原创 七伤之酒 于 2023-06-05 20:34:45 发布 138 收藏 版权
文章标签： 服务器 网络 tcp

Rtsp服务器搭建(荷载H264和AAC)

什么是RTSP协议？

RTSP是一个实时传输流协议，是一个应用层的协议

通常说的RTSP包括RTSP协议、RTP协议、RTCP协议

对于这些协议的作用简单的理解如下

RTSP协议:负责服务器与客户端之间的请求与响应

RTP协议:负责传输媒体数据

RTCP协议:在RTP传输过程中提供传输信息

rtsp承载与rtp和rtcp之上，rtsp并不会发送媒体数据，而是使用rtp协议传输

rtp并没有规定发送方式，可以选择udp发送或者tcp发送

RTSP协议详解

rtsp的交互过程就是客户端请求，服务器响应，下面看一看请求和响应的数据格式

RTSP客户端请求

```
method url vesion\r\n
CSeq: x\r\n
xxx\r\n
...
\r\n
```

method：方法，表明这次请求的方法，rtsp定义了很多方法，稍后介绍

url：格式一般为rtsp://ip:port/session，ip表主机ip，port表端口好，如果不写那么就是默认端口，rtsp的默认端口为554，session表明请求哪一个会话

version：表示rtsp的版本，现在为RTSP/1.0

CSeq：序列号，每个RTSP请求和响应都对应一个序列号，序列号是递增的

RTSP服务端的响应格式

```
vesion 200 OK\r\n
CSeq: x\r\n
xxx\r\n
...
\r\n
```

version：表示rtsp的版本，现在为 `RTSP/1.0`

CSeq：序列号，这个必须与对应请求的序列号相同

RTSP请求的常用方法

方法	描述
OPTIONS	获取服务端提供的可用方法
DESCRIBE	向服务端获取对应会话的媒体描述信息
SETUP	向服务端发起建立请求，建立连接会话
PLAY	向服务端发起播放请求
TEARDOWN	向服务端发起关闭连接会话请求

OPTIONS

C->S

```
OPTIONS rtsp://192.168.31.115:8554/live RTSP/1.0\r\n
CSeq: 2\r\n
\r\n
```

S->C

```
RTSP/1.0 200 OK\r\n
CSeq: 2\r\n
Public: OPTIONS, DESCRIBE, SETUP, TEARDOWN, PLAY\r\n
\r\n
```

DESCRIBE

C->S

```
DESCRIBE rtsp://192.168.31.115:8554/live RTSP/1.0\r\n
CSeq: 3\r\n
Accept: application/sdp\r\n
\r\n
```

S->C

```
RTSP/1.0 200 OK \r\n
CSeq: 2\r\n
Content-Base: rtsp://192.168.31.115:8554\r\n
Content-type: application/sdp\r\n
Content-length: 311\r\n

v=0\r\n
o=-91685885859 1 IN IP4 192.168.72.129\r\n
t=0 0\r\n
a=control:*\r\n
m=video 0 RTP/AVP 96\r\n
a=rtpmap:96 H264/90000\r\n
a=control:track0\r\n
m=audio 1 RTP/AVP/TCP 97\r\n
a=rtpmap:97 mpeg4-generic/44100/2\r\n
a=fmtp:97 profile-level-id=1;mode=AAC-hbr;sizelength=13;indexlength=3;indexdeltalength=3;config=1210;\r\n
a=control:track1\r\n

```

SETUP

C->S

```
SETUP rtsp://192.168.31.115:8554/live/track0 RTSP/1.0\r\n
CSeq: 4\r\n
Transport: RTP/AVP;unicast;client_port=54492-54493\r\n
\r\n
```

客户端发送建立请求，请求建立连接会话，准备接收音视频数据

解析一下Transport: RTP/AVP;unicast;client_port=54492-54493\r\n

RTP/AVP：表示RTP通过UDP发送，如果是RTP/AVP/TCP则表示RTP通过TCP发送

unicast：表示单播，如果是multicast则表示多播

client_port=54492-54493：由于这里希望采用的是RTP OVER UDP，所以客户端发送了两个用于传输数据的端口，客户端已经将这两个端口绑定到两个udp套接字上，54492表示是RTP端口，54493表示RTCP端口(RTP端口为某个偶数，RTCP端口为RTP端口+1)

S->C

```
RTSP/1.0 200 OK\r\n
CSeq: 4\r\n
Transport: RTP/AVP;unicast;client_port=54492-54493;server_port=56400-56401\r\n
Session: 66334873\r\n
\r\n
```

服务端接收到请求之后，得知客户端要求采用RTP OVER UDP发送数据，单播，客户端用于传输RTP数据的端口为54492，RTCP的端口为54493

服务器也有两个udp套接字，绑定好两个端口，一个用于传输RTP，一个用于传输RTCP，这里的端口号为56400-56401

之后客户端会使用54492-54493这两端口和服务器通过udp传输数据，服务器会使用56400-56401这两端口和这个客户端传输数据

PLAY

C->S

```
PLAY rtsp://192.168.31.115:8554/live RTSP/1.0\r\n
CSeq: 5\r\n
Session: 66334873\r\n
Range: npt=0.000-\r\n
\r\n
```

客户端请求播放媒体

S->C

```
RTSP/1.0 200 OK\r\n
CSeq: 5\r\n
Range: npt=0.000-\r\n
Session: 66334873; timeout=60\r\n
\r\n
```

服务器回复之后，会开始使用RTP通过udp向客户端的54492端口发送数据

TEARDOWN

C->S

```
TEARDOWN rtsp://192.168.31.115:8554/live RTSP/1.0\r\n
CSeq: 6\r\n
Session: 66334873\r\n
\r\n
```

S->C

```
RTSP/1.0 200 OK\r\n
CSeq: 6\r\n
\r\n
```

RTP协议

RTP头部

[外链图片转存失败,源站可能有防盗链机制,建议将图片保存下来直接上传(img-bTovyteN-1685968165403)(图片/image-20230604215011643.png)]

版本号(V)：2Bit，用来标志使用RTP版本

填充位§：1Bit，如果该位置位，则该RTP包的尾部就包含填充的附加字节

扩展位(X)：1Bit，如果该位置位，则该RTP包的固定头部后面就跟着一个扩展头部

CSRC技术器(CC)：4Bit，含有固定头部后面跟着的CSRC的数据

标记位(M)：1Bit，该位的解释由配置文档来承担

载荷类型(PT)：7Bit，标识了RTP载荷的类型

序列号(SN)：16Bit，发送方在每发送完一个RTP包后就将该域的值增加1，可以由该域检测包的丢失及恢复

包的序列。序列号的初始值是随机的

时间戳：32比特，记录了该包中数据的第一个字节的采样时刻

同步源标识符(SSRC)：32比特，同步源就是RTP包源的来源。在同一个RTP会话中不能有两个相同的SSRC值

贡献源列表(CSRC List)：0-15项，每项32比特，这个不常用

RTP建立

RtpHeader

```
class RtpHeader
{
public:
    /*byte 0*/
    uint8_t csrcLen : 4;
    uint8_t extension : 1;
    uint8_t padding : 1;
    uint8_t version : 2;
    /*byte 1*/
    uint8_t payloadType : 7;
    uint8_t marker : 1;
    /*bytes 2,3*/
    uint16_t seq;
    /*bytes 4-7*/
    uint32_t timestamp;
    /*bytes 8-11*/
    uint32_t ssrc;
```

```
};
```

RtpPacket

```
class RtpPacket
{
public:
 RtpHeader rtpHeader;
    uint8_t payload[0];
};
```

初始化RTP包

```
void rtpHeaderInit(RtpPacket*rtpPacket,uint8_t csrclen,uint8_t extension,
        uint8_t padding,uint8_t version,uint8_t payloadType,uint8_t marker,
        uint16_t seq,uint32_t timestamp,uint32_t ssrc);

```

```
void rtpHeaderInit(RtpPacket* rtpPacket, uint8_t csrclen, uint8_t extension, uint8_t padding, uint8_t version, 
{
    rtpPacket->rtpHeader.csrcLen = csrclen;
    rtpPacket->rtpHeader.extension = extension;
    rtpPacket->rtpHeader.padding = padding;
    rtpPacket->rtpHeader.version = version;
    rtpPacket->rtpHeader.payloadType = payloadType;
    rtpPacket->rtpHeader.marker = marker;
    rtpPacket->rtpHeader.seq = seq;
    rtpPacket->rtpHeader.timestamp = timestamp;
    rtpPacket->rtpHeader.ssrc = ssrc;
}
```

以TCP形式发送rtp数据包

```
int rtpSendPacketOverTcp(int clientSockfd, struct RtpPacket* rtpPacket, uint32_t dataSize, char channel);
```

```
int rtpSendPacketOverTcp(int clientSockfd, struct RtpPacket* rtpPacket, uint32_t dataSize, char channel)
{

    rtpPacket->rtpHeader.seq = htons(rtpPacket->rtpHeader.seq);
    rtpPacket->rtpHeader.timestamp = htonl(rtpPacket->rtpHeader.timestamp);
    rtpPacket->rtpHeader.ssrc = htonl(rtpPacket->rtpHeader.ssrc);

    uint32_t rtpSize = RTP_HEADER_SIZE + dataSize;
    char* tempBuf = (char*)malloc(4 + rtpSize);
    tempBuf[0] = 0x24;//$
    tempBuf[1] = channel;// 0x00;
    tempBuf[2] = (uint8_t)(((rtpSize) & 0xFF00) >> 8);
    tempBuf[3] = (uint8_t)((rtpSize) & 0xFF);
    memcpy(tempBuf + 4, (char*)rtpPacket, rtpSize);

    int ret = send(clientSockfd, tempBuf, 4 + rtpSize, 0);

    rtpPacket->rtpHeader.seq = ntohs(rtpPacket->rtpHeader.seq);
    rtpPacket->rtpHeader.timestamp = ntohl(rtpPacket->rtpHeader.timestamp);
    rtpPacket->rtpHeader.ssrc = ntohl(rtpPacket->rtpHeader.ssrc);

    free(tempBuf);
    tempBuf = NULL;

    return ret;
}
```

以UDP形式发送rtp数据包

```
int rtpSendPacketOverUdp(int serverRtpSockfd, const char* ip, int16_t port, struct RtpPacket* rtpPacket, uint32_
```

```
int rtpSendPacketOverUdp(int serverRtpSockfd, const char* ip, int16_t port, struct RtpPacket* rtpPacket, uint32_
{

    struct sockaddr_in addr;
    int ret;

    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    addr.sin_addr.s_addr = inet_addr(ip);

    rtpPacket->rtpHeader.seq = htons(rtpPacket->rtpHeader.seq);
    rtpPacket->rtpHeader.timestamp = htonl(rtpPacket->rtpHeader.timestamp);
    rtpPacket->rtpHeader.ssrc = htonl(rtpPacket->rtpHeader.ssrc);

    ret = sendto(serverRtpSockfd, (char*)rtpPacket, dataSize + RTP_HEADER_SIZE, 0,
        (struct sockaddr*)&addr, sizeof(addr));

    rtpPacket->rtpHeader.seq = ntohs(rtpPacket->rtpHeader.seq);
    rtpPacket->rtpHeader.timestamp = ntohl(rtpPacket->rtpHeader.timestamp);
    rtpPacket->rtpHeader.ssrc = ntohl(rtpPacket->rtpHeader.ssrc);

    return ret;

}
```

一、建立套接字

程序从进入main函数之后就创建服务器TCP套接字，绑定端口，然后开始监听端口

```
//建立套接字
int ServerSocket;
int ServerRtcpSocket, ServerRtpSocket;
//创建TCP套接字
ServerSocket = CreateTcpSocket();
if (ServerSocket < 0)
{
    cout << "TCP create fail !!!" << endl;
    exit(0);
}
//绑定端口和地址
if (BindSocketAddr(ServerSocket,"0.0.0.0",SERVER_PORT)<0)
{
    cout << "bind fail !!!" << endl;
    exit(0);
}
//监听端口
if (listen(ServerSocket, 5)<0)
{
    cout << "listen !!!" << endl;
    exit(0);
}
cout << "rtsp://"<< SERVER_IP <<":" << SERVER_PORT << endl;
```

二、接受客户端连接

在while循环中接受客户端消息,并利用函数进行处理

```
while (true)
{
    int ClientSocket;
    char  ClientIp[40];
    int ClientPort;
    //接收客户端消息
    ClientSocket = AcceptClient(ServerSocket, ClientIp, &ClientPort);
    if (ClientSocket < 0)
    {
        printf("failed to accept client\n");
        return -1;
    }
    //打印客户端信息
    cout << "accept client;client ip:" << ClientIp << ",client port:" << ClientPort << endl;
    //接收消息并做出响应
    doClient(ClientSocket, ClientIp, ClientPort);
}
```

三、解析请求

```
while (true) {
    int recvLen;

    recvLen = recv(clientSockfd, rBuf, BUF_MAX_SIZE, 0);
    if (recvLen <= 0) {
        break;
    }

    rBuf[recvLen] = '\0';
    printf("Accept request rBuf = %s \n", rBuf);

    const char* sep = "\n";
    //获取第一行数据
    char* line = strtok(rBuf, sep);
    while (line) {
        if (strstr(line, "OPTIONS") ||
            strstr(line, "DESCRIBE") ||
            strstr(line, "SETUP") ||
            strstr(line, "PLAY")) {
            if (sscanf(line, "%s %s %s\r\n", method, url, version) != 3) {
                // error
            }
        }
        else if (strstr(line, "CSeq")) {
            if (sscanf(line, "CSeq: %d\r\n", &CSeq) != 1) {
                // error
            }
        }
        else if (!strncmp(line, "Transport:", strlen("Transport:"))) {
            // Transport: RTP/AVP/UDP;unicast;client_port=13358-13359
            // Transport: RTP/AVP;unicast;client_port=13358-13359

            if (sscanf(line, "Transport: RTP/AVP/TCP;unicast;interleaved=0-1\r\n") != 0) {
                // error
                printf("parse Transport error \n");
            }
        }
        //获取下一行数据
        line = strtok(NULL, sep);
    }
```

四、处理请求

解析完客户端命令后，会调用相应的请求，处理完之后将要发送的消息打印到sbuf发送给客户端

```
if (!strcmp(method, "OPTIONS")) {
    if (handleCmd_OPTIONS(sBuf, CSeq))
    {
        printf("failed to handle options\n");
        break;
    }
}
else if (!strcmp(method, "DESCRIBE")) {
    if (handleCmd_DESCRIBE(sBuf, CSeq, url))
    {
        printf("failed to handle describe\n");
        break;
    }
}
else if (!strcmp(method, "SETUP")) {
    if (handleCmd_SETUP(sBuf, CSeq))
    {
        printf("failed to handle setup\n");
        break;
    }
}
else if (!strcmp(method, "PLAY")) {
    if (handleCmd_PLAY(sBuf, CSeq))
    {
        printf("failed to handle play\n");
        break;
    }
}
else {
    printf("Undefined method = %s \n", method);
    break;
}
printf("Response sBuf = %s \n", sBuf);
//向客户端回复消息
send(clientSockfd, sBuf, strlen(sBuf), 0);
```

五、AAC RTP打包发送

接受到"PLAY"消息后，服务器开始循环发送AAC数据

```
while (true)
{
    //读取ADTS头部
    ret = fread(frame, 1, 7, fp);
    if (ret <= 0)
    {
        printf("fread err\n");
        break;
    }
    printf("fread ret=%d \n", ret);
    //解析头部
    if (parseAdtsHeader(frame, &adtsHeader) < 0)
    {
```

```
        printf("parseAdtsHeader err\n");
        break;
    }
    //读取一帧
    ret = fread(frame, 1, adtsHeader.aacFrameLength - 7, fp);
    if (ret <= 0)
    {
        printf("fread err\n");
        break;
    }
    //Rtp打包发送
    rtpSendAACFrame(clientSockfd,
                    rtpPacket, frame, adtsHeader.aacFrameLength - 7);
    usleep(23223);//1000/43.06 * 1000
}
```

六、H264 RTP打包发送

接受到"PLAY"消息后，服务器开始循环发送H264数据

```
while (true) {
    frameSize = getFrameFromH264File(fp, frame, 500000);
    if (frameSize < 0)
    {
        printf("Read %s end , frameSize=%d \n", H264_FILE_NAME, frameSize);
        break;
    }

    if (startCode3(frame))
        startCode = 3;
    else
        startCode = 4;

    frameSize -= startCode;
    rtpSendH264Frame(clientSockfd, rtpPacket, frame + startCode, frameSize);

    rtpPacket->rtpHeader.timestamp += 90000 / 25;
    usleep(40000);//1000/25 * 1000
```

函数实现

建立aacheader

```
struct AdtsHeader
{
    unsigned int syncword;  //12 bit 同步字 '1111 1111 1111'，说明一个ADTS帧的开始
    unsigned int id;        //1 bit MPEG 标示符， 0 for MPEG-4, 1 for MPEG-2
    unsigned int layer;     //2 bit 总是'00'
    unsigned int protectionAbsent;  //1 bit 1表示没有crc，0表示有crc
    unsigned int profile;           //1 bit 表示使用哪个级别的AAC
    unsigned int samplingFreqIndex; //4 bit 表示使用的采样频率
    unsigned int privateBit;        //1 bit
    unsigned int channelCfg; //3 bit 表示声道数
    unsigned int originalCopy;         //1 bit
    unsigned int home;                  //1 bit

    /*下面的为改变的参数即每一帧都不同*/
    unsigned int copyrightIdentificationBit;   //1 bit
    unsigned int copyrightIdentificationStart; //1 bit
    unsigned int aacFrameLength;               //13 bit 一个ADTS帧的长度包括ADTS头和AAC原始流
    unsigned int adtsBufferFullness;           //11 bit 0x7FF 说明是码率可变的码流

    /* number_of_raw_data_blocks_in_frame
     *  表示ADTS帧中有number_of_raw_data_blocks_in_frame + 1个AAC原始帧
     *  所以说number_of_raw_data_blocks_in_frame == 0
     *  表示说ADTS帧中有一个AAC数据块并不是说没有。(一个AAC原始帧包含一段时间内1024个采样及相关数据)
     * /
    unsigned int numberOfRawDataBlockInFrame; //2 bit
};
```

解析aacheader

```
static int parseAdtsHeader(uint8_t* in, struct AdtsHeader* res)
{
    static int frame_number = 0;
    memset(res, 0, sizeof(*res));

    if ((in[0] == 0xFF) && ((in[1] & 0xF0) == 0xF0))
    {
     r e s->id = ((unsigned int)in[1] & 0x08) >> 3;
     r e s->layer = ((unsigned int)in[1] & 0x06) >> 1;
     r e s->protectionAbsent = (unsigned int)in[1] & 0x01;
     r e s->profile = ((unsigned int)in[2] & 0xc0) >> 6;
     r e s->samplingFreqIndex = ((unsigned int)in[2] & 0x3c) >> 2;
     r e s->privateBit = ((unsigned int)in[2] & 0x02) >> 1;
     r e s->channelCfg = ((((unsigned int)in[2] & 0x01) << 2) | (((unsigned int)in[3] & 0xc0) >> 6));
     r e s->originalCopy = ((unsigned int)in[3] & 0x20) >> 5;
     r e s->home = ((unsigned int)in[3] & 0x10) >> 4;
     r e s->copyrightIdentificationBit = ((unsigned int)in[3] & 0x08) >> 3;
     r e s->copyrightIdentificationStart = (unsigned int)in[3] & 0x04 >> 2;
     r e s->aacFrameLength = (((((unsigned int)in[3]) & 0x03) << 11) |
            (((unsigned int)in[4] & 0xFF) << 3) |
            ((unsigned int)in[5] & 0xE0) >> 5);
     r e s->adtsBufferFullness = (((unsigned int)in[5] & 0x1f) << 6 |
            ((unsigned int)in[6] & 0xfc) >> 2);
     r e s->numberOfRawDataBlockInFrame = ((unsigned int)in[6] & 0x03);

        return 0;
    }
    else
    {
        printf("failed to parse adts header\n");
        return -1;
    }
}
```

用rtp格式打包并发送AAC音频流数据

```
static int rtpSendAACFrame(int clientSockfd,
                           struct RtpPacket* rtpPacket, uint8_t* frame, uint32_t frameSize) {
    int ret;

    rtpPacket->payload[0] = 0x00;
    rtpPacket->payload[1] = 0x10;
    rtpPacket->payload[2] = (frameSize & 0x1FE0) >> 5; //高8位
    rtpPacket->payload[3] = (frameSize & 0x1F) << 3; //低5位

    memcpy(rtpPacket->payload + 4, frame, frameSize);


    ret = rtpSendPacketOverTcp(clientSockfd, rtpPacket, frameSize + 4, 0x02);
```

```
    if (ret < 0)
    {
        printf("failed to send rtp packet\n");
        return -1;
    }

    rtpPacket->rtpHeader.seq++;

    /*
     * 如果采样频率是44100
     * 一般AAC每个1024个采样为一帧
     * 所以一秒就有 44100 / 1024 = 43帧
     * 时间增量就是 44100 / 43 = 1025
     * 一帧的时间为 1 / 43 = 23ms
     * /
    rtpPacket->rtpHeader.timestamp += 1025;

    return 0;
}
```

创建TCP套接字

```
static int CreateTcpSocket()
{
    int sockfd;
    int on=1;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        return -1;
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (const char*)&on, sizeof(on));
    return sockfd;
}
```

创建UDP套接字

```
static int CreateUdpSocket()
{
    int sockfd;
    int on = 1;
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0)
        return -1;
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (const char*)&on, sizeof(on));
    return sockfd;
}
```

绑定端口和地址

```
static int BindSocketAddr(int sockfd,const char *ip,int port)
{
    sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    addr.sin_addr.s_addr = inet_addr(ip);
    if (bind(sockfd, (sockaddr*)&addr, sizeof(addr))<0)
        return -1;
    return 0;
}
```

连接客户端并接收客户端信息

```
static int AcceptClient(int sockfd,char *ip,int *port)
{
    int clientfd;
    socklen_t len = 0;
    sockaddr_in addr;
    memset(&addr, 0, sizeof(addr));
    len = sizeof(addr);
    clientfd = accept(sockfd, (sockaddr*)&addr, &len);
    if (clientfd < 0)
        return -1;
    strcpy(ip, inet_ntoa(addr.sin_addr));
    *port = ntohs(addr.sin_port);
    return clientfd;
}
```

判断是不是非h264码流(0 0 0 1)

```
static inline int startCode3(char* buf)
{
    if (buf[0] == 0 && buf[1] == 0 && buf[2] == 1)
        return -1;
    else
        return 0;
}
```

判断是不是非h264码流(0 0 0 0 1)

```
static inline int startCode4(char* buf)
{
    if (buf[0] == 0 && buf[1] == 0 && buf[2] == 0&&buf[3]==1)
        return -1;
    else
        return 0;
}
```

找下一段h264数据

```
static char* findNextStartCode(char* buf, int len)
{
    int i;
    if (len < 3)
        return NULL;
    for (i = 0; i < len - 3; i++)
    {
        if (startCode3(buf) || startCode4(buf))
            return buf;
        ++buf;
    }
    if (startCode3(buf))
        return buf;
    return NULL;
}
```

获得h264码流大小

```
static int getFrameFromH264File(int fd,char*frame,int size)
{
    int rSize, frameSize;
```

```
    char* nextStartCode;
    if (fd < 0)
        return fd;
    rSize = read(fd, frame, size);
    if (!startCode3(frame) && !startCode4(frame))
        return -1;
    nextStartCode = findNextStartCode(frame + 3, rSize - 3);
    if (!nextStartCode)
        return -1;
    else
    {
        frameSize = nextStartCode - frame;
        lseek(fd, frameSize - rSize, SEEK_CUR);
    }
    return frameSize;
}
```

用rtp格式打包并发送H264视频流数据

```
static int rtpSendH264Frame(int clientSockfd,
                            struct RtpPacket* rtpPacket, char* frame, uint32_t frameSize)
{

    uint8_t naluType; // nalu第一个字节
    int sendByte = 0;
    int ret;

    naluType = frame[0];

    printf("%s frameSize=%d \n", __FUNCTION__, frameSize);

    if (frameSize <= RTP_MAX_PKT_SIZE) // nalu长度小于最大包场：单一NALU单元模式
    {

        //*   0 1 2 3 4 5 6 7 8 9
        //*  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        //*  |F|NRI|  Type   | a single NAL unit ... |
        //*  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

        memcpy(rtpPacket->payload, frame, frameSize);
        ret = rtpSendPacketOverTcp(clientSockfd, rtpPacket, frameSize, 0x00);
        if (ret < 0)
            return -1;

        rtpPacket->rtpHeader.seq++;
        sendByte += ret;
        if ((naluType & 0x1F) == 7 || (naluType & 0x1F) == 8) // 如果是SPS、PPS就不需要加时间戳
        {

        }

    }
    else // nalu长度小于最大包：分片模式
    {

        //*  0                   1                   2
        //*  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
        //* +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        //* | FU indicator  |   FU header   |   FU payload   ...  |
        //* +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+



        //*     FU Indicator
        //*    0 1 2 3 4 5 6 7
        //*   +-+-+-+-+-+-+-+-+
        //*   |F|NRI|  Type   |
        //*   +---------------+



        //*      FU Header
        //*    0 1 2 3 4 5 6 7
        //*   +-+-+-+-+-+-+-+-+
        //*   |S|E|R|  Type   |
        //*   +---------------+


        int pktNum = frameSize / RTP_MAX_PKT_SIZE;       // 有几个完整的包
        int remainPktSize = frameSize % RTP_MAX_PKT_SIZE; // 剩余不完整包的大小
        int i, pos = 1;

        // 发送完整的包
        for (i = 0; i < pktNum; i++)
        {
            rtpPacket->payload[0] = (naluType & 0x60) | 28;
            rtpPacket->payload[1] = naluType & 0x1F;

            if (i == 0) //第一包数据
                rtpPacket->payload[1] |= 0x80; // start
            else if (remainPktSize == 0 && i == pktNum - 1) //最后一包数据
                rtpPacket->payload[1] |= 0x40; // end

            memcpy(rtpPacket->payload + 2, frame + pos, RTP_MAX_PKT_SIZE);
            ret = rtpSendPacketOverTcp(clientSockfd, rtpPacket, RTP_MAX_PKT_SIZE + 2, 0x00);
            if (ret < 0)
                return -1;

            rtpPacket->rtpHeader.seq++;
            sendByte += ret;
            pos += RTP_MAX_PKT_SIZE;
        }

        // 发送剩余的数据
        if (remainPktSize > 0)
        {
            rtpPacket->payload[0] = (naluType & 0x60) | 28;
            rtpPacket->payload[1] = naluType & 0x1F;
            rtpPacket->payload[1] |= 0x40; //end

            memcpy(rtpPacket->payload + 2, frame + pos, remainPktSize + 2);
            ret = rtpSendPacketOverTcp(clientSockfd, rtpPacket, remainPktSize + 2, 0x00);
            if (ret < 0)
                return -1;

            rtpPacket->rtpHeader.seq++;
            sendByte += ret;
        }
    }


    return sendByte;
```

```
}
```

获取报文第一行数据

```
static char* GetLineFromBuf(char* rbuf, char* line)
{
    while (*rbuf != '\n')
    {
        *line = *rbuf;
        line++;
        rbuf++;
    }
    *line = '\n';
    ++line;
    *line = '\0';
    ++rbuf;
    return rbuf;
}
```

对于客户端消息的回复

```
static int handleCmd_OPTIONS(char* result, int cseq)
{
    sprintf(result, "RTSP/1.0 200 OK\r\n"
        "CSeq: %d\r\n"
        "Public: OPTIONS, DESCRIBE, SETUP, PLAY\r\n"
        "\r\n",
     c s e q);
    return 0;
}
static int handleCmd_DESCRIBE(char* result, int cseq, char* url)
{
    char sdp[500];
    char localIp[100];
    sscanf(url, "rtsp://%[^:]:", localIp);
    sprintf(sdp, "v=0\r\n"
        "o=-9%ld 1 IN IP4 %s\r\n"
        "t=0 0\r\n"
        "a=control:*\r\n"
        "m=video 0 RTP/AVP 96\r\n"
        "a=rtpmap:96 H264/90000\r\n"
        "a=control:track0\r\n"
        "m=audio 1 RTP/AVP/TCP 97\r\n"
        "a=rtpmap:97 mpeg4-generic/44100/2\r\n"
        "a=fmtp:97 profile-level-id=1;mode=AAC-hbr;sizelength=13;indexlength=3;indexdeltalength=3;config=1210;\
        "a=control:track1\r\n",
        time(NULL), localIp);
    sprintf(result, "RTSP/1.0 200 OK \r\n"
        "CSeq: %d\r\n"
        "Content-Base: %s\r\n"
        "Content-type: application/sdp\r\n"
        "Content-length: %d\r\n"
        "\r\n"
        "%s", cseq, url, (int)strlen(sdp), sdp);
    return 0;
}
static int handleCmd_SETUP(char* result, int cseq)
{
    if (cseq == 3) {
        sprintf(result, "RTSP/1.0 200 OK\r\n"
            "CSeq: %d\r\n"
            "Transport: RTP/AVP/TCP;unicast;interleaved=0-1\r\n"
            "Session: 66334873\r\n"
            "\r\n",
         c s e q);
    }
    else if (cseq == 4) {
        sprintf(result, "RTSP/1.0 200 OK\r\n"
            "CSeq: %d\r\n"
            "Transport: RTP/AVP/TCP;unicast;interleaved=2-3\r\n"
            "Session: 66334873\r\n"
            "\r\n",
         c s e q);
    }
    return 0;
}
static int handleCmd_PLAY(char* result, int cseq)
{
    sprintf(result, "RTSP/1.0 200 OK\r\n"
        "CSeq: %d\r\n"
        "Range: npt=0.000-\r\n"
        "Session: 66334873; timeout=60\r\n"
        "\r\n", cseq);
    return 0;
}
```

接收并回复消息做出相应的响应

```
static void doClient(int clientSockfd, const char* clientIP, int clientPort) {

    char method[40];
    char url[100];
    char version[40];
    int CSeq;

    char* rBuf = (char*)malloc(BUF_MAX_SIZE);
    char* sBuf = (char*)malloc(BUF_MAX_SIZE);

    while (true) {
        int recvLen;

   r e c v L e n  = recv(clientSockfd, rBuf, BUF_MAX_SIZE, 0);
        if (recvLen <= 0) {
            break;
        }

     r B u f[recvLen] = '\0';
        printf("Accept request rBuf = %s \n", rBuf);

        const char* sep = "\n";

        char* line = strtok(rBuf, sep);
        while (line) {
            if (strstr(line, "OPTIONS") ||
                strstr(line, "DESCRIBE") ||
                strstr(line, "SETUP") ||
                strstr(line, "PLAY")) {
                if (sscanf(line, "%s %s %s\r\n", method, url, version) != 3) {
                    // error
                }
            }
            else if (strstr(line, "CSeq")) {
```

```c
                if (sscanf(line, "CSeq: %d\r\n", &CSeq) != 1) {
                    // error
                }
            }
            else if (!strncmp(line, "Transport:", strlen("Transport:"))) {
                // Transport: RTP/AVP/UDP;unicast;client_port=13358-13359
                // Transport: RTP/AVP;unicast;client_port=13358-13359

                if (sscanf(line, "Transport: RTP/AVP/TCP;unicast;interleaved=0-1\r\n") != 0) {
                    // error
                    printf("parse Transport error \n");
                }
            }
            line = strtok(NULL, sep);
        }

        if (!strcmp(method, "OPTIONS")) {
            if (handleCmd_OPTIONS(sBuf, CSeq))
            {
                printf("failed to handle options\n");
                break;
            }
        }
        else if (!strcmp(method, "DESCRIBE")) {
            if (handleCmd_DESCRIBE(sBuf, CSeq, url))
            {
                printf("failed to handle describe\n");
                break;
            }
        }
        else if (!strcmp(method, "SETUP")) {
            if (handleCmd_SETUP(sBuf, CSeq))
            {
                printf("failed to handle setup\n");
                break;
            }
        }
        else if (!strcmp(method, "PLAY")) {
            if (handleCmd_PLAY(sBuf, CSeq))
            {
                printf("failed to handle play\n");
                break;
            }
        }
        else {
            printf("Undefined method = %s \n", method);
            break;
        }
        printf("Response sBuf = %s \n", sBuf);

        send(clientSockfd, sBuf, strlen(sBuf), 0);


        //开始播放，发送RTP包
        if (!strcmp(method, "PLAY")) {

            std::thread t1([&]() {

                int frameSize, startCode;
                char* frame = new char [500000];
                RtpPacket* rtpPacket = new RtpPacket[500000];
                int fp = open(H264_FILE_NAME, O_RDONLY);
                if (!fp) {
                    printf("Read %s fail\n", H264_FILE_NAME);
                    return;
                }
                rtpHeaderInit(rtpPacket, 0, 0, 0, RTP_VESION, RTP_PAYLOAD_TYPE_H264, 0,
                    0, 0, 0x88923423);

                printf("start play\n");

                while (true) {
                    frameSize = getFrameFromH264File(fp, frame, 500000);
                    if (frameSize < 0)
                    {
                        printf("Read %s end , frameSize=%d \n", H264_FILE_NAME, frameSize);
                        break;
                    }

                    if (startCode3(frame))
                        startCode = 3;
                    else
                        startCode = 4;

                    frameSize -= startCode;
                    rtpSendH264Frame(clientSockfd, rtpPacket, frame + startCode, frameSize);

                    rtpPacket->rtpHeader.timestamp += 90000 / 25;
                    usleep(40000);//1000/25 * 1000
                }
                free(frame);
                free(rtpPacket);

                });
            std::thread t2([&]() {
                struct AdtsHeader adtsHeader;
                struct RtpPacket* rtpPacket;
                uint8_t* frame;
                int ret;

                FILE* fp = fopen(AAC_FILE_NAME, "rb");
                if (!fp) {
                    printf("Read %s fail\n", AAC_FILE_NAME);
                    return;
                }

                frame = (uint8_t*)malloc(5000);
                rtpPacket = (struct RtpPacket*)malloc(5000);

                rtpHeaderInit(rtpPacket, 0, 0, 0, RTP_VESION, RTP_PAYLOAD_TYPE_AAC, 1, 0, 0, 0x32411);

                while (true)
                {
                    //读取ADTS头部
                    ret = fread(frame, 1, 7, fp);
                    if (ret <= 0)
                    {
                        printf("fread err\n");
                        break;
                    }
                    printf("fread ret=%d \n", ret);
                    //解析头部
                    if (parseAdtsHeader(frame, &adtsHeader) < 0)
```

```
                {
                    printf("parseAdtsHeader err\n");
                    break;
                }
                //读取一帧
                ret = fread(frame, 1, adtsHeader.aacFrameLength - 7, fp);
                if (ret <= 0)
                {
                    printf("fread err\n");
                    break;
                }
                //Rtp打包发送
                rtpSendAACFrame(clientSockfd,
                rtpPacket, frame, adtsHeader.aacFrameLength - 7);
                usleep(23223);//1000/43.06 * 1000
            }

            free(frame);
            free(rtpPacket);
            });

        t1.join();
        t2.join();

        break;
    }

    memset(method, 0, sizeof(method) / sizeof(char));
    memset(url, 0, sizeof(url) / sizeof(char));
    CSeq = 0;


    }

    close(clientSockfd);
    free(rBuf);
    free(sBuf);

}
```

源代码

rtp.h

```
#include<stdint.h>

#define RTP_VESION              2

#define RTP_PAYLOAD_TYPE_H264   96
#define RTP_PAYLOAD_TYPE_AAC    97

#define RTP_HEADER_SIZE         12
#define RTP_MAX_PKT_SIZE        1400
class RtpHeader
{
    public:
    /*byte 0*/
    uint8_t csrcLen : 4;
    uint8_t extension : 1;
    uint8_t padding : 1;
    uint8_t version : 2;
    /*byte 1*/
    uint8_t payloadType : 7;
    uint8_t marker : 1;
    /*bytes 2,3*/
    uint16_t seq;
    /*bytes 4-7*/
    uint32_t timestamp;
    /*bytes 8-11*/
    uint32_t ssrc;
};
class RtpPacket
{
    public:
    RtpHeader rtpHeader;
    uint8_t payload[0];

};
//初始化rtp包
void rtpHeaderInit(RtpPacket*rtpPacket,uint8_t csrclen,uint8_t extension,
                   uint8_t padding,uint8_t version,uint8_t payloadType,uint8_t marker,
                   uint16_t seq,uint32_t timestamp,uint32_t ssrc);
//以Tcp形式发送rtp数据包
int rtpSendPacketOverTcp(int clientSockfd, struct RtpPacket* rtpPacket, uint32_t dataSize, char channel);
//以Udp形式发送rtp数据包
int rtpSendPacketOverUdp(int serverRtpSockfd, const char* ip, int16_t port, struct RtpPacket* rtpPacket, uint32_

```

rtp.cpp

```
#include<sys/socket.h>
#include<arpa/inet.h>
#include<cstdlib>
#include<string.h>
#include"rtp.h"

void rtpHeaderInit(RtpPacket* rtpPacket, uint8_t csrclen, uint8_t extension, uint8_t padding, uint8_t version, 
{
 rtpPacket->rtpHeader.csrcLen = csrclen;
 rtpPacket->rtpHeader.extension = extension;
 rtpPacket->rtpHeader.padding = padding;
 rtpPacket->rtpHeader.version = version;
 rtpPacket->rtpHeader.payloadType = payloadType;
 rtpPacket->rtpHeader.marker = marker;
 rtpPacket->rtpHeader.seq = seq;
 rtpPacket->rtpHeader.timestamp = timestamp;
 rtpPacket->rtpHeader.ssrc = ssrc;
}

int rtpSendPacketOverTcp(int clientSockfd, struct RtpPacket* rtpPacket, uint32_t dataSize, char channel)
{

    rtpPacket->rtpHeader.seq = htons(rtpPacket->rtpHeader.seq);
    rtpPacket->rtpHeader.timestamp = htonl(rtpPacket->rtpHeader.timestamp);
    rtpPacket->rtpHeader.ssrc = htonl(rtpPacket->rtpHeader.ssrc);

    uint32_t rtpSize = RTP_HEADER_SIZE + dataSize;
    char* tempBuf = (char*)malloc(4 + rtpSize);
    tempBuf[0] = 0x24;//$
    tempBuf[1] = channel;// 0x00;//表示通道
    tempBuf[2] = (uint8_t)(((rtpSize) & 0xFF00) >> 8);
```

```
    tempBuf[3] = (uint8_t)((rtpSize) & 0xFF);
    memcpy(tempBuf + 4, (char*)rtpPacket, rtpSize);

    int ret = send(clientSockfd, tempBuf, 4 + rtpSize, 0);

    rtpPacket->rtpHeader.seq = ntohs(rtpPacket->rtpHeader.seq);
    rtpPacket->rtpHeader.timestamp = ntohl(rtpPacket->rtpHeader.timestamp);
    rtpPacket->rtpHeader.ssrc = ntohl(rtpPacket->rtpHeader.ssrc);

    free(tempBuf);
    tempBuf = NULL;

    return ret;
}
int rtpSendPacketOverUdp(int serverRtpSockfd, const char* ip, int16_t port, struct RtpPacket* rtpPacket, uint32_
{

    struct sockaddr_in addr;
    int ret;

    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    addr.sin_addr.s_addr = inet_addr(ip);

    rtpPacket->rtpHeader.seq = htons(rtpPacket->rtpHeader.seq);//从主机字节序转为网络字节序
    rtpPacket->rtpHeader.timestamp = htonl(rtpPacket->rtpHeader.timestamp);
    rtpPacket->rtpHeader.ssrc = htonl(rtpPacket->rtpHeader.ssrc);

    ret = sendto(serverRtpSockfd, (char*)rtpPacket, dataSize + RTP_HEADER_SIZE, 0,
        (struct sockaddr*)&addr, sizeof(addr));

    rtpPacket->rtpHeader.seq = ntohs(rtpPacket->rtpHeader.seq);
    rtpPacket->rtpHeader.timestamp = ntohl(rtpPacket->rtpHeader.timestamp);
    rtpPacket->rtpHeader.ssrc = ntohl(rtpPacket->rtpHeader.ssrc);

    return ret;

}

```

rtp_server.cpp

```
#include<iostream>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <assert.h>
#include<unistd.h>
#include<thread>
#include "rtp.h"
#define H264_FILE_NAME   "test.h264"
#define AAC_FILE_NAME   "test.aac"
#define SERVER_PORT      8554
#define SERVER_RTP_PORT  55532
#define SERVER_RTCP_PORT 55533
#define BUF_MAX_SIZE     (1024*1024)
#define SERVER_IP "127.0.0.1"
using namespace std;
//建立aacHeader
struct AdtsHeader
{
    unsigned int syncword;  //12 bit 同步字 '1111 1111 1111'，说明一个ADTS帧的开始
    unsigned int id;        //1 bit MPEG 标示符， 0 for MPEG-4，1 for MPEG-2
    unsigned int layer;     //2 bit 总是'00'
    unsigned int protectionAbsent;  //1 bit 1表示没有crc，0表示有crc
    unsigned int profile;           //1 bit 表示使用哪个级别的AAC
    unsigned int samplingFreqIndex; //4 bit 表示使用的采样频率
    unsigned int privateBit;        //1 bit
    unsigned int channelCfg; //3 bit 表示声道数
    unsigned int originalCopy;         //1 bit
    unsigned int home;                  //1 bit

    /*下面的为改变的参数即每一帧都不同*/
    unsigned int copyrightIdentificationBit;   //1 bit
    unsigned int copyrightIdentificationStart; //1 bit
    unsigned int aacFrameLength;               //13 bit 一个ADTS帧的长度包括ADTS头和AAC原始流
    unsigned int adtsBufferFullness;           //11 bit 0x7FF 说明是码率可变的码流

    /* number_of_raw_data_blocks_in_frame
    *  表示ADTS帧中有number_of_raw_data_blocks_in_frame + 1个AAC原始帧
    *  所以说number_of_raw_data_blocks_in_frame == 0
    *  表示说ADTS帧中有一个AAC数据块并不是说没有。(一个AAC原始帧包含一段时间内1024个采样及相关数据)
    * /
    unsigned int numberOfRawDataBlockInFrame; //2 bit
};
//解析aacHeader
static int parseAdtsHeader(uint8_t* in, struct AdtsHeader* res)
{
    static int frame_number = 0;
    memset(res, 0, sizeof(*res));

    if ((in[0] == 0xFF) && ((in[1] & 0xF0) == 0xF0))
    {
     r e s->id = ((unsigned int)in[1] & 0x08) >> 3;
     r e s->layer = ((unsigned int)in[1] & 0x06) >> 1;
     r e s->protectionAbsent = (unsigned int)in[1] & 0x01;
     r e s->profile = ((unsigned int)in[2] & 0xc0) >> 6;
     r e s->samplingFreqIndex = ((unsigned int)in[2] & 0x3c) >> 2;
     r e s->privateBit = ((unsigned int)in[2] & 0x02) >> 1;
     r e s->channelCfg = ((((unsigned int)in[2] & 0x01) << 2) | (((unsigned int)in[3] & 0xc0) >> 6));
     r e s->originalCopy = ((unsigned int)in[3] & 0x20) >> 5;
     r e s->home = ((unsigned int)in[3] & 0x10) >> 4;
     r e s->copyrightIdentificationBit = ((unsigned int)in[3] & 0x08) >> 3;
     r e s->copyrightIdentificationStart = (unsigned int)in[3] & 0x04 >> 2;
     r e s->aacFrameLength = (((((unsigned int)in[3]) & 0x03) << 11) |
            (((unsigned int)in[4] & 0xFF) << 3) |
            ((unsigned int)in[5] & 0xE0) >> 5);
     r e s->adtsBufferFullness = (((unsigned int)in[5] & 0x1f) << 6 |
            ((unsigned int)in[6] & 0xfc) >> 2);
     r e s->numberOfRawDataBlockInFrame = ((unsigned int)in[6] & 0x03);

        return 0;
    }
```

```c
    else
    {
        printf("failed to parse adts header\n");
        return -1;
    }
}
//用rtp格式打包并发送AAC音频流数据
static int rtpSendAACFrame(int clientSockfd,
    struct RtpPacket* rtpPacket, uint8_t* frame, uint32_t frameSize) {
    int ret;

 rtpPacket->payload[0] = 0x00;
 rtpPacket->payload[1] = 0x10;
 rtpPacket->payload[2] = (frameSize & 0x1FE0) >> 5; //高8位
 rtpPacket->payload[3] = (frameSize & 0x1F) << 3; //低5位

    memcpy(rtpPacket->payload + 4, frame, frameSize);


 ret = rtpSendPacketOverTcp(clientSockfd, rtpPacket, frameSize + 4, 0x02);

    if (ret < 0)
    {
        printf("failed to send rtp packet\n");
        return -1;
    }

 rtpPacket->rtpHeader.seq++;

    /*
    *  如果采样频率是44100
    *  一般AAC每个1024个采样为一帧
    *  所以一秒就有 44100 / 1024 = 43帧
    *  时间增量就是 44100 / 43 = 1025
    *  一帧的时间为 1 / 43 = 23ms
    * /
 rtpPacket->rtpHeader.timestamp += 1025;

    return 0;
}
//创建TCP套接字
static int CreateTcpSocket()
{
    int sockfd;
    int on=1;
 sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        return -1;
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (const char*)&on, sizeof(on));
    return sockfd;
}
//创建UDP套接字
static int CreateUdpSocket()
{
    int sockfd;
    int on = 1;
 sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0)
        return -1;
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (const char*)&on, sizeof(on));
    return sockfd;
}
//绑定端口和地址
static int BindSocketAddr(int sockfd,const char *ip,int port)
{
 sockaddr_in addr;
 addr.sin_family = AF_INET;
 addr.sin_port = htons(port);
 addr.sin_addr.s_addr = inet_addr(ip);
    if (bind(sockfd, (sockaddr*)&addr, sizeof(addr))<0)
        return -1;
    return 0;
}
//连接客户端并接收客户端信息
static int AcceptClient(int sockfd,char *ip,int *port)
{
    int clientfd;
 socklen_t len = 0;
 sockaddr_in addr;
    memset(&addr, 0, sizeof(addr));
 len = sizeof(addr);
 clientfd = accept(sockfd, (sockaddr*)&addr, &len);
    if (clientfd < 0)
        return -1;
    strcpy(ip, inet_ntoa(addr.sin_addr));
    *port = ntohs(addr.sin_port);
    return clientfd;
}
//判断是不是非h264码流
static inline int startCode3(char* buf)
{
    if (buf[0] == 0 && buf[1] == 0 && buf[2] == 1)
        return -1;
    else
        return 0;
}
//判断是不是非h264码流
static inline int startCode4(char* buf)
{
    if (buf[0] == 0 && buf[1] == 0 && buf[2] == 0&&buf[3]==1)
```



```c
        return 0;
}
//找下一段h264数据
static char* findNextStartCode(char* buf, int len)
{
    int i;
    if (len < 3)
        return NULL;
    for (i = 0; i < len - 3; i++)
    {
        if (startCode3(buf) || startCode4(buf))
            return buf;
        ++buf;
    }
    if (startCode3(buf))
        return buf;
    return NULL;
}
//获得h264码流大小
static int getFrameFromH264File(int fd,char*frame,int size)
{
```

```c
    int rSize, frameSize;
    char* nextStartCode;
    if (fd < 0)
        return fd;
    rSize = read(fd, frame, size);
    if (!startCode3(frame) && !startCode4(frame))
        return -1;
    nextStartCode = findNextStartCode(frame + 3, rSize - 3);
    if (!nextStartCode)
        return -1;
    else
    {
        frameSize = nextStartCode - frame;
        lseek(fd, frameSize - rSize, SEEK_CUR);
    }
    return frameSize;
}
//用rtp格式打包并发送H264视频流数据
static int rtpSendH264Frame(int clientSockfd,
    struct RtpPacket* rtpPacket, char* frame, uint32_t frameSize)
{

    uint8_t naluType; // nalu第一个字节
    int sendByte = 0;
    int ret;

    naluType = frame[0];

    printf("%s frameSize=%d \n", __FUNCTION__, frameSize);

    if (frameSize <= RTP_MAX_PKT_SIZE) // nalu长度小于最大包场：单一NALU单元模式
    {

        //*   0 1 2 3 4 5 6 7 8 9
        //*  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        //*  |F|NRI|  Type   | a single NAL unit ... |
        //*  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

        memcpy(rtpPacket->payload, frame, frameSize);
        ret = rtpSendPacketOverTcp(clientSockfd, rtpPacket, frameSize, 0x00);
        if (ret < 0)
            return -1;

        rtpPacket->rtpHeader.seq++;
        sendByte += ret;
        if ((naluType & 0x1F) == 7 || (naluType & 0x1F) == 8) // 如果是SPS、PPS就不需要加时间戳
        {

        }

    }
    else // nalu长度小于最大包：分片模式
    {

        //*  0                   1                   2
        //*  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
        //* +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        //* | FU indicator  |   FU header   |   FU payload   ...  |
        //* +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+



        //*     FU Indicator
        //*    0 1 2 3 4 5 6 7
        //*   +-+-+-+-+-+-+-+-+
        //*   |F|NRI|  Type   |
        //*   +---------------+



        //*      FU Header
        //*    0 1 2 3 4 5 6 7
        //*   +-+-+-+-+-+-+-+-+
        //*   |S|E|R|  Type   |
        //*   +---------------+


        int pktNum = frameSize / RTP_MAX_PKT_SIZE;       // 有几个完整的包
        int remainPktSize = frameSize % RTP_MAX_PKT_SIZE; // 剩余不完整包的大小
        int i, pos = 1;

        // 发送完整的包
        for (i = 0; i < pktNum; i++)
        {
            rtpPacket->payload[0] = (naluType & 0x60) | 28;
            rtpPacket->payload[1] = naluType & 0x1F;

            if (i == 0) //第一包数据
                rtpPacket->payload[1] |= 0x80; // start
            else if (remainPktSize == 0 && i == pktNum - 1) //最后一包数据
                rtpPacket->payload[1] |= 0x40; // end

            memcpy(rtpPacket->payload + 2, frame + pos, RTP_MAX_PKT_SIZE);
            ret = rtpSendPacketOverTcp(clientSockfd, rtpPacket, RTP_MAX_PKT_SIZE + 2, 0x00);
            if (ret < 0)
                return -1;

            rtpPacket->rtpHeader.seq++;
            sendByte += ret;
            pos += RTP_MAX_PKT_SIZE;
        }

        // 发送剩余的数据
        if (remainPktSize > 0)
        {
            rtpPacket->payload[0] = (naluType & 0x60) | 28;
            rtpPacket->payload[1] = naluType & 0x1F;
            rtpPacket->payload[1] |= 0x40; //end

            memcpy(rtpPacket->payload + 2, frame + pos, remainPktSize + 2);
            ret = rtpSendPacketOverTcp(clientSockfd, rtpPacket, remainPktSize + 2, 0x00);
            if (ret < 0)
                return -1;

            rtpPacket->rtpHeader.seq++;
            sendByte += ret;
        }
    }


    return sendByte;
}
```

分类专栏

- c++ 1篇
- stocket 1篇
- c 3篇

```c
//获取报文第一行数据
static char* GetLineFromBuf(char* rbuf, char* line)
{
    while (*rbuf != '\n')
    {
        *line = *rbuf;
        line++;
        rbuf++;
    }
    *line = '\n';
    ++line;
    *line = '\0';
    ++rbuf;
    return rbuf;
}
static int handleCmd_OPTIONS(char* result, int cseq)
{
    sprintf(result, "RTSP/1.0 200 OK\r\n"
        "CSeq: %d\r\n"
        "Public: OPTIONS, DESCRIBE, SETUP, PLAY\r\n"
        "\r\n",
        cseq);
    return 0;
}
static int handleCmd_DESCRIBE(char* result, int cseq, char* url)
{
    char sdp[500];
    char localIp[100];
    sscanf(url, "rtsp://%[^:]:", localIp);
    sprintf(sdp, "v=0\r\n"
        "o=-9%ld 1 IN IP4 %s\r\n"
        "t=0 0\r\n"
        "a=control:*\r\n"
        "m=video 0 RTP/AVP 96\r\n"
        "a=rtpmap:96 H264/90000\r\n"
        "a=control:track0\r\n"
        "m=audio 1 RTP/AVP/TCP 97\r\n"
        "a=rtpmap:97 mpeg4-generic/44100/2\r\n"
        "a=fmtp:97 profile-level-id=1;mode=AAC-hbr;sizelength=13;indexlength=3;indexdeltalength=3;config=1210;\
        "a=control:track1\r\n",
        time(NULL), localIp);
    sprintf(result, "RTSP/1.0 200 OK \r\n"
        "CSeq: %d\r\n"
        "Content-Base: %s\r\n"
        "Content-type: application/sdp\r\n"
        "Content-length: %d\r\n"
        "\r\n"
        "%s", cseq, url, (int)strlen(sdp), sdp);
    return 0;
}
static int handleCmd_SETUP(char* result, int cseq)
{
    if (cseq == 3) {
        sprintf(result, "RTSP/1.0 200 OK\r\n"
            "CSeq: %d\r\n"
            "Transport: RTP/AVP/TCP;unicast;interleaved=0-1\r\n"
            "Session: 66334873\r\n"
            "\r\n",
            cseq);
    }
    else if (cseq == 4) {
        sprintf(result, "RTSP/1.0 200 OK\r\n"
            "CSeq: %d\r\n"
            "Transport: RTP/AVP/TCP;unicast;interleaved=2-3\r\n"
            "Session: 66334873\r\n"
            "\r\n",
            cseq);
    }
    return 0;
}
static int handleCmd_PLAY(char* result, int cseq)
{
    sprintf(result, "RTSP/1.0 200 OK\r\n"
        "CSeq: %d\r\n"
        "Range: npt=0.000-\r\n"
        "Session: 66334873; timeout=60\r\n"
        "\r\n", cseq);
    return 0;
}
//接收并回复消息做出相应的响应
static void doClient(int clientSockfd, const char* clientIP, int clientPort) {

    char method[40];
    char url[100];
    char version[40];
    int CSeq;

    char* rBuf = (char*)malloc(BUF_MAX_SIZE);
    char* sBuf = (char*)malloc(BUF_MAX_SIZE);

    while (true) {
        int recvLen;

        recvLen = recv(clientSockfd, rBuf, BUF_MAX_SIZE, 0);
        if (recvLen <= 0) {
            break;
        }

        rBuf[recvLen] = '\0';
        printf("Accept request rBuf = %s \n", rBuf);

        const char* sep = "\n";

        char* line = strtok(rBuf, sep);
        while (line) {
            if (strstr(line, "OPTIONS") ||
                strstr(line, "DESCRIBE") ||
                strstr(line, "SETUP") ||
                strstr(line, "PLAY")) {
                if (sscanf(line, "%s %s %s\r\n", method, url, version) != 3) {
                    // error
                }
            }
            else if (strstr(line, "CSeq")) {
                if (sscanf(line, "CSeq: %d\r\n", &CSeq) != 1) {
                    // error
                }
            }
            else if (!strncmp(line, "Transport:", strlen("Transport:"))) {
                // Transport: RTP/AVP/UDP;unicast;client_port=13358-13359
                // Transport: RTP/AVP;unicast;client_port=13358-13359

                if (sscanf(line, "Transport: RTP/AVP/TCP;unicast;interleaved=0-1\r\n") != 0) {
```

```
                // error
                printf("parse Transport error \n");
            }
        }
    line = strtok(NULL, sep);
    }

    if (!strcmp(method, "OPTIONS")) {
        if (handleCmd_OPTIONS(sBuf, CSeq))
        {
            printf("failed to handle options\n");
            break;
        }
    }
    else if (!strcmp(method, "DESCRIBE")) {
        if (handleCmd_DESCRIBE(sBuf, CSeq, url))
        {
            printf("failed to handle describe\n");
            break;
        }
    }
    else if (!strcmp(method, "SETUP")) {
        if (handleCmd_SETUP(sBuf, CSeq))
        {
            printf("failed to handle setup\n");
            break;
        }
    }
    else if (!strcmp(method, "PLAY")) {
        if (handleCmd_PLAY(sBuf, CSeq))
        {
            printf("failed to handle play\n");
            break;
        }
    }
    else {
        printf("Undefined method = %s \n", method);
        break;
    }
    printf("Response sBuf = %s \n", sBuf);

    send(clientSockfd, sBuf, strlen(sBuf), 0);


    //开始播放，发送RTP包
    if (!strcmp(method, "PLAY")) {

    std::thread t1([&]() {

            int frameSize, startCode;
            char* frame = new char [500000];
            RtpPacket* rtpPacket = new RtpPacket[500000];
            int fp = open(H264_FILE_NAME, O_RDONLY);
            if (!fp) {
                printf("Read %s fail\n", H264_FILE_NAME);
                return;
            }
            rtpHeaderInit(rtpPacket, 0, 0, 0, RTP_VESION, RTP_PAYLOAD_TYPE_H264, 0,
                0, 0, 0x88923423);

            printf("start play\n");

            while (true) {
                frameSize = getFrameFromH264File(fp, frame, 500000);
                if (frameSize < 0)
                {
                    printf("Read %s end , frameSize=%d \n", H264_FILE_NAME, frameSize);
                    break;
                }

                if (startCode3(frame))
                    startCode = 3;
                else
                    startCode = 4;

                frameSize -= startCode;
                rtpSendH264Frame(clientSockfd, rtpPacket, frame + startCode, frameSize);

                rtpPacket->rtpHeader.timestamp += 90000 / 25;
                usleep(40000);//1000/25 * 1000
            }
            free(frame);
            free(rtpPacket);

            });
    std::thread t2([&]() {
            struct AdtsHeader adtsHeader;
            struct RtpPacket* rtpPacket;
            uint8_t* frame;
            int ret;

            FILE* fp = fopen(AAC_FILE_NAME, "rb");
            if (!fp) {
                printf("Read %s fail\n", AAC_FILE_NAME);
                return;
            }

            frame = (uint8_t*)malloc(5000);
            rtpPacket = (struct RtpPacket*)malloc(5000);

            rtpHeaderInit(rtpPacket, 0, 0, 0, RTP_VESION, RTP_PAYLOAD_TYPE_AAC, 1, 0, 0, 0x32411);

            while (true)
            {
                //读取ADTS头部
                ret = fread(frame, 1, 7, fp);
                if (ret <= 0)
                {
                    printf("fread err\n");
                    break;
                }
                printf("fread ret=%d \n", ret);
                //解析头部
                if (parseAdtsHeader(frame, &adtsHeader) < 0)
                {
                    printf("parseAdtsHeader err\n");
                    break;
                }
                //读取一帧
                ret = fread(frame, 1, adtsHeader.aacFrameLength - 7, fp);
                if (ret <= 0)
                {
                    printf("fread err\n");
```

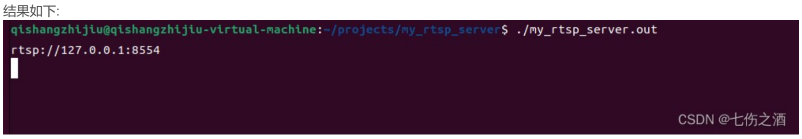
```cpp
                    break;
                }
                //Rtp打包发送
                rtpSendAACFrame(clientSockfd,
            r t p P a c k e t, frame, adtsHeader.aacFrameLength - 7);
                usleep(23223);//1000/43.06 * 1000
            }

            free(frame);
            free(rtpPacket);
            });

        t  1.join();
        t  2.join();

            break;
        }

        memset(method, 0, sizeof(method) / sizeof(char));
        memset(url, 0, sizeof(url) / sizeof(char));
    C S e q  = 0;


    }

    close(clientSockfd);
    free(rBuf);
    free(sBuf);

}
int main(int argc, char* argv[])
{
    //建立套接字
    int ServerSocket;
    int ServerRtcpSocket, ServerRtpSocket;
    //创建TCP套接字
 ServerSocket = CreateTcpSocket();
    if (ServerSocket < 0)
    {
    c o u t  << "TCP create fail !!!" << endl;
        exit(0);
    }
    //绑定端口和地址
    if (BindSocketAddr(ServerSocket,"0.0.0.0",SERVER_PORT)<0)
    {
    c o u t  << "bind fail !!!" << endl;
        exit(0);
    }
    //监听端口
    if (listen(ServerSocket, 5)<0)
    {
    c o u t  << "listen !!!" << endl;
        exit(0);
    }
  c o u t  << "rtsp://"<< SERVER_IP <<":" << SERVER_PORT << endl;
    while (true)
    {
        int ClientSocket;
        char  ClientIp[40];
        int ClientPort;
        //接收客户端消息
   C l i e n t S o c k e t  = AcceptClient(ServerSocket, ClientIp, &ClientPort);
        if (ClientSocket < 0)
        {
            printf("failed to accept client\n");
            return -1;
        }
        //打印客户端信息
    c o u t  << "accept client;client ip:" << ClientIp << ",client port:" << ClientPort << endl;
        //接收消息并做出响应
        doClient(ClientSocket, ClientIp, ClientPort);
    }
    close(ServerSocket);
    return 0;
}
```

具体实现

首先获取一个音视频文件(以test.mp4文件为例):
用ffmpeg将其拆解为h264和aac形式文件

```
ffmpeg -i test.mp4 -acodec copy -vn test.acc
ffmpeg -i test.mp4 -vcodec copy -an test.h264
```

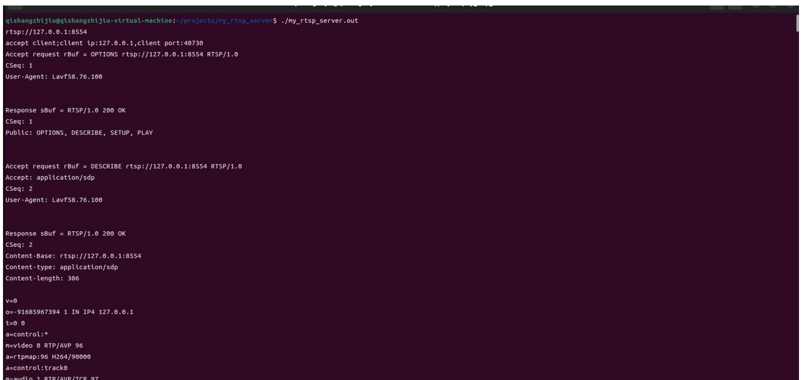
将其存进程序的项目文件夹后运行程序
结果如下:

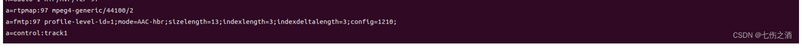


因为是tcp形式,所以需要指定为tcp形式播放
播放命令为:

```
ffplay -i -rtsp_transport tcp rtsp://127.0.0.1:8554
```

结果如下:





来自作者

七伤之酒 关注 0 0 1

一个简易的RTSP服务器 12-30
一个简易的RTSP服务器，可用于直播H264/G711A流。仅供学习之用

RTSP配置服务器说明文档 10-23
网上一些用vlc搭建TCP和UDP方式的流媒体服务器的教程都是一样的，配置了很久也没有成功，这个配置方法不一样，很简单，100%成功

1 条评论 CSDN-Ada助手 热评 恭喜您写了第7篇博客，标题为“rtsp简单服务器”。这篇文章很有深度，让我受益匪浅。希望您... 写评论

rtsp 服务器搭建 yinshipin007的博客 1万+
rtsp 服务器搭建： 今天我们搭建这个 rtsp 服务器的名称叫做：ZLMediaKit，它是一个基于 C++11 的高性能运营级流媒体服务框架，类似我之前给大家搭...

搭建RTSP流媒体服务器（用于测试分析RTSP协议）（rtsp-simple-server+FFmpeg+vlc+wireshark） qq_36819827的博客 1684
本文所搭建的流媒体服务器是在虚拟机centos7中安装rtsp-simple-server充当服务器，在windows本机中安装FFmpeg用于推流，使用vlc播放器播放rtsp-si...

RTSP服务器源码 纯C linux/windows 又更新了 09-01
学习RTSP时写的一个简易的RTSP服务器源码 可用于直播H264/G711A流 纯手工C实现 已在windows及linux上调试成功 现在更新了 支持多通道 rtp发送也...

搭建RTSP流媒体服务器的三种方式 fang.lovest.yang的博客 8425
主要用于测试目的，系统是windows，使用的是docker desktop 3.5.1 1. rtsp-simple-server 官网：https://github.com/aler9/rtsp-simple-server （1）下载...

RTSP服务器综合项目：rtsp实时监控 weixin_43796767的博客 3714
点击 编写RTSP服务器：项目索引，回到目录。 目标：添加摄像头采集编码和码流队列管理，完成实时监控功能。 protocol.h添加采集相关设置 // RTSP...

linux下实战之rtsp监控服务器 07-21
本次教程通过抓包并分析，然后在linux下一行一行的写出一个基于rtsp协议的监控服务器，对接VLC和摄像头，服务器去摄像头拉流，再把流发给VLC进行...

Rtsp服务器搭建(RTSP协议实现) weixin_66034721的博客 173
Rtsp服务器搭建(RTSP协议实现)

Rtsp服务搭建（Ffmpeg+Node.js+jsmpeg网络视频服务器） wtnu200的专栏 5855
当前有几个海康监控，想接入MES系统，去他们官方网站下载了两个web的sdk包，分别为控件开发包和无插件开发包，结果很坑，控件，要求为ie浏览器...

RTSP测试服务器 09-11
live555里面编译出来的Rtsp服务器，里面包含有视频文件和流地址，免配置可直接用于测试rtsp客户端程序。

RTSP服务器组件，， 01-02
video & audio source,very easy to use. 精炼、高效、稳定的RTSP服务器组件,调用简单灵活,轻松嵌入部署到IPCamera(ARM型、Android型)中，也可用...

用Darwin开发RTSP级联服务器 08-21
1、通过Darwin 开发的 RTSP 服务器 2、测试已经通过 3、使用很简单，简单配置一下 Device.xml 和 streamingserver.xml 即可

一个可立即运行的模拟RTSP流的RTSP流服务器 01-22
操作简单，运行RTSPServer.exe即可模拟RTSP流

rtsp-test-server：提供一些测试rtsp流的简单服务器 03-03
基于GStreamer的简单RTSP服务器。 在线演示网址 rtsp：//ipcam.stream：8554 / bars rtsp：//ipcam.stream：8554 / white rtsp：//ipcam.stream：8554...

RTSP服务器（二） 862
目录(?)[-] 一单播多播组播和广播直播点播点播试直播 1单播2多播组播 二RTSP RTMPHTTP的区别 共同点区别 三RTSPRTCPRTP区别四RTSP流程 1RT...

windows环境下，搭建RTSP视频推流服务器 热门推荐 朱豪凯的博客 1万+
windows环境下，搭建RTSP视频推流服务器

RTSP协议详解 qq_25330791的博客 9226
Rtsp

rtsp流媒体服务器搭建 最新发布 07-21
回答1： RTSP（Real Time Streaming Protocol）流媒体服务器搭建需要遵循以下步骤： 1. 选择合适的服务器软件：有一些流行的RTSP流媒体服务...

“相关推荐”对你有帮助么？

非常没帮助 没帮助 一般 有帮助 非常有帮助

七伤之酒
码龄2年 暂无认证

7	124万+	13万+	3419	
原创	周排名	总排名	访问	等级

79	3	6	5	6
积分	粉丝	获赞	评论	收藏

私信 关注

搜博主文章

热门文章

- C语言斐波那契数列解析 1370
- C语言汉诺塔问题解析 1038
- 用函数输出100~200之间的素数 407
- 链表c++ 240
- Rtsp服务器搭建(RTSP协议实现) 171

最新评论

Rtsp服务器搭建(RTSP协议实现)
CSDN-Ada助手: 恭喜您成功搭建了RTSP服务器，这篇博客非常有价值，对我们这些...

rtsp简单服务器
CSDN-Ada助手: 恭喜您写了第7篇博客，标题为“rtsp简单服务器”。这篇文章很有深...

Rtsp服务器搭建(RTSP协议实现)
魚小飛: 支持老表

Rtsp服务器搭建(RTSP协议实现)
rming364: 好文章，期待下个作品

c++网络
CSDN-Ada助手: 恭喜您写了第5篇博客，标题为“c++网络”。看到您的不断创作，我...

您愿意向朋友推荐“博客详情页”吗？

强烈不推荐 不推荐 一般般 推荐 强烈推荐

最新文章

- Rtsp服务器搭建(RTSP协议实现)
- c++网络
- 链表c++

2023年 3篇 2022年 4篇

Beta Beta 举报