc0) >> 6;
22.    //块流ID (2-63)
23.    packet->m_nChannel = (hbuf[0] & 0x3f);

```

```

24.     header++;
25.     //块流ID第1字节为0时, 块流ID占2个字节
26.     if (packet->m_nChannel == 0)
27.     {
28.         if (ReadN(r, (char *)&hbuf[1], 1) != 1)
29.         {
30.             RTMP_Log(RTMP_LOGERROR, "%s, failed to read RTMP packet header 2nd byte",
31.                 __FUNCTION__);
32.             return FALSE;
33.         }
34.         //计算块流ID (64-319)
35.         packet->m_nChannel = hbuf[1];
36.         packet->m_nChannel += 64;
37.         header++;
38.     }
39.     //块流ID第1字节为0时, 块流ID占3个字节
40.     else if (packet->m_nChannel == 1)
41.     {
42.         int tmp;
43.         if (ReadN(r, (char *)&hbuf[1], 2) != 2)
44.         {
45.             RTMP_Log(RTMP_LOGERROR, "%s, failed to read RTMP packet header 3nd byte",
46.                 __FUNCTION__);
47.             return FALSE;
48.         }
49.         tmp = (hbuf[2] << 8) + hbuf[1];
50.         //计算块流ID (64-65599)
51.         packet->m_nChannel = tmp + 64;
52.         RTMP_Log(RTMP_LOGDEBUG, "%s, m_nChannel: %0x", __FUNCTION__, packet->m_nChannel);
53.         header += 2;
54.     }
55.     //ChunkHeader的大小 (4种)
56.     nSize = packetSize[packet->m_headerType];
57.
58.     if (nSize == RTMP_LARGE_HEADER_SIZE) /* if we get a full header the timestamp is absolute */
59.         packet->m_hasAbsTimestamp = TRUE; //11字节的完整ChunkMsgHeader的TimeStamp是绝对值
60.
61.     else if (nSize < RTMP_LARGE_HEADER_SIZE)
62.     {
63.         /* using values from the last message of this channel */
64.         if (r->m_vecChannelsIn[packet->m_nChannel])
65.             memcpy(packet, r->m_vecChannelsIn[packet->m_nChannel],
66.                 sizeof(RTMPPacket));
67.     }
68.     nSize--;
69.
70.     if (nSize > 0 && ReadN(r, header, nSize) != nSize)
71.     {
72.         RTMP_Log(RTMP_LOGERROR, "%s, failed to read RTMP packet header. type: %x",
73.             __FUNCTION__, (unsigned int)hbuf[0]);
74.         return FALSE;
75.     }
76.
77.     hSize = nSize + (header - (char *)hbuf);
78.
79.     if (nSize >= 3)
80.     {
81.         //TimeStamp(注意 BigEndian to SmallEndian)(11, 7, 3字节首部都有)
82.         packet->m_nTimeStamp = AMF_DecodeInt24(header);
83.
84.         /*RTMP_Log(RTMP_LOGDEBUG, "%s, reading RTMP packet chunk on channel %x, headersz %i, timestamp %i, abs timestamp %i", __FUNCTION__,
85.             packet.m_nChannel, nSize, packet.m_nTimeStamp, packet.m_hasAbsTimestamp); */
86.         //消息长度(11, 7字节首部都有)
87.         if (nSize >= 6)
88.         {
89.             packet->m_nBodySize = AMF_DecodeInt24(header + 3);
90.             packet->m_nBytesRead = 0;
91.             RTMPPacket_Free(packet);
92.             //((11, 7字节首部都有)
93.             if (nSize > 6)
94.             {
95.                 //Msg type ID
96.                 packet->m_packetType = header[6];
97.                 //Msg Stream ID
98.                 if (nSize == 11)
99.                     packet->m_nInfoField2 = DecodeInt32LE(header + 7);
100.            }
101.            //Extend TimeStamp
102.            if (packet->m_nTimeStamp == 0xffffffff)
103.            {
104.                if (ReadN(r, header + nSize, 4) != 4)
105.                {
106.                    RTMP_Log(RTMP_LOGERROR, "%s, failed to read extended timestamp",
107.                        __FUNCTION__);
108.                    return FALSE;
109.                }
110.                packet->m_nTimeStamp = AMF_DecodeInt32(header + nSize);
111.                hSize += 4;
112.            }
113.        }

```

```

114.     RTMP_LogHexString(RTMP_LOGDEBUG2, (uint8_t *)hbuf, hSize);
115.
116.     if (packet->m_nBodySize > 0 && packet->m_body == NULL)
117.     {
118.         if (!RTMPPacket_Alloc(packet, packet->m_nBodySize))
119.         {
120.             RTMP_Log(RTMP_LOGDEBUG, "%s, failed to allocate packet", __FUNCTION__);
121.             return FALSE;
122.         }
123.         didAlloc = TRUE;
124.         packet->m_headerType = (hbuf[0] & 0xc0) >> 6;
125.     }
126.
127.     nToRead = packet->m_nBodySize - packet->m_nBytesRead;
128.     nChunk = r->m_inChunkSize;
129.     if (nToRead < nChunk)
130.         nChunk = nToRead;
131.
132.     /* Does the caller want the raw chunk? */
133.     if (packet->m_chunk)
134.     {
135.         packet->m_chunk->c_headerSize = hSize;
136.         memcpy(packet->m_chunk->c_header, hbuf, hSize);
137.         packet->m_chunk->c_chunk = packet->m_body + packet->m_nBytesRead;
138.         packet->m_chunk->c_chunkSize = nChunk;
139.     }
140.
141.     if (ReadN(r, packet->m_body + packet->m_nBytesRead, nChunk) != nChunk)
142.     {
143.         RTMP_Log(RTMP_LOGERROR, "%s, failed to read RTMP packet body. len: %lu",
144.             __FUNCTION__, packet->m_nBodySize);
145.         return FALSE;
146.     }
147.
148.     RTMP_LogHexString(RTMP_LOGDEBUG2, (uint8_t *)packet->m_body + packet->m_nBytesRead, nChunk);
149.
150.     packet->m_nBytesRead += nChunk;
151.
152.     /* keep the packet as ref for other packets on this channel */
153.     if (!r->m_vecChannelsIn[packet->m_nChannel])
154.         r->m_vecChannelsIn[packet->m_nChannel] = (RTMPPacket *) malloc(sizeof(RTMPPacket));
155.     memcpy(r->m_vecChannelsIn[packet->m_nChannel], packet, sizeof(RTMPPacket));
156.     //读取完毕
157.     if (RTMPPacket_IsReady(packet))
158.     {
159.         /* make packet's timestamp absolute */
160.         if (!packet->m_hasAbsTimestamp)
161.             packet->m_nTimeStamp += r->m_channelTimestamp[packet->m_nChannel]; /* timestamps seem to be always relative!! */
162.
163.         r->m_channelTimestamp[packet->m_nChannel] = packet->m_nTimeStamp;
164.
165.         /* reset the data from the stored packet. we keep the header since we may use it later if a new packet for this channel */
166.         /* arrives and requests to re-use some info (small packet header) */
167.         r->m_vecChannelsIn[packet->m_nChannel]->m_body = NULL;
168.         r->m_vecChannelsIn[packet->m_nChannel]->m_nBytesRead = 0;
169.         r->m_vecChannelsIn[packet->m_nChannel]->m_hasAbsTimestamp = FALSE; /* can only be false if we reuse header */
170.     }
171.     else
172.     {
173.         packet->m_body = NULL; /* so it won't be erased on free */
174.     }
175.
176.     return TRUE;
177. }
178.

```

函数代码看似很多，但是并不是很复杂，可以理解为在从事“简单重复性劳动”（和搬砖差不多）。基本上是一个字节一个字节的读取，然后按照RTMP协议规范进行解析。具体如何解析可以参考RTMP协议规范。

在RTMP_ReadPacket()函数里完成从Socket中读取数据的函数是ReadN()，继续看看它的源代码：

```

1. //从HTTP或SOCKET中读取数据
2. static int
3. ReadN(RTMP *r, char *buffer, int n)
4. {
5.     int nOriginalSize = n;
6.     int avail;
7.     char *ptr;
8.
9.     r->m_sb.sb_timedout = FALSE;
10.
11. #ifdef _DEBUG
12.     memset(buffer, 0, n);
13. #endif
14.
15.     ptr = buffer;
16.     while (n > 0)

```

```

17.     {
18.         int nBytes = 0, nRead;
19.         if (r->Link.protocol & RTMP_FEATURE_HTTP)
20.         {
21.             while (!r->m_resplen)
22.             {
23.                 if (r->m_sb.sb_size < 144)
24.                 {
25.                     if (!r->m_unackd)
26.                         HTTP_Post(r, RTMPT_IDLE, "", 1);
27.                     if (RTMPSockBuf_Fill(&r->m_sb) < 1)
28.                     {
29.                         if (!r->m_sb.sb_timedout)
30.                             RTMP_Close(r);
31.                         return 0;
32.                     }
33.                 }
34.                 HTTP_read(r, 0);
35.             }
36.             if (r->m_resplen && !r->m_sb.sb_size)
37.                 RTMPSockBuf_Fill(&r->m_sb);
38.             avail = r->m_sb.sb_size;
39.             if (avail > r->m_resplen)
40.                 avail = r->m_resplen;
41.         }
42.         else
43.         {
44.             avail = r->m_sb.sb_size;
45.             if (avail == 0)
46.             {
47.                 if (RTMPSockBuf_Fill(&r->m_sb) < 1)
48.                 {
49.                     if (!r->m_sb.sb_timedout)
50.                         RTMP_Close(r);
51.                     return 0;
52.                 }
53.                 avail = r->m_sb.sb_size;
54.             }
55.         }
56.         nRead = ((n < avail) ? n : avail);
57.         if (nRead > 0)
58.         {
59.             memcpy(ptr, r->m_sb.sb_start, nRead);
60.             r->m_sb.sb_start += nRead;
61.             r->m_sb.sb_size -= nRead;
62.             nBytes = nRead;
63.             r->m_nBytesIn += nRead;
64.             if (r->m_bSendCounter
65.                 && r->m_nBytesIn > r->m_nBytesInSent + r->m_nClientBW / 2)
66.                 SendBytesReceived(r);
67.         }
68.         /*RTMP_Log(RTMP_LOGDEBUG, "%s: %d bytes\n", __FUNCTION__, nBytes); */
69. #ifdef DEBUG
70.         fwrite(ptr, 1, nBytes, netstackdump_read);
71. #endif
72.
73.         if (nBytes == 0)
74.         {
75.             RTMP_Log(RTMP_LOGDEBUG, "%s, RTMP socket closed by peer", __FUNCTION__);
76.             /*goto again; */
77.             RTMP_Close(r);
78.             break;
79.         }
80.
81.         if (r->Link.protocol & RTMP_FEATURE_HTTP)
82.             r->m_resplen -= nBytes;
83.
84. #ifdef CRYPTO
85.         if (r->Link.rc4keyIn)
86.         {
87.             RC4_encrypt((RC4_KEY *)r->Link.rc4keyIn, nBytes, ptr);
88.         }
89. #endif
90.
91.         n -= nBytes;
92.         ptr += nBytes;
93.     }
94.
95.     return nOriginalSize - n;
96. }

```

ReadN()中实现从Socket中接收数据的函数是RTMPSockBuf_Fill(), 看看代码吧 (又是层层调用)。

```

1. //调用Socket编程中的recv () 函数，接收数据
2. int
3. RTMPSockBuf_Fill(RTMPSockBuf *sb)
4. {
5.     int nBytes;
6.
7.     if (!sb->sb_size)
8.         sb->sb_start = sb->sb_buf;
9.
10.    while (1)
11.    {
12.        //缓冲区长度：总长-未处理字节-已处理字节
13.        //|-----已处理-----|-----未处理-----|-----缓冲区-----
14.        //sb_buf      sb_start      sb_size
15.        nBytes = sizeof(sb->sb_buf) - sb->sb_size - (sb->sb_start - sb->sb_buf);
16.        #if defined(CRYPTO) && !defined(NO_SSL)
17.            if (sb->sb_ssl)
18.            {
19.                nBytes = TLS_read((SSL *)sb->sb_ssl, sb->sb_start + sb->sb_size, nBytes);
20.            }
21.            else
22.            #endif
23.            {
24.                //int recv( SOCKET s, char * buf, int len, int flags);
25.                //s：一个标识已连接套接口的描述字。
26.                //buf：用于接收数据的缓冲区。
27.                //len：缓冲区长度。
28.                //flags：指定调用方式。
29.                //从sb_start (待处理的下一字节) + sb_size () 还未处理的字节开始buffer为空，可以存储
30.                nBytes = recv(sb->sb_socket, sb->sb_start + sb->sb_size, nBytes, 0);
31.            }
32.            if (nBytes != -1)
33.            {
34.                //未处理的字节又多了
35.                sb->sb_size += nBytes;
36.            }
37.            else
38.            {
39.                int sockerr = GetSockError();
40.                RTMP_Log(RTMP_LOGDEBUG, "%s, recv returned %d. GetSockError(): %d (%s)",
41.                    __FUNCTION__, nBytes, sockerr, strerror(sockerr));
42.                if (sockerr == EINTR && !RTMP_ctrlC)
43.                    continue;
44.
45.                if (sockerr == EWOULDBLOCK || sockerr == EAGAIN)
46.                {
47.                    sb->sb_timedout = TRUE;
48.                    nBytes = 0;
49.                }
50.            }
51.            break;
52.        }
53.
54.        return nBytes;
55.    }

```

从RTMPSockBuf_Fill()代码中可以看出，调用了系统Socket的recv()函数接收RTMP连接传输过来的数据。

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/12971635>

文章标签：[rtmp](#) [rtmpdump](#) [源代码](#) [recv](#) [socket](#)

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