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

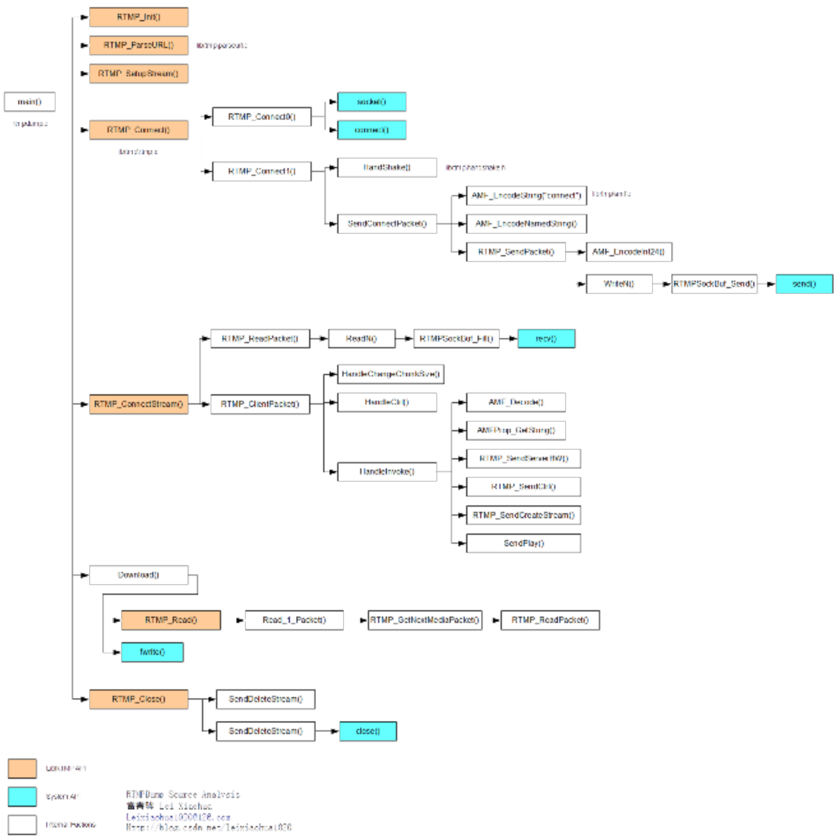
===

RTMPdump(libRTMP) 源代码分析系列文章：

===

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（Messa ge)

在这里在研究研究接收消息（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]

```cpp
//FLV文件头
static const char flvHeader[] = { 'F', 'L', 'V', 0x01,
  0x00,             /* 0x04代表有音频, 0x01代表有视频 */
  0x00, 0x00, 0x00, 0x09,
  0x00, 0x00, 0x00, 0x00
};

#define HEADERBUF   (128*1024)
int
RTMP_Read(RTMP *r, char *buf, int size)
{
  int nRead = 0, total = 0;

  /* can't continue */
fail:
  switch (r->m_read.status) {
  case RTMP_READ_EOF:
  case RTMP_READ_COMPLETE:
    return 0;
  case RTMP_READ_ERROR:  /* corrupted stream, resume failed */
    SetSockError(EINVAL);
    return -1;
  default:
    break;
  }

  /* first time thru */
  if (!(r->m_read.flags & RTMP_READ_HEADER))
    {
      if (!(r->m_read.flags & RTMP_READ_RESUME))
    {
    //分配内存，指向buf的首部和尾部
      char *mybuf = (char *) malloc(HEADERBUF), *end = mybuf + HEADERBUF;
      int cnt = 0;
      //buf指向同一地址
      r->m_read.buf = mybuf;
      r->m_read.buflen = HEADERBUF;

      //把Flv的首部复制到mybuf指向的内存
      //RTMP传递的多媒体数据是"砍头"的FLV文件
      memcpy(mybuf, flvHeader, sizeof(flvHeader));
      //m_read.buf指针后移flvheader个单位
      r->m_read.buf += sizeof(flvHeader);
      //buf长度增加flvheader长度
      r->m_read.buflen -= sizeof(flvHeader);
      //timestamp=0，不是多媒体数据
      while (r->m_read.timestamp == 0)
        {
        //读取一个Packet，到r->m_read.buf
        //nRead为读取结果标记
          nRead = Read_1_Packet(r, r->m_read.buf, r->m_read.buflen);
          //有错误
          if (nRead < 0)
        {
          free(mybuf);
          r->m_read.buf = NULL;
          r->m_read.buflen = 0;
          r->m_read.status = nRead;
          goto fail;
        }
          /* buffer overflow, fix buffer and give up */
          if (r->m_read.buf < mybuf || r->m_read.buf > end) {
            mybuf = (char *) realloc(mybuf, cnt + nRead);
        memcpy(mybuf+cnt, r->m_read.buf, nRead);
        r->m_read.buf = mybuf+cnt+nRead;
            break;
          }
          //
          //记录读取的字节数
          cnt += nRead;
          //m_read.buf指针后移nRead个单位
          r->m_read.buf += nRead;
          r->m_read.buflen -= nRead;
          //当dataType=00000101时，即有视频和音频时
          //说明有多媒体数据了
          if (r->m_read.dataType == 5)
            break;
```

```
                break;
            }
        //读入数据类型
        //注意：mybuf指针位置一直没动
        //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。
        mybuf[4] = r->m_read.dataType;
        //两个指针之间的差
        r->m_read.buflen = r->m_read.buf - mybuf;
        r->m_read.buf = mybuf;
        //这句很重要！后面memcopy
        r->m_read.bufpos = mybuf;
      }
        //flags标明已经读完了文件头
        r->m_read.flags |= RTMP_READ_HEADER;
      }

  if ((r->m_read.flags & RTMP_READ_SEEKING) && r->m_read.buf)
    {
      /* drop whatever's here */
      free(r->m_read.buf);
      r->m_read.buf = NULL;
      r->m_read.bufpos = NULL;
      r->m_read.buflen = 0;
    }

  /* If there's leftover data buffered, use it up */
  if (r->m_read.buf)
    {
      nRead = r->m_read.buflen;
      if (nRead > size)
    nRead = size;
      //m_read.bufpos指向mybuf
      memcpy(buf, r->m_read.bufpos, nRead);
      r->m_read.buflen -= nRead;
      if (!r->m_read.buflen)
    {
      free(r->m_read.buf);
      r->m_read.buf = NULL;
      r->m_read.bufpos = NULL;
    }
      else
    {
      r->m_read.bufpos += nRead;
    }
      buf += nRead;
      total += nRead;
      size -= nRead;
    }
  //接着读
  while (size > 0 && (nRead = Read_1_Packet(r, buf, size)) >= 0)
    {
      if (!nRead) continue;
      buf += nRead;
      total += nRead;
      size -= nRead;
      break;
    }
  if (nRead < 0)
    r->m_read.status = nRead;

  if (size < 0)
    total += size;
  return total;
}
```

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]

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

```
  unsigned int len;
  //获取下一个packet
  rtnGetNextMediaPacket = RTMP_GetNextMediaPacket(r, &packet);
  while (rtnGetNextMediaPacket)
    {
      char *packetBody = packet.m_body;
      unsigned int nPacketLen = packet.m_nBodySize;

      /* Return -3 if this was completed nicely with invoke message
       * Play.Stop or Play.Complete
       */
      if (rtnGetNextMediaPacket == 2)
    {
      RTMP_Log(RTMP_LOGDEBUG,
          "Got Play.Complete or Play.Stop from server. "
          "Assuming stream is complete");
      ret = RTMP_READ_COMPLETE;
      break;
    }
      //设置dataType
      r->m_read.dataType |= (((packet.m_packetType == 0x08) << 2) |
                 (packet.m_packetType == 0x09));
      //MessageID为9时，为视频数据，数据太小时。。。
      if (packet.m_packetType == 0x09 && nPacketLen <= 5)
    {
      RTMP_Log(RTMP_LOGDEBUG, "ignoring too small video packet: size: %d",
          nPacketLen);
      ret = RTMP_READ_IGNORE;
      break;
    }
      //MessageID为8时，为音频数据，数据太小时。。。
      if (packet.m_packetType == 0x08 && nPacketLen <= 1)
    {
      RTMP_Log(RTMP_LOGDEBUG, "ignoring too small audio packet: size: %d",
          nPacketLen);
      ret = RTMP_READ_IGNORE;
      break;
    }

      if (r->m_read.flags & RTMP_READ_SEEKING)
    {
      ret = RTMP_READ_IGNORE;
      break;
    }
#ifdef _DEBUG
      RTMP_Log(RTMP_LOGDEBUG, "type: %02X, size: %d, TS: %d ms, abs TS: %d",
      packet.m_packetType, nPacketLen, packet.m_nTimeStamp,
      packet.m_hasAbsTimestamp);
      if (packet.m_packetType == 0x09)
    RTMP_Log(RTMP_LOGDEBUG, "frametype: %02X", (*packetBody & 0xf0));
#endif

      if (r->m_read.flags & RTMP_READ_RESUME)
    {
      /* check the header if we get one */
        //此类packet的timestamp都是0
      if (packet.m_nTimeStamp == 0)
        {
        //messageID=18，数据消息（AMF0）
          if (r->m_read.nMetaHeaderSize > 0
          && packet.m_packetType == 0x12)
        {
        //获取metadata
          AMFObject metaObj;
          int nRes =
            AMF_Decode(&metaObj, packetBody, nPacketLen, FALSE);
          if (nRes >= 0)
            {
              AVal metastring;
              AMFProp_GetString(AMF_GetProp(&metaObj, NULL, 0),
                    &metastring);

              if (AVMATCH(&metastring, &av_onMetaData))
            {
              /* compare */
              if ((r->m_read.nMetaHeaderSize != nPacketLen) ||
                  (memcmp
                   (r->m_read.metaHeader, packetBody,
                r->m_read.nMetaHeaderSize) != 0))
                {
                  ret = RTMP_READ_ERROR;
                }
            }
              AMF_Reset(&metaObj);
              if (ret == RTMP_READ_ERROR)
            break;
            }
        }

          /* check first keyframe to make sure we got the right position
           * in the stream! (the first non ignored frame)
           */
```

```
          */
          if (r->m_read.nInitialFrameSize > 0)
        {
          /* video or audio data */
          if (packet.m_packetType == r->m_read.initialFrameType
              && r->m_read.nInitialFrameSize == nPacketLen)
            {
              /* we don't compare the sizes since the packet can
               * contain several FLV packets, just make sure the
               * first frame is our keyframe (which we are going
               * to rewrite)
               */
              if (memcmp
              (r->m_read.initialFrame, packetBody,
               r->m_read.nInitialFrameSize) == 0)
            {
              RTMP_Log(RTMP_LOGDEBUG, "Checked keyframe successfully!");
              r->m_read.flags |= RTMP_READ_GOTKF;
              /* ignore it! (what about audio data after it? it is
               * handled by ignoring all 0ms frames, see below)
               */
              ret = RTMP_READ_IGNORE;
              break;
            }
            }

          /* hande FLV streams, even though the server resends the
           * keyframe as an extra video packet it is also included
           * in the first FLV stream chunk and we have to compare
           * it and filter it out !!
           */
          //MessageID=22，聚合消息
          if (packet.m_packetType == 0x16)
            {
              /* basically we have to find the keyframe with the
               * correct TS being nResumeTS
               */
              unsigned int pos = 0;
              uint32_t ts = 0;

              while (pos + 11 < nPacketLen)
            {
              /* size without header (11) and prevTagSize (4) */
              uint32_t dataSize =
                AMF_DecodeInt24(packetBody + pos + 1);
              ts = AMF_DecodeInt24(packetBody + pos + 4);
              ts |= (packetBody[pos + 7] << 24);

#ifdef _DEBUG
              RTMP_Log(RTMP_LOGDEBUG,
                  "keyframe search: FLV Packet: type %02X, dataSize: %d, timeStamp: %d ms",
                  packetBody[pos], dataSize, ts);
#endif
              /* ok, is it a keyframe?:
               * well doesn't work for audio!
               */
              if (packetBody[pos /*6928, test 0 */ ] ==
                  r->m_read.initialFrameType
                  /* && (packetBody[11]&0xf0) == 0x10 */ )
                {
                  if (ts == r->m_read.nResumeTS)
                {
                  RTMP_Log(RTMP_LOGDEBUG,
                      "Found keyframe with resume-keyframe timestamp!");
                  if (r->m_read.nInitialFrameSize != dataSize
                      || memcmp(r->m_read.initialFrame,
                        packetBody + pos + 11,
                        r->m_read.
                        nInitialFrameSize) != 0)
                    {
                      RTMP_Log(RTMP_LOGERROR,
                      "FLV Stream: Keyframe doesn't match!");
                      ret = RTMP_READ_ERROR;
                      break;
                    }
                  r->m_read.flags |= RTMP_READ_GOTFLVK;

                  /* skip this packet?
                   * check whether skippable:
                   */
                  if (pos + 11 + dataSize + 4 > nPacketLen)
                    {
                      RTMP_Log(RTMP_LOGWARNING,
                      "Non skipable packet since it doesn't end with chunk, stream corrupt!");
                      ret = RTMP_READ_ERROR;
                      break;
                    }
                  packetBody += (pos + 11 + dataSize + 4);
                  nPacketLen -= (pos + 11 + dataSize + 4);

                  goto stopKeyframeSearch;
```

```c
              }
                else if (r->m_read.nResumeTS < ts)
              {
                /* the timestamp ts will only increase with
                 * further packets, wait for seek
                 */
                goto stopKeyframeSearch;
              }
              }
            pos += (11 + dataSize + 4);
          }
            if (ts < r->m_read.nResumeTS)
          {
            RTMP_Log(RTMP_LOGERROR,
                "First packet does not contain keyframe, all "
                "timestamps are smaller than the keyframe "
                "timestamp; probably the resume seek failed?");
          }
          stopKeyframeSearch:
            ;
            if (!(r->m_read.flags & RTMP_READ_GOTFLVK))
          {
            RTMP_Log(RTMP_LOGERROR,
                "Couldn't find the seeked keyframe in this chunk!");
            ret = RTMP_READ_IGNORE;
            break;
          }
          }
      }
      }

    if (packet.m_nTimeStamp > 0
        && (r->m_read.flags & (RTMP_READ_GOTKF|RTMP_READ_GOTFLVK)))
      {
        /* another problem is that the server can actually change from
         * 09/08 video/audio packets to an FLV stream or vice versa and
         * our keyframe check will prevent us from going along with the
         * new stream if we resumed.
         *
         * in this case set the 'found keyframe' variables to true.
         * We assume that if we found one keyframe somewhere and were
         * already beyond TS > 0 we have written data to the output
         * which means we can accept all forthcoming data including the
         * change between 08/09 <-> FLV packets
         */
        r->m_read.flags |= (RTMP_READ_GOTKF|RTMP_READ_GOTFLVK);
      }

    /* skip till we find our keyframe
     * (seeking might put us somewhere before it)
     */
    if (!(r->m_read.flags & RTMP_READ_GOTKF) &&
      packet.m_packetType != 0x16)
      {
        RTMP_Log(RTMP_LOGWARNING,
        "Stream does not start with requested frame, ignoring data... ");
        r->m_read.nIgnoredFrameCounter++;
        if (r->m_read.nIgnoredFrameCounter > MAX_IGNORED_FRAMES)
      ret = RTMP_READ_ERROR;  /* fatal error, couldn't continue stream */
        else
      ret = RTMP_READ_IGNORE;
        break;
      }
    /* ok, do the same for FLV streams */
    if (!(r->m_read.flags & RTMP_READ_GOTFLVK) &&
      packet.m_packetType == 0x16)
      {
        RTMP_Log(RTMP_LOGWARNING,
        "Stream does not start with requested FLV frame, ignoring data... ");
        r->m_read.nIgnoredFlvFrameCounter++;
        if (r->m_read.nIgnoredFlvFrameCounter > MAX_IGNORED_FRAMES)
      ret = RTMP_READ_ERROR;
        else
      ret = RTMP_READ_IGNORE;
        break;
      }

    /* we have to ignore the 0ms frames since these are the first
     * keyframes; we've got these so don't mess around with multiple
     * copies sent by the server to us! (if the keyframe is found at a
     * later position there is only one copy and it will be ignored by
     * the preceding if clause)
     */
    if (!(r->m_read.flags & RTMP_READ_NO_IGNORE) &&
      packet.m_packetType != 0x16)
      {           /* exclude type 0x16 (FLV) since it can
               * contain several FLV packets */
        if (packet.m_nTimeStamp == 0)
      {
        ret = RTMP_READ_IGNORE;
        break;
```

```
        }
          else
        {
          /* stop ignoring packets */
          r->m_read.flags |= RTMP_READ_NO_IGNORE;
        }
        }
    }

      /* calculate packet size and allocate slop buffer if necessary */
      size = nPacketLen +
    ((packet.m_packetType == 0x08 || packet.m_packetType == 0x09
      || packet.m_packetType == 0x12) ? 11 : 0) +
    (packet.m_packetType != 0x16 ? 4 : 0);

      if (size + 4 > buflen)
    {
      /* the extra 4 is for the case of an FLV stream without a last
       * prevTagSize (we need extra 4 bytes to append it) */
      r->m_read.buf = (char *) malloc(size + 4);
      if (r->m_read.buf == 0)
        {
          RTMP_Log(RTMP_LOGERROR, "Couldn't allocate memory!");
          ret = RTMP_READ_ERROR;        /* fatal error */
          break;
        }
      recopy = TRUE;
      ptr = r->m_read.buf;
    }
      else
    {
      ptr = buf;
    }
      pend = ptr + size + 4;

      /* use to return timestamp of last processed packet */

      /* audio (0x08), video (0x09) or metadata (0x12) packets :
       * construct 11 byte header then add rtmp packet's data */
      if (packet.m_packetType == 0x08 || packet.m_packetType == 0x09
      || packet.m_packetType == 0x12)
    {
      nTimeStamp = r->m_read.nResumeTS + packet.m_nTimeStamp;
      prevTagSize = 11 + nPacketLen;

      *ptr = packet.m_packetType;
      ptr++;
      ptr = AMF_EncodeInt24(ptr, pend, nPacketLen);

#if 0
        if(packet.m_packetType == 0x09) { /* video */

         /* H264 fix: */
         if((packetBody[0] & 0x0f) == 7) { /* CodecId = H264 */
         uint8_t packetType = *(packetBody+1);

         uint32_t ts = AMF_DecodeInt24(packetBody+2); /* composition time */
         int32_t cts = (ts+0xff800000)^0xff800000;
         RTMP_Log(RTMP_LOGDEBUG, "cts  : %d\n", cts);

         nTimeStamp -= cts;
         /* get rid of the composition time */
         CRTMP::EncodeInt24(packetBody+2, 0);
         }
         RTMP_Log(RTMP_LOGDEBUG, "VIDEO: nTimeStamp: 0x%08X (%d)\n", nTimeStamp, nTimeStamp);
         }
#endif

      ptr = AMF_EncodeInt24(ptr, pend, nTimeStamp);
      *ptr = (char)((nTimeStamp & 0xFF000000) >> 24);
      ptr++;

      /* stream id */
      ptr = AMF_EncodeInt24(ptr, pend, 0);
    }

      memcpy(ptr, packetBody, nPacketLen);
      len = nPacketLen;

      /* correct tagSize and obtain timestamp if we have an FLV stream */
      if (packet.m_packetType == 0x16)
    {
      unsigned int pos = 0;
      int delta;

      /* grab first timestamp and see if it needs fixing */
//    nTimeStamp = AMF_DecodeInt24(packetBody + 4);
    //  nTimeStamp |= (packetBody[7] << 24);
//    delta = packet.m_nTimeStamp - nTimeStamp;

      while (pos + 11 < nPacketLen)
```

```
        {
          /* size without header (11) and without prevTagSize (4) */
          uint32_t dataSize = AMF_DecodeInt24(packetBody + pos + 1);
          nTimeStamp = AMF_DecodeInt24(packetBody + pos + 4);
          nTimeStamp |= (packetBody[pos + 7] << 24);

//        if (delta)
//      {
//        nTimeStamp += delta;
//        AMF_EncodeInt24(ptr+pos+4, pend, nTimeStamp);
//        ptr[pos+7] = nTimeStamp>>24;
//      }

          /* set data type */
          r->m_read.dataType |= (((*(packetBody + pos) == 0x08) << 2) |
                     (*(packetBody + pos) == 0x09));

          if (pos + 11 + dataSize + 4 > nPacketLen)
        {
          if (pos + 11 + dataSize > nPacketLen)
            {
              RTMP_Log(RTMP_LOGERROR,
              "Wrong data size (%lu), stream corrupted, aborting!",
              dataSize);
              ret = RTMP_READ_ERROR;
              break;
            }
          RTMP_Log(RTMP_LOGWARNING, "No tagSize found, appending!");

          /* we have to append a last tagSize! */
          prevTagSize = dataSize + 11;
          AMF_EncodeInt32(ptr + pos + 11 + dataSize, pend,
                  prevTagSize);
          size += 4;
          len += 4;
        }
          else
        {
          prevTagSize =
            AMF_DecodeInt32(packetBody + pos + 11 + dataSize);

#ifdef _DEBUG
          RTMP_Log(RTMP_LOGDEBUG,
              "FLV Packet: type %02X, dataSize: %lu, tagSize: %lu, timeStamp: %lu ms",
              (unsigned char)packetBody[pos], dataSize, prevTagSize,
              nTimeStamp);
#endif

          if (prevTagSize != (dataSize + 11))
            {
#ifdef _DEBUG
              RTMP_Log(RTMP_LOGWARNING,
              "Tag and data size are not consitent, writing tag size according to dataSize+11: %d",
              dataSize + 11);
#endif

              prevTagSize = dataSize + 11;
              AMF_EncodeInt32(ptr + pos + 11 + dataSize, pend,
                      prevTagSize);
            }
        }

          pos += prevTagSize + 4;   /*(11+dataSize+4); */
        }
    }
      ptr += len;

      if (packet.m_packetType != 0x16)
    {
      /* FLV tag packets contain their own prevTagSize */
      AMF_EncodeInt32(ptr, pend, prevTagSize);
    }

      /* In non-live this nTimeStamp can contain an absolute TS.
       * Update ext timestamp with this absolute offset in non-live mode
       * otherwise report the relative one
       */
      /* 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); */
      r->m_read.timestamp = (r->Link.lFlags & RTMP_LF_LIVE) ? packet.m_nTimeStamp : nTimeStamp;

      ret = size;
      break;
    }

  if (rtnGetNextMediaPacket)
    RTMPPacket_Free(&packet);

  if (recopy)
    {
      len = ret > buflen ? buflen : ret;
```

```cpp
        memcpy(buf, r->m_read.buf, len);
        r->m_read.bufpos = r->m_read.buf + len;
        r->m_read.buflen = ret - len;
      }
    return ret;
  }
```

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：

```cpp
int
RTMP_GetNextMediaPacket(RTMP *r, RTMPPacket *packet)
{
  int bHasMediaPacket = 0;

  while (!bHasMediaPacket && RTMP_IsConnected(r)
     && RTMP_ReadPacket(r, packet))
    {
      if (!RTMPPacket_IsReady(packet))
    {
      continue;
    }

      bHasMediaPacket = RTMP_ClientPacket(r, packet);

      if (!bHasMediaPacket)
    {
      RTMPPacket_Free(packet);
    }
      else if (r->m_pausing == 3)
    {
      if (packet->m_nTimeStamp <= r->m_mediaStamp)
        {
          bHasMediaPacket = 0;
#ifdef _DEBUG
          RTMP_Log(RTMP_LOGDEBUG,
          "Skipped type: %02X, size: %d, TS: %d ms, abs TS: %d, pause: %d ms",
          packet->m_packetType, packet->m_nBodySize,
          packet->m_nTimeStamp, packet->m_hasAbsTimestamp,
          r->m_mediaStamp);
#endif
          continue;
        }
      r->m_pausing = 0;
    }
    }

  if (bHasMediaPacket)
    r->m_bPlaying = TRUE;
  else if (r->m_sb.sb_timedout && !r->m_pausing)
    r->m_pauseStamp = r->m_channelTimestamp[r->m_mediaChannel];

  return bHasMediaPacket;
}
```

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。

```cpp
//读取收下来的Chunk
int
RTMP_ReadPacket(RTMP *r, RTMPPacket *packet)
{
    //packet 存读取完后的的数据
    //Chunk Header最大值18
  uint8_t hbuf[RTMP_MAX_HEADER_SIZE] = { 0 };
    //header 指向的是从Socket中收下来的数据
  char *header = (char *)hbuf;
  int nSize, hSize, nToRead, nChunk;
  int didAlloc = FALSE;

  RTMP_Log(RTMP_LOGDEBUG2, "%s: fd=%d", __FUNCTION__, r->m_sb.sb_socket);
  //收下来的数据存入hbuf
  if (ReadN(r, (char *)hbuf, 1) == 0)
    {
      RTMP_Log(RTMP_LOGERROR, "%s, failed to read RTMP packet header", __FUNCTION__);
      return FALSE;
    }
  //块类型fmt
  packet->m_headerType = (hbuf[0] & 0xc0) >> 6;
  //块流ID (2-63)
  packet->m_nChannel = (hbuf[0] & 0x3f);
```

```
    header++;
    //块流ID第1字节为0时，块流ID占2个字节
    if (packet->m_nChannel == 0)
      {
        if (ReadN(r, (char *)&hbuf[1], 1) != 1)
      {
        RTMP_Log(RTMP_LOGERROR, "%s, failed to read RTMP packet header 2nd byte",
            __FUNCTION__);
        return FALSE;
      }
        //计算块流ID（64-319）
        packet->m_nChannel = hbuf[1];
        packet->m_nChannel += 64;
        header++;
      }
    //块流ID第1字节为0时，块流ID占3个字节
    else if (packet->m_nChannel == 1)
      {
        int tmp;
        if (ReadN(r, (char *)&hbuf[1], 2) != 2)
      {
        RTMP_Log(RTMP_LOGERROR, "%s, failed to read RTMP packet header 3nd byte",
            __FUNCTION__);
        return FALSE;
      }
        tmp = (hbuf[2] << 8) + hbuf[1];
        //计算块流ID（64-65599）
        packet->m_nChannel = tmp + 64;
        RTMP_Log(RTMP_LOGDEBUG, "%s, m_nChannel: %0x", __FUNCTION__, packet->m_nChannel);
        header += 2;
      }
    //ChunkHeader的大小（4种）
    nSize = packetSize[packet->m_headerType];

    if (nSize == RTMP_LARGE_HEADER_SIZE)  /* if we get a full header the timestamp is absolute */
      packet->m_hasAbsTimestamp = TRUE;    //11字节的完整ChunkMsgHeader的TimeStamp是绝对值

    else if (nSize < RTMP_LARGE_HEADER_SIZE)
      {               /* using values from the last message of this channel */
        if (r->m_vecChannelsIn[packet->m_nChannel])
      memcpy(packet, r->m_vecChannelsIn[packet->m_nChannel],
             sizeof(RTMPPacket));
      }

    nSize--;

    if (nSize > 0 && ReadN(r, header, nSize) != nSize)
      {
        RTMP_Log(RTMP_LOGERROR, "%s, failed to read RTMP packet header. type: %x",
        __FUNCTION__, (unsigned int)hbuf[0]);
        return FALSE;
      }

    hSize = nSize + (header - (char *)hbuf);

    if (nSize >= 3)
      {
      //TimeStamp(注意 BigEndian to SmallEndian)(11，7，3字节首部都有)
        packet->m_nTimeStamp = AMF_DecodeInt24(header);

        /*RTMP_Log(RTMP_LOGDEBUG, "%s, reading RTMP packet chunk on channel %x, headersz %i, timestamp %i, abs timestamp %i", __FUNCTIO
  , packet.m_nChannel, nSize, packet.m_nTimeStamp, packet.m_hasAbsTimestamp); */
      //消息长度(11，7字节首部都有)
        if (nSize >= 6)
      {
        packet->m_nBodySize = AMF_DecodeInt24(header + 3);
        packet->m_nBytesRead = 0;
        RTMPPacket_Free(packet);
      //(11，7字节首部都有)
        if (nSize > 6)
          {
            //Msg type ID
            packet->m_packetType = header[6];
            //Msg Stream ID
            if (nSize == 11)
          packet->m_nInfoField2 = DecodeInt32LE(header + 7);
          }
      }
        //Extend TimeStamp
        if (packet->m_nTimeStamp == 0xffffff)
      {
        if (ReadN(r, header + nSize, 4) != 4)
          {
            RTMP_Log(RTMP_LOGERROR, "%s, failed to read extended timestamp",
            __FUNCTION__);
            return FALSE;
          }
        packet->m_nTimeStamp = AMF_DecodeInt32(header + nSize);
        hSize += 4;
      }
      }
```

```

  RTMP_LogHexString(RTMP_LOGDEBUG2, (uint8_t *)hbuf, hSize);

  if (packet->m_nBodySize > 0 && packet->m_body == NULL)
    {
      if (!RTMPPacket_Alloc(packet, packet->m_nBodySize))
    {
      RTMP_Log(RTMP_LOGDEBUG, "%s, failed to allocate packet", __FUNCTION__);
      return FALSE;
    }
      didAlloc = TRUE;
      packet->m_headerType = (hbuf[0] & 0xc0) >> 6;
    }

  nToRead = packet->m_nBodySize - packet->m_nBytesRead;
  nChunk = r->m_inChunkSize;
  if (nToRead < nChunk)
    nChunk = nToRead;

  /* Does the caller want the raw chunk? */
  if (packet->m_chunk)
    {
      packet->m_chunk->c_headerSize = hSize;
      memcpy(packet->m_chunk->c_header, hbuf, hSize);
      packet->m_chunk->c_chunk = packet->m_body + packet->m_nBytesRead;
      packet->m_chunk->c_chunkSize = nChunk;
    }

  if (ReadN(r, packet->m_body + packet->m_nBytesRead, nChunk) != nChunk)
    {
      RTMP_Log(RTMP_LOGERROR, "%s, failed to read RTMP packet body. len: %lu",
      __FUNCTION__, packet->m_nBodySize);
      return FALSE;
    }

  RTMP_LogHexString(RTMP_LOGDEBUG2, (uint8_t *)packet->m_body + packet->m_nBytesRead, nChunk);

  packet->m_nBytesRead += nChunk;

  /* keep the packet as ref for other packets on this channel */
  if (!r->m_vecChannelsIn[packet->m_nChannel])
    r->m_vecChannelsIn[packet->m_nChannel] = (RTMPPacket *) malloc(sizeof(RTMPPacket));
  memcpy(r->m_vecChannelsIn[packet->m_nChannel], packet, sizeof(RTMPPacket));
  //读取完毕
  if (RTMPPacket_IsReady(packet))
    {
      /* make packet's timestamp absolute */
      if (!packet->m_hasAbsTimestamp)
    packet->m_nTimeStamp += r->m_channelTimestamp[packet->m_nChannel]; /* timestamps seem to be always relative!! */

      r->m_channelTimestamp[packet->m_nChannel] = packet->m_nTimeStamp;

      /* reset the data from the stored packet. we keep the header since we may use it later if a new packet for this channel */
      /* arrives and requests to re-use some info (small packet header) */
      r->m_vecChannelsIn[packet->m_nChannel]->m_body = NULL;
      r->m_vecChannelsIn[packet->m_nChannel]->m_nBytesRead = 0;
      r->m_vecChannelsIn[packet->m_nChannel]->m_hasAbsTimestamp = FALSE;   /* can only be false if we reuse header */
    }
  else
    {
      packet->m_body = NULL; /* so it won't be erased on free */
    }

  return TRUE;
}
```

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：

[cpp]

```cpp
//从HTTP或SOCKET中读取数据
static int
ReadN(RTMP *r, char *buffer, int n)
{
  int nOriginalSize = n;
  int avail;
  char *ptr;

  r->m_sb.sb_timedout = FALSE;

#ifdef _DEBUG
  memset(buffer, 0, n);
#endif

  ptr = buffer;
  while (n > 0)
```

```
  while (n > 0)
    {
      int nBytes = 0, nRead;
      if (r->Link.protocol & RTMP_FEATURE_HTTP)
        {
      while (!r->m_resplen)
        {
          if (r->m_sb.sb_size < 144)
            {
          if (!r->m_unackd)
            HTTP_Post(r, RTMPT_IDLE, "", 1);
          if (RTMPSockBuf_Fill(&r->m_sb) < 1)
            {
              if (!r->m_sb.sb_timedout)
                RTMP_Close(r);
              return 0;
            }
        }
          HTTP_read(r, 0);
        }
      if (r->m_resplen && !r->m_sb.sb_size)
        RTMPSockBuf_Fill(&r->m_sb);
          avail = r->m_sb.sb_size;
      if (avail > r->m_resplen)
        avail = r->m_resplen;
    }
      else
        {
          avail = r->m_sb.sb_size;
      if (avail == 0)
        {
          if (RTMPSockBuf_Fill(&r->m_sb) < 1)
            {
              if (!r->m_sb.sb_timedout)
                RTMP_Close(r);
              return 0;
        }
          avail = r->m_sb.sb_size;
        }
    }
      nRead = ((n < avail) ? n : avail);
      if (nRead > 0)
    {
      memcpy(ptr, r->m_sb.sb_start, nRead);
      r->m_sb.sb_start += nRead;
      r->m_sb.sb_size -= nRead;
      nBytes = nRead;
      r->m_nBytesIn += nRead;
      if (r->m_bSendCounter
          && r->m_nBytesIn > r->m_nBytesInSent + r->m_nClientBW / 2)
        SendBytesReceived(r);
    }
      /*RTMP_Log(RTMP_LOGDEBUG, "%s: %d bytes\n", __FUNCTION__, nBytes); */
#ifdef _DEBUG
      fwrite(ptr, 1, nBytes, netstackdump_read);
#endif

      if (nBytes == 0)
    {
      RTMP_Log(RTMP_LOGDEBUG, "%s, RTMP socket closed by peer", __FUNCTION__);
      /*goto again; */
      RTMP_Close(r);
      break;
    }

      if (r->Link.protocol & RTMP_FEATURE_HTTP)
    r->m_resplen -= nBytes;

#ifdef CRYPTO
      if (r->Link.rc4keyIn)
    {
      RC4_encrypt((RC4_KEY *)r->Link.rc4keyIn, nBytes, ptr);
    }
#endif

      n -= nBytes;
      ptr += nBytes;
    }

  return nOriginalSize - n;
}
```

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）。

[cpp]

```cpp
//调用Socket编程中的recv（）函数，接收数据
int
RTMPSockBuf_Fill(RTMPSockBuf *sb)
{
  int nBytes;

  if (!sb->sb_size)
    sb->sb_start = sb->sb_buf;

  while (1)
    {
    //缓冲区长度：总长-未处理字节-已处理字节
    //|-----已处理--------|-----未处理--------|---------缓冲区----------
    //sb_buf        sb_start    sb_size
      nBytes = sizeof(sb->sb_buf) - sb->sb_size - (sb->sb_start - sb->sb_buf);
#if defined(CRYPTO) && !defined(NO_SSL)
      if (sb->sb_ssl)
    {
      nBytes = TLS_read((SSL *)sb->sb_ssl, sb->sb_start + sb->sb_size, nBytes);
    }
      else
#endif
    {
    //int recv( SOCKET s, char * buf, int len, int flags);
    //s：一个标识已连接套接口的描述字。
    //buf：用于接收数据的缓冲区。
    //len：缓冲区长度。
    //flags：指定调用方式。
    //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
        nBytes = recv(sb->sb_socket, sb->sb_start + sb->sb_size, nBytes, 0);
    }
      if (nBytes != -1)
    {
    //未处理的字节又多了
      sb->sb_size += nBytes;
    }
      else
    {
      int sockerr = GetSockError();
      RTMP_Log(RTMP_LOGDEBUG, "%s, recv returned %d. GetSockError(): %d (%s)",
          __FUNCTION__, nBytes, sockerr, strerror(sockerr));
      if (sockerr == EINTR && !RTMP_ctrlC)
        continue;

      if (sockerr == EWOULDBLOCK || sockerr == EAGAIN)
        {
          sb->sb_timedout = TRUE;
          nBytes = 0;
        }
    }
      break;
    }

  return nBytes;
}
```

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

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