1 功能

2 基础知识

2.1 摄像头

UVC：USB视频类，是一种为USB视频捕获设备定义的协议标准。使用UVC技术的摄像头在使用时不用安装驱动，因为大多数系统都支持UVC设备。

2 实现

2.1 HTTP服务器

视频采集：

```
/* 摄像头初始化 */
struct vdIn{
    int fd;
    char *videodevice;
    char *status;
    char *pictName;
    struct v4l2_capability cap;  //设备的功能，比如是否是视频输入设备
    struct v4l2_format fmt;
    struct v4l2_buffer buf;
    struct v4l2_requestbuffers rb;
    void *mem[NB_BUFFER];
    int width;
    int height;
    int fps;
    int formatIn;
}
int init_videoIn(void){
    vd->videodevice = "/dev/video0";
    vd->width = 640;
    vd->height = 480;
    vd->fps = 5;
    vd->formatIn = V4L2_PIX_FMT_YUYV;
}
int init_v4l2(struct vdIn *vd){
    /* 打开设备文件 */
    vd->fd = open(vd->videodevice, O_RDWR);
    /* 获取设备支持的功能 */
    ioctl(vd->fd, VIDIOC_QUERYCAP, &vd->cap);
    if (!(vd->cap.capabilities & V4L2_CAP_VIDEO_CAPTURE)) ;  //是否支持捕获
    if (!(vd->cap.capabilities & V4L2_CAP_STREAMING)) ;  //是否支持流
    if (!(vd->cap.capabilities & V4L2_CAP_READWRITE)) ;  //是否支持读写
    /* 设置视频捕获格式 */
    vd->fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    vd->fmt.fmt.pix.width = vd->width;
    vd->fmt.fmt.pix.height = vd->height;
    vd->fmt.fmt.pix.pixelformat = vd->formatIn;
    vd->fmt.fmt.pix.field = V4L2_FIELD_ANY;
    ioctl(vd->fd, VIDIOC_S_FMT, &vd->fmt);
    /* 设置视频帧率 */
    ...;
    ioctl(vd->fd, VIDIOC_S_PARM, setfps);
    /* 申请缓冲 */
```

```
    vd->rb.count = NB_BUFFER;  //定义了4个缓冲，可提高视频采集的效率
    vd->rb.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    vd->rb.memory = V4L2_MEMORY_MMAP;
    ioctl(vd->fd, VIDIOC_REQBUFS, &vd->rb);
    /* 获取每个缓冲区的信息，并映射到内存 */
    for(i = 0; i < NB_BUFFER; i++){
        memset(&vd->buf, 0, sizeof(struct v4l2_buffer));
        vd->buf.index = i;
        vd->buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        vd->buf.memory = V4L2_MEMORY_MMAP;
        ioctl(vd->fd, VIDIOC_QUERYBUF, &vd->buf);
        vd->mem[i] = mmap(0, vd->buf.length, PROT_READ,
                          MAP_SHARED, vd->fd, vd->buf.m.offset);
    }
    /* 排列缓冲区(缓冲入队) */
    ioctl(vd->fd, VIDIOC_QBUF, &vd->buf);
}
int video_enable(struct vdIn *vd){
    /* 开始采集视频 */
    int type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    ioctl(vd->fd, VIDIOC_STREAMON, &type);
}
/* 开始采集视频 */
int input_run(void){
    pthread_create(&cam, 0, cam_thread, NULL);
    pthread_detach(cam);
}
/* 该线程获取一个帧并将其复制到全局缓冲区 */
void *cam_thread(void *arg){
    while(!pglobal->stop){
        /* 取出FIFO缓存中已经采样的帧缓冲(缓冲出队) */
        memset(&vd->buf, 0, sizeof(struct v4l2_buffer));
        vd->buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        vd->buf.memory = V4L2_MEMORY_MMAP;
        ioctl(vd->fd, VIDIOC_DQBUF, &vd->buf);
        memcpy (vd->framebuffer, vd->mem[vd->buf.index],
                (size_t) vd->buf.bytesused);
        /* 将处理完的缓冲重新入队，这样可以循环采集 */
        ioctl(vd->fd, VIDIOC_QBUF, &vd->buf);
        /* 将JPG图片复制到全局缓冲区 */
        pthread_mutex_lock( &pglobal->db );
        /* 捕获到YUV格式，则立即转换为JPEG格式 */
        pglobal->size = compress_yuyv_to_jpeg(videoIn, pglobal->buf,
                                              videoIn->framesizeIn,
gquality,
                                              videoIn-
>fmt.fmt.pix.pixelformat);
        /* 发送帧更新信号 */
        pthread_cond_broadcast(&pglobal->db_update);
        pthread_mutex_unlock( &pglobal->db);
    }
}

int input_cmd(int_cmd_type cmd, int value){
    switch(cmd){
        case IN_CMD_RESET:
        case IN_CMD_RESET_PAN_TILT: //平移/倾斜
        case IN_CMD_PAN_SET:  //平移量设置
```

```
        case IN_CMD_TILT_SET:  //倾斜量设置
        case IN_CMD_SATURATION_PLUS:  //饱和度增加
        case IN_CMD_CONTRAST_PLUS:  //对比度增加
        case IN_CMD_BRIGHTNESS_PLUS:  //亮度增加
        case IN_CMD_GAIN_MINUS:  //获得
        case IN_CMD_FOCUS_PLUS:  //焦点
        case IN_CMD_LED_ON:  //开启闪光灯
        default:
    }
}

void server_thread(void *arg){
    /* 初始化套接字 */
    while(!pglobal->stop){
        accept(...);
        pthread_create(..., &client_thread);
        pthread_detach(...);
    }
}

void client_thread(void *arg){
    /* 读取请求行 */
    /* 判断请求类型 */
    /* 读取请求头部 */
    /* 响应请求 */
    switch(req.type){
        case A_SNAPSHOT:
        case A_STREAM:
            send_stream(...);
            break;
        case A_COMMAND:
        case A_FILE:
        default:
    }
}

void send_stream(int fd){
    /* 发送响应头部 */
    while(!pglobal->stop){
        /* 等待新的帧到来 */
        pthread_cond_wait(&pglobal->db_update, &pglobal->db);
        /* 读取缓冲区并发送 */
        ...;
        pthread_mutex_unlock( &pglobal->db );
    }
}
```

2.2 QT客户端

3 问题及解决

3.1 客户端关闭套接字导致程序退出

问题：当客户端发送action=stream请求时，服务器在一个循环中不断发送视频流到客户端。此时，当客户端关闭连接时，服务器就会自动退出。

原因：将错误定位到send_stream函数中，发现当客户端关闭连接后，函数执行到write调用就停止了，故问题出在write函数上。

解决方法:

如果是阻塞模式，服务端recv则会阻塞。服务端send，则会产生SIGPIPE信号中断程序。

如果是非阻塞模式，服务端recv会返回-1。服务端send，则会产生SIGPIPE信号中断程序。

1）忽略SIGPIPE信号，即可避免程序中断而退出。`signal(SIGPIPE, SIG_IGN);`

2）在write之前先用非阻塞模式读套接字，若返回-1则跳出循环。

3.2 内存返回越界导致程序退出

问题：程序异常退出，并输出glibc detected malloc(): memory corruption: 0x00001122.

原因：使用malloc函数为request结构体分配堆内存，在对request结构体的一个字符数组进行memcpy操作时发送内存访问越界，导致堆的数据结构被破坏，因此程序之后调用malloc函数就会产生错误。若分配在栈中，则不会产生这种错误。

3.3 读取客户端请求时没有读到文件末尾，导致响应消息出错