第 3 章

RTP 与 RTCP 流媒体协议

RTP 是针对 Internet 上多媒体数据流的一个传输协议，由 IETF(Internet 工程任务组)作为 RFC 1889 发布。RTP 被定义为在一对一或一对多的传输情况下工作，其目的是提供时间信息和实现流同步。RTP 的典型应用建立在 UDP 上，但也可以在 TCP 或 ATM 等其他协议之上工作。RTP 本身只保证实时数据的传输，并不能为按顺序传送数据包提供可靠的传送机制，也不提供流量控制或拥塞控制，它依靠 RTCP 提供这些服务。

多媒体应用的一个显著特点是数据量大，并且许多应用对实时性要求比较高。传统的 TCP 协议是一个面向连接的协议，它的重传机制和拥塞控制机制都不适用于实时多媒体传输。RTP 是一个应用型的传输层协议，它并不提供任何传输可靠性的保证和流量的拥塞控制机制。RTP 位于 UDP(User Datagram Protocol) 之上。UDP 虽然没有 TCP 那么可靠，并且无法保证实时业务的服务质量，需要 RTCP 实时监控数据传输和服务质量，但是，由于 UDP 的传输时延低于 TCP，能与声频和视频很好地配合，因此在实际应用中 RTP/ RTCP/ UDP 用于传输声频/视频媒体，而 TCP 用于数据和控制信令的传输。目前支持流媒体传输的协议主要有实时传输协议(Realtime Transport Protocol，RTP)、实时传输控制协议(Realtime Transport Control Protocol，RTCP)和实时流协议(Real Time Streaming Protocol，RTSP)等。RTP 标准可以参考 RFC 3550，网址为 https://datatracker.ietf.org/doc/html/rfc3550#section-5.3.1。

3.1 RTP

RTP 用来为语音、图像、传真等多种需要实时传输的多媒体数据提供端到端的实时传输服务。RTP 为 Internet 上端到端的实时传输提供时间信息和流同步，但并不保证服务质量，服务质量由 RTCP 来提供。RTP 用来提供实时传输，因而可以看成传输层的一个子层。流媒体应用中的一个典型的协议体系结构，如图 3-1 所示。

RTP 用于在单播或多播网络中传送实时数据，它们典型的应用场合有以下几个：

(1) 简单的多播声频会议。语音通信通过一个多播地址和一对端口实现。一个用于声频数据(RTP)，另一个用于控制包(RTCP)。

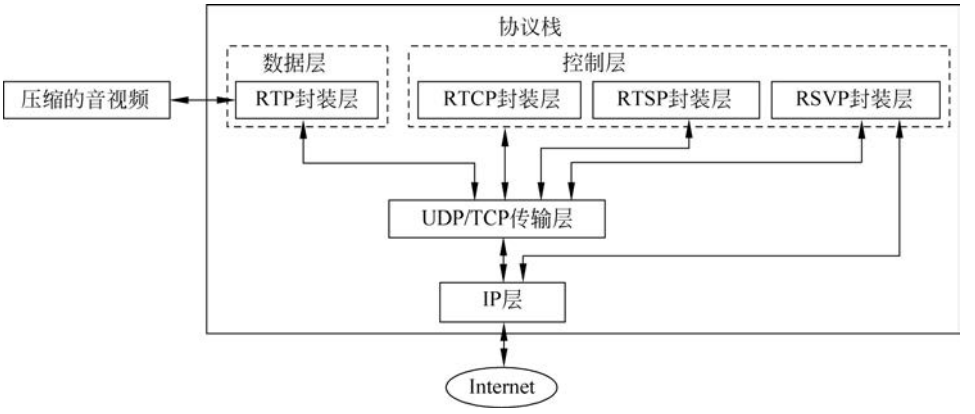

图 3-1　流媒体体系结构的协议栈

(2) 声频和视频会议。如果在一次会议中同时使用了声频和视频会议,这两种媒体将分别在不同的 RTP 会话中传送,每个会话使用不同的传输地址(IP 地址+端口)。如果一个用户同时使用了两个会话,则每个会话对应的 RTCP 包都使用规范化名字 CNAME (Canonical Name)。与会者可以根据 RTCP 包中的 CNAME 获取相关联的声频和视频,然后根据 RTCP 包中的计时协议相关信息(Network Time Protocol)实现声频和视频的同步。

(3) 翻译器和混合器。翻译器和混合器都是 RTP 级的中继系统。翻译器用在通过 IP 多播不能直接到达的用户区,例如发送者和接收者之间存在防火墙。当与会者能接收的声频编码格式不一样,例如有一个与会者通过一条低速链路接入到高速会议,这时就要使用混合器。在进入声频数据格式需要变化的网络前,混合器将来自一个源或多个源的声频包进行重构,并把重构后的多个声频合并,采用另一种声频编码进行编码后,再转发这个新的 RTP 包。从一个混合器出来的所有数据包要用混合器作为它们的同步源(SSRC)来识别,可以通过贡献源列表(CSRC)确认谈话者。

RTP 详细说明了在互联网上传递声频和视频的标准数据包格式。它一开始被设计为一个多播协议,但后来被用在很多单播应用中。RTP 常用于流媒体系统(配合 RTCP 或者 RTSP)。因为 RTP 自身具有时间戳(Timestamp),所以在 FFmpeg 中被用作一种 Format (封装格式)。

3.1.1　RTP 格式

RTP 格式,如图 3-2 所示。

RTP 的各个字段,解释如下。

(1) V: RTP 的版本号,占 2b,当前协议版本号为 2。

(2) P: 填充标志,占 1b,如果 P=1,则在该报文的尾部填充一个或多个额外的八位组,它们不是有效载荷的一部分。

(3) X: 扩展标志,占 1b,如果 X=1,则在 RTP 报头后跟有一个扩展报头。

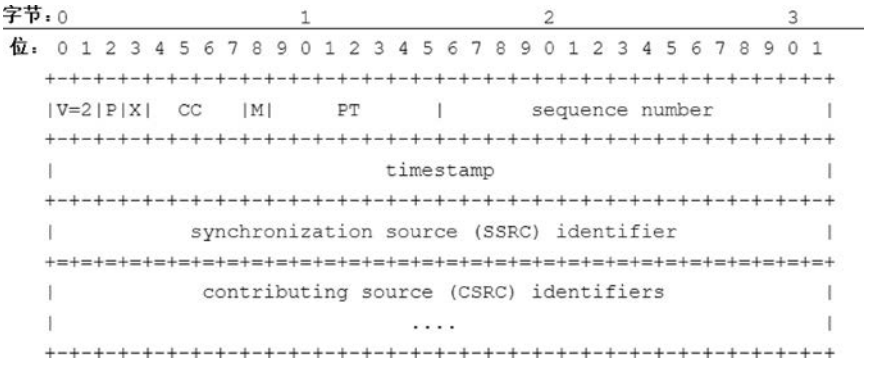

图 3-2　RTP 格式

(4) CC：CSRC 计数器，占 4b，指示 CSRC 标识符的个数。

(5) M：标记，占 1b，不同的有效载荷有不同的含义，对于视频，标记一帧的结束；对于声频，标记会话的开始。

(6) PT：有效荷载类型，占 7b，用于说明 RTP 报文中有效载荷的类型，如 GSM 声频、JPEG 图像等，在流媒体中大部分用来区分声频流和视频流，这样便于客户端进行解析。RTP 的负载类型，如表 3-1 所示。

表 3-1　RTP 负载类型

负载类型	编 码 类 型	A/V	时钟频率	声道数	参 考 文 档
0	PCMU	A	8000	1	[RFC 3551]
1	Reserved				
2	Reserved				
3	GSM	A	8000	1	[RFC 3551]
4	G723	A	8000	1	[Vineet_Kumar][RFC 3551]
5	DVI4	A	8000	1	[RFC 3551]
6	DVI4	A	16000	1	[RFC 3551]
7	LPC	A	8000	1	[RFC 3551]
8	PCMA	A	8000	1	[RFC 3551]
9	G722	A	8000	1	[RFC 3551]
10	L16	A	44100	2	[RFC 3551]
11	L16	A	44100	1	[RFC 3551]
12	QCELP	A	8000	1	[RFC 3551]
13	CN	A	8000	1	[RFC 3389]
14	MPA	A	90000		[RFC 3551][RFC 2250]
15	G728	A	8000	1	[RFC 3551]
16	DVI4	A	11025	1	[Joseph_Di_Pol]
17	DVI4	A	22050	1	[Joseph_Di_Pol]
18	G729	A	8000	1	[RFC 3551]

续表

负载类型	编码类型	A/V	时钟频率	声道数	参考文档
19	Reserved	A			
20	Unassigned	A			
21	Unassigned	A			
22	Unassigned	A			
23	Unassigned	A			
24	Unassigned	V			
25	CelB	V	90000		[RFC 2029]
26	JPEG	V	90000		[RFC 2435]
27	Unassigned	V			
28	nv	V	90000		[RFC 3551]
29	Unassigned	V			
30	Unassigned	V			
31	H261	V	90000		[RFC 4587]
32	MPV	V	90000		[RFC 2250]
33	MP2T	AV	90000		[RFC 2250]
34	H263	V	90000		[Chunrong_Zhu]
35-71	Unassigned	?			
72-76	Reserved for RTCP conflict avoidance				[RFC 3551]
77-95	Unassigned	?			
96-127	dynamic	?			[RFC 3551]

注意：基本的 A/V 列中的 A 表示声频，V 表示视频，? 表示未知。

(7) 序列号(Sequence Number)：占 16b，用于标识发送者所发送的 RTP 报文的序列号，每发送一个报文，序列号增 1。这个字段目前层的承载协议用 UDP 时，网络状况不好的时候可以用来检查丢包。同时在出现网络抖动的情况下可以用来对数据进行重新排序，序列号的初始值是随机的，同时声频包和视频包的 sequence 分别进行记数。

(8) 时间戳(Timestamp)：占 32b，必须使用 90kHz 时钟频率。时间戳反映了该 RTP 报文的第 1 个八位组的采样时刻。接收者使用时间戳来计算延迟和延迟抖动，并进行同步控制。

(9) 同步信源(SSRC)标识符：占 32b，用于标识同步信源。该标识符是随机选择的，参加同一视频会议的两个同步信源不能有相同的 SSRC。

(10) 贡献信源(CSRC)标识符：每个 CSRC 标识符占 32b，可以有 0～15 个。每个 CSRC 标识了包含在该 RTP 报文有效载荷中的所有贡献信源。

注意：基本的 RTP 说明并不定义任何头扩展，如果遇到 X=1，则需要特殊处理。

下面是一段示例码流，用十六进制显示如下：

```
80 e0 00 1e 00 00 d2 f0 00 00 00 00 41 9b 6b 49 ?....??....A?kI
e1 0f 26 53 02 1a ff06 59 97 1d d2 2e 8c 50 01 ?.&S....Y?.?.?P.
cc 13 ec 52 77 4e e50e 7b fd 16 11 66 27 7c b4 ?.?RwN?.{?..f'|?
f6 e1 29 d5 d6 a4 ef3e 12 d8 fd 6c 97 51 e7 e9 ??)????>.??l?Q??
cfc7 5e c8 a9 51 f6 82 65 d6 48 5a 86 b0 e0 8c ??^??Q??e?HZ????
```

各字节的值及对应的字段，如表 3-2 所示。

表 3-2　RTP 示例码流的值及对应字段

十六进制	对应字段	十六进制	对应字段
80	V、P、X、CC	00 00 d2 f0	Timestamp
e0	M、PT	00 00 00 00	SSRC
00 1e	SequenceNumber		

把前两字节(0x80e0)换成二进制，即 1000 0000 1110 0000，按顺序解释，如表 3-3 所示。

表 3-3　示例码流的前两字节的二进制及对应字段

前两字节的二进制	对应字段
10	V，即版本是 2
0	P，填充标志，0 表示没有
0	X，扩展标志，0 表示没有
0000	CC，CSRC 计数器
1	M，标记位；对于视频，标记一帧的结束
110 0000	PT，有效荷载类型，这里的值为 96，代表 H.264

3.1.2　RTP 封装 H.264

RTP 荷载格式定义了 3 种不同的基本荷载结构，接收者可以通过 RTP 荷载的第 1 字节的后 5 位识别荷载结构，如图 3-3 所示。

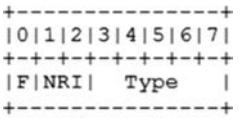

图 3-3　RTP 荷载第 1 字节的字段结构

(1) 单个 H.264 的 NAL 单元包：荷载中只包含一个 NAL 单元。NAL 头类型域等于原始 NAL 单元类型，即在范围 1 到 23 之间。

(2) 聚合包：本类型用于将多个 NAL 单元聚合到单个 RTP 荷载中。本包有 4 个版本，包括单时间聚合包类型 A(STAP-A)、单时间聚合包类型 B(STAP-B)、多时间聚合包类

型(MTAP)16 位位移(MTAP16)和多时间聚合包类型(MTAP)24 位位移(MTAP24)。赋予 STAP-A、STAP-B、MTAP16、MTAP24 的 NAL 单元类型号分别是 24、25、26、27。

(3) 分片单元：用于将单个 NAL 单元分片到多个 RTP 包。现存两个版本 FU-A 和 FU-B，分别用 NAL 单元类型 28 和 29 标识。

常用的打包时的分包规则是：如果小于 MTU，则采用单个 NAL 单元包，如果大于 MTU，则采用分片方式(FUs)。因为常用的打包方式是单个 NAL 包和 FU-A 方式，这里只解析这两种。

1. 单个 NAL 单元包

定义在此的 NAL 单元包必须只包含一个。这意味着聚合包和分片单元不可以用在单个 NAL 单元包中，并且 RTP 序号必须符合 NAL 单元的解码顺序。NAL 单元的第一字节和 RTP 荷载头的第 1 字节重合，如图 3-4 所示。打包 H.264 码流时，只需要在帧前面加上 12 字节的 RTP 头。

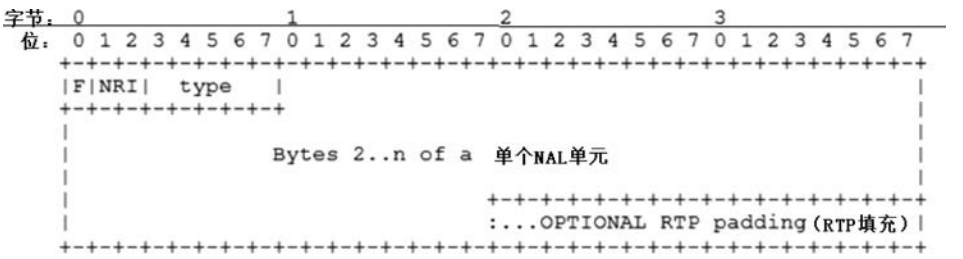

图 3-4 RTP 荷载单个 NAL 单元

2. 分片单元(FU-A)

分片只定义于单个 NAL 单元，不用于任何聚合包。NAL 单元的一个分片由整数个连续 NAL 单元字节组成。每个 NAL 单元字节必须正好是该 NAL 单元一个分片的一部分。相同 NAL 单元的分片必须使用递增的 RTP 序号连续按顺序发送，即第一和最后分片之间没有其他的 RTP 包。类似地，NAL 单元必须按照 RTP 顺序号的顺序装载。

当一个 NAL 单元被分片运送在分片单元(FUs)中时，被引用为分片 NAL 单元。STAPs 和 MTAPs 不可以被分片。FUs 不可以嵌套，即一个 FU 不可以包含另一个 FU。运送 FU 的 RTP 时戳被设置成分片 NAL 单元的 NALU 时刻。

FU-A 的 RTP 荷载格式，如图 3-5 所示。FU-A 由 1 字节的分片单元指示(FU indicator，如图 3-3 所示)、1 字节的分片单元头(FU header，如图 3-6 所示)和分片单元荷载(FU payload)组成。

RTP 的分片单元头(FU-A header)的各个字段(如图 3-6 所示)，解释如下。

(1) S：长度为 1b，当设置成 1 时，开始位指示分片 NAL 单元的开始。当跟随的 FU 荷载不是分片 NAL 单元荷载的开始时，开始位设为 0。

(2) E：长度为 1b，当设置成 1 时，结束位指示分片 NAL 单元的结束，即荷载的最后字节也是分片 NAL 单元的最后一字节。当跟随的 FU 荷载不是分片 NAL 单元的最后分片

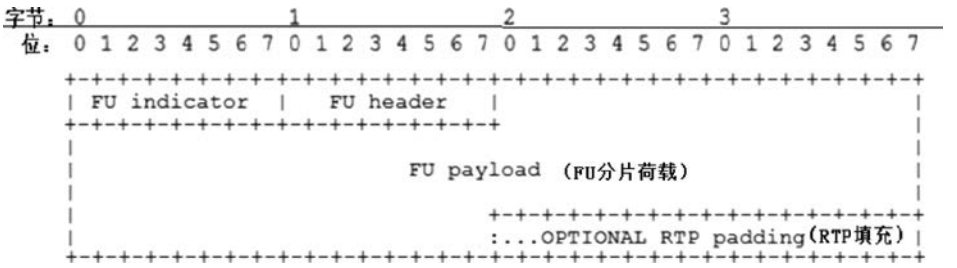

图 3-5　RTP 荷载的分片单元格式(FU-A)

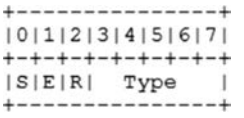

图 3-6　RTP 的分片单元头(FU-A header)

时，结束位设置为 0。

(3) R：长度为 1b，保留位，必须设置为 0，接收者必须忽略该位。

(4) Type：长度为 5b，打包时，原始的 NAL 头的前三位为 FU indicator 的前三位，原始的 NAL 头的后五位为 FU header 的后五位，即这里的 Type 字段，表示 H.264 的 NALU 类型。

3. RTP 示例码流分析

下面是一段 RTP 示例码流，用十六进制显示如下：

```
80 60 01 0f 00 0e 10 00 00 00 00 00 7c 85 88 82 €`.........|???
00 0a 7f ca 94 05 3b 7f 3e 7f fe 14 2b 27 26 f8 ...??.;.>.?. + '&?
89 88 dd 85 62 e1 6d fc 33 01 38 1a 10 35 f2 14 ????b?m?3.8..5?.
84 6e 21 24 8f 72 62 f0 51 7e 10 5f 0d 42 71 12 ?n! $?rb?Q～._.Bq.
17 65 62 a1 f1 44 dc df 4b 4a 38 aa 96 b7 dd 24 .eb??D??KJ8???$
```

(1) 前 12B 是 RTP Header，具体字段格式如图 3-2 所示。

(2) 7c 是 FU indicator。

(3) 85 是 FU header。

(4) FU indicator(0x7c)和 FU header(0x85)换成二进制，即 0111 1100 1000 0101，按顺序解释，如表 3-4 所示。

表 3-4　FU indicator 和 FU header 的字段的二进制值及解释

二进制对应的值	字 段 解 释
0	F，即版本是 2
11	NRI，填充标志，0 表示没有
11100	FU Type，这里是 28，即 FU-A 分片模式
1	S，Start，1 表示是分片的第 1 个包。这里的值为 1，说明是分片的第一包

续表

二进制对应的值	字 段 解 释
0	E,End,如果是分片的最后一包,设置为 1。这里是 0,表示不是分片的最后一个包
0	R,Remain,保留位,总是 0
00101	NAL Type,这里是 5,说明是关键帧

打包时,FU indicator 的 F、NRI 是 NAL header 中的 F、NRI,Type 是 28; FU header 的 S、E、R 分别按照分片起始位置设置,Type 是 NAL header 中的 Type。

解包时,取 FU indicator 的前三位和 FU header 的后五位,即 0110 0101(0x65)为 H.264 的 NALU header(共 8b)。

3.1.3 RTP 的会话过程

当应用程序建立一个 RTP 会话时,应用程序将确定一对目的地传输地址。目的地传输地址由一个网络地址和一对端口组成,有两个端口:一个给 RTP 包,另一个给 RTCP 包,使 RTP/RTCP 数据能够正确发送。RTP 数据发向偶数的 UDP 端口,而对应的控制信号 RTCP 数据发向相邻的奇数 UDP 端口(偶数的 UDP 端口+1),这样就构成一个 UDP 端口对。RTP 的发送过程如下,接收过程则相反。

(1) RTP 从上层接收流媒体信息码流(如 H.264),封装成 RTP 数据包; RTCP 从上层接收控制信息,封装成 RTCP 控制包。

(2) RTP 将 RTP 数据包发往 UDP 端口对中的偶数端口; RTCP 将 RTCP 控制包发往 UDP 端口对中的接收端口。

3.1.4 RTP 的抓包分析

安装好 Wireshark 之后,双击“RTSP 交互流程抓包分析(章节 2.8.3).pcapng”(可从本书对应的代码资料中下载),弹出界面,可以看出包含声频流和视频流,如图 3-7 所示。

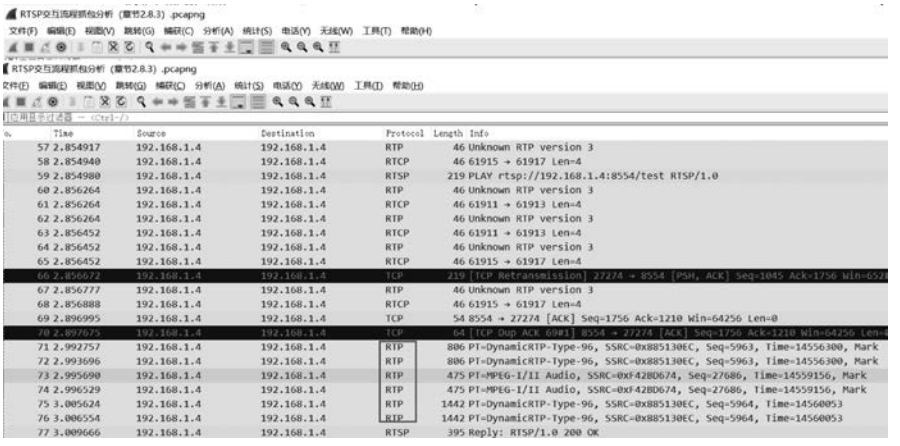

图 3-7 RTP 的抓包流程截图

然后，双击序号为 71 的那一行，弹出界面如图 3-8 所示。

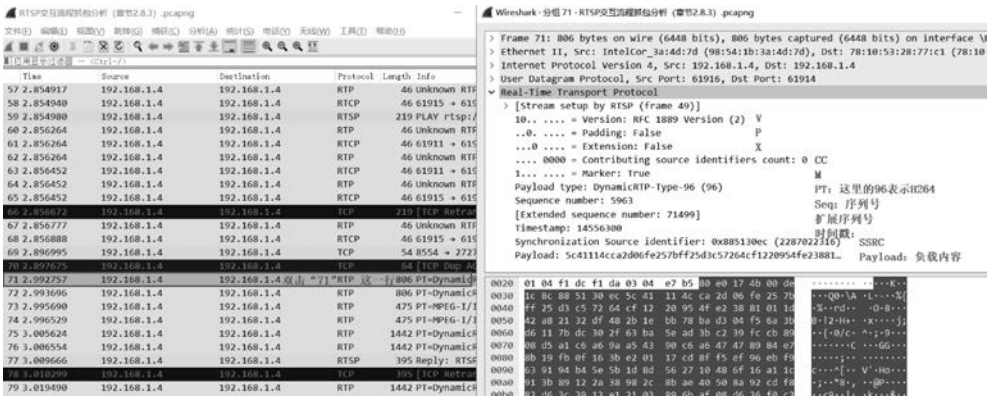

图 3-8　RTP 包的详细字段解析

该 RTP 包的各个字段如下：

```
10.. .... = Version: RFC 1889 Version (2)                  //版本号为 2
..0. .... = Padding: False                                 //没有填充位
...0 .... = Extension: False                               //没有扩展
.... 0000 = Contributing source identifiers count: 0       //CC
1... .... = Marker: True                                   //M,标记位
Payload type: DynamicRTP-Type-96 (96)                      //PT:负载类型,这里为 H.264 码流
Sequence number: 5963                                      //序列号
[Extended sequence number: 71499]
Timestamp: 14556300                                        //时间戳
Synchronization Source identifier: 0x885130ec (2287022316)      //SSRC
Payload: 5c41114cca2d06fe257bff25d3c57264cf1220954fe23881…      //负载
```

3.2　RTCP

实时传输控制协议(Real-time Transport Control Protocol，RTCP)负责管理传输质量并在当前应用进程之间交换控制信息。在 RTP 会话期间，各参与者周期性地传送 RTCP 包，包中含有已发送的数据包的数量、丢失的数据包的数量等统计资料，因此，服务器可以利用这些信息动态地改变传输速率，甚至改变有效载荷类型。RTP 和 RTCP 配合使用，能以有效的反馈和最小的开销使传输效率最佳化，故特别适合传送网上的实时数据。

当应用程序开始一个 RTP 会话时将使用两个端口：一个给 RTP，另一个给 RTCP。RTP 本身并不能为按顺序传送数据包提供可靠的传送机制，也不提供流量控制或拥塞控制，它依靠 RTCP 提供这些服务。在 RTP 的会话之间周期性地发放一些 RTCP 包以用来监听服务质量和交换会话用户信息等功能。RTCP 包中含有已发送的数据包的数量、丢失

的数据包的数量等统计资料，因此，服务器可以利用这些信息动态地改变传输速率，甚至改变有效载荷类型。RTP 和 RTCP 配合使用，它们能以有效的反馈和最小的开销使传输效率最佳化，因而特别适合传送网上的实时数据。根据用户间的数据传输反馈信息，可以制定流量控制的策略，而会话用户信息的交互，可以制定会话控制的策略。

3.2.1 RTCP 的 5 种分组类型

RTP 需要 RTCP 为其服务质量提供保证，RTCP 的主要功能是服务质量的监视与反馈、媒体间的同步及多播组中成员的标识。在 RTP 会话期间，各参与者周期性地传送 RTCP 包。RTCP 包中含有已发送的数据包的数量、丢失的数据包的数量等统计资料，因此，各参与者可以利用这些信息动态地改变传输速率，甚至改变有效载荷类型。RTP 和 RTCP 配合使用，它们能以有效的反馈和最小的开销使传输效率最佳化，因而特别适合传送网上的实时数据。RTCP 也是用 UDP 来传送的，但 RTCP 封装的仅仅是一些控制信息，因而分组很短，所以可以将多个 RTCP 分组后封装在一个 UDP 包中。在 RTCP 通信控制中，RTCP 的功能是通过不同的 RTCP 数据报实现的，有 5 种分组类型，如表 3-5 所示。

表 3-5 RTCP 的 5 种分组类型

类　型	缩　写	用　途
200	SR(Sender Report)	发送端报告
201	RR(Receiver Report)	接收端报告
202	SDES(Source DEScription)	源描述
203	BYE	通知离开
204	APP	应用程序自定义

(1) SR：发送端报告，所谓发送端是指发出 RTP 数据报的应用程序或者终端，发送端同时也可以是接收端。

(2) RR：接收端报告，所谓接收端是指仅接收但不发送 RTP 数据报的应用程序或者终端。

(3) SDES：源描述，主要功能是作为会话成员有关标识信息的载体，如用户名、邮件地址、电话号码等，此外还具有向会话成员传达会话控制信息的功能。

(4) BYE：通知离开，主要功能是指示某一个或者几个源不再有效，即通知会话中的其他成员自己将退出会话。

(5) APP：由应用程序自己定义，解决了 RTCP 的扩展性问题，并且为协议的实现者提供了很大的灵活性。

3.2.2 RTCP 包结构

上述 5 种分组的封装大同小异，下面只讲述 SR 类型，而其他类型可参考 RFC 3550。发送端报告分组 SR(Sender Report)用来使发送端以多播方式向所有接收端报告发送情

况。SR 分组的主要内容有相应的 RTP 流的 SSRC、RTP 流中最新产生的 RTP 分组的时间戳和 NTP、RTP 流包含的分组数，以及 RTP 流包含的字节数。SR 包的封装结构，如图 3-9 所示。

位：0　　8　　16　　24　　32

V=2	P	RC	PT=SR=200	Length
SSRC of sender				
NTP timestamp,most significant word				
NTP timestamp,least significant word				
RTP timestamp				
Sender's packet count				
Sender's octet count				
SSRC_1(SSRC of first source)				
Fraction lost			Cumulative number of packets lost	
Extended highest sequence number received				
Interarrival jitter				
Last SR(LSR)				
Delay since last SR(DLSR)				
SSRC_2(SSRC of second source)				
…				
Profile-specific extensions				

图 3-9　RTCP 的头格式

RTCP 头部格式的各个字段解释如下。

(1) 版本(V)：同 RTP 包头域。

(2) 填充(P)：同 RTP 包头域。

(3) 接收报告计数器(RC)：5b，该 SR 包中的接收报告块的数目，可以为零。

(4) 包类型(PT)：8b，SR 包是 200。

(5) 长度域(Length)：16b，其中存放的是该 SR 包以 32b 为单位的总长度减一。

(6) 同步源(SSRC)：SR 包发送者的同步源标识符。与对应 RTP 包中的 SSRC 一样。

(7) NTP(Network Time Protocol)timestamp：SR 包发送时的绝对时间值。NTP 的作用是同步不同的 RTP 媒体流。

(8) RTP timestamp：与 NTP 时间戳对应，与 RTP 数据包中的 RTP 时间戳具有相同的单位和随机初始值。

(9) Sender's packet count：从开始发送包到产生这个 SR 包这段时间里，发送者发送的 RTP 数据包的总数。SSRC 改变时，这个域清零。

(10) Sender's octet count：从开始发送包到产生这个 SR 包这段时间里，发送者发送的净荷数据的总字节数(不包括头部和填充)。发送者改变其 SSRC 时，这个域要清零。

(11) 同步源 n 的 SSRC 标识符：该报告块中包含的是从该源接收的包的统计信息。

(12) 丢失率(Fraction Lost)：表明从上一个 SR 或 RR 包发出以来从同步源 n(SSRC_n)来的 RTP 数据包的丢失率。

(13) 累计的包丢失数目：从开始收到 SSRC_n 的包到发送 SR，从 SSRC_n 传过来的 RTP 数据包的丢失总数。

(14) 收到的扩展最大序列号：从 SSRC_n 收到的 RTP 数据包中最大的序列号。

(15) 接收抖动(Interarrival Jitter)：RTP 数据包接受时间的统计方差估计。

(16) 上次 SR 时间戳(Last SR，LSR)：取最近从 SSRC_n 收到的 SR 包中的 NTP 时间戳的中间 32b。如果目前还没收到 SR 包，则该域清零。

(17) 上次 SR 以来的延时(Delay since Last SR，DLSR)：上次从 SSRC_n 收到 SR 包到发送本报告的延时。

3.2.3 RTCP 的注意事项

不同类型的 RTCP 信息包可堆叠，不需要插入任何分隔符就可以将多个 RTCP 包连接起来形成一个 RTCP 组合包，然后由低层协议用单一包发送出去。由于需要低层协议提供整体长度来决定组合包的结尾，在组合包中没有单个 RTCP 包的显式计数。

组合包中每个 RTCP 包可独立处理，而不需要按照包组合的先后顺序处理。在组合包中有以下几条强制约束：

(1) 只要带宽允许，在 SR 包或 RR 包中的接收统计应该经常发送，因此每个周期发送的组合 RTCP 包中应包含报告包。

(2) 每个组合包中都应该包含 SDES CNAME，因为新接收者需要通过接收 CNAME 来识别源，并与媒体联系进行同步。

(3) 组合包前面是包类型数量，其增长应该受到限制。

所有 RTCP 包至少必须以两个包组合的形式发送，推荐格式如下：

(1) 加密前缀(Encryption Prefix)：

(2) 仅当组合包被加密，才加上一个 32 位随机数用于每个组合包发送。

对于 SR 或 RR，组合包中第 1 个 RTCP 包必须是一个报告包，以帮助分组头的确认。即使没有数据发送，也没有收到数据，也要发送一个空 RR，哪怕组合包中 RTCP 包为 BYE。对于附加 RR，如报告统计源数目超过 31，在初始报告包后应该有附加 RR 包。对于 SDES，包含 CNAME 项的 SDES 包必须包含在每个组合 RTCP 包中；SDES 包可能包括其他源描述项，这要根据特别的应用需要，并同时考虑带宽限制。对于 BYE 或 APP，除了 BYE 应作为最后一个包发送，其他 RTCP 包类型可以按任意顺序排列，包类型的出现次数可不止一次。

混合器从多个源组合单个 RTCP 包，如组合包整体长度超过网络路径最大传输单元，则可分成多个较短组合包并用低层协议以单个包形式发送。注意，每个组合包必须以 SR 或 RR 包开始。附加 RTCP 包类型可在 Internet Assigned Numbers Authority (IANA)处注册，以获得合法的类型号。

1. RTCP 传输间隔

由于 RTP 被设计成允许应用自动扩展，所以可从几个人的小规模系统扩展成上千人的大规模系统，而每个会话参与者可周期性地向所有其他参与者发送 RTCP 控制信息包，如每个参与者以固定速率发送接收报告，控制流量将随参与者的数量线性增长。由于网络资源有限，相应的数据包就要减少，直接影响用户关心的数据传输。为了限制控制信息的流量，RTCP 控制信息包速率必须按比例下降。一旦确认加入 RTP 会话中，即使后来被标记成非活动站，地址的状态仍会被保留，地址应继续计入共享 RTCP 带宽地址的总数中，时间要保证能扫描典型网络分区，建议为 30 分钟。注意，这仍大于 RTCP 报告间隔最大值的五倍。

2. SR 源报告包和 RR 接收者报告包

SR 源报告包和 RR 接收者报告包用于提供接收质量反馈，除包类型代码外，SR 与 RR 间唯一的差别是源报告包含一个 20 字节发送者信息段。RR 针对每个信源都提供信息包丢失数、已收信息包最大序列号、到达时间抖动、接收最后一个 SR 的时间、接收最后一个 SR 的延迟等信息。SR 不仅提供接收质量反馈信息(与 RR 相同)，而且提供 SSRC 标识符、NTP 时间戳、RTP 时间戳、发送包数及发送字节数等。根据接收者是否为发送者来决定使用 SR 还是 RR 包，活动源在发出最后一个数据包之后或前一个数据包与下一个数据包间隔期间发送 SR；否则，就发送 RR；SR 和 RR 包都可没有接收报告块，也可以包括多个接收报告块，其发布报告表示的源不一定是在 CSRC 列表上的起作用的源，每个接收报告块提供从特殊源接收数据的统计。最大可有 31 个接收报告块嵌入在 SR 或 RR 包中，丢失包累计数差别给出间隔期间丢包的数量，而系列号的差别给出间隔期间希望发送的包数量，两者之比等于经过间隔期间包丢失百分比。从发送者信息，第三方监控器可计算载荷平均数据速率与没收到数据间隔的平均包速率，两者比值给出平均载荷大小。如假设包丢失与包大小无关，则特殊接收者收到的包数量给出此接收者收到的表观流量。

3. SDES 源描述包

SDES 源描述包提供了直观的文本信息，用来描述会话的参加者，包括 CNAME、NAME、EMAIL、PHONE、LOC 等源描述项，这些为接收方获取发送方的有关信息提供了方便。SDES 包由包头与数据块组成，数据块可以没有，也可有多个。包头由版本(V)、填充(P)、长度指示、包类型(PT)和源计数(SC)组成。PT 占 8 位，用于识别 RTCP 的 SDES 包，SC 占 5 位，用于指示包含在 SDES 包中的 SSRC/CSRC 块的数量，零值有效，但没有意义。数据块由源描述项组成，源描述项的内容如下：

(1) CNAME 用于规范终端标识 SDES 项，类似 SSRC 标识，RTCP 为 RTP 连接中每个参加者赋予唯一一个 CNAME 标识。在发生冲突或重启程序时，由于随机分配的 SSRC 标识可能发生变化，CNAME 项可以提供从 SSRC 标识到仍为常量的源标识的绑定。为了方便第三方监控，CNAME 应适合程序或人员定位源。

(2) NAME 是指用户名称 SDES 项，是用于描述源的真正的名称，如 John Doe、Bit Recycler、Megacorp 等，可以是用户想要的任意形式。由于采用文本信息来描述，对诸如会

议应用，可以对参加者直接列表显示，NAME 项是除 CNAME 项以外发送最频繁的项目。NAME 值在一次 RTP 会话期间应该保持为常数，但它不该成为连接的所有参加者中的唯一依赖。

(3) EMAIL 是指电子邮件地址 SDES 项，邮件地址格式由 RFC 822 规定，如 John. Doe @megacorp. com。一次 RTP 会话期间，EMAIL 项的内容应保持不变。

(4) PHONE 是指电话号码 SDES 项，电话号码应带有加号，代替国际接入代码，如 +1 908 555 1212 为美国电话号码。

(5) LOC 是指用户地理位置 SDES 项，根据应用，此项具有不同程度的细节。对会议应用，字符串(如 Murray Hill、New Jersey)就足够了，然而，对活动标记系统，字符串(如 Room 2A244, AT&T BL MH)就适用。细节留给实施者或用户，但格式和内容可用来设置指示。在一次 RTP 会话期间，除移动主机外，LOC 值应保持不变。

(6) TOOL 是指应用或工具名称 SDES 项，包含一个字符串，表示产生流的应用的名称与版本，如 videotool 1.2。这部分信息对调试很有用，类似于邮件或邮件系统版本 SMTP 头。TOOL 值在一次 RTP 会话期间应保持不变。

(7) NOTE 是指通知/状态 SDES 项，旨在描述源当前状态的过渡信息，如 on the phone、can't talk，或在讲座期间用于传送谈话的题目，它的语法可在设置中显式定义。NOTE 项一般只用于携带例外信息，而不应包含在全部参加者中，因为这将降低接收报告和 CNAME 发送的速度，降低协议的性能。一般 NOTE 项不作为用户设置文件的项目，也不会自动产生。由于 NOTE 项对显示很重要，当会话的参加者处于活动状态时，其他非 CNAME 项(如 NAME)传输速率将会降低，结果使 NOTE 项占用 RTCP 部分带宽。若过渡信息不活跃，则 NOTE 项会继续以同样的速度重复发送几次，并以一个串长为零的字符串通知接收者。

(8) PRIV 专用扩展 SDES 项，用于定义实验或应用特定的 SDES 扩展，它是由长字符串对组成的前缀、后跟填充该项其他部分和携带所需信息的字符串值组成。前缀长度段为 8 位。前缀字符串是定义 PRIV 项人员选择的名称，唯一对应应用接收的其他 PRIV 项。应用实现者可选择使用应用名称，如有必要，外加附加子类型标识。另外，推荐其他人根据其代表的实体选择名称，然后在实体内部协调名称的使用。注意，前缀应尽可能短。SDES 的 PRIV 项前缀没在 IANA 处注册。如证实某些形式的 PRIV 项具有通用性，则 IANA 应给它分配一个正式的 SDES 项类型，这样就不再需要前缀，从而简化应用，并提高传输的效率。

4. BYE 断开 RTCP 包

如混合器接收到一个 BYE 包，混合器转发 BYE 包，而不改变 SSRC/CSRC 标识。如混合器关闭，在关闭之前它应该发出一个 BYE 包，列出混合器处理的所有源，而不只是自己的 SSRC 标识。作为可选项，BYE 包可包括一个 8 位八进制计数，后跟文本信息，表示离开原因，如 cameramalfunction 或 RTPloop detected。字符串的编码与在 SDES 项中所描述的编码相同。如字符串信息至 BYE 包下 32 位边界结束处，字符串就不以空结尾；否则，BYE 包以空八进制填充。

5. APP 特殊应用包

APP 包用于开发新应用和新特征的实验，不要求注册包类型值。带有不可识别名称的 APP 包应被忽略。测试后，如确定应用广泛，推荐重新定义每个 APP 包，而不用向 IANA 注册子类型和名称段。

3.2.4　RTCP 的抓包分析

安装好 Wireshark 之后，双击“RTSP 交互流程抓包分析(章节 2.8.3).pcapng”(可从本书对应的代码资料中下载)，弹出界面，可以看出包含声频流和视频流，如图 3-10 所示。

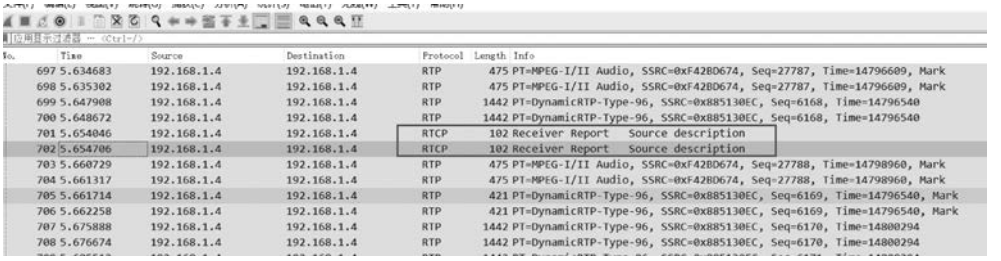

图 3-10　RTCP 抓包流程截图

双击序号为 702 的那一行，弹出界面如图 3-11 所示。

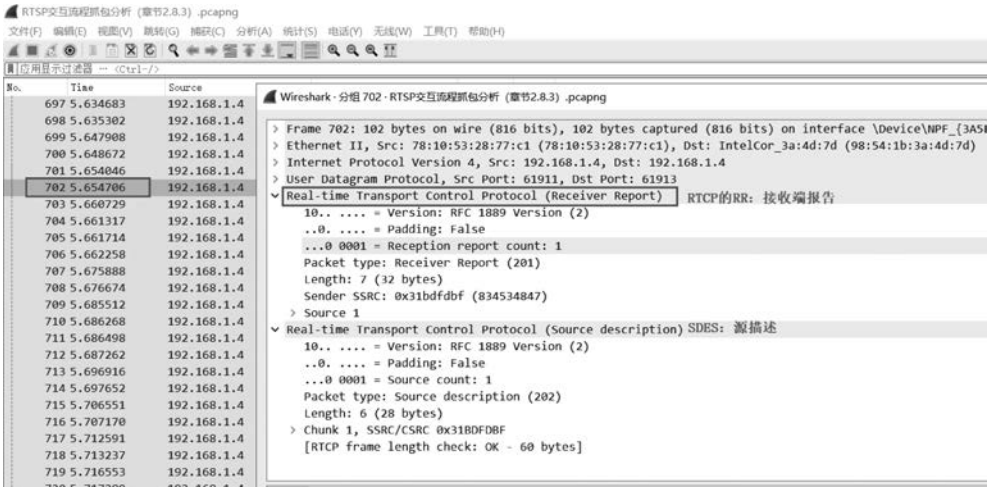

图 3-11　RTCP 包的详细字段

该 RTCP 包的各个字段(可参考“3.2.2 RTCP 包结构”)如下：

```
//chapter3/rtcp.pack.filelds.txt
#Real-time Transport Control Protocol (Receiver Report)
10.. .... = Version: RFC 1889 Version (2)
```

```
..0. .... = Padding: False
...0 0001 = Reception report count: 1
Packet type: Receiver Report (201)
Length: 7 (32 Bytes)
Sender SSRC: 0x31bdfdbf (834534847)
#Source 1
Identifier: 0xf42bd674 (4096513652)
SSRC contents
Extended highest sequence number received: 93323
Interarrival jitter: 109
Last SR timestamp: 0 (0x00000000)
Delay since last SR timestamp: 0 (0 milliseconds)

#Real-time Transport Control Protocol (Source description)
10.. .... = Version: RFC 1889 Version (2)
..0. .... = Padding: False
...0 0001 = Source count: 1
Packet type: Source description (202)
Length: 6 (28 Bytes)
#Chunk 1, SSRC/CSRC 0x31BDFDBF
Identifier: 0x31bdfdbf (834534847)
#SDES items
Type: CNAME (user and domain) (1)
Length: 15
Text: DESKTOP-NUCTJFU
Type: END (0)
负载
```

3.3 RTP/RTCP 与 RTSP 的关系

RTP 与 RTCP 通常会结合在一起使用，RTP 用于传输媒体流，而 RTSP 是媒体控协议。RTP/RTCP 一般是基于 UDP 来传输数据的，但也可以基于 TCP，因此可以将 RTP 划分到传输层。RTP、RTCP 与 RTSP 的网络层级关系如图 3-12 所示。

RTP 是用于 Internet 上针对多媒体数据流的一种传输层协议，详细说明了在互联网上传递声频和视频的标准数据包格式。RTP 协议常用于流媒体系统，并配合 RTCP、视频会议和一键通系统(配合 H.323 或 SIP)，使它成为 IP 电话产业的技术基础。RTP 和控制协议 RTCP 一起使用，通常建立在 UDP 上。RTP 本身并没有提供按时发送机制或其他服务质量(QoS)保证，它依赖于低层服务去实现这一过程。RTP 并不保证传送或防止无序传送，也不确定底层网络的可靠性。RTP 实行有序传送，RTP 中的序列号允许接收方重组发送方的包序列，同时序列号也能用于决定适当的包位置，例如在视频解码中就不需要按顺序解码。可以把广义的 RTP 理解为由两个紧密连接的部分组成，即 RTP 和 RTCP。RTP 用

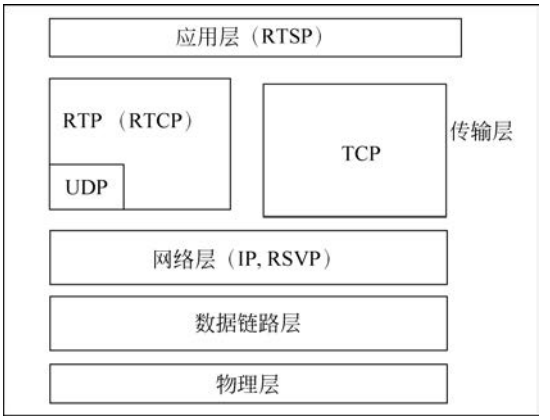

图 3-12　RTP、RTCP、RTSP 的网络层级关系

于传送具有实时属性的数据；RTCP 用于监控服务质量并传送正在进行的会话参与者的相关信息。

RTCP 是 RTP 的一个姐妹协议。RTCP 为 RTP 媒体流提供信道外(out-of-band)控制。RTCP 本身并不传输数据，但和 RTP 一起协作将多媒体数据打包和发送。RTCP 定期在流多媒体会话参加者之间传输控制数据。RTCP 的主要功能是为 RTP 所提供的服务质量(Quality of Service，QoS)提供反馈。RTCP 收集相关媒体连接的统计信息，例如传输字节数、传输分组数、丢失分组数、jitter、单向和双向网络延迟等。网络应用程序可以利用 RTCP 所提供的信息试图提高服务质量，例如限制信息流量或改用压缩比较小的编解码器。RTCP 本身不提供数据加密或身份认证。SRTCP 可以用于此类用途。

安全实时传输协议(Secure Real-time Transport Protocol，SRTP)是在 RTP 的基础上定义的一个协议，旨在为单播和多播应用程序中的 RTP 的数据提供加密、消息认证、完整性保证和重放保护。它是由思科公司和爱立信公司开发的，并最早由 IETF 于 2004 年 3 月作为 RFC 3711 发布。由于 RTP 和可以被用来控制 RTP 会话的 RTCP 有着紧密的联系，SRTP 同样也有一个伴生协议，它被称为安全实时传输控制协议(Secure RTCP，SRTCP)。SRTCP 为 RTCP 提供类似的与安全有关的特性，就像 SRTP 为 RTP 提供的那些一样。在使用 RTP 或 RTCP 时，使不使用 SRTP 或 SRTCP 是可选的，但即使使用了 SRTP 或 SRTCP，它们提供的所有特性(如加密和认证)也都是可选的，这些特性可以被独立地使用或禁用。唯一的例外是在使用 SRTCP 时，必须使用其消息认证特性。

RTSP 是用来控制声音或影像的多媒体串流协议，并允许同时多个串流需求控制，传输时所用的网络通信协定并不在其定义的范围内，服务器端可以自行选择使用 TCP 或 UDP 来传送串流内容，它的语法和运作跟 HTTP 1.1 类似，但并不特别强调时间同步，所以比较能容忍网络延迟，而前面提到的允许同时多个串流需求控制(Multicast)，除了可以降低服务器端的网络用量，更进而支持多方视频会议(Video Conference)。因为与 HTTP 1.1 的运作方式相似，所以代理服务器的缓存功能也同样适用于 RTSP，并因 RTSP 具有重新导向

功能，可根据实际负载情况来转换提供服务的服务器，以避免过大的负载集中于同一服务器而造成延迟。

RTP 不像 HTTP 和 FTP 那样可完整地下载整个影视文件，它是以固定的数据率在网络上发送数据，客户端也是按照这种速度观看影视文件，当影视画面播放过后，就不可以再重复播放了，除非重新向服务器端请求数据。RTSP 与 RTP 最大的区别在于：RTSP 是一种双向实时数据传输协议，它允许客户端向服务器端发送请求，如回放、快进、倒退等操作。当然，RTSP 可基于 RTP 来传送数据，还可以选择 TCP、单播 UDP 或组播 UDP 等通道来发送数据，具有很好的扩展性。它是一种类似于 HTTP 的网络应用层协议。例如一个应用场景，服务器端实时采集、编码并发送两路视频，客户端接收并显示这两路视频。由于客户端不必对视频数据做任何回放、倒退等操作，所以可直接采用 UDP＋RTP＋组播实现。RTP、RTCP 与 RTSP 的关系与作用，如图 3-13 所示。

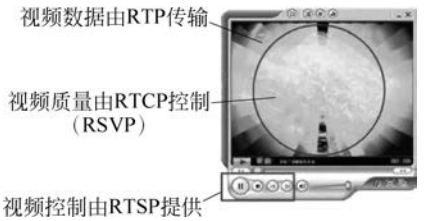

图 3-13　RTP、RTCP 与 RTSP 的关系与作用

3.4　开源库 JRTPLIB 简介

JRTPLIB 是一个基于 C++、面向对象的 RTP 封装库，目前的较新版本是 3. x. x。为了与 RFC 3550 相兼容，3. x. x 版本已被完全重写，现在它提供了一些非常有用的组件，这些组件为构建各种各样的 RTP 应用程序开发提供了有用的帮助。较旧的 2. x. x 版本依然可用，但是不兼容 RFC 3550。JRTPLIB 支持定义于 RFC 3550 中的 RTP，它使发送和接收 RTP 报文变得非常简单，用户不用担心 SSRC 冲突，也不用考虑如何传输 RTCP 数据，因为 RTCP 功能完全在内部实现，不需用户手动操作。当发送 RTP 报文时，用户只需简单地给发送函数提供负载数据；当接收数据时，JRTPLIB 提供了访问传入的 RTP 和 RTCP 数据的接口。

目前为止，JRTPLIB 支持的平台主要包括 GNU/Linux、MS-Windows 和 Solaris 等，也可以运行于其他类 UNIX 环境。

JRTPLIB 可以使用 JTHREAD 库在后台自动轮询传入的数据，所以安装 JTHREAD 库是个很好的选择。如果没有安装 JTHREAD 库，JRTPLIB 也能正常工作，但是需要用户自己轮询传入的数据。3. x. x 版本的 JRTPLIB 至少需要 1. 3. 0 版本的 JTHREAD 库。JRTPLIB 官网网址是 http://research. edm. uhasselt. be/jori/page/Main/HomePage. html，可以从官网上下载源码。

3.4.1　Windows 10＋VS 2015 编译 JRTPLIB

操作系统使用 Windows 10，读者自己安装好 CMake 3.12.0，以及 Visual Studio 2015/2017/2019。由于 JRTPLIB 依赖于 JTHREAD 库，需要下载以下内容。

(1) jrtplib：http://research.edm.uhasselt.be/jori/jrtplib/jrtplib-3.11.1.zip。

(2) jthread：http://research.edm.uhasselt.be/jori/jthread/jthread-1.3.3.zip。

(3) cmake：https://cmake.org/files/v3.12/cmake-3.12.0-win64-x64.msi。

注意：如果下载网址无法使用，读者则可以自己百度，也可以从清华大学出版社网站上本书对应的课件资料中下载(jrtplib-3.11.1.zip、jthread-1.3.3.zip、cmake-3.12.0-win64-x64.msi)。

详细的编译步骤如下：

(1) 安装 cmake-gui。

(2) 将下载的 jrtplib 和 jthread 压缩包进行解压缩，同时在同目录下创建 build 文件夹。

(3) 以下过程主要是编译 jthread 并生成 jthread.lib 和 jthread_d.lib。

打开 cmake-gui，首先添加输入源(where..)和输出路径(where to…)，单击 Configure，目标选择 VS 2015/2017/2019 默认编译器，然后详细检查参数，如图 3-14 所示。确认无误后再单击 configure，最后单击 Generate，生成 VS 2015/2017/2019 工程文件，如图 3-15 所示。

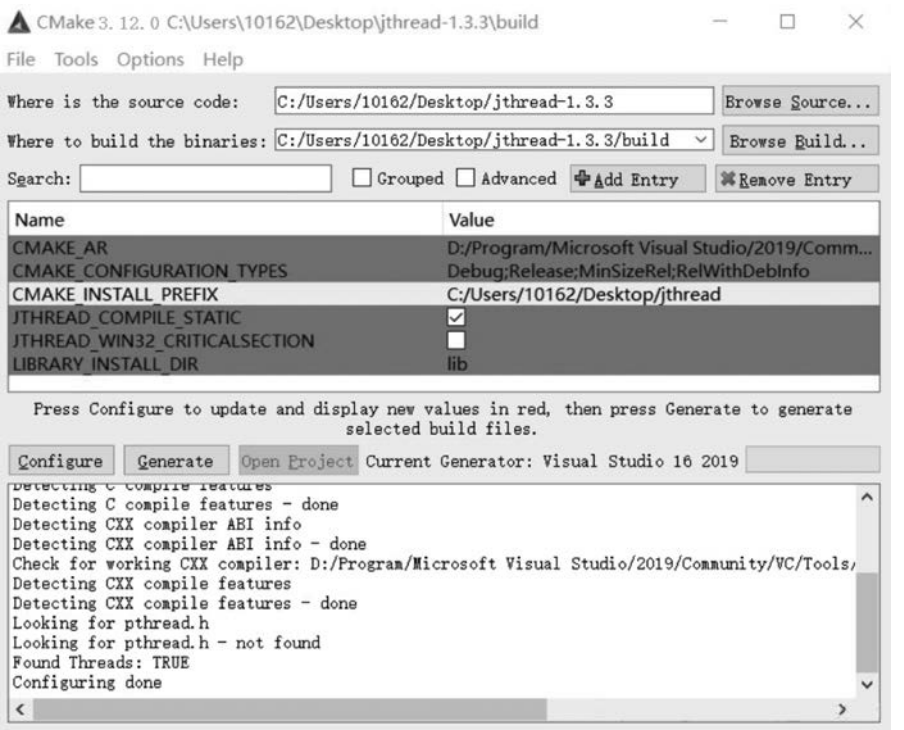

图 3-14　CMake 配置 JTHREAD

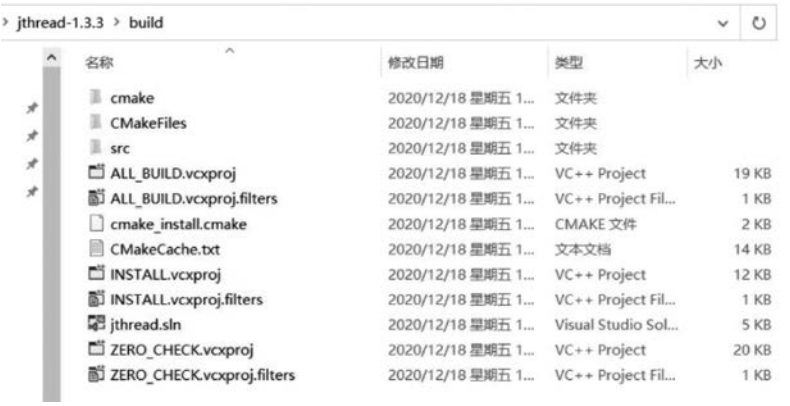

图 3-15　CMake 生成的 JTHREAD 工程文件

单击 Open Project 打开工程，如图 3-16 所示。

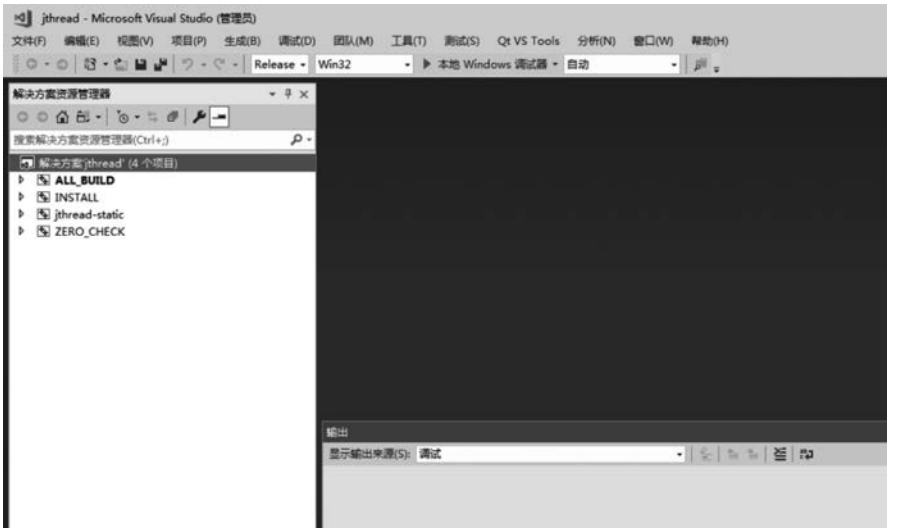

图 3-16　VS 打开 JTHREAD 工程

编译的具体方法为选择解决方案资源管理器里的解决方案 jthread，运行“重新生成解决方案”；如果没有出现错误，再选择 INSTALL 项目，运行“生成”。debug 和 release 各进行一次上述操作即可。如果编译成功(如图 3-16 所示)，则会在对应目录的\jthread\include\jthread 下生成头文件；在 lib 文件夹下生成 lib 和 cmake 文件。

(4) 以下过程主要是编译 jrtplib，生成 jrtplib. lib 和 jrtplib_d. lib。

大致的步骤与上述相同，但在编译和 configure 时需要添加一些配置，同样先输入源(where. .)和输出路径(where to…)，单击 configure，目标选择 VS 2015/2017/2019 默认编译器，初始的配置结果如图 3-17 所示。

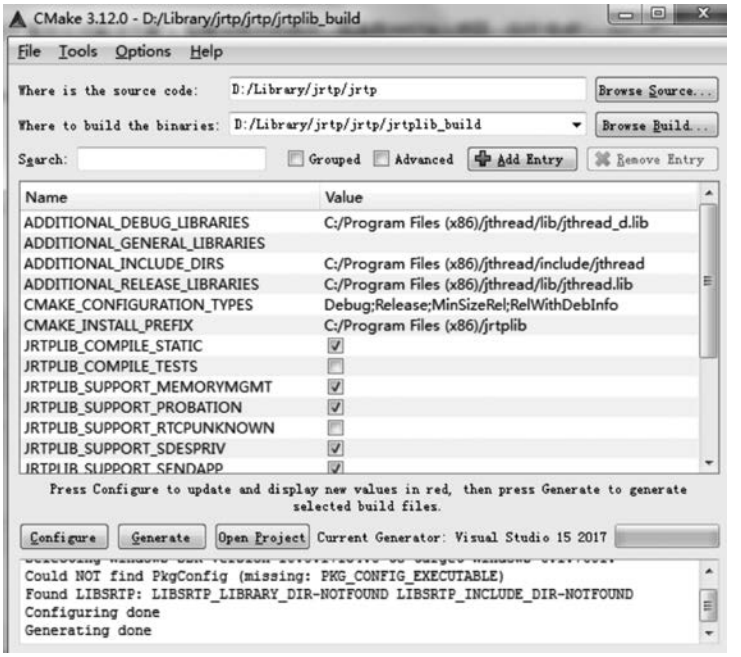

图 3-17　CMake 配置 JRTPLIB

注意这里需要添加几个路径，包括 ADDITIONAL_DEBUG_LIBRARIES、ADDITIONAL_RELEASE_LIBRARIES 和 ADDITIONAL_INCLUDE_DIRS 的路径。确认无误后再单击 Configure，最后单击 Generate，生成 VS 2017 工程文件，然后单击 Open Project 打开工程，如图 3-18 所示。

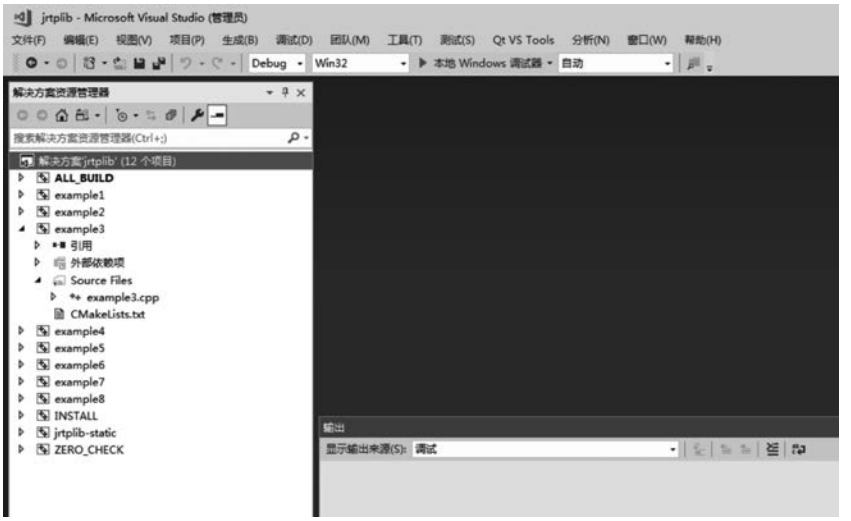

图 3-18　VS 打开 JRTPLIB 工程

编译的具体方法为选择解决方案资源管理器里的解决方案 jrtplib，运行“重新生成解决方案”；如果没有出现错误，则选择 INSTALL 项目，运行“生成”。debug 和 release 各进行一次上述操作即可。如果编译成功，则会在对应的目录\jrtplib\include\jthread 生成头文件；在 lib 文件夹下生成 lib 和 cmake 文件。

3.4.2 Ubuntu 18 编译 JRTPLIB

Linux 下编译 JRTPLIB 比较简单，遵循基本的三部曲，即 configure、make、make install，这里以 Ubuntu 18 为例进行讲解。

(1) 源码下载，需要从 JRTPLIB 官网下载源码，这里需要下载两个库，包括 JTHREAD 和 JRTPLIB，如图 3-19 所示。

注意：官网网址 http://research.edm.uhasselt.be/jori/page/Main/HomePage.html。

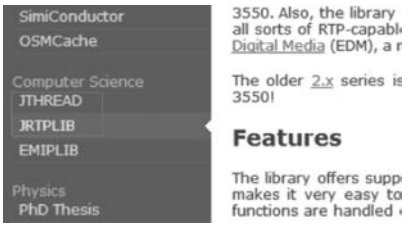

图 3-19 JRTPLIB 官网

分别下载两个库 JTHREAD 和 JRTPLIB，如图 3-20 所示。

图 3-20 JRTPLIB 和 JTHREAD 下载

(2) 安装相关工具，JRTPLIB 使用 CMake 构建，这里需要安装较新版本的 CMake，可以使用以下命令安装：

```
sudo apt install cmake
```

(3) 编译并安装，先编译 JTHREAD，使用下面的命令解压 jthread-1.3.3.tar.gz：

```
tar xvzf jthread-1.3.3.tar.gz
```

在源码文件外创建一个 jthread-build 文件夹，用于放编译文件，这样不会对源码有任何改动，命令如下：

```
mkdir jthread-build
```

进入 jthread-build 文件夹：

```
cd jthread-build/
```

执行 cmake 命令生成 makefile 文件：

```
cmake ../jthread-1.3.3 -DCMAKE_INSTALL_PREFIX=/usr
```

使用 make 命令进行源码的编译：

```
make
```

使用 make install 命令进行安装：

```
sudo make install
```

此时可以看到链接库安装在/usr/lib/目录下，头文件安装在/usr/include/jthread/目录下。

然后来编译 JRTPLIB，使用如下命令解压 jrtplib-3.11.1.tar.gz：

```
tar xvzf jrtplib-3.11.1.tar.gz
```

在源码文件外创建一个 jrtplib-build 文件夹，用于放编译文件，这样不会对源码有任何改动：

```
mkdir jrtplib-build
```

进入 jrtplib-build 文件夹：

```
cd jrtplib-build/
```

执行 cmake 命令生成 makefile 文件：

```
cmake ../jrtplib-3.11.1 -DCMAKE_INSTALL_PREFIX=/usr
```

使用 make 命令进行源码的编译：

```
make
```

使用 make install 命令进行安装：

```
sudo make install
```

以上命令执行完后，库文件安装在/usr/lib/目录下，头文件在/usr/include/jrtplib3/目录下。

3.4.3 使用 VS 2015 搭建 JRTPLIB 开发环境并收发包案例解析

上文介绍了 jrtplib 源码库的编译，下面介绍一个简单的 JRTPLIB 接收端和客户端。在使用 JRTPLIB 之前需要将其添加进工程，笔者以 VS 2015（VS 2017/2019 与此大同小异）作为 IDE，编写一个 VC 程序，其他 IDE 参考 VS 即可，调用外部库通常有三点注意事项。

（1）引用时需要的头文件. h/. hpp。

（2）编译时需要的库文件. dll/. lib/. a。

（3）运行时需要的动态库. dll/. so。

下面介绍详细的配置步骤。

1. 新建 jrtplibdemo 工程

使用 VS 2015 新建 VC++空工程，移除创建的项目，然后添加 sender 和 recver 两个项目，如图 3-21 和图 3-22 所示。

图 3-21 VS 2015 创建 VC++新建项目

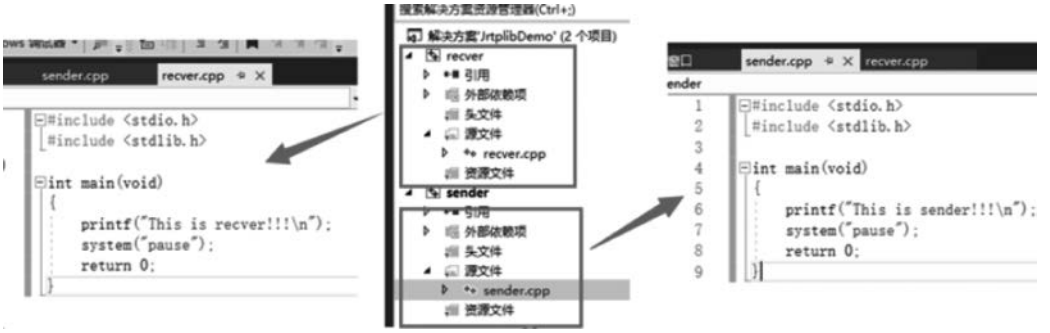

图 3-22　VS 2015 添加两个新项目

为了调试方便，启用多个项目调试，即运行时可设置运行调试哪些项目，如图 3-23 所示。

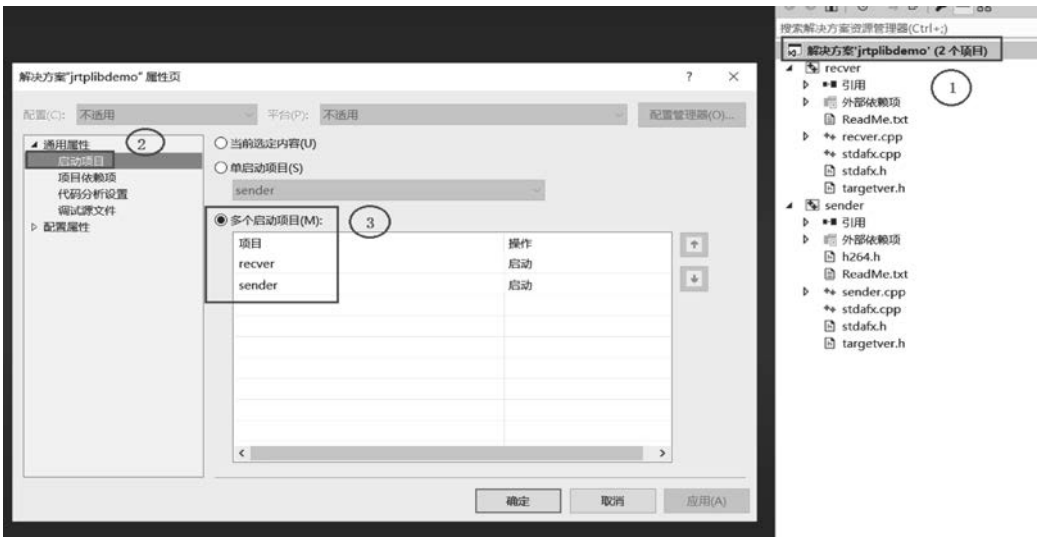

图 3-23　VS 2015 设置多项目启动

运行时，如图 3-24 所示。

图 3-24　VS 2015 运行多项目

2. 引用 JRTPLIB 头文件和库文件

新建一个文件夹 jthreaddevs，将 JRTPLIB 和 JTHREAD 的头文件和库文件整合到一起，如图 3-25 所示。

图 3-25　JTHREAD 和 JRTPLIB 的头文件和库文件

为 sender 项目引入头文件和库文件目录（recver 项目设置完全相同），如图 3-26 所示。其中，需要注意相对路径..\..\，需要将刚才创建的 jthreaddevs 文件夹放置到 jrtplibdemoSenderRecver 的同目录下，如图 3-27 所示。

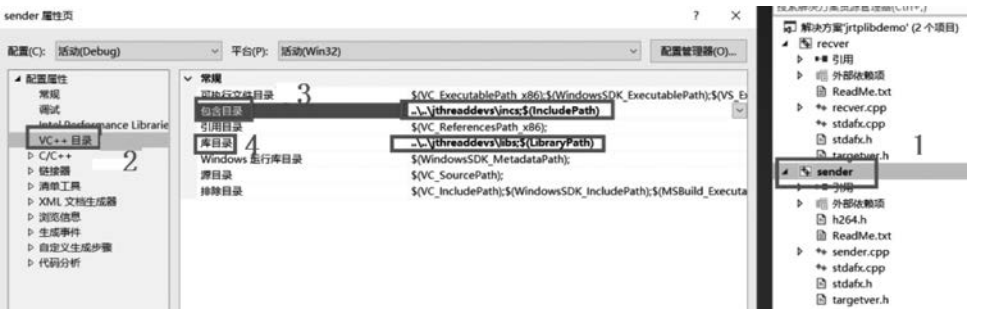

图 3-26　为 sender/recver 项目设置目录和库目录

图 3-27　jthreaddevs 的相对路径为..\..\

复制库文件，如果将 jrtplib 编译为静态库，则运行时不需要其他库；如果将 jrtplib 编译为动态库，则运行时需要使用 dll 动态库，需要将 DLL 文件复制到 exe 文件输出目录下，如图 3-28 所示。

3. JRTPLIB 示例代码实现发送和接收功能

发送端（参考 sender 项目下的 sender.cpp 文件）代码如下：

图 3-28　动态编译则运行时需要单独的 DLL 文件

```
//chapter3/jrtplibDemoRecverSender/jrtplibdemoSenderRecver/sender/sender.cpp
#include <stdio.h>
#include <stdlib.h>

//RTP 库依赖 socket,必须在 RTP 库引入之前添加,否则会出现各种错误
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib")

//RTP 库引入
#include "jrtplib/rtpsession.h"
#include "jrtplib/rtpudpv4transmitter.h"
#include "jrtplib/rtpipv4address.h"
#include "jrtplib/rtpsessionparams.h"
#include "jrtplib/rtperrors.h"
//注意引入对应的 lib 文件
#pragma comment(lib, "jrtplib_d.lib")          //jthread_d.lib;jrtplib_d.lib;
#pragma comment(lib, "jthread_d.lib")          //jthread 的 lib 库文件

using namespace jrtplib;                       //引入 jrtplib 命名空间

int main(void)
{
    //RTPSession 代表一个 RTP 会话
    RTPSession rtpSession;
    //设置会话参数
    RTPSessionParams rtpSessionParams;
    RTPUDPv4TransmissionParams rtpUdpv4Transmissionparams;     //UDP 地址

    char buf[1024] = { 0x00 };
    char ip[16] = { 0x00 };
    int port = 0;                              //端口号,需要运行时输入
    int ret = 0;
```

```
    //容易忽略,因为自写代码中没有调用 socket,RTP 有调用,但是没有初始化
    WSADATA dat;
    WSAStartup(MAKEWORD(2, 2), &dat);

    printf("This is sender!!!\n");

    //目的 ip 与 port
    printf("Input destination ip:");
    scanf("%s", ip);
    printf("Input destination port:");
    scanf("%d", &port);
    printf("Destination %s:%d\n", ip, port);

    //设置 RTP 会话的参数
    rtpSessionParams.SetOwnTimestampUnit(1.0 / 1);          //时间戳单元
    rtpSessionParams.SetUsePollThread(true);                //线程轮询
    rtpSessionParams.SetAcceptOwnPackets(false);是否接收自己的数据包
    ret = rtpSession.Create(rtpSessionParams,&rtpUdpv4Transmissionparams);     //创建会话
    if (ret < 0)
    {
     printf("Failed to RtpSession::Create, ret=%d\n", ret);
    }

    RTPIPv4Address addr(ntohl(inet_addr(ip)), port);
    rtpSession.AddDestination(addr);                        //将目的地址添加到 RTP 会话中

    while (true)            //开启循环,连续发送消息,输入 exit 退出循环
    {
     printf("Input message:");
     scanf("%s", buf);
     if (strcmp(buf, "exit") == 0)                          //输入 exit 退出循环
     {
       break;
     }
     //发送数据包,JRTPLIB 会将原始数据封装为 RTP 包格式,自动添加 RTP 包头
     ret = rtpSession.SendPacket((void *)buf, strlen(buf), 0, false, 1);
     if (ret < 0)
     {
       printf("Failed to RtpSession::SendPacket, ret=%d\n", ret);
       continue;
     }
     else {
       printf("Succeed to RtpSession::SendPacket!!!\n");
     }
     RTPTime::Wait(RTPTime(0, 100));
    }
    return 0;
}
```

接收端(参考 recver 项目下的 recver.cpp 文件)代码如下：

```
//chapter3/jrtplibDemoRecverSender/jrtplibdemoSenderRecver/recver/recver.cpp
#include <stdio.h>
#include <stdlib.h>

//RTP 库依赖 socket,必须在 RTP 库引入之前添加,否则会出现各种错误
#include <WinSock2.h>
#pragma comment(lib, "ws2_32.lib")

//RTP 库引入
#include "jrtplib/rtpsession.h"
#include "jrtplib/rtpudpv4transmitter.h"
#include "jrtplib/rtpipv4address.h"
#include "jrtplib/rtpsessionparams.h"
#include "jrtplib/rtperrors.h"
#include "jrtplib/rtppacket.h"
#pragma comment(lib, "jrtplib_d.lib")
#pragma comment(lib, "jthread_d.lib")

using namespace jrtplib;

int main(void)
{
    RTPSession rtpSession;                    //RTP 会话
    RTPSessionParams rtpSessionParams;        //RTP 会话参数
    RTPUDPv4TransmissionParams rtpUdpv4Transmissionparams;

    char ip[16] = "127.0.0.1";
    int port = 0;
    int ret = 0;
    char buf[1024] = { 0x00 };

    //容易忽略,因为自写代码中没有调用 socket,RTP 有调用,但是没有初始化
    WSADATA dat;
    WSAStartup(MAKEWORD(2, 2), &dat);

    printf("This is recver!!!\n");

    printf("Input local port:");
    scanf("%d", &port);
    printf("recv %s:%d\n", ip, port);

    //设置 RTP 会话参数
    rtpSessionParams.SetOwnTimestampUnit(1.0 / 1);
    rtpSessionParams.SetUsePollThread(true);
```

```
    rtpSessionParams.SetAcceptOwnPackets(true);
    rtpUdpv4Transmissionparams.SetPortbase(port);
    ret = rtpSession.Create(rtpSessionParams, &rtpUdpv4Transmissionparams);   //创建 RTP 会话
    if (ret < 0)
    {
            printf("Failed to RtpSession::Create, ret = %d\n", ret);
    }


    RTPIPv4Address addr(ntohl(inet_addr(ip)), port);
#if 0
    //组播
    rtpSession.JoinMulticastGroup(addr);
#else
    //本机接收,127.0.0.1
    rtpSession.AddDestination(addr);
#endif

    while (true)                                  //开启循环,接收消息
    {
        rtpSession.BeginDataAccess();             //开始数据访问
        if (rtpSession.GotoFirstSourceWithData())
        {
            do {
                RTPPacket *packet;
  //注意以下缩进是为了方便读者阅读,源码中建议使用 Tab 键对齐
  //GetNextPacket 用于接收下一个可用的数据包
  while ((packet = rtpSession.GetNextPacket()) != NULL)
  {
    //RTP 数据包的负载内容的长度,不包含包头
    unsigned int recvSize = packet->GetPayloadLength();
    //RTP 负载的具体内容
    unsigned char * recvData = (unsigned char *)packet->GetPayloadData();
    memcpy(buf, recvData, recvSize);
    buf[recvSize] = '\0';
printf("recv %d, message: %s\n", recvSize, buf);
    rtpSession.DeletePacket(packet);          //用完后,需要删除数据包
  }
                } while (rtpSession.GotoNextSourceWithData());
        }
        rtpSession.EndDataAccess();           //结束数据访问
        //时间间隔,真实项目中要根据帧率设置
        RTPTime::Wait(RTPTime(0, 100));
    }
    return 0;
}
```

在 VS 中运行程序，效果演示，如图 3-29 所示。

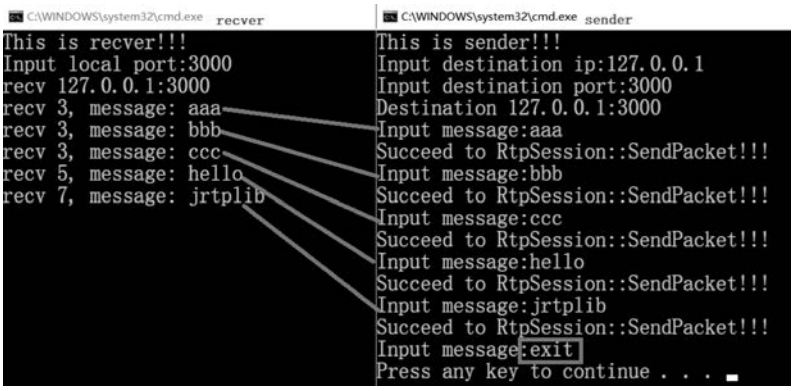

图 3-29　recver 和 sender 的收发效果演示

4. JRTPLIB 的重要数据结构及用法

JRTPLIB 的交互流程中涉及一些重要的数据结构及相关的 API，包括 RTPSession、RTPSessionParams、RTPUDPv4TransmissionParams、RTPIPv4Address 等。

创建一个 RTP 会话需要以下几个步骤：

(1) 使用 RTPSession 类创建一个会话对象 sess/session。

(2) 通过 RTP 会话的参数类 RTPSessionParams 创建一个参数设置的对象 sessparams/sessionparams。具体设置的属性有时间戳单元(SetOwnTimestampUnit)、是否允许接收自己的数据(SetAcceptOwnPackets)等。

(3) 传递参数类(RTPTransmissionParams)下面的有基于 UDP 和 IPv4 的传递参数的派生类(RTPUDPv4TransmissionParams)，也有基于 UDP 和 IPv6 的派生类，一般使用第 1 个。用于设置本机的数据传递参数，主要是 RTP 数据包中的首部参数，本机要使用的端口号(注意是偶数)。

(4) 通过会话对象 sess/session 的 Create 方法调用步骤 3 和步骤 4 中的两个类完成对话的创建。

(5) 在 IPv4 目标地址类(RTPIPv4Address)创建的时候，完成目标端口号和目标 IP 地址。

(6) 通过会话对象 sess/session 的方法 AddDestination 将创建的目标地址对象添加到发送队列中，这里可以添加多个目标地址。

以上是使用 jrtplib 包进行 TCP 通信的通用步骤。端口号选择偶数是因为在 JRTPLIB 中偶数位用于 RTP 通信，然后自动将下一数字(奇数)用于 RTCP 通信。创建 RTP 会话依靠 RTPSession 类的方法，其他的 3 个类 RTPSessionParams、RTPTransmissionParams、RTPIPv4Address 作为 RTPSession 的参数分别为其提供 RTP 会话所需要的时间戳单元、本机的传递参数设置及目标主机的 IP 地址和端口号，样例中一般先创建前两个类，完成

RTP 对话的创建，最后使用第 3 个类添加目标主机的地址，也可以添加多个目标主机，即发往不同的主机上。

JRTPLIB 发送数据是由重载函数实现的，代码如下：

```
1. intjrtplib::RTPSession::SendPacket(const void * data,size_t len)
2. intjrtplib::RTPSession::SendPacket(const void * data,size_t len,uint8_t pt,bool mark,
uint32_t timetamp inc)
```

第 2 个函数相对于第 1 个函数多了 pt、M 标志位及时间戳，这几个参数都是 RTP 首部包中需要的部分。如果需要使用第 1 个函数发送数据，则需要先将这 3 个参数设置成默认值。具体设置可通过 RTP 会话对象的 3 个属性完成，代码如下：

```
session.SetDefaultPayloadType(96);
session.SetDefaultMark(false);
session.SetDefaultTimestampIncrement(160);
```

在创建一个 TCP 对话和发送数据中两次用到了与时间戳相关的参数，这里解释一下这两处分别代表什么意思，以及该如何设置。第一处是 RTP 会话的参数类 RTPSessionParams，这里需要设置基本的时钟单元，例如，一个 8000Hz 的声音信号，接收端每接收一个采样信号的时间要增加 1/8000s。这个是传递这个信号最基本的时钟单元，sessionparams.SetOwnTimestampUnit(1.0/8000.0)。再例如，一个 20fps 的图像，每传送一张图像过去，时钟信号需要增加 1/20s。第二处是 RTP 会话对象的默认时间戳属性，假设每个 RTP 包中包含一个 20ms 的声音信号（对于声音而言，每字节是一个采样信号），则这一个包传送过去，时间需要增加 0.02s/(1/8000s)=160。得到的 160 是以第一处设置的基本时钟单元为单位传送这 20ms 的声音信号所需要的时间。

JRTPLIB 接收端首先要使用 RTPSession 的成员函数 poll 接收来自不同会话参与者的数据（不同的 IP 地址发来的数据，这里称为会话参与者）。在数据接收端，关于 RTP 会话参与者及数据包检索等信息，可以在对 RTPSession 类的成员函数 RTPSession::BegainDataAccess 和 RTPSession::EndDataAceess 的调用之间完成。这种方式是为了保证所使用的数据不会被其他隐藏线程所调用。使用 RTPSession::GotoFirstSource 和 RTPSession::GotoNextSource 两个成员函数的迭代找出所需要的会话参与者，当前被选中的会话参与者的数据包可以通过 RTPSession::GetNextPacket 函数获取，该函数返回一个 RTPPacket 类的对象。下面是接收端示例，代码如下：

```
//chapter/jrtplib/recvdemo.txt
status = session.Poll();
check(status);

status = session.BeginDataAccess();
```

```
if (session.GotoFirstSource())
{
    do
    {
        RTPPacket * packet;
        while ((packet = session.GetNextPacket()) != 0)
        {
            std::cout <<"Got packet with extended sequence number "
<< packet->GetExtendedSequenceNumber()
<<" from SSRC "<< packet->GetSSRC()
<< std::endl;
            session.DeletePacket(packet);
        }
    } while (session.GotoNextSource());
}
session.EndDataAccess();
```

有以下几个注意事项：

（1）RTPSession::GetNextPacket 函数用于获取当前参与者的数据包，而不是下一个。

（2）packet－>GetPayloadData 函数用于获取负载数据。

（3）如果想要获取发送报告、接收报告、SDES 等数据可以通过 RTPSsion 类的成员函数 GetCurrentSourceInfo 函数获取，返回的是一个 RTPSourceData 类。

（4）会话参与者是指数据接收端接收不止一个 IP 主机发送过来的数据，不同的参与者指的是不同的 IP 主机。

另外，RTPSession 提供了一个 OnRTPPacket()虚函数，可以更加方便快捷地接收数据包，代码如下：

```
virtual void OnRTPPacket(RTPPacket * pack,const RTPTime &receivetime, const RTPAddress *
senderaddress)
```

使用 OnRTPPacket 函数时，需要注意几个问题：

（1）重载 OnRTPPacket 替代之前的接收方法，会出现内存泄漏，因为数据处理完后，无法调用 DeletePacket。

（2）即使能通过修改部分源码，释放内存，这种方式也不可取，因为 OnRTPPacket 收到的数据包是即时的，没经过排序。

JRTPLIB 与时间相关的类是 RTPTime，有两个构造函数，代码如下：

```
//这个函数的单位不同，一个是 s，一个是 s + ms
jrtplib::RTPTime::RTPTime(double t)
jrtplib::RTPTime::RTPTime(int64_t seconds, uint32_t microseconds )
```

通过 RTPTime 构造一个延时类，例如 RTPTimedelay(0.02)，此时并没有延时，只是构

造了一个需要延时 0.02s 的 dealy 对象。

RTPTime 有两个成员函数，代码如下：

```
RTPTime jrtplib::RTPTime::CurrentTime()
void jrtplib::RTPTime::Wait ( constRTPTime & delay)
```

(1) RTPTime jrtplib::RTPTime::CurrentTime() 函数用于获取当前的时间，这个时间是以秒为单位，从世界标准时间的 1970 年 1 月 1 日 00:00:00 开始算起为第 0s，代码如下：

```
RTPTimestarttime = RTPTime::CurrentTime();
//构造一个 RTPTime 的对象，并且对其进行类的成员函数进行操作，等同于
RTPTime starttime;
starttime.CurrentTime();
```

(2) jrtplib::RTPTime::Wait (constRTPTime & delay) 函数用于等待延时，传入的是构造函数中的对象，例如 RTPTime::Wait(delay)。

3.4.4 RTP 与 H.264 的相关结构体

使用 RTP 可以封装 H.264 码流的数据包，详细过程可以参考“3.1.2 RTP 封装 H.264”节。这里详细介绍几个重要的结构体，主要包括 RTP_FIXED_HEADER、NALU_HEADER、FU_INDICATOR、FU_HEADER 和 NALU_t 等，详细的解释可参考下面的代码。

关于 H.264 的拆包流程，按照 FU-A 方式主要分为以下几个步骤。

(1) 第 1 个 FU-A 包的 FU indicator：F 应该为当前 NALU 头的 F，而 NRI 应该为当前 NALU 头的 NRI，Type 等于 28，表明它是 FU-A 包。FU header 生成方法：S=1、E=0、R=0，而 Type 等于 NALU 头中的 Type。

(2) 后续的 N 个 FU-A 包的 FU indicator 和第 1 个是完全一样的，如果不是最后一个包，则 FU header 应该为 S=0、E=0、R=0，Type 等于 NALU 头中的 Type。

(3) 最后一个 FU-A 包 FU header 应该为 S=0、E=1、R=0，Type 等于 NALU 头中的 Type。

RTP 与 H.264 的 NALU 数据结构，代码如下：

```
//chapter3/rtpnalu.txt
#define MAX_RTP_PKT_LENGTH  1360                //包长度不要超过 1400
#define H264                96                  //H.264 的负载类型
typedef struct //RTP 固定头
{
    /* Byte 0 */
    unsigned char csrc_len:4;                   /* 4 位:csrn 长度 */
```

```
    unsigned char extension:1;              /* 1 位:扩展 */
    unsigned char padding:1;                /* 1 位:填充 */
    unsigned char version:2;                /* 2 位:版本号 */
    /* Byte 1 */
    unsigned char payload:7;                /* 7 位:负载类型 */
    unsigned char marker:1;                 /* 1 位:标志位 */
    /* Bytes 2, 3 */
    unsigned short seq_no;                  //RTP 序列号
    /* Bytes 4 - 7 */                       //RTP 时间戳
    unsigned long timestamp;
    /* Bytes 8 - 11 */
    unsigned long ssrc;                     /* ssrc */
} RTP_FIXED_HEADER;

/* 1 BYTES */
typedef struct { //NALU_HEADER
    //Byte 0
    unsigned char TYPE:5;                   //类型
    unsigned char NRI:2;                    //优先级
    unsigned char F:1;                      //禁止位

} NALU_HEADER;

typedef struct {
    //Byte 0
    unsigned char TYPE:5;
    unsigned char NRI:2;
    unsigned char F:1;
} FU_INDICATOR; /** // * 1 BYTES */

typedef struct {//FU_HEADER
    //Byte 0
    unsigned char TYPE:5;
    unsigned char R:1;                      //保留位,必须设置为 0,接收者必须忽略该位
    unsigned char E:1;                      //结束位指示分片 NAL 单元的结束
    unsigned char S:1;                      //开始位指示分片 NAL 单元的开始
} FU_HEADER; /* 1 BYTES */

typedef struct
{
    //4B for parameter sets and first slice in picture,
    //3B for everything else
    //如果是 SPS、PPS,则第 1 个 slice 为 4B,其他的为 3B
    int startcodeprefix_len;                //起始码的长度,3 或 4 字节

    //! Length of the NAL unit (Excluding the start code, which does not belong to the NALU)
```

```
    //NALU 的字节数,不包括起始码
    unsigned len;
    unsigned max_size;            //! Nal Unit Buffer size,NALU 的缓冲区大小
    int forbidden_bit;            //! should be always FALSE,禁止位
    int nal_reference_idc;        //! NALU_PRIORITY_xxxx,优先级
    int nal_unit_type;            //! NALU_TYPE_xxxx,NALU 的类型

    //! contains the first Byte followed by the EBSP
    unsigned char * buf;          //真正的帧数据缓冲区

    //! true, if packet loss is detected,是否丢包
    unsigned short lost_packets;
} NALU_t;
```

3.4.5 使用 JRTPLIB 发送 H.264 码流

使用JRTPLIB发送H.264码流,需要根据起始码获取NALU,自动封装为RTP格式,然后发送出去,核心代码及解释如下所示,完整的代码可以参考课件资料中的jrtplibdemoSendH264工程。重要的结构体及注释说明可参考“3.4.4 RTP与H.264的相关结构体”。

```
//chapter3/jrtplib/sendh264.cpp
#ifndef H264_H_
#define H264_H_
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


#define PACKET_BUFFER_END       (unsigned int)0x00000000
#define MAX_RTP_PKT_LENGTH      1360
#define H264                    96


#define SSRC          100
#define DEST_IP_STR   "127.0.0.1"
#define DEST_PORT     12500
#define BASE_PORT     9400

//static bool flag = true;
static int info2 = 0, info3 = 0;
RTP_FIXED_HEADER     * rtp_hdr;
FILE * bits = NULL;           //!< the bit stream file:位流文件
NALU_HEADER          * nalu_hdr;
```

```
FU_INDICATOR         * fu_ind;
FU_HEADER          * fu_hdr;

//查找开始字符 0x000001
static int FindStartCode2 (unsigned char * Buf)
{
    if(Buf[0]!= 0 || Buf[1]!= 0 || Buf[2] != 1) return 0;  //判断是否为 0x000001。如果是,则返回 1
    else return 1;
}

//查找开始字符 0x00000001
static int FindStartCode3 (unsigned char * Buf)
{
    //判断是否为 0x00000001,如果是,则返回 1
    if(Buf[0]!= 0 || Buf[1]!= 0 || Buf[2] != 0 || Buf[3] != 1) return 0;else return 1;
}

//打开 H.264 裸流文件
void OpenBitstreamFile (char * fn)
{
    if (NULL == (bits = fopen(fn, "rb")))     //操作二进制必须为 rb/wb
    {
            printf("open file error\n");
            exit(0);
    }
}


//为 NALU_t 结构体分配内存空间
NALU_t * AllocNALU(int buffersize)
{
    NALU_t * n;

    if ((n = (NALU_t * )calloc (1, sizeof (NALU_t))) == NULL)
    {
            printf("AllocNALU: n");
            exit(0);
    }

    n -> max_size = buffersize;
    //默认 80000 字节的大小
    if ((n -> buf = (unsigned char * )calloc (buffersize, sizeof (char))) == NULL)
    {
            free (n);
            printf ("AllocNALU: n -> buf");
            exit(0);
```

```
    }

    return n;
}


//释放
void FreeNALU(NALU_t *n)
{
    if (n)
    {
        if (n->buf)
        {
            free(n->buf);
            n->buf = NULL;
        }
        free (n);
    }
}


//这个函数的输入为一个 NAL 结构体
//得到一个完整的 NALU 并保存在 NALU_t 的 buf 中,获取它的长度,填充 F、IDC、TYPE 位
//并且返回两个开始字符之间间隔的字节数,即包含前缀的 NALU 的长度
int GetAnnexbNALU (NALU_t *nalu)
{
    int pos = 0;
    int StartCodeFound, rewind;
    unsigned char *Buf;

    if ((Buf = (unsigned char*)calloc (nalu->max_size , sizeof(char))) == NULL)
            printf ("GetAnnexbNALU: Could not allocate Buf memory\n");

    nalu->startcodeprefix_len = 3;              //初始化码流序列的开始字符为 3 字节

    if (3 != fread (Buf, 1, 3, bits))           //从码流中读 3 字节
    {
            free(Buf);
            return 0;
    }
    info2 = FindStartCode2 (Buf);               //判断是否为 0x000001
    if(info2 != 1)
    {
            //如果不是,则再读一字节
            if(1 != fread(Buf+3, 1, 1, bits))   //读一字节
            {
```

```
                free(Buf);
                return 0;
            }
            info3 = FindStartCode3 (Buf);        //判断是否为 0x00000001
            if (info3 != 1)                      //如果不是,则返回 - 1
            {
                free(Buf);
                return - 1;
            }
            else
            {
                    //如果是 0x00000001,则得到的开始前缀为 4 字节
                    pos = 4;
                    nalu - > startcodeprefix_len = 4;
        }
    }
    else
    {
            //如果是 0x000001,则得到的开始前缀为 3 字节
            nalu - > startcodeprefix_len = 3;
            pos = 3;
    }
    //查找下一个开始字符的标志位
    StartCodeFound = 0;
    info2 = 0;
    info3 = 0;

    while (!StartCodeFound)
    {
            if (feof (bits))                     //判断是否到了文件尾
            {
    nalu - > len = (pos - 1) - nalu - > startcodeprefix_len;
    memcpy (nalu - > buf, &Buf[nalu - > startcodeprefix_len], nalu - > len);
    nalu - > forbidden_bit = nalu - > buf[0] & 0x80;       //1 bit
    nalu - > nal_reference_idc = nalu - > buf[0] & 0x60;   //2 bit
    nalu - > nal_unit_type = (nalu - > buf[0]) & 0x1f;     //5 bit
    free(Buf);
    return pos - 1;
            }
            Buf[pos++] = fgetc (bits);                   //将一字节读到 Buf 中
            info3 = FindStartCode3(&Buf[pos - 4]);       //判断是否为 0x00000001
            if(info3 != 1)
            info2 = FindStartCode2(&Buf[pos - 3]);       //判断是否为 0x000001
            StartCodeFound = (info2 == 1 || info3 == 1);
    }
```

```
        //Here, we have found another start code
        //(and read length of startcode Bytes more than we should
        //have. Hence, go back in the file
        //已经发现了一个起始码
        rewind = (info3 == 1)? -4 : -3;

        //把文件指针指向前一个 NALU 的末尾
        if (0 != fseek (bits, rewind, SEEK_CUR))
        {
                free(Buf);
                printf("GetAnnexbNALU: Cannot fseek in the bit stream file");
        }

        //Here the Start code, the complete NALU,
        //and the next start code is in the Buf.
        //完整的 NALU 已经在缓冲区中了
        //The size of Buf is pos
        //pos + rewind are the number of Bytes excluding the next
        //start code, and (pos + rewind) - startcodeprefix_len
        //is the size of the NALU excluding the start code

        nalu->len = (pos + rewind) - nalu->startcodeprefix_len;
        //复制一个完整 NALU,不复制起始前缀 0x000001 或 0x00000001
        memcpy (nalu->buf, &Buf[nalu->startcodeprefix_len], nalu->len);
        nalu->forbidden_bit = nalu->buf[0] & 0x80;          //1 bit
        nalu->nal_reference_idc = nalu->buf[0] & 0x60;      //2 bit
        nalu->nal_unit_type = (nalu->buf[0]) & 0x1f;        //5 bit
        free(Buf);

        //返回两个开始字符之间间隔的字节数,即包含前缀的 NALU 的长度
        return (pos + rewind);}

    #endif
```

可以使用 VLC 直接打开 show.sdp 来播放 RTP 封装的 H.264 码流。本地 IP 地址为 127.0.0.1,端口号是 12500,使用 RTP 封装格式,帧率为 25,RTP 负载类型是 H.264,代码如下:

```
m = video 12500 RTP/AVP 96
a = rtpmap:96 H264
a = framerate:25
c = IN IP4 127.0.0.1
```

使用 JRTPLIB 发送 H.264 码流,以及 VLC 播放的效果,如图 3-30 所示。

图 3-30　JRTPLIB 发送 H.264 码流后用 VLC 播放

3.5　RTP 扩展头结构

RTP 提供扩展机制允许实现个性化：某些新的与负载格式独立的功能要求的附加信息在 RTP 数据包头中传输。设计此方法可以使其他没有扩展的交互忽略此头扩展。

3.5.1　RTP 单扩展头

RTP 扩展头跟在确定头后，如果有 CSRC，就跟在 CSRC 后。RTP 单扩展头的格式如图 3-31 所示。

图 3-31　RTP 单扩展头结构

若 RTP 固定头中的可扩展比特位置为 1，则一个长度可变的头扩展部分被加到 RTP 固定头之后。头扩展包含 16b 的长度域，指示扩展项中 32b 的个数，不包括 4 字节扩展头（因此零是有效值）。RTP 固定头之后只允许有一个头扩展。为允许多个互操作独立生成不同的头扩展，或某种特定实现有多种不同的头扩展，扩展项的前 16b 用以识别标识符或参

数。这 16b 的格式由具体实现的上层协议定义。基本的 RTP 说明并不定义任何头扩展本身。RTP 标准可以参考 RFC 3550，网址为 https://datatracker.ietf.org/doc/html/rfc3550#section-5.3.1。

3.5.2 RTP 多扩展头

上面介绍的这种形式只能够附加一个扩展头，为了支持多个扩展头，RFC 5285 对 defined by profile 进行了扩展，可以参考网址 https://datatracker.ietf.org/doc/html/rfc5285。

扩展头为 One-Byte Header 的情况，如图 3-32 所示。RTP 头后的第 1 个 16b 为固定的 0XBEDE 标志，意味着这是一个 One-Byte 扩展，length=3 说明后面有 3 个扩展头，每个扩展头首先以一字节开始，前 4 位是这个扩展头的 ID，后 4 位是 data 的长度减 1。例如 L=0 意味着后面有 1B 的 data，同理第 2 个扩展头的 L=1 说明后面还有 2B 的 data，但是注意，其后没有紧跟第 3 个扩展头，而是添加了 2B 大小的全 0 的 data，这是为了使填充对齐，因为扩展头是以 32b 填充对齐的。

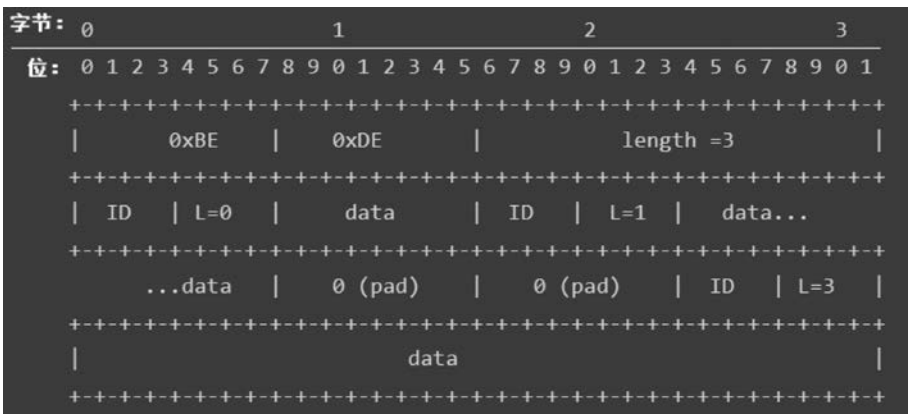

图 3-32 RTP 扩展头 One-Byte Header 结构

扩展头为 Two-Byte Header 的情况，如图 3-33 所示。RTP 头后的第 1 个 16b 为 0x100+appbits，其中 appbits 可以用来填充应用层级别的数据。

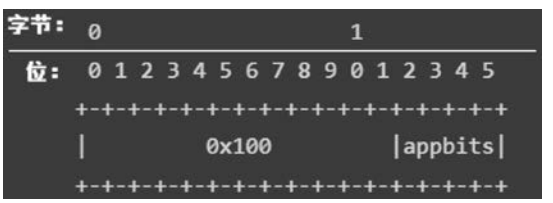

图 3-33 RTP 扩展头 Two-Byte Header 结构

扩展头为 Two-Byte Header 的一个案例，如图 3-34 所示。可以看到开头为 0x100＋0x0，接下来的 length＝3 表示有 3 个头，然后是扩展头和数据，扩展头除了 ID 和 L 相对于 One-Byte Header 从 4b 变成了 8b 之外，其余都一样。

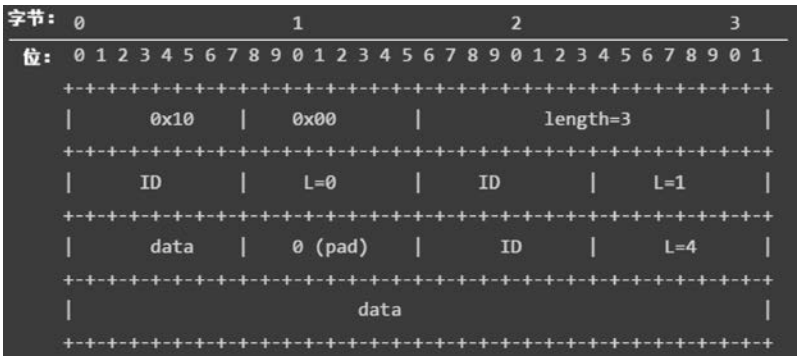

图 3-34　RTP 扩展头 Two-Byte Header 案例