

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

2013年10月23日 15:03:32 阅读数：16506

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 \(NetStream部分 2\)](#)

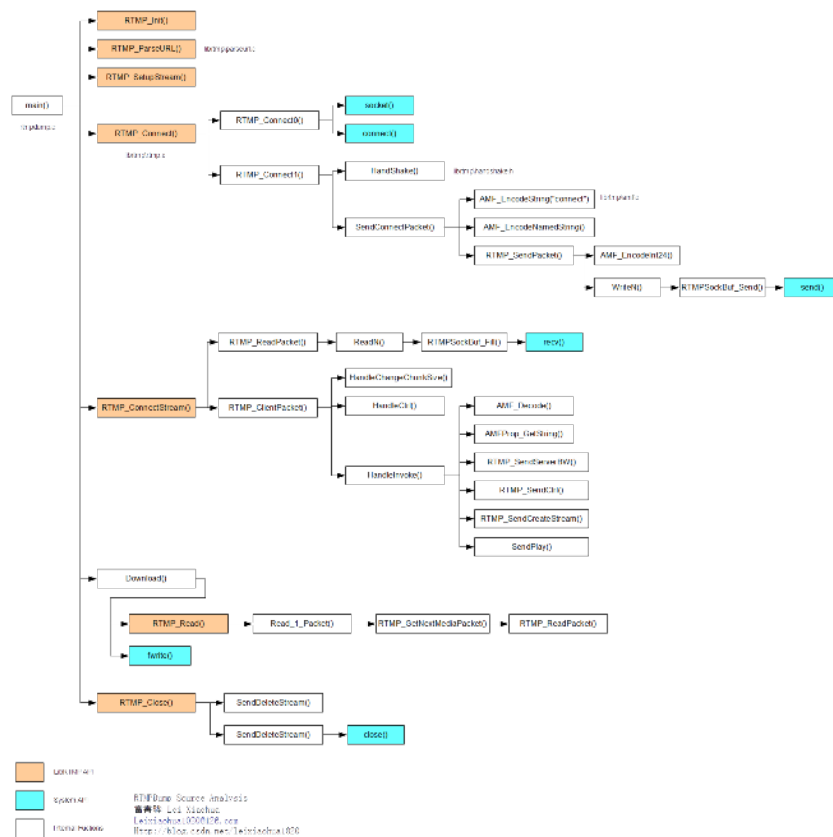
[RTMPdump \(libRTMP\) 源代码分析8：发送消息 \(Message\)](#)

[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

之前写了一系列的文章介绍RTMPDump各种函数。比如怎么建立网络连接（NetConnection），怎么建立网络流（NetStream）之类的，唯独没有介绍这些发送或接收

的数据，在底层到底是怎么实现的。本文就是要剖析一下其内部的实现。即这些消息（Message）到底是怎么发送和接收的。先来看看发送消息吧。

- 发送connect命令使用函数SendConnectPacket()
- 发送createstream命令使用RTMP_SendCreateStream()
- 发送releaseStream命令使用SendReleaseStream()
- 发送publish命令使用SendPublish()
- 发送deleteStream的命令使用SendDeleteStream()
- 发送pause命令使用RTMP_SendPause()

不再一一例举，发现函数命名有两种规律：RTMP_Send***()或者Send***()，其中*号代表命令的名称。

SendConnectPacket()这个命令是每次程序开始运行的时候发送的第一个命令消息，内容比较多，包含了很多AMF编码的内容，在此不多做分析，贴上代码：

```
[cpp]
1. //发送“connect”命令
2. static int
3. SendConnectPacket(RTMP *r, RTMPPacket *cp)
4. {
5.     RTMPPacket packet;
6.     char pbuf[4096], *pend = pbuf + sizeof(pbuf);
7.     char *enc;
8.
9.     if (cp)
10.         return RTMP_SendPacket(r, cp, TRUE);
11.
12.     packet.m_nChannel = 0x03; /* control channel (invoke) */
13.     packet.m_headerType = RTMP_PACKET_SIZE_LARGE;
14.     packet.m_packetType = 0x14; /* INVOKE */
15.     packet.m_nTimeStamp = 0;
16.     packet.m_nInfoField2 = 0;
17.     packet.m_hasAbsTimestamp = 0;
18.     packet.m_body = pbuf + RTMP_MAX_HEADER_SIZE;
19.
20.     enc = packet.m_body;
21.     enc = AMF_EncodeString(enc, pend, &av_connect);
22.     enc = AMF_EncodeNumber(enc, pend, ++r->m_numInvokes);
23.     *enc++ = AMF_OBJECT;
24.
25.     enc = AMF_EncodeNamedString(enc, pend, &av_app, &r->Link.app);
26.     if (!enc)
27.         return FALSE;
28.     if (r->Link.protocol & RTMP_FEATURE_WRITE)
29.     {
30.         enc = AMF_EncodeNamedString(enc, pend, &av_type, &av_nonprivate);
31.         if (!enc)
32.             return FALSE;
33.     }
34.     if (r->Link.flashVer.av_len)
35.     {
36.         enc = AMF_EncodeNamedString(enc, pend, &av_flashVer, &r->Link.flashVer);
37.         if (!enc)
38.             return FALSE;
39.     }
40.     if (r->Link.swfUrl.av_len)
41.     {
42.         enc = AMF_EncodeNamedString(enc, pend, &av_swfUrl, &r->Link.swfUrl);
43.         if (!enc)
44.             return FALSE;
45.     }
46.     if (r->Link.tcUrl.av_len)
47.     {
48.         enc = AMF_EncodeNamedString(enc, pend, &av_tcUrl, &r->Link.tcUrl);
49.         if (!enc)
50.             return FALSE;
51.     }
52.     if (!(r->Link.protocol & RTMP_FEATURE_WRITE))
53.     {
54.         enc = AMF_EncodeNamedBoolean(enc, pend, &av_fpad, FALSE);
55.         if (!enc)
56.             return FALSE;
57.         enc = AMF_EncodeNamedNumber(enc, pend, &av_capabilities, 15.0);
58.         if (!enc)
59.             return FALSE;
60.         enc = AMF_EncodeNamedNumber(enc, pend, &av_audioCodecs, r->m_fAudioCodecs);
61.         if (!enc)
62.             return FALSE;
63.         enc = AMF_EncodeNamedNumber(enc, pend, &av_videoCodecs, r->m_fVideoCodecs);
64.         if (!enc)
65.             return FALSE;
66.         enc = AMF_EncodeNamedNumber(enc, pend, &av_videoFunction, 1.0);
67.         if (!enc)
68.             return FALSE;
69.         if (r->Link.pageUrl.av_len)
```

```

70.     {
71.         enc = AMF_EncodeNamedString(enc, pend, &av_pageUrl, &r->Link.pageUrl);
72.         if (!enc)
73.             return FALSE;
74.     }
75. }
76. if (r->m_fEncoding != 0.0 || r->m_bSendEncoding)
77. { /* AMF0, AMF3 not fully supported yet */
78.     enc = AMF_EncodeNamedNumber(enc, pend, &av_objectEncoding, r->m_fEncoding);
79.     if (!enc)
80.         return FALSE;
81. }
82. if (enc + 3 >= pend)
83.     return FALSE;
84. *enc++ = 0;
85. *enc++ = 0; /* end of object - 0x00 0x00 0x09 */
86. *enc++ = AMF_OBJECT_END;
87.
88. /* add auth string */
89. if (r->Link.auth.av_len)
90. {
91.     enc = AMF_EncodeBoolean(enc, pend, r->Link.lFlags & RTMP_LF_AUTH);
92.     if (!enc)
93.         return FALSE;
94.     enc = AMF_EncodeString(enc, pend, &r->Link.auth);
95.     if (!enc)
96.         return FALSE;
97. }
98. if (r->Link.extras.o_num)
99. {
100.     int i;
101.     for (i = 0; i < r->Link.extras.o_num; i++)
102.     {
103.         enc = AMFProp_Encode(&r->Link.extras.o_props[i], enc, pend);
104.         if (!enc)
105.             return FALSE;
106.     }
107. }
108. packet.m_nBodySize = enc - packet.m_body;
109. //-----
110. r->dlg->AppendMLInfo(20,1,"命令消息","Connect");
111. //-----
112. return RTMP_SendPacket(r, &packet, TRUE);
113. }

```

RTMP_SendCreateStream()命令相对而言比较简单，代码如下：

```

[cpp]
1. //发送"createstream"命令
2. int
3. RTMP_SendCreateStream(RTMP *r)
4. {
5.     RTMPPacket packet;
6.     char pbuf[256], *pend = pbuf + sizeof(pbuf);
7.     char *enc;
8.
9.     packet.m_nChannel = 0x03; /* control channel (invoke) */
10.    packet.m_headerType = RTMP_PACKET_SIZE_MEDIUM;
11.    packet.m_packetType = 0x14; /* INVOKE */
12.    packet.m_nTimeStamp = 0;
13.    packet.m_nInfoField2 = 0;
14.    packet.m_hasAbsTimestamp = 0;
15.    packet.m_body = pbuf + RTMP_MAX_HEADER_SIZE;
16.
17.    enc = packet.m_body;
18.    enc = AMF_EncodeString(enc, pend, &av_createStream);
19.    enc = AMF_EncodeNumber(enc, pend, ++r->m_numInvokes);
20.    *enc++ = AMF_NULL; /* NULL */
21.
22.    packet.m_nBodySize = enc - packet.m_body;
23.    //-----
24.    r->dlg->AppendMLInfo(20,1,"命令消息","CreateStream");
25.    //-----
26.    return RTMP_SendPacket(r, &packet, TRUE);
27. }

```

同样，SendReleaseStream()内容也比较简单，我对其中部分内容作了注释：

```

1. //发送ReleaseStream命令
2. static int
3. SendReleaseStream(RTMP *r)
4. {
5.     RTMPPacket packet;
6.     char pbuf[1024], *pend = pbuf + sizeof(pbuf);
7.     char *enc;
8.
9.     packet.m_nChannel = 0x03; /* control channel (invoke) */
10.    packet.m_headerType = RTMP_PACKET_SIZE_MEDIUM;
11.    packet.m_packetType = 0x14; /* INVOKE */
12.    packet.m_nTimeStamp = 0;
13.    packet.m_nInfoField2 = 0;
14.    packet.m_hasAbsTimestamp = 0;
15.    packet.m_body = pbuf + RTMP_MAX_HEADER_SIZE;
16.
17.    enc = packet.m_body;
18.    //对"releaseStream"字符串进行AMF编码
19.    enc = AMF_EncodeString(enc, pend, &av_releaseStream);
20.    //对传输ID (0) 进行AMF编码?
21.    enc = AMF_EncodeNumber(enc, pend, ++r->m_numInvokes);
22.    //命令对象
23.    *enc++ = AMF_NULL;
24.    //对播放路径字符串进行AMF编码
25.    enc = AMF_EncodeString(enc, pend, &r->Link.playpath);
26.    if (!enc)
27.        return FALSE;
28.
29.    packet.m_nBodySize = enc - packet.m_body;
30.    //-----
31.    r->dlg->AppendMLInfo(20,1,"命令消息","ReleaseStream");
32.    //-----
33.    return RTMP_SendPacket(r, &packet, FALSE);
34. }

```

再来看一个SendPublish()函数，用于发送“publish”命令

```

1. //发送Publish命令
2. static int
3. SendPublish(RTMP *r)
4. {
5.     RTMPPacket packet;
6.     char pbuf[1024], *pend = pbuf + sizeof(pbuf);
7.     char *enc;
8.     //块流ID为4
9.     packet.m_nChannel = 0x04; /* source channel (invoke) */
10.    packet.m_headerType = RTMP_PACKET_SIZE_LARGE;
11.    //命令消息,类型20
12.    packet.m_packetType = 0x14; /* INVOKE */
13.    packet.m_nTimeStamp = 0;
14.    //流ID
15.    packet.m_nInfoField2 = r->m_stream_id;
16.    packet.m_hasAbsTimestamp = 0;
17.    packet.m_body = pbuf + RTMP_MAX_HEADER_SIZE;
18.    //指向Chunk的负载
19.    enc = packet.m_body;
20.    //对"publish"字符串进行AMF编码
21.    enc = AMF_EncodeString(enc, pend, &av_publish);
22.    enc = AMF_EncodeNumber(enc, pend, ++r->m_numInvokes);
23.    //命令对象为空
24.    *enc++ = AMF_NULL;
25.    enc = AMF_EncodeString(enc, pend, &r->Link.playpath);
26.    if (!enc)
27.        return FALSE;
28.
29.    /* FIXME: should we choose live based on Link.lFlags & RTMP_LF_LIVE? */
30.    enc = AMF_EncodeString(enc, pend, &av_live);
31.    if (!enc)
32.        return FALSE;
33.
34.    packet.m_nBodySize = enc - packet.m_body;
35.    //-----
36.    r->dlg->AppendMLInfo(20,1,"命令消息","Publish");
37.    //-----
38.    return RTMP_SendPacket(r, &packet, TRUE);
39. }

```

其他的命令不再一一例举，总体的思路是声明一个RTMPPacket类型的结构体，然后设置各种属性值，最后交给RTMP_SendPacket()进行发送。

RTMPPacket类型的结构体定义如下，一个RTMPPacket对应RTMP协议规范里面的一个块（Chunk）。

```

1. //Chunk信息
2. typedef struct RTMPPacket
3. {
4.     uint8_t m_headerType;//ChunkMsgHeader的类型 (4种)
5.     uint8_t m_packetType;//Message type ID (1-7协议控制;8, 9音视频;10以后为AMF编码消息)
6.     uint8_t m_hasAbsTimestamp; /* Timestamp 是绝对值还是相对值? */
7.     int m_nChannel; //块流ID
8.     uint32_t m_nTimeStamp; // Timestamp
9.     int32_t m_nInfoField2; /* last 4 bytes in a long header,消息流ID */
10.    uint32_t m_nBodySize; //消息长度
11.    uint32_t m_nBytesRead;
12.    RTMPPChunk *m_chunk;
13.    char *m_body;
14. } RTMPPacket;

```

下面我们来看看RTMP_SendPacket()吧, 各种的RTMPPacket (即各种Chunk) 都需要用这个函数进行发送。

```

1. //自己编一个数据报发送出去!
2. //非常常用
3. int
4. RTMP_SendPacket(RTMP *r, RTMPPacket *packet, int queue)
5. {
6.     const RTMPPacket *prevPacket = r->m_vecChannelsOut[packet->m_nChannel];
7.     uint32_t last = 0;
8.     int nSize;
9.     int hSize, cSize;
10.    char *header, *hptr, *hend, hbuf[RTMP_MAX_HEADER_SIZE], c;
11.    uint32_t t;
12.    char *buffer, *tbuf = NULL, *toff = NULL;
13.    int nChunkSize;
14.    int tlen;
15.    //不是完整ChunkMsgHeader
16.    if (prevPacket && packet->m_headerType != RTMP_PACKET_SIZE_LARGE)
17.    {
18.        /* compress a bit by using the prev packet's attributes */
19.        //获取ChunkMsgHeader的类型
20.        //前一个Chunk和这个Chunk对比
21.        if (prevPacket->m_nBodySize == packet->m_nBodySize
22.            && prevPacket->m_packetType == packet->m_packetType
23.            && packet->m_headerType == RTMP_PACKET_SIZE_MEDIUM)
24.            packet->m_headerType = RTMP_PACKET_SIZE_SMALL;
25.
26.        if (prevPacket->m_nTimeStamp == packet->m_nTimeStamp
27.            && packet->m_headerType == RTMP_PACKET_SIZE_SMALL)
28.            packet->m_headerType = RTMP_PACKET_SIZE_MINIMUM;
29.        //上一个packet的TimeStamp
30.        last = prevPacket->m_nTimeStamp;
31.    }
32.
33.    if (packet->m_headerType > 3) /* sanity */
34.    {
35.        RTMP_Log(RTMP_LOGERROR, "sanity failed!! trying to send header of type: 0x%02x.",
36.            (unsigned char)packet->m_headerType);
37.        return FALSE;
38.    }
39.
40.    //chunk包头大小; packetSize[] = { 12, 8, 4, 1 }
41.    nSize = packetSize[packet->m_headerType];
42.    hSize = nSize; cSize = 0;
43.    //相对的TimeStamp
44.    t = packet->m_nTimeStamp - last;
45.
46.    if (packet->m_body)
47.    {
48.        //Header的Start
49.        //m_body是指向负载数据首地址的指针; "-"号用于指针前移
50.        header = packet->m_body - nSize;
51.        //Header的End
52.        hend = packet->m_body;
53.    }
54.    else
55.    {
56.        header = hbuf + 6;
57.        hend = hbuf + sizeof(hbuf);
58.    }
59.    //当ChunkStreamID大于319时
60.    if (packet->m_nChannel > 319)
61.        //ChunkBasicHeader是3个字节
62.        cSize = 2;
63.    //当ChunkStreamID大于63时
64.    else if (packet->m_nChannel > 63)
65.        //ChunkBasicHeader是2个字节
66.        cSize = 1;
67.    if (cSize)
68.    {
69.        //header指针指向ChunkMsgHeader

```

```

70.     header -= cSize;
71.     //hSize加上ChunkBasicHeader的长度
72.     hSize += cSize;
73. }
74. //相对TimeStamp大于0xffffffff, 此时需要使用ExtendTimeStamp
75. if (nSize > 1 && t >= 0xffffffff)
76. {
77.     header -= 4;
78.     hSize += 4;
79. }
80.
81. hptr = header;
82. //把ChunkBasicHeader的Fmt类型左移6位
83. c = packet->m_headerType << 6;
84. switch (cSize)
85. {
86.     //把ChunkBasicHeader的低6位设置成ChunkStreamID
87.     case 0:
88.         c |= packet->m_nChannel;
89.         break;
90.     //同理, 但低6位设置成000000
91.     case 1:
92.         break;
93.     //同理, 但低6位设置成000001
94.     case 2:
95.         c |= 1;
96.         break;
97. }
98. //可以拆分成两句*hptr=c;hptr++, 此时hptr指向第2个字节
99. *hptr++ = c;
100. //CSize>0, 即ChunkBasicHeader大于1字节
101. if (cSize)
102. {
103.     //将要放到第2字节的内容tmp
104.     int tmp = packet->m_nChannel - 64;
105.     //获取低位存储与第2字节
106.     *hptr++ = tmp & 0xff;
107.     //ChunkBasicHeader是最大的3字节时
108.     if (cSize == 2)
109.     //获取高位存储于最后1个字节 (注意: 排序使用大端序列, 和主机相反)
110.     *hptr++ = tmp >> 8;
111. }
112. //ChunkMsgHeader。注意一共有4种, 包含的字段数不同。
113. //TimeStamp(3B)
114. if (nSize > 1)
115. {
116.     //相对TimeStamp和绝对TimeStamp?
117.     hptr = AMF_EncodeInt24(hptr, hend, t > 0xffffffff ? 0xffffffff : t);
118. }
119. //MessageLength+MessageTypeID(4B)
120. if (nSize > 4)
121. {
122.     //MessageLength
123.     hptr = AMF_EncodeInt24(hptr, hend, packet->m_nBodySize);
124.     //MessageTypeID
125.     *hptr++ = packet->m_packetType;
126. }
127. //MessageStreamID(4B)
128. if (nSize > 8)
129.     hptr += EncodeInt32LE(hptr, packet->m_nInfoField2);
130.
131. //ExtendedTimeStamp
132. if (nSize > 1 && t >= 0xffffffff)
133.     hptr = AMF_EncodeInt32(hptr, hend, t);
134. //负载长度, 指向负载的指针
135. nSize = packet->m_nBodySize;
136. buffer = packet->m_body;
137. //Chunk大小, 默认128字节
138. nChunkSize = r->m_outChunkSize;
139.
140. RTMP_Log(RTMP_LOGDEBUG2, "%s: fd=%d, size=%d", __FUNCTION__, r->m_sb.sb_socket,
141.     nSize);
142. /* send all chunks in one HTTP request */
143. //使用HTTP
144. if (r->Link.protocol & RTMP_FEATURE_HTTP)
145. {
146.     //nSize:Message负载长度; nChunkSize: Chunk长度;
147.     //例nSize: 307, nChunkSize:128;
148.     //可分为 (307+128-1) / 128=3个
149.     //为什么+nChunkSize-1? 因为除法会只取整数部分!
150.     int chunks = (nSize+nChunkSize-1) / nChunkSize;
151.     //Chunk个数超过一个
152.     if (chunks > 1)
153.     {
154.         //注意: CSize=1表示ChunkBasicHeader是2字节
155.         //消息分n块后总的开销:
156.         //n个ChunkBasicHeader, 1个ChunkMsgHeader, 1个Message负载
157.         //实际中只有第一个Chunk是完整的, 剩下的只有ChunkBasicHeader
158.         tlen = chunks * (cSize + 1) + nSize + hSize;
159.         //分配内存
160.         tbuf = (char *) malloc(tlen);
161.         if (!tbuf)

```

```

162.         return FALSE;
163.         toff = tbuf;
164.     }
165.     //消息的负载+头
166. }
167. while (nSize + hSize)
168. {
169.     int wrote;
170.     //消息负载<Chunk大小 (不用分块)
171.     if (nSize < nChunkSize)
172.     //Chunk可能小于设定值
173.     nChunkSize = nSize;
174.
175.     RTMP_LogHexString(RTMP_LOGDEBUG2, (uint8_t *)header, hSize);
176.     RTMP_LogHexString(RTMP_LOGDEBUG2, (uint8_t *)buffer, nChunkSize);
177.     if (tbuf)
178.     {
179.     //void *memcpy(void *dest, const void *src, int n);
180.     //由src指向地址为起始地址的连续n个字节的数据复制到以dest指向地址为起始地址的空间内
181.     memcpy(toff, header, nChunkSize + hSize);
182.     toff += nChunkSize + hSize;
183.     }
184.     else
185.     {
186.     wrote = WriteN(r, header, nChunkSize + hSize);
187.     if (!wrote)
188.     return FALSE;
189.     }
190.     //消息负载长度-Chunk负载长度
191.     nSize -= nChunkSize;
192.     //Buffer指针前移1个Chunk负载长度
193.     buffer += nChunkSize;
194.     hSize = 0;
195.
196.     //如果消息没有发完
197.     if (nSize > 0)
198.     {
199.     //ChunkBasicHeader
200.     header = buffer - 1;
201.     hSize = 1;
202.     if (cSize)
203.     {
204.         header -= cSize;
205.         hSize += cSize;
206.     }
207.     //ChunkBasicHeader第1个字节
208.     *header = (0xc0 | c);
209.     //ChunkBasicHeader大于1字节
210.     if (cSize)
211.     {
212.         int tmp = packet->m_nChannel - 64;
213.         header[1] = tmp & 0xff;
214.         if (cSize == 2)
215.             header[2] = tmp >> 8;
216.     }
217.     }
218.     }
219.     if (tbuf)
220.     {
221.     //
222.     int wrote = WriteN(r, tbuf, toff-tbuf);
223.     free(tbuf);
224.     tbuf = NULL;
225.     if (!wrote)
226.     return FALSE;
227.     }
228.
229.     /* we invoked a remote method */
230.     if (packet->m_packetType == 0x14)
231.     {
232.         AVal method;
233.         char *ptr;
234.         ptr = packet->m_body + 1;
235.         AMF_DecodeString(ptr, &method);
236.         RTMP_Log(RTMP_LOGDEBUG, "Invoking %s", method.av_val);
237.         /* keep it in call queue till result arrives */
238.         if (queue) {
239.             int txn;
240.             ptr += 3 + method.av_len;
241.             txn = (int)AMF_DecodeNumber(ptr);
242.             AV_queue(&r->m_methodCalls, &r->m_numCalls, &method, txn);
243.         }
244.     }
245.
246.     if (!r->m_vecChannelsOut[packet->m_nChannel])
247.         r->m_vecChannelsOut[packet->m_nChannel] = (RTMPPacket *) malloc(sizeof(RTMPPacket));
248.     memcpy(r->m_vecChannelsOut[packet->m_nChannel], packet, sizeof(RTMPPacket));
249.     return TRUE;
250. }

```

这个函数乍一看好像非常复杂，其实不然，他只是按照RTMP规范将数据编码成符合规范的块（Chunk），规范可以参考相关的文档。

具体怎么编码成块（Chunk）就不多分析了，在这里需要注意一个函数：WriteN()。该函数完成了将数据发送出去的功能。

来看一下WriteN()函数：

```
[cpp]
1. //发送数据的时候调用（连接，buffer，长度）
2. static int
3. WriteN(RTMP *r, const char *buffer, int n)
4. {
5.     const char *ptr = buffer;
6. #ifdef CRYPTO
7.     char *encrypted = 0;
8.     char buf[RTMP_BUFFER_CACHE_SIZE];
9.
10.    if (r->Link.rc4keyOut)
11.    {
12.        if (n > sizeof(buf))
13.            encrypted = (char *)malloc(n);
14.        else
15.            encrypted = (char *)buf;
16.        ptr = encrypted;
17.        RC4_encrypt2((RC4_KEY *)r->Link.rc4keyOut, n, buffer, ptr);
18.    }
19. #endif
20.
21.    while (n > 0)
22.    {
23.        int nBytes;
24.        //因方式的不同而调用不同函数
25.        //如果使用的是HTTP协议进行连接
26.        if (r->Link.protocol & RTMP_FEATURE_HTTP)
27.            nBytes = HTTP_Post(r, RTMPT_SEND, ptr, n);
28.        else
29.            nBytes = RTMPSockBuf_Send(&r->m_sb, ptr, n);
30.        /*RTMP_Log(RTMP_LOGDEBUG, "%s: %d\n", __FUNCTION__, nBytes); */
31.        //成功发送字节数<0
32.        if (nBytes < 0)
33.        {
34.            int sockerr = GetSockError();
35.            RTMP_Log(RTMP_LOGERROR, "%s, RTMP send error %d (%d bytes)", __FUNCTION__,
36.                sockerr, n);
37.
38.            if (sockerr == EINTR && !RTMP_ctrlC)
39.                continue;
40.
41.            RTMP_Close(r);
42.            n = 1;
43.            break;
44.        }
45.
46.        if (nBytes == 0)
47.            break;
48.
49.        n -= nBytes;
50.        ptr += nBytes;
51.    }
52.
53. #ifdef CRYPTO
54.     if (encrypted && encrypted != buf)
55.         free(encrypted);
56. #endif
57.
58.     return n == 0;
59. }
```

该函数中，RTMPSockBuf_Send()完成了数据发送的功能，再看看这个函数（函数调用真是好多啊。。。。）


```

1. //Socket发送 (指明套接字, buffer缓冲区, 数据长度)
2. //返回所发数据量
3. int
4. RTMPSockBuf_Send(RTMPSockBuf *sb, const char *buf, int len)
5. {
6.     int rc;
7.
8. #ifdef _DEBUG
9.     fwrite(buf, 1, len, netstackdump);
10. #endif
11.
12. #if defined(CRYPTO) && !defined(NO_SSL)
13.     if (sb->sb_ssl)
14.     {
15.         rc = TLS_write((SSL *)sb->sb_ssl, buf, len);
16.     }
17.     else
18. #endif
19.     {
20.         //向一个已连接的套接口发送数据。
21.         //int send( SOCKET s, const char * buf, int len, int flags);
22.         //s: 一个用于标识已连接套接口的描述字。
23.         //buf: 包含待发送数据的缓冲区。
24.         //len: 缓冲区中数据的长度。
25.         //flags: 调用执行方式。
26.         //rc: 所发数据量。
27.         rc = send(sb->sb_socket, buf, len, 0);
28.     }
29.     return rc;
30. }
31.
32. int
33. RTMPSockBuf_Close(RTMPSockBuf *sb)
34. {
35. #if defined(CRYPTO) && !defined(NO_SSL)
36.     if (sb->sb_ssl)
37.     {
38.         TLS_shutdown((SSL *)sb->sb_ssl);
39.         TLS_close((SSL *)sb->sb_ssl);
40.         sb->sb_ssl = NULL;
41.     }
42. #endif
43.     return closesocket(sb->sb_socket);
44. }

```

到这个函数的时候,发现一层的调用终于完成了,最后调用了系统Socket的send()函数完成了数据的发送功能。

之前贴过一张图总结这个过程,可能理解起来要方便一些: RTMPDump源代码分析 0: 主要函数调用分析

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/12958747>

文章标签: [rtmpdump](#) [rtmp](#) [消息](#) [源代码](#) [send](#)

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