1. freerdp_client_load_addins, 此接口在pre_connect中调用。此函数主要做了三件事: 1. 加载static_channel 2.加载dynamic_channel 3.加载device_channel, 需要注意的是, 一些add channel信息的操作在freerdp_client_command_line_post_filter中就已经做了。涉及到的主要函数有freerdp_static_channel_collection_find、freerdp_dynamic_channel_collection_find、freerdp_device_collection_find、freerdp_client_add_device_channel、freerdp_client_add_static_channel、freerdp_client_add_dynamic_channel、freerdp_client_load_static_channel_addin。rdp中的通道主要分三类, 设备通道(device), 静态通道(static), 动态通道(dynamic)。每种通道中又包含具体的小通道。

```
BOOL freerdp_client_load_addins(rdpChannels* channels, rdpSettings* settings)
{
    UINT32 index;
    ADDIN_ARGV* args;

    if ((freerdp_static_channel_collection_find(settings, "rdpsnd")) ||
        (freerdp_dynamic_channel_collection_find(settings, "tsmf")))
    {
```

- 这里主要是查看device集合中是否存在指定名称的device, 如果没有就加进去, 加进去什么时候? 创建device通道以后, 触发connected后会创建所有集合中的device对象。下张图表示出了使用的位置。rdpdr_process_connect这个函数里面。此函数在rdpdr虚拟通道链接成功以后触发。
- *_find函数是查找settings->DeviceArray中是否存在同名数据, 如果没有则调用*_add_*函数, 这个函数主要是根据传进去的params创建device数据对象, 然后添加进settings->DeviceArray中。

```
if (settings->RedirectDrives)
{
    if (!freerdp_device_collection_find(settings, "drive"))
    {
        char* params[3];
        params[0] = "drive";
        params[1] = "media";
        params[2] = "*";

        if (!freerdp_client_add_device_channel(settings, 3, (char**) params))
            return FALSE;
    }
}
```

```
for (index = 0; index < settings->DeviceCount; index++)
{
    device = settings->DeviceArray[index];

    if (device->Type == RDPDR_DTYP_FILESYSTEM)
    {
        RDPDR_DRIVE* drive = (RDPDR_DRIVE*)device;

        if (drive->Path && (strcmp(drive->Path, "*") == 0))
        {
```

- 这个和上面的设备的处理类似, 只不过是检查和添加到settings->StaticChannelArray中。

```
if (!freerdp_static_channel_collection_find(settings, "rdpsnd"))
{
    char* params[2];
    params[0] = "rdpsnd";
    params[1] = "sys:fake";

    if (!freerdp_client_add_static_channel(settings, 2, (char**) params))
        return FALSE;
}
```

- 这个和上面的设备的处理类似, 只不过是检查和添加到settings->DynamicChannelArray中。

```
if (settings->SupportEchoChannel)
{
    char* p[1];
    int count;
    count = 1;
    p[0] = "echo";

    if (!freerdp_client_add_dynamic_channel(settings, count, p))
        return FALSE;
}
```

- settings->DynamicChannelArray在drdynvc_virtual_channel_event_connected中使用。

```
for (index = 0; index < settings->DynamicChannelCount; index++)
{
    args = settings->DynamicChannelArray[index];
    error = dvcman_load_addin(drdynvc, drdynvc->channel_mgr, args, settings);

    if (CHANNEL_RC_OK != error)
        goto error;
}
```

2.freerdp_client_load_static_channel_addin。主要两件事:1.获取通道的入口函数地址; 2.创建通道相关数据, 并且调用通道入口函数。

```
static BOOL freerdp_client_load_static_channel_addin(rdpChannels* channels,
        rdpSettings* settings, char* name, void* data)
{
    PVIRTUALCHANNELENTRY entry = NULL;
    PVIRTUALCHANNELENTRYEX entryEx = NULL;
    entryEx = (PVIRTUALCHANNELENTRYEX)(void*)freerdp_load_channel_addin_entry(name, NULL, NULL,
              FREERDP_ADDIN_CHANNEL_STATIC | FREERDP_ADDIN_CHANNEL_ENTRYEX);

    if (!entryEx)
        entry = freerdp_load_channel_addin_entry(name, NULL, NULL, FREERDP_ADDIN_CHANNEL_STATIC);

    if (entryEx)
    {
        if (freerdp_channels_client_load_ex(channels, settings, entryEx, data) == 0)
        {
            WLog_INFO(TAG, "loading channelEx %s", name);
            return TRUE;
        }
    }
    else if (entry)
    {
        if (freerdp_channels_client_load(channels, settings, entry, data) == 0)
        {
            WLog_INFO(TAG, "loading channel %s", name);
            return TRUE;
        }
    }

    return FALSE;
}
```

- freerdp_load_channel_addin_entry, 主要分为静态获取和动态获取。静态获取简单说就是从事先配置好的stables中获得接口地址。动态加载则需要先加载lib库, 再从lib中获取接口地址。

```
PVIRTUALCHANNELENTRY freerdp_load_channel_addin_entry(LPCSTR pszName,
        LPCSTR pszSubsystem, LPCSTR pszType, DWORD dwFlags)
{
    PVIRTUALCHANNELENTRY entry = NULL;

    if (freerdp_load_static_channel_addin_entry)
        entry = freerdp_load_static_channel_addin_entry(pszName, pszSubsystem, pszType, dwFlags);

    if (!entry)
        entry = freerdp_load_dynamic_channel_addin_entry(pszName, pszSubsystem, pszType, dwFlags);

    if (!entry)
        WLog_WARN(TAG, "Failed to load channel %s [%s]", pszName, pszSubsystem);

    return entry;
}
```

- freerdp_channels_client_load_ex, 1. 初始化clientDataList对应的pChannelClientData的值, 最主要的就是为entry成员赋值, 这个成员存储的就是通道的入口函数地址。2.初始化initDataList中对应的pChannelInitData的值, 最主要的就是设置channels成员。3. 最主要的是设置EntryPoints的成员, 主要是生命周期回调例如FreeRDP_VirtualChannelInitEx(此函数再entry中调用), 还有open(此函数在)close等。4. 调用entry入口函数。pChannelClientData中主要存放三种类型的数据, 1.通道入口函数地址. 2.通道的init回调函数地址. 3.每个通道特有的数据结构

```
pChannelClientData = &channels->clientDataList[channels->clientDataCount];
pChannelClientData->entryEx = entryEx;
pChannelInitData = &(channels->initDataList[channels->initDataCount++]);
pInitHandle = pChannelInitData;
pChannelInitData->channels = channels;
ZeroMemory(&EntryPointsEx, sizeof(CHANNEL_ENTRY_POINTS_FREERDP_EX));
EntryPointsEx.cbSize = sizeof(EntryPointsEx);
EntryPointsEx.protocolVersion = VIRTUAL_CHANNEL_VERSION_WIN2000;
EntryPointsEx.pVirtualChannelInitEx = FreeRDP_VirtualChannelInitEx;
EntryPointsEx.pVirtualChannelOpenEx = FreeRDP_VirtualChannelOpenEx;
EntryPointsEx.pVirtualChannelCloseEx = FreeRDP_VirtualChannelCloseEx;
EntryPointsEx.pVirtualChannelWriteEx = FreeRDP_VirtualChannelWriteEx;
EntryPointsEx.MagicNumber = FREERDP_CHANNEL_MAGIC_NUMBER;
EntryPointsEx.pExtendedData = data;
EntryPointsEx.context = ((freerdp*) settings->instance)->context;
/* enable VirtualChannelInit */
channels->can_call_init = TRUE;
EnterCriticalSection(&channels->channelsLock);
status = pChannelClientData->entryEx((PCHANNEL_ENTRY_POINTS_EX) &EntryPointsEx, pInitHandle);
LeaveCriticalSection(&channels->channelsLock);
/* disable MyVirtualChannelInit */
channels->can_call_init = FALSE;
```

3. 通道入口函数entry, 主要初始化通道自己专有的数据结构, 并且调用freerdp_channels_client_load_ex中赋值的init函数即FreeRDP_VirtualChannelInitEx。比较主要的就是设置通道名称了标识, 还有就是创建通道执行的环境channel contex, 再有就是将负责管理通道生命周期回调的对象赋值给通道专属结构, 再在有就是repcontex也记录在专属结构中。

```
    drdynvc->channelDef.options =
        CHANNEL_OPTION_INITIALIZED |
        CHANNEL_OPTION_ENCRYPT_RDP |
        CHANNEL_OPTION_COMPRESS_RDP;
    sprintf_s(drdynvc->channelDef.name, ARRAYSIZE(drdynvc->channelDef.name), "drdynvc");
    drdynvc->state = DRDYNVC_STATE_INITIAL;
    pEntryPointsEx = (CHANNEL_ENTRY_POINTS_FREERDP_EX*) pEntryPoints;

    if ((pEntryPointsEx->cbSize >= sizeof(CHANNEL_ENTRY_POINTS_FREERDP_EX)) &&
        (pEntryPointsEx->MagicNumber == FREERDP_CHANNEL_MAGIC_NUMBER))
    {
        context = (DrdynvcClientContext*) calloc(1, sizeof(DrdynvcClientContext));

        if (!context)
        {
            WLog_Print(drdynvc->log, WLOG_ERROR, "calloc failed!");
            free(drdynvc);
            return FALSE;
        }

        context->handle = (void*) drdynvc;
        context->custom = NULL;
        drdynvc->context = context;
        context->GetVersion = drdynvc_get_version;
        drdynvc->rdpcontext = pEntryPointsEx->context;
    }

    drdynvc->log = WLog_Get(TAG);
    WLog_Print(drdynvc->log, WLOG_DEBUG, "VirtualChannelEntryEx");
    CopyMemory(&(drdynvc->channelEntryPoints), pEntryPoints, sizeof(CHANNEL_ENTRY_POINTS_FREERDP_EX));
```

- 上图是初始化drdynvc, 这个对象就是动态虚拟通道的专有数据结构。

- 调用pVirtualChannelInitEx,间接调用FreeRDP_VirtualChannelInitEx, 这里主要目的就是将刚刚生成的通道专属数据结构保存到channels.clientlist中, 并且绑定initevent回调。

```
rc = drdynvc->channelEntryPoints.pVirtualChannelInitEx(drdynvc, context, pInitHandle,
        &drdynvc->channelDef, 1, VIRTUAL_CHANNEL_VERSION_WIN2000, drdynvc_virtual_channel_init_event_ex)
```

4. FreeRDP_VirtualChannelInitEx, 1. 为channelinit的一些成员赋值 2.为channelopendata的一些成员赋值 2.将通道专属的数据结构和通道专属的事件回调函数赋值给channelclentdate

```
static UINT VCAPITYPE FreeRDP_VirtualChannelInitEx(LPVOID lpUserParam, LPVOID clientContext,
        LPVOID pInitHandle,
        PCHANNEL_DEF pChannel, INT channelCount, ULONG versionRequested,
        PCHANNEL_INIT_EVENT_EX_FN pChannelInitEventProcEx)
{
    INT index;
    CHANNEL_DEF* channel;
    rdpSettings* settings;
    PCHANNEL_DEF pChannelDef;
    CHANNEL_INIT_DATA* pChannelInitData;
    CHANNEL_OPEN_DATA* pChannelOpenData;
    CHANNEL_CLIENT_DATA* pChannelClientData;
    rdpChannels* channels = (rdpChannels*) pInitHandle;
```

- 给channelinitdata成员赋值

```
pChannelInitData = (CHANNEL_INIT_DATA*) pInitHandle;
channels = pChannelInitData->channels;
pChannelInitData->pInterface = clientContext;
```

- 检查channels是否成功打开了

```
for (index = 0; index < channelCount; index++)
{
    pChannelDef = &pChannel[index];

    if (freerdp_channels_find_channel_open_data_by_name(channels, pChannelDef->name) != 0)
    {
        return CHANNEL_RC_BAD_CHANNEL;
    }
}
```

- 将通道专属数据结构对象和专属init回调赋值给channelclientdata对象

```
pChannelClientData = &channels->clientDataList[channels->clientDataCount];
pChannelClientData->pChannelInitEventProcEx = pChannelInitEventProcEx;
pChannelClientData->pInitHandle = pInitHandle;
pChannelClientData->lpUserParam = lpUserParam;
channels->clientDataCount++;
settings = channels->instance->context->settings;
```

- 初始化channelopendata对象, 为什么要循环目前还不清楚, 这里的openhandle其实就是一个int类型的值, 主要作用就是作为hashtable的key, 用来保存opendata的

值。opendata中主要存放了openevent回调, channel的name和opt, 通道专属数据结构, 自己在hashtable中的序号, channels对象等

```
for (index = 0; index < channelCount; index++)
{
    pChannelDef = &pChannel[index];
    pChannelOpenData = &channels->openDataList[channels->openDataCount];
    pChannelOpenData->OpenHandle = InterlockedIncrement(&g_OpenHandleSeq);
    pChannelOpenData->channels = channels;
    pChannelOpenData->lpUserParam = lpUserParam;
    HashTable_Add(channels->openHandles, (void*)(UINT_PTR) pChannelOpenData->OpenHandle,
                  (void*) pChannelOpenData);
    pChannelOpenData->flags = 1; /* init */
    strncpy(pChannelOpenData->name, pChannelDef->name, CHANNEL_NAME_LEN);
    pChannelOpenData->options = pChannelDef->options;

    if (settings->ChannelCount < CHANNEL_MAX_COUNT)
    {
        channel = &settings->ChannelDefArray[settings->ChannelCount];
        strncpy(channel->name, pChannelDef->name, 7);
        channel->options = pChannelDef->options;
        settings->ChannelCount++;
    }

    channels->openDataCount++;
}
```

到这里通道的初始化操作就完成了, 接下来是通道的链接过程。

5.freerdp_channels_pre_connect这个接口会在rdp_client_connect链接之前调用, 其内部主要调用了pChannelInitEventProc回调, 这个回调实在通道入口函数entry中调用*_init_*类型的函数的时候绑定的。

```
UINT freerdp_channels_pre_connect(rdpChannels* channels, freerdp* instance)
{
    UINT error = CHANNEL_RC_OK;
    int index;
    CHANNEL_CLIENT_DATA* pChannelClientData;

    for (index = 0; index < channels->clientDataCount; index++)
    {
        pChannelClientData = &channels->clientDataList[index];

        if (pChannelClientData->pChannelInitEventProc)
        {
            pChannelClientData->pChannelInitEventProc(
                pChannelClientData->pInitHandle, CHANNEL_EVENT_INITIALIZED, 0, 0);
        }
        else if (pChannelClientData->pChannelInitEventProcEx)
        {
            pChannelClientData->pChannelInitEventProcEx(pChannelClientData->lpUserParam,
                    pChannelClientData->pInitHandle, CHANNEL_EVENT_INITIALIZED, 0, 0);
        }

        if (CHANNEL_RC_OK != getChannelError(instance->context))
            break;
    }

    return error;
}
```

- 多数通道的*_init_event*类型的函数都如下图, 我们这里传入的是init类型的event所有即使调用了, 也不会处理, 因为没有编写相应的处理逻辑。找了一圈只有rdpsnd通道中有event_init类型的处理逻辑rdpsnd_virtual_channel_event_initialized也仅仅是创建了一个stop事件。

```
switch (event)
{
    case CHANNEL_EVENT_CONNECTED:
        if ((error = rail_virtual_channel_event_connected(rail, pData, dataLength)))
            WLog_ERR(TAG, "rail_virtual_channel_event_connected failed with error %"PRIu32"!",
                     error);

        break;

    case CHANNEL_EVENT_DISCONNECTED:
        if ((error = rail_virtual_channel_event_disconnected(rail)))
            WLog_ERR(TAG, "rail_virtual_channel_event_disconnected failed with error %"PRIu32"!",
                     error);

        break;

    case CHANNEL_EVENT_TERMINATED:
        rail_virtual_channel_event_terminated(rail);
        break;

    case CHANNEL_EVENT_ATTACHED:
    case CHANNEL_EVENT_DETACHED:
    default:
        break;
```

6. 发送channels信息rdp_client_connect中调用mcs_client_begin进而调用mcs_send_connect_initial

```
if (status)
    status2 = freerdp_channels_pre_connect(instance->context->channels,
                                           instance);

if (settings->KeyboardLayout == KBD_JAPANESE_INPUT_SYSTEM_MS_IME2002)
{
    settings->KeyboardType = 7;
    settings->KeyboardSubType = 2;
    settings->KeyboardFunctionKey = 12;
}

if (!status || (status2 != CHANNEL_RC_OK))
{
    if (!freerdp_get_last_error(rdp->context))
        freerdp_set_last_error(instance->context, FREERDP_ERROR_PRE_CONNECT_FAILED);

    WLog_ERR(TAG, "freerdp_pre_connect failed");
    goto freerdp_connect_finally;
}

status = rdp_client_connect(rdp);
```

- 首先会设置nego的一些参数, 然后调用nego_connect链接服务器, 创建socket

```
if (!nego_connect(rdp->nego))
{
    if (!freerdp_get_last_error(rdp->context))
    {
        freerdp_set_last_error(rdp->context, FREERDP_ERROR_SECURITY_NEGO_CONNECT_FAILED);
        WLog_ERR(TAG, "Error: protocol security negotiation or connection failure");
    }

    return FALSE;
}
```

- 成功建立链接以后调用mcs_client_begin, 在这里面会通知服务器, mcs信息

```
if (rdp->state != CONNECTION_STATE_NLA)
{
    if (!mcs_client_begin(rdp->mcs))
        return FALSE;
}
```

- 发送msc信息, 主要是setting中的

```
if (!mcs_send_connect_initial(mcs))
{
    if (!freerdp_get_last_error(context))
        freerdp_set_last_error(context, FREERDP_ERROR_MCS_CONNECT_INITIAL_ERROR);

    WLog_ERR(TAG, "Error: unable to send MCS Connect Initial");
    return FALSE;
}
```

- 这里发送channels信息

```
mcs_initialize_client_channels(mcs, mcs->settings);
client_data = Stream_New(NULL, 512);
```

- 这个就是将所有channel信息通知服务器了, channelDefArray中的信息来自于FreeRDP_VirtualChannelInitEx

```
for (index = 0; index < mcs->channelCount; index++)
{
    CopyMemory(mcs->channels[index].Name, settings->ChannelDefArray[index].name, 8);
    mcs->channels[index].options = settings->ChannelDefArray[index].options;
}
```

7.成功链接以后调用freerdp_channels_post_connect

```
if (status)
{
    pointer_cache_register_callbacks(instance->context->update);
    IFCALLRET(instance->PostConnect, status, instance);
    instance->ConnectionCallbackState = CLIENT_STATE_POSTCONNECT_PASSED;

    if (status)
        status2 = freerdp_channels_post_connect(instance->context->channels, instance);
}
```

- 调用pChannelInitEventProc这个回调在通道入口函数entry中通过调用*_init_*类型函数时绑定。这里的类型是connected

```
if (pChannelClientData->pChannelInitEventProc)
{
    pChannelClientData->pChannelInitEventProc(
        pChannelClientData->pInitHandle, CHANNEL_EVENT_CONNECTED, hostname, hostnameLength);
}
else if (pChannelClientData->pChannelInitEventProcEx)
{
    pChannelClientData->pChannelInitEventProcEx(pChannelClientData->lpUserParam,
            pChannelClientData->pInitHandle, CHANNEL_EVENT_CONNECTED, hostname, hostnameLength);
}
```

- 然后广播connected事件, 这里的监听回调是在wf_client中绑定的, pre_connecte

```
pChannelOpenData = &channels->openDataList[index];
EventArgsInit(&e, "freerdp");
e.name = pChannelOpenData->name;
e.pInterface = pChannelOpenData->pInterface;
PubSub_OnChannelConnected(instance->context->pubSub, instance->context, &e);
```

- 为设备动态虚拟通道绑定回调, 如果要开通这个通道的话

```
if (channels->drdynvc)
{
    channels->drdynvc->custom = (void*) channels;
    channels->drdynvc->OnChannelConnected = freerdp_drdynvc_on_channel_connected;
    channels->drdynvc->OnChannelDisconnected =
        freerdp_drdynvc_on_channel_disconnected;
    channels->drdynvc->OnChannelAttached = freerdp_drdynvc_on_channel_attached;
    channels->drdynvc->OnChannelDetached = freerdp_drdynvc_on_channel_detached;
}
```

8.通过调用pChannelInitEventProc会回调rail_virtual_channel_init_event_ex

```
static VOID VCAPITYPE rail_virtual_channel_init_event_ex(LPVOID lpUserParam, LPVOID pInitHandle,
        UINT event, LPVOID pData, UINT dataLength)
{
    UINT error = CHANNEL_RC_OK;
    railPlugin* rail = (railPlugin*) lpUserParam;

    if (!rail || (rail->InitHandle != pInitHandle))
    {
        WLog_ERR(TAG,  "error no match");
        return;
    }

    switch (event)
    {
        case CHANNEL_EVENT_CONNECTED:
            if ((error = rail_virtual_channel_event_connected(rail, pData, dataLength)))
                WLog_ERR(TAG, "rail_virtual_channel_event_connected failed with error %"PRIu32"!",
                         error);
```

- 然后根据调用的时候设置的event类型, 会走rail_virtual_channel_event_connected的逻辑

```
switch (event)
{
    case CHANNEL_EVENT_CONNECTED:
        if ((error = rail_virtual_channel_event_connected(rail, pData, dataLength)))
            WLog_ERR(TAG, "rail_virtual_channel_event_connected failed with error %"PRIu32"!",
                     error);
```

9. 这里面主要做了三件事情: 1. 调用pVirtualChannelOpenEx并且绑定rail_virtual_channel_open_event_ex回调。2.为通道专属结构创建消息队列。3.创建通道专属线程, 这个线程中, 主要就是处理消息队列中的消息。

```
    UINT status;
    status = rail->channelEntryPoints.pVirtualChannelOpenEx(rail->InitHandle,
             &rail->OpenHandle, rail->channelDef.name, rail_virtual_channel_open_event_ex);

    if (status != CHANNEL_RC_OK)
    {
        WLog_ERR(TAG, "pVirtualChannelOpen failed with %s [%08"PRIX32"]",
                 WTSErrorToString(status), status);
        return status;
    }

    rail->queue = MessageQueue_New(NULL);

    if (!rail->queue)
    {
        WLog_ERR(TAG, "MessageQueue_New failed!");
        return CHANNEL_RC_NO_MEMORY;
    }

    if (!(rail->thread = CreateThread(NULL, 0,
                                      rail_virtual_channel_client_thread, (void*) rail, 0,
                                      NULL)))
    {
        WLog_ERR(TAG, "CreateThread failed!");
        MessageQueue_Free(rail->queue);
        rail->queue = NULL;
        return ERROR_INTERNAL_ERROR;
    }

    return CHANNEL_RC_OK;
```

- FreeRDP_VirtualChannelOpenEx比较简单, 将通道专属的*_open_event_*回调绑定到channelopendata数据对象中, 并且改变通道当前状态。这个回调会在收到服务器发送的消息后, 分发到对应通道以后调用。

```
        return CHANNEL_RC_ALREADY_OPEN;

    pChannelOpenData->flags = 2; /* open */
    pChannelOpenData->pInterface = pInterface;
    pChannelOpenData->pChannelOpenEventProcEx = pChannelOpenEventProcEx;
    *pOpenHandle = pChannelOpenData->OpenHandle;
    return CHANNEL_RC_OK;
```

- 这个比较简单了, 创建一个消息队列, 并赋值给rail通道的消息队列成员

```
rail->queue = MessageQueue_New(NULL);

if (!rail->queue)
{
    WLog_ERR(TAG, "MessageQueue_New failed!");
    return CHANNEL_RC_NO_MEMORY;
}
```

- 创建通道处理消息的线程, 这个线程中主要就是通过MessageQueue_Wait等待消息队列里面有消息, 然后调用rail_order_recv分发处理消息。更详细的介绍在消息流程文档中详细介绍。

```
if (!(rail->thread = CreateThread(NULL, 0,
                                  rail_virtual_channel_client_thread, (void*) rail, 0,
                                  NULL)))
{
    WLog_ERR(TAG, "CreateThread failed!");
    MessageQueue_Free(rail->queue);
    rail->queue = NULL;
    return ERROR_INTERNAL_ERROR;
}
```

10. PubSub_OnChannelConnected这个就是通知监听channelConnected的线程处理channelConnect事件。在哪里监听的呢? 在wf_pre_connect中监听的。

```
                            (int) GetKeyboardLayout(0) & 0x0000FFFF);
    PubSub_SubscribeChannelConnected(instance->context->pubSub,
                                     wf_OnChannelConnectedEventHandler);
    PubSub_SubscribeChannelDisconnected(instance->context->pubSub,
                                        wf_OnChannelDisconnectedEventHandler);
    return TRUE;
```

- wf_OnChannelConnectedEventHandler主要做了三件事: 1.为gid对象的各种函数回调成员填充回调函数。2.为rail对象的各种函数回调成员填充回调函数。3.为cliprdr对象的各种函数回调成员填充回调函数。这些绑定的回调在收到服务器消息以后处理消息的时候调用, 就是9中创建的通道专属线程中调用。

```
else if (strcmp(e->name, RDPGFX_DVC_CHANNEL_NAME) == 0)
{
    if (!settings->SoftwareGdi)
        WLog_WARN(TAG,
                  "Channel "RDPGFX_DVC_CHANNEL_NAME" does not support hardware acceleration, using

    gdi_graphics_pipeline_init(wfc->context.gdi, (RdpgfxClientContext*) e->pInterface);
}
else if (strcmp(e->name, RAIL_SVC_CHANNEL_NAME) == 0)
{
    wf_rail_init(wfc, (RailClientContext*) e->pInterface);
}
else if (strcmp(e->name, CLIPRDR_SVC_CHANNEL_NAME) == 0)
{
    wf_cliprdr_init(wfc, (CliprdrClientContext*) e->pInterface);
}
else if (strcmp(e->name, ENCOMSP_SVC_CHANNEL_NAME) == 0)
{
}
```

- 这里还不是很清楚, 貌似是虚拟通道相关的

```
if (channels->drdynvc)
{
    channels->drdynvc->custom = (void*) channels;
    channels->drdynvc->OnChannelConnected = freerdp_drdynvc_on_channel_connected;
    channels->drdynvc->OnChannelDisconnected =
        freerdp_drdynvc_on_channel_disconnected;
    channels->drdynvc->OnChannelAttached = freerdp_drdynvc_on_channel_attached;
    channels->drdynvc->OnChannelDetached = freerdp_drdynvc_on_channel_detached;
}
```

到这里通道初始化以及建立链接的过程就描述清楚了。

11.有几个通道是静态加载的vc、sound、rdpsnd、cliprdr

12. 说一下虚拟通道, 虚拟通道有一个总的, 还有一些被总的虚拟通道管理的成员虚拟通道, 通过tables就可以观察出来。之前有说过通道的加载逻辑, 这里再*_post_*函数调用以后, 如果是虚拟通道, 会有一定的区别。这里动态虚拟通道和非动态虚拟通道会有一定的区别。

```
if (pChannelClientData->pChannelInitEventProc)
{
    pChannelClientData->pChannelInitEventProc(
        pChannelClientData->pInitHandle, CHANNEL_EVENT_CONNECTED, hostname, hostnameLength);
}
```

- 这个是动态虚拟通道的*_init_*函数, 看图貌似与非动态虚拟通道没什么不同, 多了几个函数, 例如*_attached还有detached。主要还是*_connected(...)内部有所不同

```
switch (event)
{
    case CHANNEL_EVENT_CONNECTED:
        if ((error = drdynvc_virtual_channel_event_connected(drdynvc, pData, dataLength)))
            WLog_Print(drdynvc->log, WLOG_ERROR,
                       "drdynvc_virtual_channel_event_connected failed with error %"PRIu32"", error);

        break;

    case CHANNEL_EVENT_DISCONNECTED:
        if ((error =  drdynvc_virtual_channel_event_disconnected(drdynvc)))
            WLog_Print(drdynvc->log, WLOG_ERROR,
                       "drdynvc_virtual_channel_event_disconnected failed with error %"PRIu32"", erro

        break;

    case CHANNEL_EVENT_TERMINATED:
        if ((error =  drdynvc_virtual_channel_event_terminated(drdynvc)))
            WLog_Print(drdynvc->log, WLOG_ERROR,
                       "drdynvc_virtual_channel_event_terminated failed with error %"PRIu32"", error)

        break;

    case CHANNEL_EVENT_ATTACHED:
        if ((error =  drdynvc_virtual_channel_event_attached(drdynvc)))
            WLog_Print(drdynvc->log, WLOG_ERROR,
                       "drdynvc_virtual_channel_event_attached failed with error %"PRIu32"", error);
```

- 这是*_connected(...)内部前面这些基本都一样, 绑定*_open_event()回调用来接受服务器数据以后做分发处理。

```
status = drdynvc->channelEntryPoints.pVirtualChannelOpenEx(drdynvc->InitHandle,
         &drdynvc->OpenHandle, drdynvc->channelDef.name, drdynvc_virtual_channel_open_event_ex);

if (status != CHANNEL_RC_OK)
{
    WLog_Print(drdynvc->log, WLOG_ERROR, "pVirtualChannelOpen failed with %s [%08"PRIX32"]",
               WTSErrorToString(status), status);
    return status;
}

drdynvc->queue = MessageQueue_New(NULL);

if (!drdynvc->queue)
{
    error = CHANNEL_RC_NO_MEMORY;
    WLog_Print(drdynvc->log, WLOG_ERROR, "MessageQueue_New failed!");
    goto error;
}
```

- 这里就有所不同了。这里重要的建立了一个channel_mgr也就是动态虚拟通道管理对象, 用来管理子通道。

```
drdynvc->queue->object.fnObjectFree = drdynvc_queue_object_free;
drdynvc->channel_mgr = dvcman_new(drdynvc);

if (!drdynvc->channel_mgr)
{
    error = CHANNEL_RC_NO_MEMORY;
    WLog_Print(drdynvc->log, WLOG_ERROR, "dvcman_new failed!");
    goto error;
}

settings = (rdpSettings*) drdynvc->channelEntryPoints.pExtendedData;

for (index = 0; index < settings->DynamicChannelCount; index++)
{
    args = settings->DynamicChannelArray[index];
    error = dvcman_load_addin(drdynvc, drdynvc->channel_mgr, args, settings);

    if (CHANNEL_RC_OK != error)
        goto error;
}

if ((error = dvcman_init(drdynvc, drdynvc->channel_mgr)))
{
    WLog_Print(drdynvc->log, WLOG_ERROR, "dvcman_init failed with error %"PRIu32"!", error);
    goto error;
}
```

- dvcman_load_addin主要做的是, 要将通道对象添加进channel_mgr需要用到的回调。例如: dvcman_register_plugin就是将新的虚拟通道加入到channel_mgr中, dvcman_get_plugin就是查看channel_mgr中是否包含带加入的虚拟通道。dvcman_get_plugin_data这个函数就是获得通道相关的设置的值, 即/gfs: AVC420等, dvcman_get_rdp_settings当然就是获得setting信息了。最后会调用改自虚拟动态通道的入口函数pDVCPluginEntry(...)。

```
static UINT dvcman_load_addin(drdynvcPlugin* drdynvc,
                              IWTSVirtualChannelManager* pChannelMgr,
                              ADDIN_ARGV* args,
                              rdpSettings* settings)
{
    DVCMAN_ENTRY_POINTS entryPoints;
    PDVC_PLUGIN_ENTRY pDVCPluginEntry = NULL;
    WLog_Print(drdynvc->log, WLOG_INFO, "Loading Dynamic Virtual Channel %s", args->argv[0]);
    pDVCPluginEntry = (PDVC_PLUGIN_ENTRY) freerdp_load_channel_addin_entry(
                          args->argv[0],
                          NULL, NULL, FREERDP_ADDIN_CHANNEL_DYNAMIC);

    if (pDVCPluginEntry)
    {
        entryPoints.iface.RegisterPlugin = dvcman_register_plugin;
        entryPoints.iface.GetPlugin = dvcman_get_plugin;
        entryPoints.iface.GetPluginData = dvcman_get_plugin_data;
        entryPoints.iface.GetRdpSettings = dvcman_get_rdp_settings;
        entryPoints.dvcman = (DVCMAN*) pChannelMgr;
        entryPoints.args = args;
        entryPoints.settings = settings;
        return pDVCPluginEntry((IDRDYNVC_ENTRY_POINTS*) &entryPoints);
    }

    return ERROR_INVALID_FUNCTION;
}
```

- DVCPluginEntry(...)虚拟动态通道人口函数。首先通过GetPlugin检查channel_mng中是否加入过改通道, 如果没有则创建一个新的虚拟动态通道对象。

```
UINT DVCPluginEntry(IDRDYNVC_ENTRY_POINTS* pEntryPoints)
{
    UINT error = CHANNEL_RC_OK;
    RDPGFX_PLUGIN* gfx;
    RdpgfxClientContext* context;
    gfx = (RDPGFX_PLUGIN*) pEntryPoints->GetPlugin(pEntryPoints, "rdpgfx");

    if (!gfx)
    {
        gfx = (RDPGFX_PLUGIN*) calloc(1, sizeof(RDPGFX_PLUGIN));

        if (!gfx)
        {
            WLog_ERR(TAG, "calloc failed!");
            return CHANNEL_RC_NO_MEMORY;
        }
```

- 为生命周期回调赋值, init ternimated等, init函数在dvcman_init中调用, 此函数会循环调用所有的动态虚拟子通道的init函数。这个init目前会创建一个lisener, 每一个channel都会创建一个lisener貌似, 然后会绑定一个rdpgfx_on_new_channel_connection函数。

```
gfx->settings = (rdpSettings*) pEntryPoints->GetRdpSettings(pEntryPoints);
gfx->iface.Initialize = rdpgfx_plugin_initialize;
gfx->iface.Connected = NULL;
gfx->iface.Disconnected = NULL;
gfx->iface.Terminated = rdpgfx_plugin_terminated;
gfx->rdpcontext = ((freerdp*)gfx->settings->instance)->context;
gfx->SurfaceTable = HashTable_New(TRUE);
```

- rdpgfx_on_new_channel_connection在收到服务器开启改虚拟通道消息的时候回调, 之后会绑定datareceived等回调。接受并处理属于这个通道的数据的逻辑主要就在这里面。

```
    callback->iface.OnDataReceived = rdpgfx_on_data_received;
    callback->iface.OnOpen = rdpgfx_on_open;
    callback->iface.OnClose = rdpgfx_on_close;
    callback->plugin = listener_callback->plugin;
    callback->channel_mgr = listener_callback->channel_mgr;
    callback->channel = pChannel;
    listener_callback->channel_callback = callback;
    *ppCallback = (IWTSVirtualChannelCallback*) callback;
```

- 设置动态虚拟通道所需要的一些必要参数, 这个不同的动态虚拟子通道会不同

```
gfx->ThinClient = gfx->settings->GfxThinClient;
gfx->SmallCache = gfx->settings->GfxSmallCache;
gfx->Progressive = gfx->settings->GfxProgressive;
gfx->ProgressiveV2 = gfx->settings->GfxProgressiveV2;
gfx->H264 = gfx->settings->GfxH264;
gfx->AVC444 = gfx->settings->GfxAVC444;
gfx->SendQoeAck = gfx->settings->GfxSendQoeAck;
gfx->capsFilter = gfx->settings->GfxCapsFilter;
```

- 绑定一些必要的回调

```
context->handle = (void*) gfx;
context->GetSurfaceIds = rdpgfx_get_surface_ids;
context->SetSurfaceData = rdpgfx_set_surface_data;
context->GetSurfaceData = rdpgfx_get_surface_data;
context->SetCacheSlotData = rdpgfx_set_cache_slot_data;
context->GetCacheSlotData = rdpgfx_get_cache_slot_data;
context->CapsAdvertise = rdpgfx_send_caps_advertise_pdu;
context->FrameAcknowledge = rdpgfx_send_frame_acknowledge_pdu;
gfx->iface.pInterface = (void*) context;
gfx->zgfx = zgfx_context_new(FALSE);
```

- 最后会将自己注册进channels_mng

```
error = pEntryPoints->RegisterPlugin(pEntryPoints, "rdpgfx", (IWTSPlugin*) gfx);
```

13. 这些虚拟通道加紧channel_mng以后要做什么呢? 之前提到过, 虚拟通道在加载的过程中会创建一个专有的线程负责处理属于自己这个通道的消息。

```
static DWORD WINAPI drdynvc_virtual_channel_client_thread(LPVOID arg)
{
    wStream* data;
    wMessage message;
    UINT error = CHANNEL_RC_OK;
    drdynvcPlugin* drdynvc = (drdynvcPlugin*) arg;

    if (!drdynvc)
    {
        ExitThread((DWORD) CHANNEL_RC_BAD_CHANNEL_HANDLE);
        return CHANNEL_RC_BAD_CHANNEL_HANDLE;
    }

    while (1)
    {
        if (!MessageQueue_Wait(drdynvc->queue))
        {
            WLog_Print(drdynvc->log, WLOG_ERROR, "MessageQueue_Wait failed!");
            error = ERROR_INTERNAL_ERROR;
            break;
        }

        if (!MessageQueue_Peek(drdynvc->queue, &message, TRUE))
        {
            WLog_Print(drdynvc->log, WLOG_ERROR, "MessageQueue_Peek failed!");
```

- 然后会调用一个自己的消息分发函数

```
if (message.id == 0)
{
    data = (wStream*) message.wParam;

    if ((error = drdynvc_order_recv(drdynvc, data)))
    {
        Stream_Free(data, TRUE);
        WLog_Print(drdynvc->log, WLOG_ERROR, "drdynvc_order_recv failed with error %"
        break;
    }

    Stream_Free(data, TRUE);
}
```

- 然后众多消息中, 有一条消息是要打开新的虚拟通道的。

```
case CREATE_REQUEST_PDU:
    return drdynvc_process_create_request(drdynvc, Sp, cbChId, s);
```

- 这里会做三件是, 1. 创建虚拟通道dvcman_create_channel。2.通知服务器, create完毕。3. dvcman_open_channel

```
            ChannelId, name);
channel_status = dvcman_create_channel(drdynvc, drdynvc->channel_mgr, ChannelId, name);
data_out = Stream_New(NULL, pos + 4);

if (!data_out)
{
    WLog_Print(drdynvc->log, WLOG_ERROR, "Stream_New failed!");
    return CHANNEL_RC_NO_MEMORY;
}

Stream_Write_UINT8(data_out, (CREATE_REQUEST_PDU << 4) | cbChId);
Stream_SetPosition(s, 1);
Stream_Copy(s, data_out, pos - 1);

if (channel_status == CHANNEL_RC_OK)
{
    WLog_Print(drdynvc->log, WLOG_DEBUG, "channel created");
    Stream_Write_UINT32(data_out, 0);
}
else
{
    WLog_Print(drdynvc->log, WLOG_DEBUG, "no listener");
    Stream_Write_UINT32(data_out, (UINT32)0xC0000001); /* same code used by mstsc */
}

status = drdynvc_send(drdynvc, data_out);
```

- dvcman_create_channel主要就是创建一个channels对象,然后赋予名字 ID 还有channel_mng

```
    }

    channel->dvcman = (DVCMAN*) pChannelMgr;
    channel->channel_id = ChannelId;
    channel->channel_name = _strdup(ChannelName);
```

- 将新创建的channel对象添加进channel_mng

```
channel->status = ERROR_NOT_CONNECTED;
ArrayList_Add(dvcman->channels, channel);
```

- 调用listenner的OnNewChannelConnection, 创建channel的callback, 然后将callback添加到channel中, 目前也就这一个功能。这些回调主要是用来接受channel数据的, 还有open和close看了一下, 也不太清楚里面的逻辑。

```
if ((error = listener->listener_callback->OnNewChannelConnection(
                listener->listener_callback,
                (IWTSVirtualChannel*) channel, NULL, &bAccept, &pCallback)) == CHANNEL_RC_OK
    && bAccept)
{
    WLog_Print(drdynvc->log, WLOG_DEBUG, "listener %s created new channel %"PRIu32"",
               listener->channel_name, channel->channel_id);
    channel->status = CHANNEL_RC_OK;
    channel->channel_callback = pCallback;
    channel->pInterface = listener->iface.pInterface;
    context = dvcman->drdynvc->context;
```

- 调用OnChannelConnected,

```
    context = dvcman->drdynvc->context;
    IFCALLRET(context->OnChannelConnected, error, context, ChannelName,
              listener->iface.pInterface);
```

- 这个回调在freerdp_channels_post_connect中调用。

```
if (channels->drdynvc)
{
    channels->drdynvc->custom = (void*) channels;
    channels->drdynvc->OnChannelConnected = freerdp_drdynvc_on_channel_connected;
    channels->drdynvc->OnChannelDisconnected =
        freerdp_drdynvc_on_channel_disconnected;
    channels->drdynvc->OnChannelAttached = freerdp_drdynvc_on_channel_attached;
    channels->drdynvc->OnChannelDetached = freerdp_drdynvc_on_channel_detached;
}
```

- freerdp_drdynvc_on_channel_connected主要调用了pubsub, 通知wf_client中绑定的回调

```
PubSub_SubscribeChannelConnected(instance->context->pubSub,
                                 wf_OnChannelConnectedEventHandler);
PubSub_SubscribeChannelDisconnected(instance->context->pubSub,
                                    wf_OnChannelDisconnectedEventHandler);
return TRUE;
```

- 通知服务器的部分很简单再次不做叙述, 通知服务器的原因就是告诉服务器, 我这边处理消息的回调已经绑定完成, 你可以发送属于我的消息了。

dvcman_open_channel内部其实就是调用了channel_callback的onopen, 就是设置一些flags然后给服务器发送一条消息。

```
if (channel->status == CHANNEL_RC_OK)
IMgr, UINT32 ChannelId)
    pCallback = channel->channel_callback;

    if ((pCallback->OnOpen) && (error = pCallback->OnOpen(pCallback)))
    {
        WLog_Print(drdynvc->log, WLOG_ERROR, "OnOpen failed with error %"PRIu32"!", error);
        return error;
    }

    WLog_Print(drdynvc->log, WLOG_DEBUG, "open_channel: ChannelId %"PRIu32"", ChannelId);
}
```

14. channel的消息发送流程。通道的代码文件中可能会写一些发送通道消息的接口, 例如动态虚拟通道drdynvc, 就有一个drdynvc_send接口用来发送消息。其内部调用pVirtualChannelWriteEx发送消息, 这个接口是在*_load_ex中同*_init_ex一起绑定的。

```
static UINT drdynvc_send(drdynvcPlugin* drdynvc, wStream* s)
{
    UINT status;

    if (!drdynvc)
        status = CHANNEL_RC_BAD_CHANNEL_HANDLE;
    else
    {
        status = drdynvc->channelEntryPoints.pVirtualChannelWriteEx(drdynvc->InitHandle,
                 drdynvc->OpenHandle, Stream_Buffer(s), (UINT32) Stream_GetPosition(s), s);
    }
```

- 然后看一下绑定的地方

```
EntryPointsEx.protocolVersion = VIRTUAL_CHANNEL_VERSION_WIN2000;
EntryPointsEx.pVirtualChannelInitEx = FreeRDP_VirtualChannelInitEx;
EntryPointsEx.pVirtualChannelOpenEx = FreeRDP_VirtualChannelOpenEx;
EntryPointsEx.pVirtualChannelCloseEx = FreeRDP_VirtualChannelCloseEx;
EntryPointsEx.pVirtualChannelWriteEx = FreeRDP_VirtualChannelWriteEx;
EntryPointsEx.MagicNumber = FREERDP_CHANNEL_MAGIC_NUMBER;
```

- FreeRDP_VirtualChannelWriteEx这个函数逻辑不复杂, 主要创建了消息对象, 然后通过dispach函数放到channel的queue中等待发送。

```
message.context = channels;
message.id = 0;
message.wParam = pChannelOpenEvent;
message.lParam = NULL;
message.Free = channel_queue_message_free;

if (!MessageQueue_Dispatch(channels->queue, &message))
{
    channel_queue_message_free(&message);
    return CHANNEL_RC_NO_MEMORY;
}
```

- 加入到channel消息队列中的消息是如何发送出去的呢? 在主线程中, freerdp_check_event_handles函数内部, 这个函数除了检测收到的消息

freerdp_check_fds还负责处理待发送的消息freerdp_channels_check_fds。这些都在主线程的while循环中做到的。

```
BOOL status;
status = freerdp_check_fds(context->instance);

if (!status)
{
    if (freerdp_get_last_error(context) == FREERDP_ERROR_SUCCESS)
        WLog_ERR(TAG, "freerdp_check_fds() failed - %"PRIi32"", status);

    return FALSE;
}

status = freerdp_channels_check_fds(context->channels, context->instance);
```

- freerdp_channels_check_fds内部很简单, 等待消息时间, 如果有消息就调用freerdp_channels_process_sync处理

```
BOOL freerdp_channels_check_fds(rdpChannels* channels, freerdp* instance)
{
    if (WaitForSingleObject(MessageQueue_Event(channels->queue),
                            0) == WAIT_OBJECT_0)
    {
        freerdp_channels_process_sync(channels, instance);
    }

    return TRUE;
}
```

- freerdp_channels_process_sync做了四件事, 去channels消息队列中的消息

```
while (MessageQueue_Peek(channels->queue, &message, TRUE))
{
    if (message.id == WMQ_QUIT)
```

- 然后获取channel信息, 并调用SendChannelData发送出去

```
if (channel)
    instance->SendChannelData(instance, channel->ChannelId, item->Data, item->DataLength);
```

- 然后调用通道绑定好的openevent回调, 这个有的回调是没有写逻辑的, 根据需要吧

```
if (pChannelOpenData->pChannelOpenEventProc)
{
    pChannelOpenData->pChannelOpenEventProc(
        pChannelOpenData->OpenHandle, CHANNEL_EVENT_WRITE_COMPLETE, item->UserData, item->D
        item->DataLength, 0);
}
else if (pChannelOpenData->pChannelOpenEventProcEx)
{
    pChannelOpenData->pChannelOpenEventProcEx(pChannelOpenData->lpUserParam,
            pChannelOpenData->OpenHandle, CHANNEL_EVENT_WRITE_COMPLETE, item->UserData, ite
            item->DataLength, 0);
}
```

- 最后释放掉消息的资源

```
IFCALL(message.Free, &message);
```

15. SendChannelData是在哪里绑定的呢? 这里一共绑定了两个回调, 一个是收消息的一个是发送消息的。

```
freerdp* freerdp_new()
{
    freerdp* instance;
    instance = (freerdp*) calloc(1, sizeof(freerdp));

    if (!instance)
        return NULL;

    instance->ContextSize = sizeof(rdpContext);
    instance->SendChannelData = freerdp_send_channel_data;
    instance->ReceiveChannelData = freerdp_channels_data;
    return instance;
}
```

16. channel收消息的逻辑。主线程的thile循环中调用freerdp_check_event_handles

```
{
    if (!freerdp_check_event_handles(context))
    {
        if (client_auto_reconnect(instance))
            continue;

        WLog_ERR(TAG, "Failed to check FreeRDP file descriptor");
        break;
    }
}
```

- 调用freerdp_check_fds

```
    BOOL status;
    status = freerdp_check_fds(context->instance);

    if (!status)
    {
        if (freerdp_get_last_error(context) == FREERDP_ERROR_SUCCESS)
            WLog_ERR(TAG, "freerdp_check_fds() failed - %"PRIi32"", status);

        return FALSE;
    }

    status = freerdp_channels_check_fds(context->channels, context->instance);
```

- 就一个主要函数, rdp_check_fds, 其他都是检测进入条件和推出条件

```
BOOL freerdp_check_fds(freerdp* instance)
{
    int status;
    rdpRdp* rdp;

    if (!instance)
        return FALSE;

    if (!instance->context)
        return FALSE;

    if (!instance->context->rdp)
        return FALSE;

    rdp = instance->context->rdp;
    status = rdp_check_fds(rdp);

    if (status < 0)
    {
        TerminateEventArgs e;
        rdpContext* context = instance->context;
        WLog_DBG(TAG, "rdp_check_fds() - %i", status);
        EventArgsInit(&e, "freerdp");
        e.code = 0;
        PubSub_OnTerminate(context->pubSub, context, &e);
        return FALSE;
    }
```

- transport_check_fds是最主要的获取数据的接口

```
        if (tsg_get_state(tsg) != TSG_STATE_PIPE_CREATED)
            return 1;
    }

    status = transport_check_fds(transport);

    if (status == 1)
    {
        status = rdp_client_redirect(rdp); /* session redirection */
    }
```

- 在while循环中取数据, 通过调用transport_read_pdu

```
    return -1;

status = transport_read_layer_bytes(transport, s, pduLength - Stream_GetPosition(s));

if (status != 1)
    return status;
```

- 前面有乱七八糟的数据大小什么的检测, 不管他, 最主要的是transport_read_layer_bytes

```
while (now < dueDate)
{
    if (freerdp_shall_disconnect(transport->context->instance))
    {
        return -1;
    }

    /**
     * Note: transport_read_pdu tries to read one PDU from
     * the transport layer.
     * The ReceiveBuffer might have a position > 0 in case of a non blocking
     * transport. If transport_read_pdu returns 0 the pdu couldn't be read at
     * this point.
     * Note that transport->ReceiveBuffer is replaced after each iteration
     * of this loop with a fresh stream instance from a pool.
     */
    if ((status = transport_read_pdu(transport, transport->ReceiveBuffer)) <= 0)
    {
        if (status < 0)
            WLog_Print(transport->log, WLOG_DEBUG, "transport_check_fds: transport_read_pdu

        return status;
    }

    received = transport->ReceiveBuffer;
```

- 然后内部调用transport_read_layer

```
static SSIZE_T transport_read_layer_bytes(rdpTransport* transport, wStream* s,
                                          size_t toRead)
{
    SSIZE_T status;
    if (toRead > SSIZE_MAX)
        return 0;

    status = transport_read_layer(transport, Stream_Pointer(s), toRead);

    if (status <= 0)
        return status;

    Stream_Seek(s, (size_t)status);
    return status == (SSIZE_T)toRead ? 1 : 0;
}
```

- 其他的不管值看这个是最主要的从socket去数据的接口了, 取出来的数据放在哪里了呢? 要往回倒一下

```
while (read < (SSIZE_T)bytes)
{
    const SSIZE_T tr = (SSIZE_T)bytes - read;
    int r = (int)((tr > INT_MAX) ? INT_MAX : tr);
    int status = status = BIO_read(transport->frontBio, data + read, r);

    if (status <= 0)
    {
```

- 上面有一个函数transport_check_fds. 里面调用了transport_read_pdu, 将数据存放到transport->ReceiveBuffer中, 经过一系列的处理, 再调用ReceiveCallback

```
received = transport->ReceiveBuffer;

if (!(transport->ReceiveBuffer = StreamPool_Take(transport->ReceivePool, 0)))
    return -1;

/**
 * status:
 *  -1: error
 *   0: success
 *   1: redirection
 */
recv_status = transport->ReceiveCallback(transport, received, transport->ReceiveExtra);
Stream_Release(received);
```

- 这个ReceiveCallback是在rdp_client_connect中绑定的, 这个函数就是通过nego链接rdp服务器的主要逻辑函数。之前有介绍过, 在rdp_client_thread中调用。客户端收到的所有消息都会触发这个回调, 然后在这里面做分发处理。这里面包含链接消息处理, 通道消息处理, 用户名密码消息处理等, 还有就是画面呈现消息的处理了。

```
/* everything beyond this point is event-driven and non blocking
rdp->transport->ReceiveCallback = rdp_recv_callback;
rdp->transport->ReceiveExtra = rdp;
transport_set_blocking_mode(rdp->transport, FALSE);

if (rdp->state != CONNECTION_STATE_NLA)
```

- 最主要的一个函数就是rdp_recv_pdu, 通道数据和成像数据都会经过这个函数。

```
    break;

case CONNECTION_STATE_ACTIVE:
    status = rdp_recv_pdu(rdp, s);

    if (status < 0)
```

- 我们发现这里面有两个不同的分发路线, 一个是rdp_recv_tpkt_pdu通道数据走的就是这个, 另一个是rdp_recv_fastpath_pdu桌面位图数据走的就是这个。我们这里看通道数据的这个分支。

```
static int rdp_recv_pdu(rdpRdp* rdp, wStream* s)
{
    if (tpkt_verify_header(s))
        return rdp_recv_tpkt_pdu(rdp, s);
    else
        return rdp_recv_fastpath_pdu(rdp, s);
}
```

- 通道数据分支, 进来以后发现有两个主要的分支, global_channel和其他

```
if (channelId == MCS_GLOBAL_CHANNEL_ID) { ... }
else if (rdp->mcs->messageChannelId && channelId == rdp->mcs->messageChannelId)
{
    if (!rdp->settings->UseRdpSecurityLayer)
        if (!rdp_read_security_header(s, &securityFlags, NULL))
            return -1;

    return rdp_recv_message_channel_pdu(rdp, s, securityFlags);
}
else
{
    if (!freerdp_channel_process(rdp->instance, s, channelId))
    {
        WLog_ERR(TAG, "rdp_recv_tpkt_pdu: freerdp_channel_process() fail");
        return -1;
    }
}
```

- 根据调试过程发现, 我们最近研究的参数配置, 会走非globle_channel分支

```
BOOL freerdp_channel_process(freerdp* instance, wStream* s, UINT16 channelId)
{
    UINT32 length;
    UINT32 flags;
    int chunkLength;

    if (Stream_GetRemainingLength(s) < 8)
        return FALSE;

    Stream_Read_UINT32(s, length);
    Stream_Read_UINT32(s, flags);
    chunkLength = Stream_GetRemainingLength(s);
    IFCALL(instance->ReceiveChannelData, instance,
           channelId, Stream_Pointer(s), chunkLength, flags, length);
    return TRUE;
}
```

- 内部分简单, 主要调用了ReceiveChannelData, 这个回调前面讲过, 实在FreeRdp_new中绑定的, 这个函数中一共绑定了两个回调, 另一个是sendchanneldata回调,

```
freerdp* freerdp_new()
{
    freerdp* instance;
    instance = (freerdp*) calloc(1, sizeof(freerdp));

    if (!instance)
        return NULL;

    instance->ContextSize = sizeof(rdpContext);
    instance->SendChannelData = freerdp_send_channel_data;
    instance->ReceiveChannelData = freerdp_channels_data;
    return instance;
}
```

- 看一下freerdp_channels_data回调, 逻辑很简单, 显示通过传进来的channelId在所有channels中寻找到对应的channel对象, 然后再通过channlename寻找pChannelOpenData, 最后调用pChannelOpenData->pChannelOpenEventProc或者pChannelOpenData->pChannelOpenEventProcEx这两个函数实在对应通道的*_connected函数中绑定的, 主要就是将消息放到自己通道的消息队列中

```
for (index = 0; index < mcs->channelCount; index++)
{
    if (mcs->channels[index].ChannelId == channelId)
    {
        channel = &mcs->channels[index];
        break;
    }
}

if (!channel)
{
    return 1;
}

pChannelOpenData = freerdp_channels_find_channel_open_data_by_name(channels, channel->Name);

if (!pChannelOpenData)
{
    return 1;
}

if (pChannelOpenData->pChannelOpenEventProc)
{
    pChannelOpenData->pChannelOpenEventProc(
        pChannelOpenData->OpenHandle, CHANNEL_EVENT_DATA_RECEIVED, data, dataSize, totalSize, flags);
}
else if (pChannelOpenData->pChannelOpenEventProcEx)
{
    pChannelOpenData->pChannelOpenEventProcEx(pChannelOpenData->lpUserParam,
            pChannelOpenData->OpenHandle, CHANNEL_EVENT_DATA_RECEIVED, data, dataSize, totalSize, flags
}
```

- 那drdynvc通道距离, 这个回调就是drdynvc_virtual_channel_open_event_ex, 主要逻辑全在这里了。

```
static void VCAPITYPE drdynvc_virtual_channel_open_event_ex(LPVOID lpUserParam, DWORD openHandle,
        UINT event, LPVOID pData, UINT32 dataLength, UINT32 totalLength, UINT32 dataFlags)
{
    UINT error = CHANNEL_RC_OK;
    drdynvcPlugin* drdynvc = (drdynvcPlugin*) lpUserParam;

    if (!drdynvc || (drdynvc->OpenHandle != openHandle))
    {
        WLog_ERR(TAG, "drdynvc_virtual_channel_open_event: error no match");
        return;
    }

    switch (event)
    {
        case CHANNEL_EVENT_DATA_RECEIVED:
            if ((error = drdynvc_virtual_channel_event_data_received(drdynvc, pData, dataLength, totalLeng
                        dataFlags)))
                WLog_Print(drdynvc->log, WLOG_ERROR,
                           "drdynvc_virtual_channel_event_data_received failed with error %"PRIu32"", erro

            break;

        case CHANNEL_EVENT_WRITE_COMPLETE:
            break;

        case CHANNEL_EVENT_USER:
            break;
    }
```

- 重点是这个drdynvc_virtual_channel_event_data_received, 内部逻辑很多, 其实大多数实在处理消息, 相当于反序列化, 然后调用最主要的函数MessageQueue_Post将反序列化好的消息放入到通道对应的queue中

```
if (dataFlags & CHANNEL_FLAG_LAST)
{
    if (Stream_Capacity(data_in) != Stream_GetPosition(data_in))
    {
        WLog_Print(drdynvc->log, WLOG_ERROR, "drdynvc_plugin_process_received: read error
        return ERROR_INVALID_DATA;
    }

    drdynvc->data_in = NULL;
    Stream_SealLength(data_in);
    Stream_SetPosition(data_in, 0);

    if (!MessageQueue_Post(drdynvc->queue, NULL, 0, (void*) data_in, NULL))
    {
        WLog_Print(drdynvc->log, WLOG_ERROR, "MessageQueue_Post failed!");
        return ERROR_INTERNAL_ERROR;
    }
}
```

- 到这里这个接受消息的线程以及处理逻辑全部完成了, 剩下的一部分是消息处理逻辑, 这部分逻辑再另一个线程, 就是通道专属的线程。那drdynvc来说就是drdynvc_virtual_channel_client_thread线程。这个线程里面等待消息队列中的消息, 然后通过peek取出消息。

```
static DWORD WINAPI drdynvc_virtual_channel_client_thread(LPVOID arg)
{
    wStream* data;
    wMessage message;
    UINT error = CHANNEL_RC_OK;
    drdynvcPlugin* drdynvc = (drdynvcPlugin*) arg;

    if (!drdynvc)
    {
        ExitThread((DWORD) CHANNEL_RC_BAD_CHANNEL_HANDLE);
        return CHANNEL_RC_BAD_CHANNEL_HANDLE;
    }

    while (1)
    {
        if (!MessageQueue_Wait(drdynvc->queue))
        {
            WLog_Print(drdynvc->log, WLOG_ERROR, "MessageQueue_Wait failed!");
            error = ERROR_INTERNAL_ERROR;
            break;
        }

        if (!MessageQueue_Peek(drdynvc->queue, &message, TRUE))
        {
            WLog_Print(drdynvc->log, WLOG_ERROR, "MessageQueue_Peek failed!");
```

- 最后同通过drdynvc_order_recv进行最终的分发

```
if (message.id == 0)
{
    data = (wStream*) message.wParam;

    if ((error = drdynvc_order_recv(drdynvc, data)))
    {
        Stream_Free(data, TRUE);
        WLog_Print(drdynvc->log, WLOG_ERROR, "drdynvc_order_recv failed with error %"PRIu32"
        break;
    }

    Stream_Free(data, TRUE);
}
```

- 这里是drdynvc的所有分发逻辑目前是这样，其中*_request是创建子动态虚拟通道消息。*_process_data是处理图像逻辑。找这里消息分发处理逻辑就介绍完了，需要看其他文档。

```c
switch (Cmd)
{
    case CAPABILITY_REQUEST_PDU:
        return drdynvc_process_capability_request(drdynvc, Sp, cbChId, s);

    case CREATE_REQUEST_PDU:
        return drdynvc_process_create_request(drdynvc, Sp, cbChId, s);

    case DATA_FIRST_PDU:
        return drdynvc_process_data_first(drdynvc, Sp, cbChId, s);

    case DATA_PDU:
        return drdynvc_process_data(drdynvc, Sp, cbChId, s);

    case CLOSE_REQUEST_PDU:
        return drdynvc_process_close_request(drdynvc, Sp, cbChId, s);

    default:
        WLog_Print(drdynvc->log, WLOG_ERROR, "unknown drdynvc cmd 0x%x", Cmd);
        return ERROR_INTERNAL_ERROR;
```