JT同学
码龄4年 暂无认证
105 原创 2万+ 周排名 91万+ 总排名 46万+ 访问 等级
5440 积分 1861 粉丝 1016 获赞 346 评论 3945 收藏
私信 关注

搜博主文章

热门文章
从零开始写一个RTSP服务器（一）RTSP协议讲解 57162
C++ deque的用法与示例 38004
Linux I2C驱动框架（超详细） 35491
我的开源项目-RtspServer 25554
从零开始写一个RTSP服务器（二）RTSP协议的实现 21860

最新评论
我的开源项目-RtspServer
kaifa3k: 我测试 ./h264_rtsp_server test.h264 也会这样的问题？请问解决了吗？m...
使用mp4v2封装H.264成mp4最简单示例
热心兔斯基: 应该是播放器的原因。换个播放器试试
使用mp4v2封装H.264成mp4最简单示例
ai604233436: h264的标准中，好像默认的设置的90KHZ。我所以我理解timeScale...
使用mp4v2封装H.264成mp4最简单示例
ai604233436: 有遇到，不知道原因呢。
深入学习Linux摄像头（二）v4l2驱动框架
qq_43745897: 请问往缓存区里面采集图像数据的帧率是多少呢？请问如果缓存区里...

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
深入浅出MySQL事务（二）MVCC的实现原理
深入浅出MySQL事务（一）事务隔离
深入浅出MySQL索引（二）InnoDB存储引擎的索引
2020年 4篇 2019年 95篇
2018年 7篇

从零开始写一个RTSP服务器（三）RTP传输H.264

原创 JT同学 于 2019-08-10 17:09:20 发布 19311 收藏 75 版权
分类专栏： 从零开始写一个RTSP服务器 文章标签： rtsp 流媒体 音视频 H.264 rtp
从零开始写一个RTS... 专栏收录该内容 187 订阅 10 篇文章 订阅专栏

从零开始写一个RTSP服务器系列

★我的开源项目-RtspServer

从零开始写一个RTSP服务器（一）RTSP协议讲解

从零开始写一个RTSP服务器（二）RTSP协议的实现

从零开始写一个RTSP服务器（三）RTP传输H.264

从零开始写一个RTSP服务器（四）一个传输H.264的RTSP服务器

从零开始写一个RTSP服务器（五）RTP传输AAC

从零开始写一个RTSP服务器（六）一个传输AAC的RTSP服务器

从零开始写一个RTSP服务器（七）多播传输RTP包

从零开始写一个RTSP服务器（八）一个多播的RTSP服务器

从零开始写一个RTSP服务器（九）一个RTP OVER RTSP/TCP的RTSP服务器

从零开始写一个RTSP服务器（三）RTP传输H.264

文章目录

本篇文章目标，使用vlc打开sdp文件后，可以观看到视频数据

一、RTP封装

1.1 RTP数据结构

RTP包格式前面已经比较详细的介绍过，参考从零开始写一个RTSP服务器（一）不一样的RTSP协议讲解

看一张RTP头的格式图回忆一下

V	P	X	CC	M	PT	序列号
时戳						
同步信源(SSRC)标识符						
特约信源(CSRC)标识符						
...						

RTP报头格式

每个RTP包都包含这样一个RTP头部和RTP数据，为了方便，我将这个头部封装成一个结构体，还有发送包封装成一个函数，下面来看一看

- **RTP头结构体**

```
/*
* 作者: _JT_
* 博客: https://blog.csdn.net/weixin_42462202
*/

struct RtpHeader
{
    /* 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;
};
```

其中的 `:n` 是一种位表示法，这个结构体跟RTP的头部一一对应

RTP的发包函数

RTP包

```
struct RtpPacket
{
    struct RtpHeader rtpHeader;
    uint8_t payload[0];
};
```

这是我封装的一个RTP包，包含一个RTP头部和RTP载荷，`uint8_t payload[0]` 并不占用空间，它表示rtp头部接下来紧跟着的地址

RTP的发包函数

```
/*
 * 函数功能: 发送RTP包
 * 参数 socket: 表示本机的udp套接字
 * 参数 ip: 表示目的ip地址
 * 参数 port: 表示目的的端口号
 * 参数 rtpPacket: 表示rtp包
 * 参数 dataSize: 表示rtp包中载荷的大小
 * 返回值: 发送字节数
 */
int rtpSendPacket(int socket, char* ip, int16_t port, struct RtpPacket* rtpPacket, uint32_t dataSize)
{
    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(socket, (void*)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;
}
```

仔细看这个函数你应该可以看懂

目录
从零开始写一个RTSP服务器（三）RTP...
文章目录
一、RTP封装
1.1 RTP数据结构
1.2 源码
二、H.264的RTP打包
2.1 H.264格式
2.2 H.264的RTP打包方式
2.3 H.264 RTP包的时间戳计算
2.4 源码
三、H.264 RTP打包的sdp描述

分类专栏
C/C++ 1篇
STL源码剖析 18篇
MySQL 4篇
Linux内核 14篇
分布式
nginx
从零开始写一个RTSP服... 10篇
live555源码分析与应用 9篇
Linux驱动 19篇

我们设置好一个包之后，就会调用这个函数发送指定目标

这个函数中多处使用 `htons` 等函数，是因为RTP是采用网络字节序（大端模式），所以要将主机字节字节序转换为网络字节序

下面给出源码，`rtp.h` 和 `rtp.c`，这两个文件在后面讲经常使用

1.2 源码

rtp.h

```c
/*
 * 作者: _JT_
 * 博客: https://blog.csdn.net/weixin_42462202
 */

#ifndef _RTP_H_
#define _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

/*
 *
 *    0                   1                   2                   3
 *    7 6 5 4 3 2 1 0|7 6 5 4 3 2 1 0|7 6 5 4 3 2 1 0|7 6 5 4 3 2 1 0
 *   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 *   |V=2|P|X|  CC   |M|     PT      |       sequence number         |
 *   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 *   |                           timestamp                           |
 *   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 *   |           synchronization source (SSRC) identifier            |
 *   +=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
 *   |            contributing source (CSRC) identifiers             |
 *   :                             ....                              :
 *   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 *
 */
struct RtpHeader
{
    /* 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;
};

struct RtpPacket
{
    struct RtpHeader rtpHeader;
    uint8_t payload[0];
};

void rtpHeaderInit(struct 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);
int rtpSendPacket(int socket, char* ip, int16_t port, struct RtpPacket* rtpPacket, uint32_t dataSize);

#endif //_RTP_H_
```

rtp.c

```c
/*
 * 作者: _JT_
 * 博客: https://blog.csdn.net/weixin_42462202
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "rtp.h"

void rtpHeaderInit(struct 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)
{
    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 rtpSendPacket(int socket, char* ip, int16_t port, struct RtpPacket* rtpPacket, uint32_t dataSize)
{
    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(socket, (void*)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;
}
```

二、H.264的RTP打包

2.1 H.264格式

H.264由一个一个的NALU组成，每个NALU之间使用 `00 00 00 01` 或 `00 00 01` 分隔开

每个NALU的第一次字节都有特殊的含义，其内容如下

位	描述
bit[7]	必须为0
bit[5-6]	标记该NALU的重要性
bit[0-4]	NALU单元的类型

好，对于H.264格式了解这么多就够了，我们的目的是想从一个H.264的文件中将一个一个的NALU提取出来，然后封装成RTP包，下面介绍如何将NALU封装成RTP包

2.2 H.264的RTP打包方式

H.264可以由三种RTP打包方式

单NALU打包

一个RTP包包含一个完整的NALU

聚合打包

对于较小的NALU，一个RTP包可包含多个完整的NALU

分片打包

对于较大的NALU，一个NALU可以分为多个RTP包发送

注意：这里要区分好概念，每一个RTP包都包含一个RTP头部和RTP荷载，这是固定的。而H.264发送数据可支持三种RTP打包方式

比较常用的是 单NALU打包 和 分片打包，本文也只介绍这两种

单NALU打包

所谓单NALU打包就是将一整个NALU的数据放入RTP包的载荷中

这是最简单的一种方式，无需过多的讲解

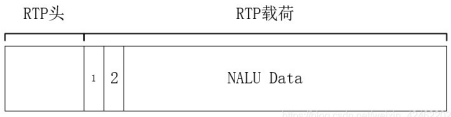
分片打包

每个RTP包都有大小限制的，因为RTP一般都是使用UDP发送，UDP没有流量控制，所以要限制每一次发送的大小，所以如果一个NALU的太大，就需要分成多个RTP包发送，如何分成多个RTP包，下面来好好讲一讲

首先要明确，RTP包的格式是绝不会变的，永远多是RTP头+RTP载荷

RTP头部是固定的，那么只能在RTP载荷中去添加额外信息来说明这个RTP包是表示同一个NALU

如果是分片打包的话，那么在RTP载荷开始有两个字节的信息，然后再是NALU的内容



第一个字节位 `FU Indicator`，其格式如下

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

高三位：与NALU第一个字节的高三位相同

Type：28，表示该RTP包一个分片，为什么是28？因为H.264的规范中定义的，此外还有许多其他Type，这里不详讲

第二个字节位 `FU Header`，其格式如下

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

S：标记该分片打包的第一个RTP包

E：比较该分片打包的最后一个RTP包

Type：NALU的Type

2.3 H.264 RTP包的时间戳计算

RTP包的时间戳起始值是随机的

RTP包的时间戳增量怎么计算？

假设时钟频率为90000，帧率为25

频率为90000表示一秒用90000点来表示

帧率为25，那么一帧就是1/25秒

所以一帧有90000*(1/25)=3600个点来表示

因此每一帧数据的时间增量为3600

2.4 源码

rtp_h264.c

这里给出rtp发送H.264的源码

```c
/*
 * 作者: _JT_
 * 博客: https://blog.csdn.net/weixin_42462202
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#include "rtp.h"

#define H264_FILE_NAME  "test.h264"
#define CLIENT_IP       "127.0.0.1"
#define CLIENT_PORT     9832

#define FPS             25

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

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

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;
}

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)
    {
        lseek(fd, 0, SEEK_SET);
```

```
        frameSize = rSize;
    }
    else
    {
        frameSize = (nextStartCode-frame);
        lseek(fd, frameSize-rSize, SEEK_CUR);
    }

    return frameSize;
}

static int createUdpSocket()
{
    int fd;
    int on = 1;

    fd = socket(AF_INET, SOCK_DGRAM, 0);
    if(fd < 0)
        return -1;

    setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, (const char*)&on, sizeof(on));

    return fd;
}

static int rtpSendH264Frame(int socket, char* ip, int16_t port,
                            struct RtpPacket* rtpPacket, uint8_t* frame, uint32_t frameSize)
{
    uint8_t naluType; // nalu第一个字节
    int sendBytes = 0;
    int ret;

    naluType = frame[0];

    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 = rtpSendPacket(socket, ip, port, rtpPacket, frameSize);
        if(ret < 0)
            return -1;

        rtpPacket->rtpHeader.seq++;
        sendBytes += ret;
        if ((naluType & 0x1F) == 7 || (naluType & 0x1F) == 8) // 如果是SPS、PPS就不需要加时间戳
            goto out;
    }
    else // nalu长度小于最大包场：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 = rtpSendPacket(socket, ip, port, rtpPacket, RTP_MAX_PKT_SIZE+2);
            if(ret < 0)
                return -1;

            rtpPacket->rtpHeader.seq++;
            sendBytes += 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 = rtpSendPacket(socket, ip, port, rtpPacket, remainPktSize+2);
            if(ret < 0)
                return -1;

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

out:

    return sendBytes;
}

int main(int argc, char* argv[])
{
    int socket;
    int fd;
    int fps = 25;
    int startCode;
    struct RtpPacket* rtpPacket;
    uint8_t* frame;
    uint32_t frameSize;

    fd = open(H264_FILE_NAME, O_RDONLY);
    if(fd < 0)
    {
        printf("failed to open %s\n", H264_FILE_NAME);
        return -1;
    }

    socket = createUdpSocket();
    if(socket < 0)
    {
        printf("failed to create socket\n");
        return -1;
    }

    rtpPacket = (struct RtpPacket*)malloc(500000);
    frame = (uint8_t*)malloc(500000);

    rtpHeaderInit(rtpPacket, 0, 0, 0, RTP_VESION, RTP_PAYLOAD_TYPE_H264, 0,
                  0, 0, 0x88923423);

    while(1)
    {
        frameSize = getFrameFromH264File(fd, frame, 500000);
        if(frameSize < 0)
        {
            printf("read err\n");
            continue;
        }

```

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

        frameSize -= startCode;
        rtpSendH264Frame(socket, CLIENT_IP, CLIENT_PORT,
                         rtpPacket, frame+startCode, frameSize);
        rtpPacket->rtpHeader.timestamp += 90000/FPS;

        usleep(1000*1000/fps);
    }

    free(rtpPacket);
    free(frame);

    return 0;
}
```

三、H.264 RTP打包的sdp描述

sdp文件有什么用?

sdp描述着媒体信息，当使用vlc打开这个sdp文件后，会根据这些信息做相应的操作（创建套接字...），然后等待接收RTP包

这里给出`RTP打包H.264`的sdp文件，并描述每一行是什么意思

```
m=video 9832 RTP/AVP 96
a=rtpmap:96 H264/90000
a=framerate:25
c=IN IP4 127.0.0.1
```

这个一个媒体级的sdp描述，关于sdp文件描述详情可看从零开始写一个RTSP服务器（一）不一样的RTSP协议讲解

m=video 9832 RTP/AVP 96

格式为 m=<媒体类型> <端口号> <传输协议> <媒体格式 >
媒体类型：video，表示这是一个视频流

端口号：9832，**表示UDP发送的目的端口为9832**

传输协议：RTP/AVP，表示RTP OVER UDP，通过UDP发送RTP包

媒体格式：表示负载类型(payload type)，一般使用96表示H.264

a=rtpmap:96 H264/90000

格式为a=rtpmap:<媒体格式><编码格式>/<时钟频率>

a=framerate:25

表示帧率

c=IN IP4 127.0.0.1

IN：表示internet

IP4：表示IPV4

127.0.0.1：**表示UDP发送的目的地址为127.0.0.1**

特别注意：这段sdp文件描述的udp发送的目的IP为127.0.0.1，目的端口为9832

四、测试

讲上面给出的源码`rtp.c`、`rtp.h`、`rtp_h264.c`保存下来，然后编译运行

注意：该程序默认打开的是`test.h264`，如果你没有视频源，可以从RtspServer的example目录下获取

```
# gcc rtp.c rtp_h264.c
# ./a.out
```

讲上面的sdp文件保存为`rtp_h264.sdp`，使用vlc打开，即可观看到视频

```
# vlc rtp_h264.sdp
```

运行效果

至此，我们已经完成了RTSP协议交互和RTP打包H.264，下一篇文章就可以来实现一个播放H.264的RTSP服务器了