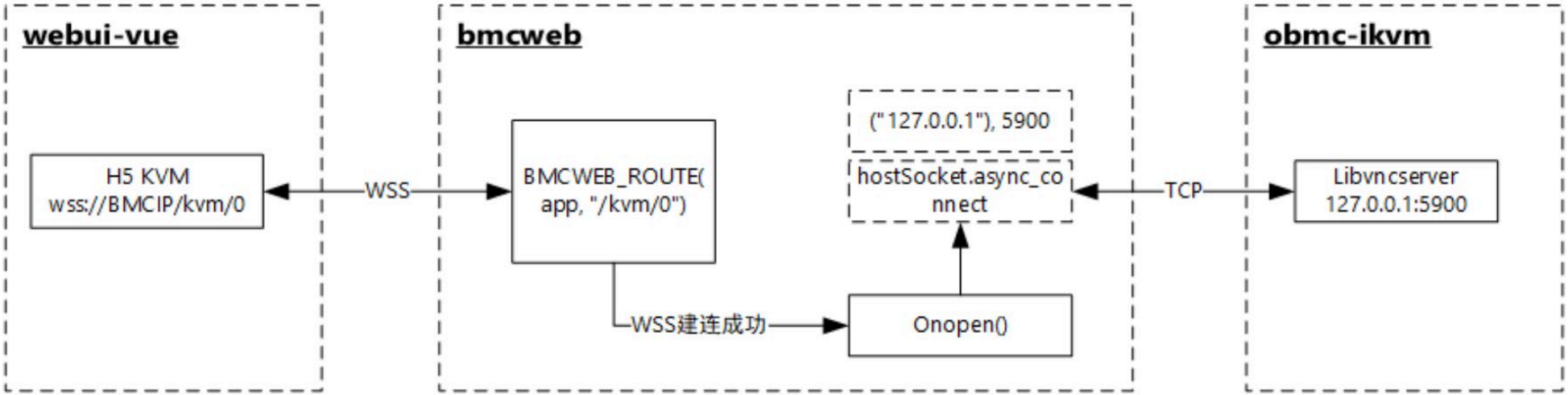
OpenBMC development obmc-ikvm video transmission

原创 Lemon in the Rain Posted on 2025-04-22 09:25:17 Read 1k Collection 18 Likes 22

Category Column: OpenBMC Article Tags: Audio and Video server linux C++

1. Connection establishment process

The KVM connection process mainly involves four code packages: webui-vue, bmcweb, obmc-ikvm and libvncserver. The connection logic diagram is shown in the following figure:



1.1 WebUI click "Start KVM"

Remote Control- >Remote Console->Start H5KVM, call the openTerminal() method on the front end, create an RFB client instance and connect to the websocket service via wss://10.18.35.109/kvm/0

cpp AI generated projects 登录复制 run

```cpp
openTerminal() {
  // 从Vuex store获取认证令牌
  const token = this.$store.getters['authentication/token'];

  /* 初始化RFB客户端连接
   * - 使用DOM元素作为显示容器
   * - 构造带认证token的WebSocket地址
   * - 通过wsProtocols传递token进行身份验证 */
  this.rfb = new RFB(
    this.$refs.panel,
    `wss://${window.location.host}/kvm/0`,
    { wsProtocols: [token] }
  );
  ............
}}
```

收起 ^

1.2 Bmcweb router detects WSS connection

The bmcweb/include/kvm_websokcet.hpp code implements the routing registration and session management of KVM WebSocket, supports permission control, connection management and message processing, and is suitable for real-time KVM data transmission scenarios.

Registered a WebSocket route with the path /kvm/0

cpp AI generated projects 登录复制 run

```cpp
inline void requestRoutes(App& app)
{
    sessions.reserve(maxSessions);

    BMCWEB_ROUTE(app, "/kvm/0")
        .privileges({{"ConfigureComponents", "ConfigureManager"}})
        .websocket()
        .onopen([](crow::websocket::Connection& conn) {
        BMCWEB_LOG_DEBUG("Connection {} opened", logPtr(&conn));

        if (sessions.size() == maxSessions)
        {
            conn.close("Max sessions are already connected");
            return;
        }

        sessions[&conn] = std::make_shared<KvmSession>(conn);
    })
        .onclose([](crow::websocket::Connection& conn, const std::string&) {
        sessions.erase(&conn);
    })
        .onmessage([](crow::websocket::Connection& conn,
                      const std::string& data, bool) {
        if (sessions[&conn])
        {
            sessions[&conn]->onMessage(data);
        }
    });
}
```

收起 ^

After the websocket connection between webui and bmcweb is successfully established, the onpen() event will be triggered, and the KvmSession object will be created in its event processing function: sessions[&conn] = std::make_shared(conn);

When the KvmSession object is constructed, a TCP endpoint ("127.0.0.1"), 5900) is created and the KVM service is connected via hostSocket.async_connect();

cpp AI generated projects 登录复制 run

```cpp
class KvmSession : public std::enable_shared_from_this<KvmSession>              //继承的目的是防止异步操作中对象提前销毁导致空指针问题
{                                                                               //为了能安全的在类内部获取指向自身的std::shared_ptr
  public:
    explicit KvmSession(crow::websocket::Connection& connIn) :
        conn(connIn), hostSocket(conn.getIoContext())
    {

```

```
        boost::asio::ip::tcp::endpoint endpoint(
            boost::asio::ip::make_address("127.0.0.1"), 5900);
        hostSocket.async_connect(
            endpoint, [this, &connIn](const boost::system::error_code& ec) {
            if (ec)
            {
                BMCWEB_LOG_ERROR(
                    "conn:{}, Couldn't connect to KVM socket port: {}",
                    logPtr(&conn), ec);
                if (ec != boost::asio::error::operation_aborted)
                {
                    connIn.close("Error in connecting to KVM port");
                }
                return;
            }

            doRead();
        });
    }
    ......................
}
```

收起 ^

1.3 Relationship between obmc-ikvm and libvncserver

The service or code location where bmcweb establishes a connection through hostSocket.async_connect() is actually in libvncserver

rfbGetScreen() function initializes and assigns port + interface

rfbInitSockets()->rfbScreen->listenSock = rfbListenOnTCPPort(rfbScreen->port, iface) creates a socket socket(), bind(), and then listen()

c AI generated projects 登录复制 run

```c
rfbScreenInfoPtr rfbGetScreen(int* argc,char** argv,
 int width,int height,int bitsPerSample,int samplesPerPixel,
 int bytesPerPixel)
{
    screen->port=5900;
    screen->ipv6port=5900;
    screen->listenInterface = htonl(INADDR_ANY);
    ......
}
```

c AI generated projects 登录复制 run

```c
rfbListenOnTCPPort(int port,
                   in_addr_t iface)
```

```
{
    struct sockaddr_in addr;
    rfbSocket sock;
    int one = 1;

    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    addr.sin_addr.s_addr = iface;

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == RFB_INVALID_SOCKET) {
    return RFB_INVALID_SOCKET;
    }
    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
           (const char *)&one, sizeof(one)) < 0) {
    rfbCloseSocket(sock);
    return RFB_INVALID_SOCKET;
    }
    if (bind(sock, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
    rfbCloseSocket(sock);
    return RFB_INVALID_SOCKET;
    }
    if (listen(sock, 32) < 0) {
    rfbCloseSocket(sock);
    return RFB_INVALID_SOCKET;
    }

    return sock;
}
```

收起 ^

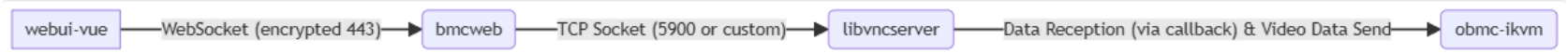
rfbProcessEvents()->rfbCheckFds(screen,usec) listens for new connections accept() and processes communication data

1.4 Establishing a connection

So far, the connection between the front-end webui-vue and the back-end obmc-ikvm is established:

1. Between webui-vue and bmcweb: establish a websocket connection, the external port number is encrypted 443
2. Between bmcweb and libvncserver: establish a TCP socket, the port number is 5900 by default, and supports customizing other ports, and the interface is localloop
3. Between libvncserver and obmc-ikvm: registering a callback function in obmc-ikmv to handle the reception of front-end data and the sending of underlying video data

It should be noted here that `libvncserver` this is a library that implements the VNC (Virtual Network Computing) protocol. It is responsible for handling standard VNC protocol handshake, authentication, data encoding and transmission details. By registering the processing function in obmc-ikvm, the communication data is parsed and processed in the obmc-ikvm service.



2. Data Transfer

2.1 Logic between Manager.run() and serverThread() threads

1. After the obmc-ikvm service is started, the overall startup process is: main() -> manager.run(), and the thread serverThread() is started at the same time, as shown in the following code:
2. continueExecuting = true, after creating the KVM session, server.wantsFrame() = true, cd->skipFrame = video.getFrameRate();
3. Loop1: Execute video.start() to get fd = open("/dev/video0", O_RDWR), video buffer adjustment Video::resize(), resizeAfterOpen = true;
4. Loop1: Execute video.getFrame() to get the underlying video data and store it in the buffer (char*)buffers[lastFrameIndex].data;
5. Loop1: Execute server.sendFrame(). If (cd->skipFrame){cd->skipFrame–;continue;} will delay 30 frames to send data to the front end.
6. Loop1: if (video.needsResize()) = true, waitServer() waits for the thread to finish executing server.run(), then sets manager->setServerDone(), the thread waits for Video to be set, waitVideo();
7. Loop1: Then videoDone = false; video.resize(); server.resize(); setVideoDone(); Then, the thread executes
8. Loop2-N: Then the main thread Manager::run() and the thread serverThread are cross-executed. The cross-point is controlled by the unique_lock mutex, which ensures the cross-polling processing of the video acquisition and sending of the main thread and the rfbProcessEvents(server, processTime) in the server.run() in the thread;


```cpp
void Manager::run()
{
    std::thread run(serverThread, this);

    while (continueExecuting)
    {
        if (server.wantsFrame())
        {
            video.start();
            video.getFrame();
            server.sendFrame();
        }
        else
        {
            video.stop();
        }

        if (video.needsResize())
        {
            waitServer();
            videoDone = false;
```

```
twen
twen
twen
 25          video.resize();
 26          server.resize();
 27          setVideoDone();
 28      }
 29      else
 30      {
 31          setVideoDone();             //A: 视频数据已经准备完毕，等待发送
 32          waitServer();               //B: 阻塞等待server状态
 33      }
 34  }
```

收起 ^

cpp AI generated projects 登录复制 run

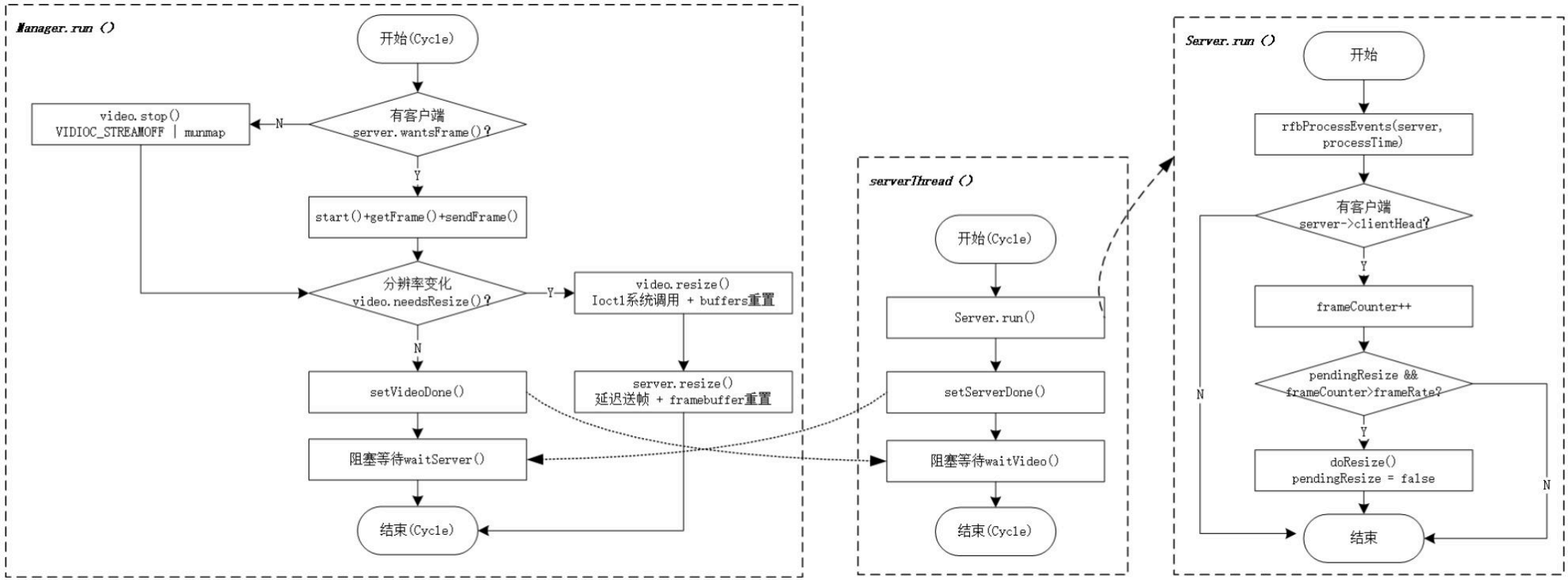
```cpp
void Manager::serverThread(Manager* manager)
{
    while (manager->continueExecuting)
    {
        manager->server.run();       //执行rfbProcessEvents处理libvncserve事件
        manager->setServerDone();    //C: server处理完毕，释放互斥锁
        manager->waitVideo();        //D: 等待视频数据处理状态
    }
}
```

cpp AI generated projects 登录复制 run

```cpp
void Server::run()
{
    rfbProcessEvents(server, processTime);

    if (server->clientHead)
    {
        frameCounter++;
        if (pendingResize && frameCounter > video.getFrameRate())
        {
            doResize();
            pendingResize = false;
        }
    }
}
```

收起 ^

One thing to note is that in the open source framework, videoDone is only set to false when video.needResize() is called, and is true in all other scenarios. Therefore, the overall processing flow chart is as follows:



2.2 The whole process of data processing

For example, a user enters a keyboard key value through the front-end KVM interface:

1. The front end sends data to bmcweb through websocket, bmcweb triggers the onMessage() event and reads the data into inputBuffer, and finally calls doWrite()
2. The doWrite() function calls hostSocket.async_write_some() to send inputBuffer.data() data to libvncserver
3. The following processing is performed in libvncserver: rfbProcessEvents->rfbCheckFds->rfbProcessClientMessage->rfbProcessClientNormalMessage->kbdAddEvent(), and the kbdAddEvent function is defined and processed in obmc-ikvm
4. Finally, in the obmc-ikvm service, kbdAddEvent->input->writeKeyboard(input->keyboardReport) sends the 8-byte keyboard data to the Host via USB.


```c
static void
rfbProcessClientNormalMessage(rfbClientPtr cl)
{
    int n=0;

```

```
    rfbClientToServerMsg msg;
    char *str;
    int i;
    ........
    if ((n = rfbReadExact(cl, (char *)&msg, 1)) <= 0) {
        if (n != 0)
            rfbLogPerror("rfbProcessClientNormalMessage: read");
        rfbCloseClient(cl);
        return;
    }

    switch (msg.type) {

        case rfbKeyEvent:

        if ((n = rfbReadExact(cl, ((char *)&msg) + 1,
                    sz_rfbKeyEventMsg - 1)) <= 0) {
            if (n != 0)
            rfbLogPerror("rfbProcessClientNormalMessage: read");
            rfbCloseClient(cl);
            return;
        }

        rfbStatRecordMessageRcvd(cl, msg.type, sz_rfbKeyEventMsg, sz_rfbKeyEventMsg);

        if(!cl->viewOnly) {
            cl->screen->kbdAddEvent(msg.ke.down, (rfbKeySym)Swap32IfLE(msg.ke.key), cl);
        }

        return;
        ..........
        }

}
```

收起 ^

So, how to transmit the video data obtained from the CPU to the front-end KVM?

1. Get the underlying video data through Video::getFrame() and store it in buffers, and encapsulate it through Video::getData(): (char*)buffers[lastFrameIndex].data
2. Call Manager.run()->server.sendFrame() to send, where sendFrame() mainly does the following things:

- Get the underlying data: char* data = video.getData()
- Detect the current session through the iterator: cl = rfbClientIteratorNext(it)

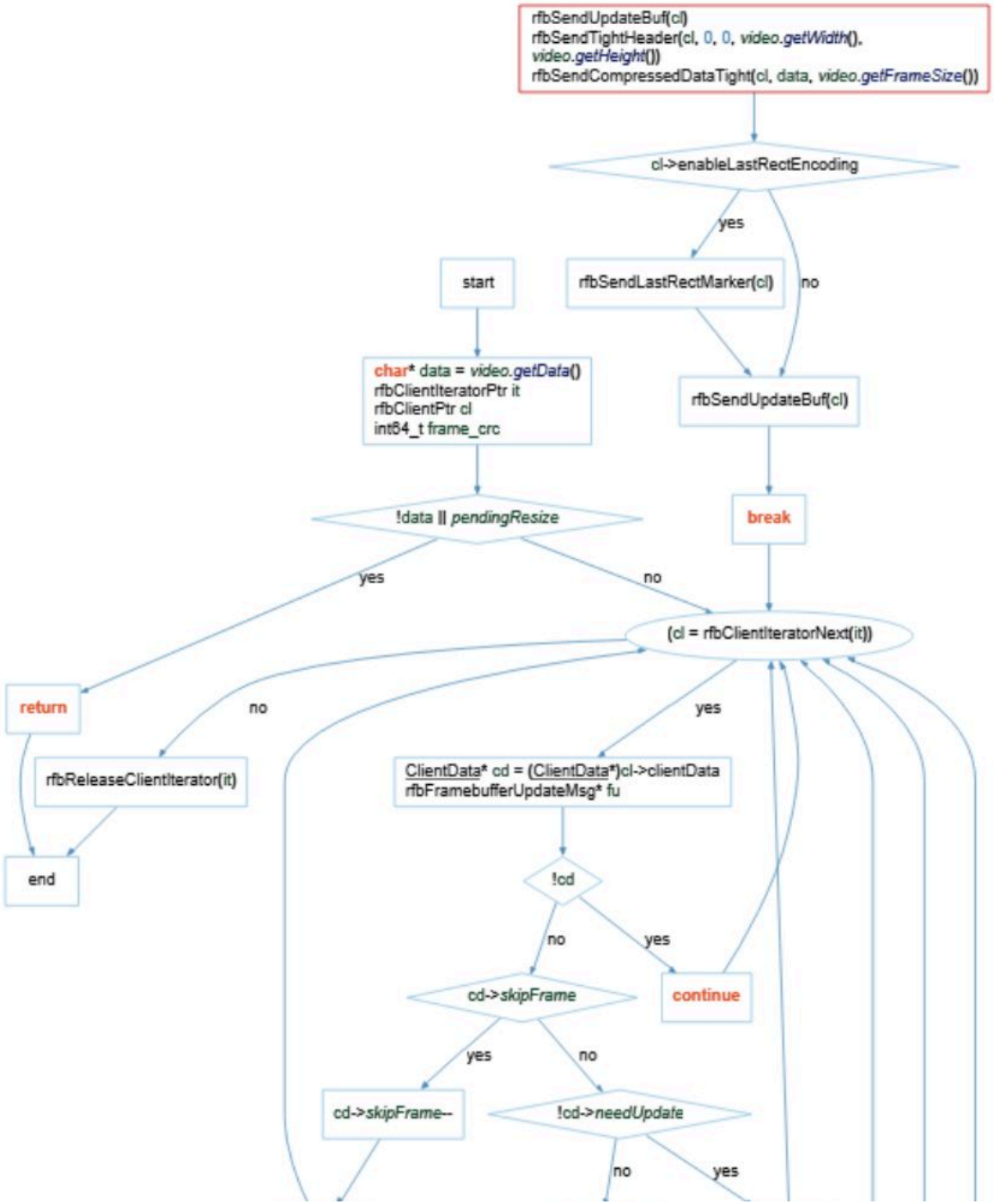
- Check key variables such as cd, skipFrame, needUpdata, etc. to confirm whether to send data to the front end immediately
- Send video data to the front end according to the data format video.getPixelformat()

c AI generated projects 登录复制 run

```c
void Server::sendFrame()
{
    // 获取视频帧数据
    char* data = video.getData();
    rfbClientIteratorPtr it;
    rfbClientPtr cl;
    int64_t frame_crc = -1;

    // 如果数据为空或需要调整大小，则直接返回
    if (!data || pendingResize)
    {
        return;
    }

    // 获取客户端迭代器
    it = rfbGetClientIterator(server);

    // 遍历所有客户端
    while ((cl = rfbClientIteratorNext(it)))
    {
        // 获取客户端数据
        ClientData* cd = (ClientData*)cl->clientData;
        rfbFramebufferUpdateMsg* fu = (rfbFramebufferUpdateMsg*)cl->updateBuf;

        // 如果客户端数据为空，则跳过
        if (!cd)
        {
            continue;
        }

        // 如果客户端需要跳过帧，则减少跳过帧计数并跳过
        if (cd->skipFrame)
        {
            cd->skipFrame--;
            continue;
        }

        // 如果客户端不需要更新，则跳过
        if (!cd->needUpdate)
        {
            continue;
```

```
        }

        // 如果需要计算帧的 CRC 校验和
        if (calcFrameCRC)
        {
            // 如果尚未计算 CRC，则计算帧的 CRC（跳过 JFIF 头部的 0x30 字节）
            if (frame_crc == -1)
            {
                frame_crc =
                    boost::crc<32, 0x04C11DB7, 0xFFFFFFFF, 0xFFFFFFFF, true, true>(
                        data + 0x30, video.getFrameSize() - 0x30);
            }

            // 如果当前帧的 CRC 与上一帧相同，则跳过
            if (cd->last_crc == frame_crc)
            {
                continue;
            }

            // 更新客户端的 CRC
            cd->last_crc = frame_crc;
        }

        // 标记客户端不需要更新
        cd->needUpdate = false;

        // 设置更新消息中的矩形数量
        if (cl->enableLastRectEncoding)
        {
            fu->nRects = 0xFFFF; // 启用最后矩形编码
        }
        else
        {
            fu->nRects = Swap16IfLE(1); // 单个矩形更新
        }

        // 根据视频像素格式处理帧数据
        switch (video.getPixelformat())
        {
            case V4L2_PIX_FMT_RGB24:
                // 将帧数据复制到帧缓冲区
                framebuffer.assign(data, data + video.getFrameSize());
                // 标记整个帧为已修改
                rfbMarkRectAsModified(server, 0, 0, video.getWidth(),
                                      video.getHeight());
                break;
```

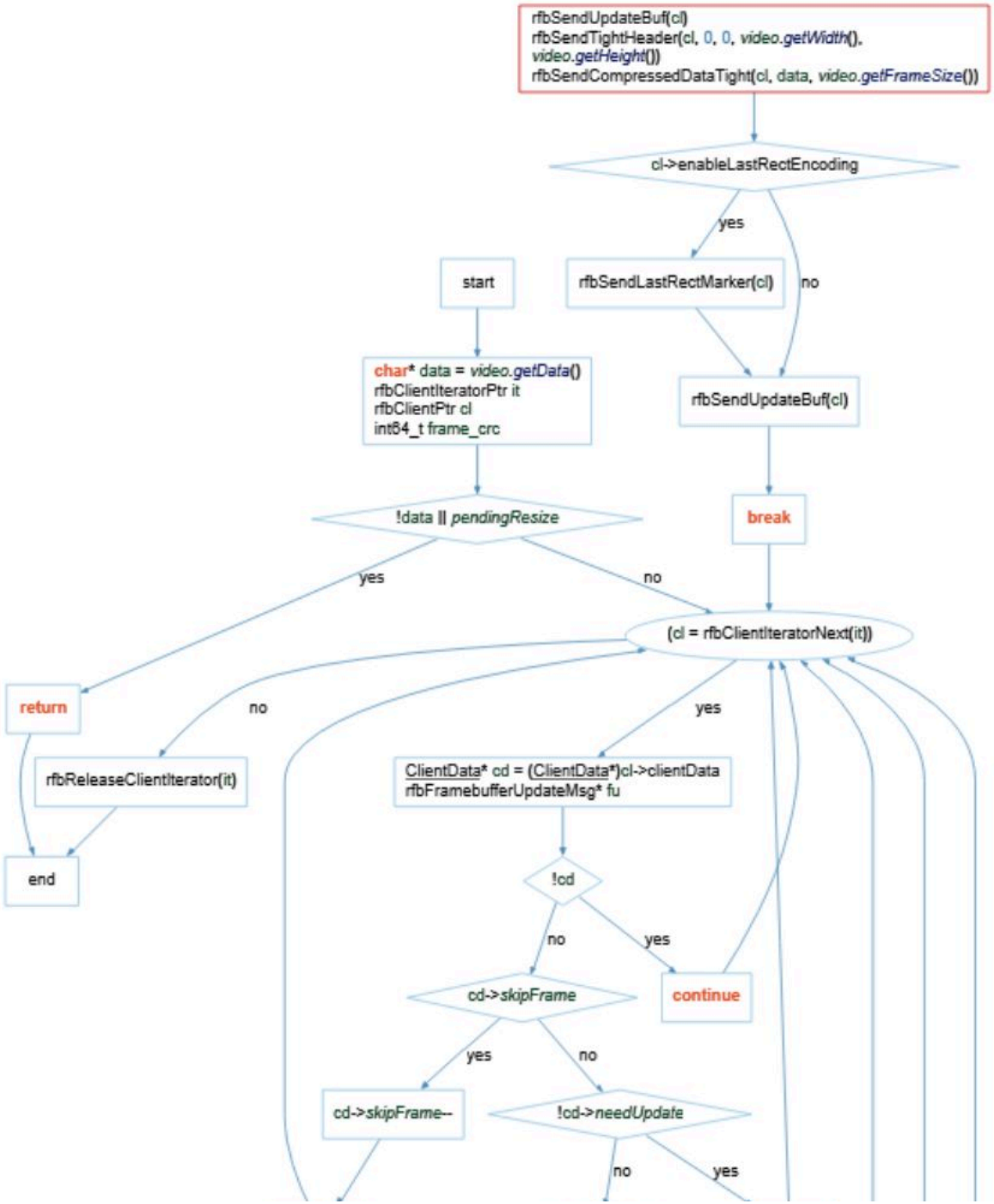
```
            case V4L2_PIX_FMT_JPEG:
                // 发送帧更新消息
                fu->type = rfbFramebufferUpdate;
                cl->ublen = sz_rfbFramebufferUpdateMsg;
                rfbSendUpdateBuf(cl);
                // 设置 Tight 编码
                cl->tightEncoding = rfbEncodingTight;
                rfbSendTightHeader(cl, 0, 0, video.getWidth(), video.getHeight());
                // 发送 JPEG 数据
                cl->updateBuf[cl->ublen++] = (char)(rfbTightJpeg << 4);
                rfbSendCompressedDataTight(cl, data, video.getFrameSize());
                // 如果启用了最后矩形编码，则发送最后矩形标记
                if (cl->enableLastRectEncoding)
                {
                    rfbSendLastRectMarker(cl);
                }
                // 发送更新缓冲区
                rfbSendUpdateBuf(cl);
                break;

            default:
                break;
        }
    }

    // 释放客户端迭代器
    rfbReleaseClientIterator(it);
}
```


rfbSendUpdateBuf(cl)
rfbSendTightHeader(cl, 0, 0, video.getWidth(), video.getHeight())
rfbSendCompressedDataTight(cl, data, video.getFrameSize())
cl->enableLastRectEncoding
yes
no
start
rfbSendLastRectMarker(cl)
char* data = video.getData()
rfbClientIteratorPtr it
rfbClientPtr cl
int64_t frame_crc
rfbSendUpdateBuf(cl)
!data || pendingResize
break
yes
no
(cl = rfbClientIteratorNext(it))
return
no
yes
rfbReleaseClientIterator(it)
ClientData* cd = (ClientData*)cl->clientData
rfbFramebufferUpdateMsg* fu
end
!cd
no
yes
cd->skipFrame
continue
yes
no
cd->skipFrame--
!cd->needUpdate
no
yes

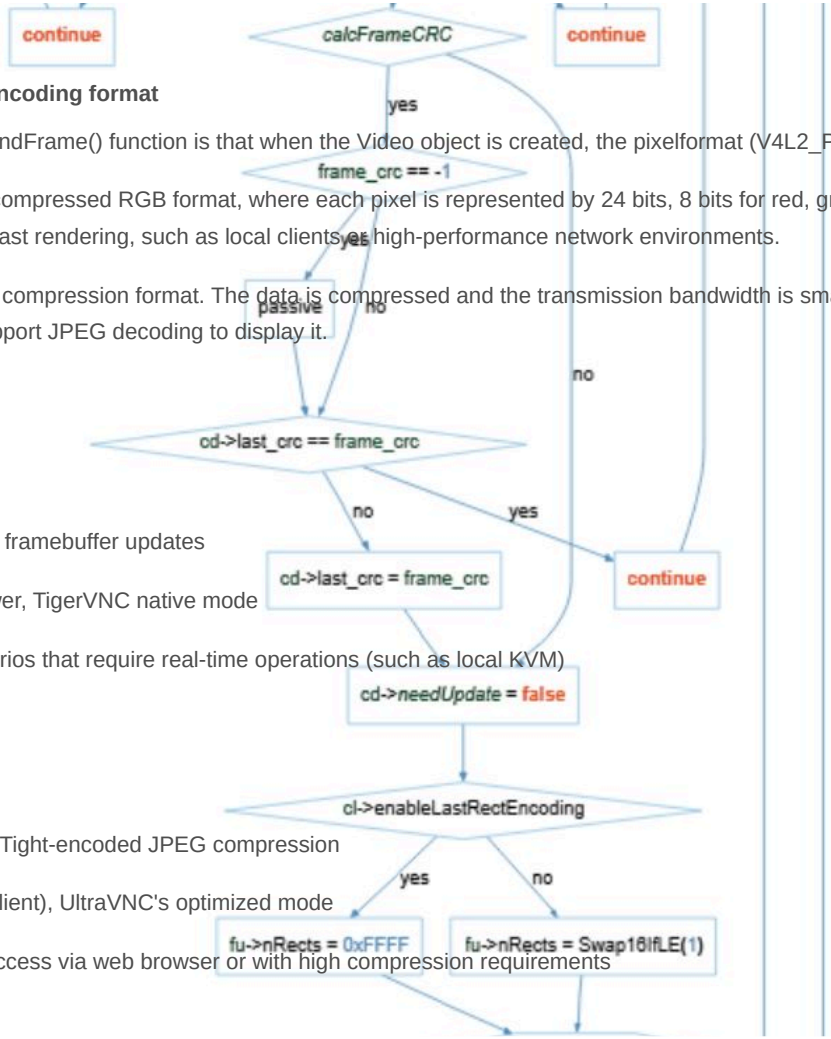
```cpp
void Video::getFrame()
{
    int rc(0);
    int fd_flags;
    v4l2_buffer buf;
    fd_set fds;
    timeval tv;

    if (fd < 0)
    {
        return;
    }

    FD_ZERO(&fds);
    FD_SET(fd, &fds);

    tv.tv_sec = 1;
    tv.tv_usec = 0;

    memset(&buf, 0, sizeof(v4l2_buffer));
    buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
    buf.memory = V4L2_MEMORY_MMAP;

    // Switch to non-blocking in order to safely dequeue all buffers; if the
    // video signal is lost while blocking to dequeue, the video driver may
    // wait forever if signal is not re-acquired
    fd_flags = fcntl(fd, F_GETFL);
    fcntl(fd, F_SETFL, fd_flags | O_NONBLOCK);

    rc = select(fd + 1, &fds, NULL, NULL, &tv);
    if (rc > 0)
    {
        do
        {
            rc = ioctl(fd, VIDIOC_DQBUF, &buf);
            if (rc >= 0)
            {
                buffers[buf.index].queued = false;

                if (!(buf.flags & V4L2_BUF_FLAG_ERROR))
                {
                    lastFrameIndex = buf.index;
                    buffers[lastFrameIndex].payload = buf.bytesused;
                    break;
                }
```

```
                else
                {
                    buffers[buf.index].payload = 0;
                }
            }
        } while (rc >= 0);
    }

    fcntl(fd, F_SETFL, fd_flags);

    for (unsigned int i = 0; i < buffers.size(); ++i)
    {
        if (i == (unsigned int)lastFrameIndex)
        {
            continue;
        }

        if (!buffers[i].queued)
        {
            memset(&buf, 0, sizeof(v4l2_buffer));
            buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
            buf.memory = V4L2_MEMORY_MMAP;
            buf.index = i;

            rc = ioctl(fd, VIDIOC_QBUF, &buf);
            if (rc)
            {
                log<level::ERR>("Failed to queue buffer",
                                entry("ERROR=%s", strerror(errno)));
            }
            else
            {
                buffers[i].queued = true;
            }
        }
    }
}
```

收起 ^


rfbSendUpdateBuf(cl)
rfbSendTightHeader(cl, 0, 0, video.getWidth(), video.getHeight())
rfbSendCompressedDataTight(cl, data, video.getFrameSize())
cl->enableLastRectEncoding
yes
no
start
rfbSendLastRectMarker(cl)
char* data = video.getData()
rfbClientIteratorPtr it
rfbClientPtr cl
int64_t frame_crc
rfbSendUpdateBuf(cl)
!data || pendingResize
break
yes
no
(cl = rfbClientIteratorNext(it))
return
no
yes
rfbReleaseClientIterator(it)
ClientData* cd = (ClientData*)cl->clientData
rfbFramebufferUpdateMsg* fu
end
!cd
no
yes
cd->skipFrame
continue
yes
no
cd->skipFrame--
!cd->needUpdate
no
yes

3. Encoding format



3.1 Introduction to obmc-ikvm encoding format

One thing to note in the Server::sendFrame() function is that when the Video object is created, the pixelformat (V4L2_PIX_FMT_JPEG) format is selected by default.

V4L2_PIX_FMT_RGB24 is an uncompressed RGB format, where each pixel is represented by 24 bits, 8 bits for red, green and blue. This format has a large amount of data, but is easy to process and is suitable for scenes that require fast rendering, such as local clients or high-performance network environments.

V4L2_PIX_FMT_JPEG is a JPEG compression format. The data is compressed and the transmission bandwidth is small. It is suitable for network transmission, especially in the case of limited bandwidth. The client needs to support JPEG decoding to display it.

Client support

1. RGB24 format:

- Requires client to support raw framebuffer updates
- Typical clients: RealVNC Viewer, TigerVNC native mode
- Suitable for low-latency scenarios that require real-time operations (such as local KVM)

2. JPEG format:

- Requires the client to support Tight-encoded JPEG compression
- Typical clients: noVNC (web client), UltraVNC's optimized mode
- Suitable for remote desktop access via web browser or with high compression requirements

c AI generated projects 登录复制 run

```c
// RGB24格式处理（代码片段）
case V4L2_PIX_FMT_RGB24:
    // 原始RGB像素数据直接拷贝到帧缓冲区
    framebuffer.assign(data, data + video.getFrameSize());
    // 标记整个画面区域需要更新
    rfbMarkRectAsModified(server, 0, 0, video.getWidth(), video.getHeight());
    break;

// JPEG格式处理（代码片段）
case V4L2_PIX_FMT_JPEG:
    // 使用Tight编码发送JPEG压缩数据
```

```
    cl->tightEncoding = rfbEncodingTight;
    rfbSendTightHeader(...);
    // 添加JPEG压缩标记
    cl->updateBuf[cl->ublen++] = (char)(rfbTightJpeg << 4);
    // 发送压缩后的JPEG数据
    rfbSendCompressedDataTight(cl, data, video.getFrameSize());
    break;
```


3.2 Encoding format definition

The specific encoding format is not defined in obmc-ikvm, but in the driver. The specific path is: drivers/media/platform/aspeed/aspeed-video.c/static int aspeed_video_probe(struct platform_device *pdev)

aspeed_video_probe->aspeed_video_setup_video->video->pix_fmt.pixelformat = V4L2_PIX_FMT_JPEG, the specific code is as follows:


```c
static int aspeed_video_setup_video(struct aspeed_video *video)
{
    const u64 mask = ~(BIT(V4L2_JPEG_CHROMA_SUBSAMPLING_444) |
                BIT(V4L2_JPEG_CHROMA_SUBSAMPLING_420));
    struct v4l2_device *v4l2_dev = &video->v4l2_dev;
    struct vb2_queue *vbq = &video->queue;
    struct video_device *vdev = &video->vdev;
    struct v4l2_ctrl_handler *hdl = &video->ctrl_handler;
    int rc;

    video->pix_fmt.pixelformat = V4L2_PIX_FMT_JPEG;
    video->pix_fmt.field = V4L2_FIELD_NONE;
    video->pix_fmt.colorspace = V4L2_COLORSPACE_SRGB;
    video->pix_fmt.quantization = V4L2_QUANTIZATION_FULL_RANGE;
    video->v4l2_input_status = V4L2_IN_ST_NO_SIGNAL;
    ..............
}
```


3.3 V4L2_PIX_FMT_RGB24 and V4L2_PIX_FMT_JPEG settings and differences

The main differences between the two are shown in the following table:

characteristic	V4L2_PIX_FMT_RGB24	V4L2_PIX_FMT_JPEG
Data format	Uncompressed RGB raw pixel data	JPEG lossy compressed streaming data
Bandwidth usage	High (24 bits per pixel, no compression)	Low (depends on compression factor)

characteristic	V4L2_PIX_FMT_RGB24	V4L2_PIX_FMT_JPEG
Transport Protocol	Pixel-by-pixel transmission via RAW encoding of RFB	Send compressed data using Tight encoding + JPEG specific tags
Hardware Dependency	No hardware encoding required, only frame buffer	The device must support JPEG encoding or software transcoding
Typical usage scenarios	Low latency and high quality scenes in LAN	Remote access scenarios with limited bandwidth

If the data is in V4L2_PIX_FMT_RGB24 format, it will be copied from buffers to framebuffer via framebuffer.assign(data, data + video.getFrameSize()), and then rfbMarkRectAsModified(server, 0, 0, video.getWidth(), video.getHeight()) will be called to notify the libvncserver service to send it to the front end.

If the data is in V4L2_PIX_FMT_JPEG format, it will be packaged and compressed in Tight format and sent to the KVM or VNC front end through rfbSendUpdateBuf(cl).

4. Two important buffer zones

So, what are the functions and differences between framebuffer and std::vector buffers?

framebuffer (member of Server class)

Function: As the frame buffer of RFB service, it stores the complete frame image data to be rendered.

Data source: When the pixel format is RGB24, the raw data captured by the video device is copied to this buffer via framebuffer.assign(data, data + frameSize). The hardware buffer is not directly mapped, and the content is filled by the application layer.

Life cycle: dynamic resizing, resize() when resolution changes, automatic release of memory (RAII) when destroyed.

std::vector buffers (member of the Video class)

Purpose: Manage the DMA/MMAP buffer of the video device for efficient zero-copy capture of frame data. Request the driver to allocate the buffer through VIDIOC_REQBUFS, and mmap it to user space. Circularly use the buffer through VIDIOC_QBUF/VIDIOC_DQBUF (producer-consumer model).

Performance advantage: Avoid data copying between user space and kernel, and reduce CPU usage.

Lifecycle: allocated/released in Video::start() and Video::stop(), synchronized with device opening/closing.

The key differences are compared as follows:

characteristic	framebuffer	buffers
Belong to the object	Server Class	Video
Data Source	Capture from Video devices and copy or convert	Direct-mapped device hardware/DMA buffers
Access method	Direct access through the two-dimensional array maintained by the Server	Access through V4L2's mmap memory mapping
Content change trigger	By `<font style="color:rgba(255, 255, 245, 0.9);">rfbMarkRectAsModified</font>` tag	Triggered when VIDIOC_DQBUF returns a new frame

characteristic	framebuffer	buffers
Memory Management	STL container automatic management	Manual munmap release
Format related	Stores only RGB24 data	The original output format of the storage device (such as JPEG/RGB24)