

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

2013年10月23日 00:41:04 阅读数：12717

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 \(NetStream部分 2\)](#)

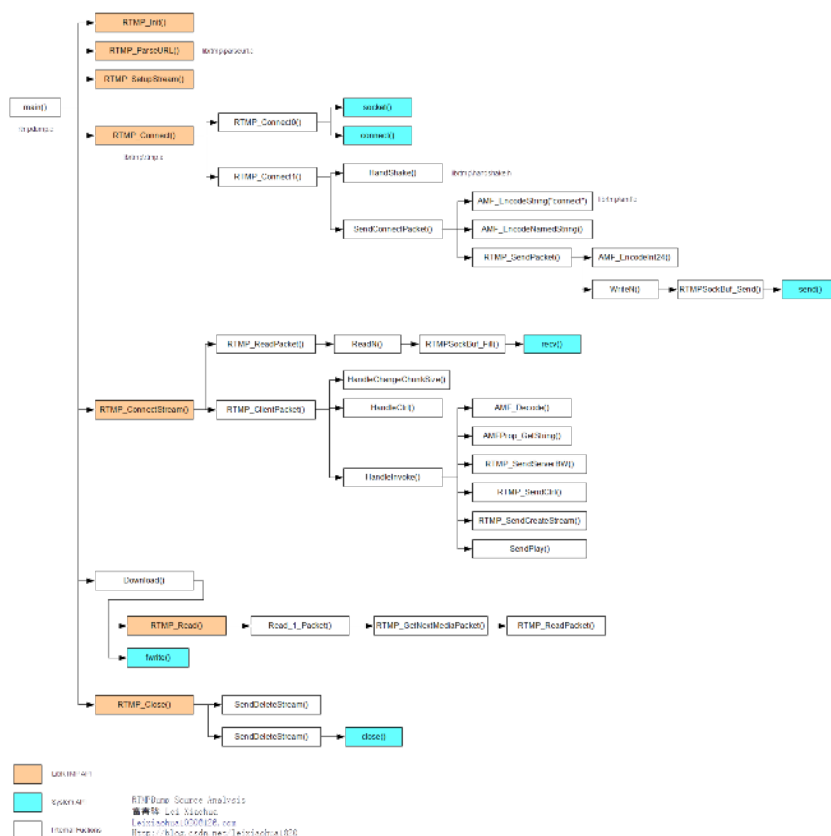
[RTMPdump \(libRTMP\) 源代码分析8：发送消息 \(Message\)](#)

[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

前文已经分析了 RTMPdump中建立一个NetConnection的过程：[RTMPdump 源代码分析 5：建立一个流媒体连接（NetConnection部分）](#)

多余的话不多说，下面先来看看RTMP_ConnectStream()，该函数主要用于在NetConnection基础上建立一个NetStream。

RTMP_ConnectStream()

```
[cpp]
1. //创建流
2. int
3. RTMP_ConnectStream(RTMP *r, int seekTime)
4. {
5.     RTMPPacket packet = { 0 };
6.
7.     /* seekTime was already set by SetupStream / SetupURL.
8.      * This is only needed by ReconnectStream.
9.      */
10.    if (seekTime > 0)
11.        r->Link.seekTime = seekTime;
12.
13.    r->m_mediaChannel = 0;
14.
15.    while (!r->m_bPlaying && RTMP_IsConnected(r) && RTMP_ReadPacket(r, &packet))
16.    {
17.        if (RTMPPacket_IsReady(&packet))
18.        {
19.            if (!packet.m_nBodySize)
20.                continue;
21.            if ((packet.m_packetType == RTMP_PACKET_TYPE_AUDIO) ||
22.                (packet.m_packetType == RTMP_PACKET_TYPE_VIDEO) ||
23.                (packet.m_packetType == RTMP_PACKET_TYPE_INFO))
24.            {
25.                RTMP_Log(RTMP_LOGWARNING, "Received FLV packet before play()! Ignoring.");
26.                RTMPPacket_Free(&packet);
27.                continue;
28.            }
29.            //处理Packet!
30.            //-----
31.            r->dlog->AppendCInfo("建立网络流：处理收到的数据。开始处理收到的数据");
32.            //-----
33.            RTMP_ClientPacket(r, &packet);
34.            //-----
35.            r->dlog->AppendCInfo("建立网络流：处理收到的数据。处理完毕，清除数据。");
36.            //-----
37.            RTMPPacket_Free(&packet);
38.        }
39.    }
40.
41.    return r->m_bPlaying;
42. }
```

乍一看，这个函数的代码量好像挺少的，实际上不然，其复杂度还是挺高的。我觉得比RTMP_Connect()要复杂不少。

其关键就在于这个While()循环。首先，循环的三个条件都满足，就能进行循环。只有出错或者建立网络流（NetStream）的步骤完成后，才能跳出循环。

在这个函数中有两个函数尤为重要：

RTMP_ReadPacket()

RTMP_ClientPacket()

第一个函数的作用是读取通过Socket接收下来的消息（Message）包，但是不做任何处理。第二个函数则是处理消息（Message），并做出响应。这两个函数结合，就可以完成接收消息然后响应消息的步骤。

下面来开一下RTMP_ReadPacket()：

```
[cpp]
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__, packet.m_nChannel, nSize, packet.m_nTimeStamp, packet.m_hasAbsTimestamp); */
85.     //消息长度(11, 7字节首部都有)
86.     if (nSize >= 6)
87.     {
88.         packet->m_nBodySize = AMF_DecodeInt24(header + 3);
89.         packet->m_nBytesRead = 0;
90.         RTMPPacket_Free(packet);
91.         //(11, 7字节首部都有)
92.         if (nSize > 6)
93.         {
94.             //Msg type ID
95.             packet->m_packetType = header[6];
96.             //Msg Stream ID
97.             if (nSize == 11)
98.                 packet->m_nInfoField2 = DecodeInt32LE(header + 7);
99.         }
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.
115. RTMP_LogHexString(RTMP_LOGDEBUG2, (uint8_t *)hbuf, hSize);
116.
117. if (packet->m_nBodySize > 0 && packet->m_body == NULL)
118. {
119.     if (!RTMPPacket_Alloc(packet, packet->m_nBodySize))
120.     {
121.         RTMP_Log(RTMP_LOGDEBUG, "%s, failed to allocate packet", __FUNCTION__);
122.         return FALSE;
123.     }
124.     didAlloc = TRUE;
125.     packet->m_headerType = (hbuf[0] & 0xc0) >> 6;
126. }
127.
128. nToRead = packet->m_nBodySize - packet->m_nBytesRead;
129. nChunk = r->m_inChunkSize;
130. if (nToRead < nChunk)
131.     nChunk = nToRead;
132.
133. /* Does the caller want the raw chunk? */
134. if (packet->m_chunk)
135. {
136.     packet->m_chunk->c_headerSize = hSize;
137.     memcpy(packet->m_chunk->c_header, hbuf, hSize);
138.     packet->m_chunk->c_chunk = packet->m_body + packet->m_nBytesRead;
139.     packet->m_chunk->c_chunkSize = nChunk;
140. }
141.
142. if (ReadN(r, packet->m_body + packet->m_nBytesRead, nChunk) != nChunk)
143. {
144.     RTMP_Log(RTMP_LOGERROR, "%s, failed to read RTMP packet body. len: %lu",
145.         __FUNCTION__, packet->m_nBodySize);
146.     return FALSE;
147. }
148.
149. RTMP_LogHexString(RTMP_LOGDEBUG2, (uint8_t *)packet->m_body + packet->m_nBytesRead, nChunk);
150.
151. packet->m_nBytesRead += nChunk;
152.
153. /* keep the packet as ref for other packets on this channel */
154. if (!r->m_vecChannelsIn[packet->m_nChannel])
155.     r->m_vecChannelsIn[packet->m_nChannel] = (RTMPPacket *) malloc(sizeof(RTMPPacket));
156. memcpy(r->m_vecChannelsIn[packet->m_nChannel], packet, sizeof(RTMPPacket));
157. //读取完毕
158. if (RTMPPacket_IsReady(packet))
159. {
160.     /* make packet's timestamp absolute */
161.     if (!packet->m_hasAbsTimestamp)
162.         packet->m_nTimeStamp += r->m_channelTimestamp[packet->m_nChannel]; /* timestamps seem to be always relative!! */
163.
164.     r->m_channelTimestamp[packet->m_nChannel] = packet->m_nTimeStamp;
165.
166.     /* reset the data from the stored packet. we keep the header since we may use it later if a new packet for this channel */
167.     /* arrives and requests to re-use some info (small packet header) */
168.     r->m_vecChannelsIn[packet->m_nChannel]->m_body = NULL;
169.     r->m_vecChannelsIn[packet->m_nChannel]->m_nBytesRead = 0;
170.     r->m_vecChannelsIn[packet->m_nChannel]->m_hasAbsTimestamp = FALSE; /* can only be false if we reuse header */
171. }
172. else
173. {
174.     packet->m_body = NULL; /* so it won't be erased on free */
175. }
176.
177. return TRUE;
178. }

```

在这里要注意的是，接收下来的实际上是块（Chunk）而不是消息（Message），因为消息（Message）在网络上传播的时候，实际上要分割成块（Chunk）。

这里解析的就是块（Chunk）

可参考：[RTMP规范简单分析](#)

具体的解析代码我就不多说了，直接参考RTMP协议规范就可以了，一个字节一个字节的解析就OK了。

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](#) [流媒体](#)

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