在Linux层处理数据并推流到Android层的红外图像处理软件

一、RTSP部分

1.1 Project CMakeLists.txt

```
PROJECT(RK3568_APP)

CMAKE_MINIMUM_REQUIRED(VERSION 3.5)

SET(COMPILER_PATH "/home/xjt/Gogs/OK3568-linux-source/buildroot/output/OK3568/host/bin/")

SET(CMAKE_C_COMPILER ${COMPILER_PATH}aarch64-buildroot-linux-gnu-gcc)
SET(CMAKE_CXX_COMPILER ${COMPILER_PATH}aarch64-buildroot-linux-gnu-g++)

SET(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -s -O3 -lrt")
# SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++17 -s -O3 -lrt -lstdc++fs -lopencv_imgcodecs")
SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++17 -s -O3 -lrt")

ADD_SUBDIRECTORY(src bin)
```

1.2 Src CMakeLists.txt

```
FILE(
    GLOB_RECURSE SRC_LIST
    ./*.c
    ./*.cpp
)

# Exe output path
SET(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR/bin})

ADD_EXECUTABLE(demo ${SRC_LIST})

# Link lib and so
TARGET_LINK_LIBRARIES(
    demo
    # drm
    libdrm.so;
    # ffmpeg
    libavformat.so;
    libswscale.so;
    libavcodec.so;
    libavutil.so;
    libavdevice.so;
    libavdevice.so;
    libswresample.so;
    # png
```

```
    libpng.so;
    # jpg
    libjpeg.so;
    # input
    libinput.so;
    # gst
    libgstreamer-1.0.so;
    libglib-2.0.so;
    libgobject-2.0.so;
    libgstapp-1.0.so;
    # opencv
    libopencv_core.so;
    libopencv_highgui.so;
    libopencv_imgproc.so;
    # xkb
    libxkbcommon.so;
    # wayland
    libwayland-cursor.so;
    libwayland-client.so;
    #alsa
    libasound.so
    # pthread
    pthread;
)
```

1.3 main.cpp

```
#include <chrono>
#include <thread>
#include <iostream>
extern "C"
{
#include <stdio.h>
#include <libavcodec/avcodec.h>
#include <libavformat/avformat.h>
#include <libavutil/opt.h>
#include <libavutil/time.h>
}

using namespace std;

static int video_is_eof;

#define STREAM_FRAME_RATE 120
#define STREAM_PIX_FMT   AV_PIX_FMT_YUV420P /* default pix_fmt */
#define VIDEO_CODEC_ID AV_CODEC_ID_MPEG4

/* video output */
static AVFrame *frame;
static AVPicture src_picture, dst_picture;

/* Add an output stream. */
static AVStream *add_stream(AVFormatContext *oc, AVCodec **codec, enum
AVCodecID codec_id)
{
```

```
    AVCodecContext *c;
    AVStream *st;

    /* find the encoder */
    *codec = avcodec_find_encoder(codec_id);
    if (!(*codec)) {
        av_log(NULL, AV_LOG_ERROR, "Could not find encoder for '%s'.\n",
avcodec_get_name(codec_id));
    }
    else {
        st = avformat_new_stream(oc, *codec);
        if (!st) {
            av_log(NULL, AV_LOG_ERROR, "Could not allocate stream.\n");
        }
        else {
            st->id = oc->nb_streams - 1;
            st->time_base.den = STREAM_FRAME_RATE;
            st->time_base.num = 1;

            c = st->codec;
            c->codec_id = codec_id;
            c->bit_rate = 1600000;
            c->width = 1920;
            c->height = 1080;
            c->time_base.den = STREAM_FRAME_RATE;
            c->time_base.num = 1;
            c->gop_size = 0; /* with out inter frame, only have intra frame
*/
            c->pix_fmt = STREAM_PIX_FMT;
        }
    }

    return st;
}

static int open_video(AVFormatContext *oc, AVCodec *codec, AVStream *st)
{
    int ret;
    AVCodecContext *c = st->codec;
    // AVCodecContext *c = avcodec_alloc_context3(codec);

    /* open the codec */
    ret = avcodec_open2(c, codec, NULL);
    if (ret < 0) {
        av_log(NULL, AV_LOG_ERROR, "Could not open video codec.\n",
avcodec_get_name(c->codec_id));
    }
    else {

        /* allocate and init a re-usable frame */
        frame = av_frame_alloc();
        if (!frame) {
            av_log(NULL, AV_LOG_ERROR, "Could not allocate video
frame.\n");
            ret = -1;
        }
```

```
        else {
            frame->format = c->pix_fmt;
            frame->width = c->width;
            frame->height = c->height;

            /* Allocate the encoded raw picture. */
            ret = avpicture_alloc(&dst_picture, c->pix_fmt, c->width, c->height);
            if (ret < 0) {
                av_log(NULL, AV_LOG_ERROR, "Could not allocate picture.\n");
            }
            else {
                /* copy data and linesize picture pointers to frame */
                *((AVPicture *)frame) = dst_picture;
            }
        }
    }

    return ret;
}

/* Prepare a dummy image. */
static void fill_yuv_image(AVPicture *pict, int frame_index, int width, int height)
{
    int x, y, i;

    i = frame_index;

    /* Y */
    for (y = 0; y < height; y++)
        for (x = 0; x < width; x++)
            pict->data[0][y * pict->linesize[0] + x] = x + y + i * 3;

    /* Cb and Cr */
    for (y = 0; y < height / 2; y++) {
        for (x = 0; x < width / 2; x++) {
            pict->data[1][y * pict->linesize[1] + x] = 128 + y + i * 2;
            pict->data[2][y * pict->linesize[2] + x] = 64 + x + i * 5;
        }
    }
}

static int write_video_frame(AVFormatContext *oc, AVStream *st, int64_t frameCount)
{
    int ret = 0;
    AVCodecContext *c = st->codec;
    // AVCodecContext *c = avcodec_alloc_context3(st->codecpar);

    fill_yuv_image(&dst_picture, frameCount, c->width, c->height);

    AVPacket pkt = { 0 };
    int got_packet;
    av_init_packet(&pkt);
```

```
132
133     /* encode the image */
134     frame->pts = frameCount;
135     ret = avcodec_encode_video2(c, &pkt, frame, &got_packet);
136
137     if (ret < 0) {
138         av_log(NULL, AV_LOG_ERROR, "Error encoding video frame.\n");
139     }
140     else {
141         if (got_packet) {
142             pkt.stream_index = st->index;
143             pkt.pts = av_rescale_q_rnd(pkt.pts, c->time_base, st-
>time_base, AVRounding(AV_ROUND_NEAR_INF | AV_ROUND_PASS_MINMAX));
144             ret = av_write_frame(oc, &pkt);
145
146             if (ret < 0) {
147                 av_log(NULL, AV_LOG_ERROR, "Error while writing video
frame.\n");
148             }
149         }
150     }
151
152     return ret;
153 }
154
155 int main(int argc, char* argv[])
156 {
157     printf("starting...\n");
158
159     const char *url = "rtsp://192.168.50.84:8554/stream";
160
161     AVFormatContext *outContext;
162     AVStream *video_st;
163     AVCodec *video_codec;
164     int ret = 0;
165     int64_t frameCount = 0;
166
167     av_log_set_level(AV_LOG_DEBUG);
168
169     av_register_all();
170     avformat_network_init();
171
172     avformat_alloc_output_context2(&outContext, NULL, "rtsp", url);
173
174     if (!outContext) {
175         av_log(NULL, AV_LOG_FATAL, "Could not allocate an output context
for '%s'.\n", url);
176     }
177
178     if (!outContext->oformat) {
179         av_log(NULL, AV_LOG_FATAL, "Could not create the output format for
'%s'.\n", url);
180     }
181
182     video_st = add_stream(outContext, &video_codec, VIDEO_CODEC_ID);
183
```

```
    /* Now that all the parameters are set, we can open the video codec and allocate the necessary encode buffers. */
    if (video_st) {
        av_log(NULL, AV_LOG_DEBUG, "Video stream codec %s.\n ", avcodec_get_name(video_st->codec->codec_id));

        ret = open_video(outContext, video_codec, video_st);
        if (ret < 0) {
            av_log(NULL, AV_LOG_FATAL, "Open video stream failed.\n");
        }
    }
    else {
        av_log(NULL, AV_LOG_FATAL, "Add video stream for the codec '%s' failed.\n", avcodec_get_name(VIDEO_CODEC_ID));
    }

    av_dump_format(outContext, 0, url, 1);

    ret = avformat_write_header(outContext, NULL);
    if (ret != 0) {
        av_log(NULL, AV_LOG_ERROR, "Failed to connect to RTSP server for '%s'.\n", url);
    }

    while (video_st) {
        frameCount++;

        ret = write_video_frame(outContext, video_st, frameCount);

        if (ret < 0) {
            av_log(NULL, AV_LOG_ERROR, "Write video frame failed.\n", url);
            goto end;
        }
    }

    if (video_st) {
        avcodec_close(video_st->codec);
        av_free(src_picture.data[0]);
        av_free(dst_picture.data[0]);
        av_frame_free(&frame);
    }

    avformat_free_context(outContext);

end:
    printf("finished.\n");

    getchar();

    return 0;
}
```