

在上一篇文章《内核封装WSK网络通信接口》中，LyShark 已经带大家看过了如何通过WSK接口实现套接字通信，但 WSK 实现的通信是内核与内核模块之间的，而如果需要内核与应用层之间通信则使用 TDI 会更好一些因为它更接近应用层，本章将使用 TDI 实现，TDI 全称传输驱动接口，其主要负责连接 socket 和协议驱动，用于实现访问传输层的功能，该接口比 NDIS 更接近于应用层，在早期 Win 系统中常用于实现过滤防火墙。

TDI (Transport Driver Interface) 是 Windows 内核中的一个网络通信接口，提供了一种统一的方式，用于在传输层上实现网络通信。在 TDI 接口中，应用程序可以使用一系列的 API 函数，来创建并控制网络连接、发送和接收数据等操作。

经过封装后也可实现通信功能，本章将运用 TDI 接口实现驱动与应用层之间传输字符串，结构体，多线程收发等技术。

- TDI 传输字符串
- TDI 多线程收发
- TDI 传数结构实现认证

我们将如下通用 SDK 库拷贝下来，并命名为 `MyTDI.hpp` 放入到代码同级目录下。

```
#include <ntifs.h>
#include <tdikrn1.h>
#include <ntstatus.h>

// 定义 TCP 驱动设备名称
#define COMM_TCP_DEV_NAME L"\Device\Tcp"

// 定义地址转换的宏
#define INETADDR(a, b, c, d) (a + (b<<8) + (c<<16) + (d<<24))
#define HTONL(a) (((a & 0xFF)<<24) + ((a & 0xFF00)<<8) + ((a & 0xFF0000)>>8) +
(a&0xFF000000)>>24)
#define HTONS(a) (((a & 0xFF)<<8) + ((a & 0xFF00)>>8))

// 完成回调函数
NTSTATUS TdiCompletionRoutine(PDEVICE_OBJECT pDevObj, PIRP pIrp, PVOID pContext)
{
    if (NULL != pContext)
    {
        KeSetEvent((PKEVENT)pContext, IO_NO_INCREMENT, FALSE);
    }
    return STATUS_MORE_PROCESSING_REQUIRED;
}

// 初始化设置
NTSTATUS Tdiopen(PDEVICE_OBJECT *ppTdiAddressDevObj, PFILE_OBJECT *ppTdiEndPointFileObject,
HANDLE *phTdiAddress, HANDLE *phTdiEndPoint)
{
    NTSTATUS status = STATUS_UNSUCCESSFUL;
    PFILE_FULL_EA_INFORMATION pAddressEaBuffer = NULL;
    ULONG ulAddressEaBufferLength = 0;
    PTA_IP_ADDRESS pTaIpAddr = NULL;
    UNICODE_STRING ustrTDIDevName;
    OBJECT_ATTRIBUTES ObjectAttributes = { 0 };
    IO_STATUS_BLOCK iobs = { 0 };
    HANDLE hTdiAddress = NULL;
```

```

PFILE_OBJECT pTdiAddressFileObject = NULL;
PDEVICE_OBJECT pTdiAddressDevObj = NULL;
PFILE_FULL_EA_INFORMATION pContextEaBuffer = NULL;
ULONG ulContextEaBufferLength = 0;
HANDLE hTdiEndPoint = NULL;
PFILE_OBJECT pTdiEndPointFileObject = NULL;
KEVENT irpCompleteEvent = { 0 };
PIRP pIrp = NULL;

do
{
    ulAddressEaBufferLength = sizeof(FILE_FULL_EA_INFORMATION) +
TDI_TRANSPORT_ADDRESS_LENGTH + sizeof(TA_IP_ADDRESS);
    pAddressEaBuffer = (PFILE_FULL_EA_INFORMATION)ExAllocatePool(NonPagedPool,
ulAddressEaBufferLength);
    if (NULL == pAddressEaBuffer)
    {
        break;
    }
    RtlZeroMemory(pAddressEaBuffer, ulAddressEaBufferLength);
    RtlCopyMemory(pAddressEaBuffer->EaName, TdiTransportAddress, (1 +
TDI_TRANSPORT_ADDRESS_LENGTH));
    pAddressEaBuffer->EaNameLength = TDI_TRANSPORT_ADDRESS_LENGTH;
    pAddressEaBuffer->EaValueLength = sizeof(TA_IP_ADDRESS);

    // 初始化IP地址与端口
    pTaIpAddr = (PTA_IP_ADDRESS)((PUCHAR)pAddressEaBuffer->EaName + pAddressEaBuffer-
>EaNameLength + 1);
    pTaIpAddr->TAAddressCount = 1;
    pTaIpAddr->Address[0].AddressLength = TDI_ADDRESS_LENGTH_IP;
    pTaIpAddr->Address[0].AddressType = TDI_ADDRESS_TYPE_IP;
    pTaIpAddr->Address[0].Address[0].sin_port = 0;      // 0表示本机任意随机端口
    pTaIpAddr->Address[0].Address[0].in_addr = 0;       // 0表示本机本地IP地址
    RtlZeroMemory(pTaIpAddr->Address[0].Address[0].sin_zero, sizeof(pTaIpAddr-
>Address[0].Address[0].sin_zero));

    RtlInitUnicodeString(&ustrTDIDevName, COMM_TCP_DEV_NAME);
    InitializeObjectAttributes(&ObjectAttributes, &ustrTDIDevName, OBJ_CASE_INSENSITIVE |
OBJ_KERNEL_HANDLE, NULL, NULL);

    status = ZwCreateFile(&hTdiAddress, GENERIC_READ | GENERIC_WRITE | SYNCHRONIZE,
    &ObjectAttributes, &iobs, NULL, FILE_ATTRIBUTE_NORMAL,
    FILE_SHARE_READ, FILE_OPEN, 0, pAddressEaBuffer, ulAddressEaBufferLength);
    if (!NT_SUCCESS(status))
    {
        break;
    }

    status = ObReferenceObjectByHandle(hTdiAddress,
    FILE_ANY_ACCESS, 0, KernelMode, &pTdiAddressFileobject, NULL);
    if (!NT_SUCCESS(status))
    {
        break;
    }
}

```

```

}

pTdiAddressDevObj = IoGetRelatedDeviceObject(pTdiAddressFileObject);
if (NULL == pTdiAddressDevObj)
{
    break;
}

ulContextEaBufferLength = FIELD_OFFSET(FILE_FULL_EA_INFORMATION, EaName) +
TDI_CONNECTION_CONTEXT_LENGTH + 1 + sizeof(CONNECTION_CONTEXT);
pContextEaBuffer = (PFILE_FULL_EA_INFORMATION)ExAllocatePool(NonPagedPool,
ulContextEaBufferLength);
if (NULL == pContextEaBuffer)
{
    break;
}
RtlZeroMemory(pContextEaBuffer, ulContextEaBufferLength);
RtlCopyMemory(pContextEaBuffer->EaName, TdiConnectionContext, (1 +
TDI_CONNECTION_CONTEXT_LENGTH));
pContextEaBuffer->EaNameLength = TDI_CONNECTION_CONTEXT_LENGTH;
pContextEaBuffer->EaValueLength = sizeof(CONNECTION_CONTEXT);

status = ZwCreateFile(&hTdiEndPoint, GENERIC_READ | GENERIC_WRITE | SYNCHRONIZE,
&ObjectAttributes, &iobs, NULL, FILE_ATTRIBUTE_NORMAL, FILE_SHARE_READ,
FILE_OPEN, 0, pContextEaBuffer, ulContextEaBufferLength);
if (!NT_SUCCESS(status))
{
    break;
}
status = ObReferenceObjectByHandle(hTdiEndPoint,
FILE_ANY_ACCESS, 0, KernelMode, &pTdiEndPointFileObject, NULL);
if (!NT_SUCCESS(status))
{
    break;
}

KeInitializeEvent(&irpCompleteEvent, NotificationEvent, FALSE);

pIrp = TdiBuildInternalDeviceControlIrp(TDI_ASSOCIATE_ADDRESS,
pTdiAddressDevObj, pTdiEndPointFileObject, &irpCompleteEvent, &iobs);
if (NULL == pIrp)
{
    break;
}

TdiBuildAssociateAddress(pIrp, pTdiAddressDevObj, pTdiEndPointFileObject, NULL, NULL,
hTdiAddress);

IoSetCompletionRoutine(pIrp, TdiCompletionRoutine, &irpCompleteEvent, TRUE, TRUE, TRUE);

status = IoCallDriver(pTdiAddressDevObj, pIrp);
if (STATUS_PENDING == status)
{
}

```

```

        KeWaitForSingleObject(&irpCompleteEvent, Executive, KernelMode, FALSE, NULL);
    }

    *ppTdiAddressDevObj = pTdiAddressDevObj;
    *ppTdiEndPointFileObject = pTdiEndPointFileObject;
    *phTdiAddress = hTdiAddress;
    *phTdiEndPoint = hTdiEndPoint;

} while (FALSE);

if (pTdiAddressFileobject)
{
    ObDereferenceObject(pTdiAddressFileobject);
}

if (pContextEaBuffer)
{
    ExFreePool(pContextEaBuffer);
}

if (pAddressEaBuffer)
{
    ExFreePool(pAddressEaBuffer);
}

return status;
}

// TCP连接服务器
NTSTATUS Tdiconnection(PDEVICE_OBJECT pTdiAddressDevObj, PFILE_OBJECT
pTdiEndPointFileObject, LONG *pServerIp, LONG lServerPort)
{
    NTSTATUS status = STATUS_SUCCESS;
    IO_STATUS_BLOCK iosb = { 0 };
    PIRP pIrp = NULL;
    KEVENT connEvent = { 0 };
    TA_IP_ADDRESS serverTaIpAddr = { 0 };
    ULONG serverIpAddr = 0;
    USHORT serverPort = 0;
    TDI_CONNECTION_INFORMATION serverConnection = { 0 };

    KeInitializeEvent(&connEvent, NotificationEvent, FALSE);

    pIrp = TdiBuildInternalDeviceControlIrp(TDI_CONNECT, pTdiAddressDevObj,
pTdiEndPointFileObject, &connEvent, &iosb);
    if (NULL == pIrp)
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    serverIpAddr = INETADDR(pServerIp[0], pServerIp[1], pServerIp[2], pServerIp[3]);
    serverPort = HTONS(lServerPort);
    serverTaIpAddr.TAAddressCount = 1;
    serverTaIpAddr.Address[0].AddressLength = TDI_ADDRESS_LENGTH_IP;
    serverTaIpAddr.Address[0].AddressType = TDI_ADDRESS_TYPE_IP;
    serverTaIpAddr.Address[0].Address[0].sin_port = serverPort;
}

```

```

serverTaIpAddr.Address[0].Address[0].in_addr = serverIpAddress;
serverConnection.UserDataLength = 0;
serverConnection(userData = 0;
serverConnection.OptionsLength = 0;
serverConnection.Options = 0;
serverConnection.RemoteAddressLength = sizeof(TA_IP_ADDRESS);
serverConnection.RemoteAddress = &serverTaIpAddr;

TdiBuildConnect(pIrp, pTdiAddressDevObj, pTdiEndPointFileObject, NULL, NULL, NULL,
&serverConnection, 0);

IoSetCompletionRoutine(pIrp, TdiCompletionRoutine, &connEvent, TRUE, TRUE, TRUE);

status = IoCallDriver(pTdiAddressDevObj, pIrp);
if (STATUS_PENDING == status)
{
    KeWaitForSingleObject(&connEvent, Executive, KernelMode, FALSE, NULL);
}
return status;
}

// TCP发送信息
NTSTATUS Tdisend(PDEVICE_OBJECT pTdiAddressDevObj, PFILE_OBJECT pTdiEndPointFileObject,
PUCHAR pSendData, ULONG ulSendDataLength)
{
    NTSTATUS status = STATUS_SUCCESS;
    KEVENT sendEvent;
    PIRP pIrp = NULL;
    IO_STATUS_BLOCK iosb = { 0 };
    PMDL pSendMdl = NULL;

    KeInitializeEvent(&sendEvent, NotificationEvent, FALSE);

    pIrp = TdiBuildInternalDeviceControlIrp(TDI_SEND, pTdiAddressDevObj,
pTdiEndPointFileObject, &sendEvent, &iosb);
    if (NULL == pIrp)
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    pSendMdl = IoAllocateMdl(pSendData, ulSendDataLength, FALSE, FALSE, pIrp);
    if (NULL == pSendMdl)
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }
    MmProbeAndLockPages(pSendMdl, KernelMode, IoModifyAccess);

    TdiBuildSend(pIrp, pTdiAddressDevObj, pTdiEndPointFileObject, NULL, NULL, pSendMdl, 0,
ulSendDataLength);

    IoSetCompletionRoutine(pIrp, TdiCompletionRoutine, &sendEvent, TRUE, TRUE, TRUE);

    status = IoCallDriver(pTdiAddressDevObj, pIrp);
}

```

```
if (STATUS_PENDING == status)
{
    KeWaitForSingleObject(&sendEvent, Executive, KernelMode, FALSE, NULL);
}

if (pSendMdl)
{
    IoFreeMdl(pSendMdl);
}
return status;
}

// TCP接收信息
ULONG_PTR TdiRecv(PDEVICE_OBJECT pTdiAddressDevObj, PFILE_OBJECT pTdiEndPointFileObject,
PUCHAR pRecvData, ULONG ulRecvDataLength)
{
    NTSTATUS status = STATUS_SUCCESS;
    KEVENT recvEvent;
    PIRP pIrp = NULL;
    IO_STATUS_BLOCK iosb = { 0 };
    PMDL pRecvMdl = NULL;
    ULONG_PTR ulRecvSize = 0;

    KeInitializeEvent(&recvEvent, NotificationEvent, FALSE);

    pIrp = TdiBuildInternalDeviceControlIrp(TDI_RECV, pTdiAddressDevObj,
pTdiEndPointFileObject, &recvEvent, &iosb);
    if (NULL == pIrp)
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }

    pRecvMdl = IoAllocateMdl(pRecvData, ulRecvDataLength, FALSE, FALSE, pIrp);
    if (NULL == pRecvMdl)
    {
        return STATUS_INSUFFICIENT_RESOURCES;
    }
    MmProbeAndLockPages(pRecvMdl, KernelMode, IoModifyAccess);

    TdiBuildReceive(pIrp, pTdiAddressDevObj, pTdiEndPointFileObject, NULL, NULL, pRecvMdl,
TDI_RECEIVE_NORMAL, ulRecvDataLength);

    IoSetCompletionRoutine(pIrp, TdiCompletionRoutine, &recvEvent, TRUE, TRUE, TRUE);

    status = IoCallDriver(pTdiAddressDevObj, pIrp);
    if (STATUS_PENDING == status)
    {
        KeWaitForSingleObject(&recvEvent, Executive, KernelMode, FALSE, NULL);
    }

    ulRecvSize = pIrp->IoStatus.Information;

    if (pRecvMdl)
```

```

    {
        IoFreeMdl(pRecvMdl);
    }
    return status;
}

// 关闭释放
VOID TdiClose(PFILE_OBJECT pTdiEndPointFileObject, HANDLE hTdiAddress, HANDLE hTdiEndPoint)
{
    if (pTdiEndPointFileObject)
    {
        ObDereferenceObject(pTdiEndPointFileObject);
    }
    if (hTdiEndPoint)
    {
        ZwClose(hTdiEndPoint);
    }
    if (hTdiAddress)
    {
        ZwClose(hTdiAddress);
    }
}

```

传输字符串

服务端在应用层侦听，客户端是驱动程序，驱动程序加载后自动连接应用层并发送消息。

首先来看 [应用层\(服务端\)](#) 代码，具体我就不说了，来看教程的都是有基础的。

```

#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <winsock2.h>

#pragma comment(lib,"ws2_32.lib")
#define PORT 8888

int main(int argc, char *argv[])
{
    printf("hello lyshark.com \n");
    WSADATA WSADATA;
    SOCKET sock, msgsock;
    struct sockaddr_in ServerAddr;

    if (WSAStartup(MAKEWORD(2, 0), &WSADATA) != SOCKET_ERROR)
    {
        ServerAddr.sin_family = AF_INET;
        ServerAddr.sin_port = htons(PORT);
        ServerAddr.sin_addr.s_addr = INADDR_ANY;

        sock = socket(AF_INET, SOCK_STREAM, 0);
        int BindRet = bind(sock, (LPSOCKADDR)&ServerAddr, sizeof(ServerAddr));
        int LinsRet = listen(sock, 10);
    }
}

```

```

while (1)
{
    char buf[1024] = { 0 };
    msgsock = accept(sock, (LPSOCKADDR)0, (int *)0);
    memset(buf, 0, sizeof(buf));

    recv(msgsock, buf, 1024, 0);
    printf("内核返回: %s \n", buf);

    char send_buffer[1024] = { 0 };
    memset(send_buffer, 0, 1024);
    strcpy(send_buffer, "Hi, RO !");
    send(msgsock, send_buffer, strlen(send_buffer), 0);
    closesocket(msgsock);
}

closesocket(sock);
WSACleanup();
return 0;
}

```

再来是驱动层代码，如下所示；

```

#include "MyTDI.hpp"

// 发送接收数据
NTSTATUS SendOnRecv()
{
    NTSTATUS status = STATUS_SUCCESS;
    HANDLE hTdiAddress = NULL;
    HANDLE hTdiEndPoint = NULL;
    PDEVICE_OBJECT pTdiAddressDevObj = NULL;
    PFILE_OBJECT pTdiEndPointFileObject = NULL;
    LONG pServerIp[4] = { 127, 0, 0, 1 };
    LONG lServerPort = 8888;
    UCHAR szSendData[] = "Hello Tyshark";
    ULONG ulSendDataLength = 1 + strlen(szSendData);
    HANDLE hThread = NULL;

    // TDI初始化
    status = TdiOpen(&pTdiAddressDevObj, &pTdiEndPointFileObject, &hTdiAddress,
&hTdiEndPoint);
    if (!NT_SUCCESS(status))
    {
        return STATUS_SUCCESS;
    }

    // TDI TCP连接服务器
    status = TdiConnection(pTdiAddressDevObj, pTdiEndPointFileObject, pServerIp, lServerPort);
    if (!NT_SUCCESS(status))
    {
        return STATUS_SUCCESS;
    }
}

```

```

// TDI TCP发送信息
status = TdiSend(pTdiAddressDevObj, pTdiEndPointFileObject, szSendData, ulSendDataLength);
if (!NT_SUCCESS(status))
{
    return STATUS_SUCCESS;
}
DbgPrint("发送: %s\n", szSendData);

// 创建接收信息多线程，循环接收信息

char szRecvData[1024] = { 0 };
ULONG ulRecvDataLenngth = 1024;
RtlZeroMemory(szRecvData, ulRecvDataLenngth);

// TDI TCP接收信息
do
{
    ulRecvDataLenngth = TdiRecv(pTdiAddressDevObj, pTdiEndPointFileObject, szRecvData,
ulRecvDataLenngth);
    if (0 < ulRecvDataLenngth)
    {
        DbgPrint("接收数据: %s\n", szRecvData);
        break;;
    }
} while (TRUE);

// 释放
TdiClose(pTdiEndPointFileObject, hTdiAddress, hTdiEndPoint);
return STATUS_SUCCESS;
}

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("驱动卸载成功 \n");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    for (int x = 0; x < 10; x++)
    {
        SendOnRecv();
    }

    DbgPrint("驱动加载成功 \n");
    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}

```

首先运行应用层开启服务端侦听，然后运行驱动程序，会输出如下信息：

多线程收发包

实现驱动内部发送数据包后开启一个线程用于等待应用层返回并输出结果，多线程收发在发送数据包后需要创建新的线程等待接收。

首先是服务端代码。

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <winsock2.h>

#pragma comment(lib,"ws2_32.lib")
#define PORT 8888

int main(int argc, char *argv[])
{
    printf("hello lyshark.com \n");
    WSADATA WSADATA;
    SOCKET sock, msgsock;
    struct sockaddr_in ServerAddr;

    if (WSAStartup(MAKEWORD(2, 0), &WSADATA) != SOCKET_ERROR)
    {
        ServerAddr.sin_family = AF_INET;
        ServerAddr.sin_port = htons(PORT);
        ServerAddr.sin_addr.s_addr = INADDR_ANY;

        sock = socket(AF_INET, SOCK_STREAM, 0);
        int BindRet = bind(sock, (LPSOCKADDR)&ServerAddr, sizeof(ServerAddr));
        int LinsRet = listen(sock, 10);
    }
}
```

```

while (1)
{
    char buf[1024] = { 0 };
    msgsock = accept(sock, (LPSOCKADDR)0, (int *)0);
    memset(buf, 0, sizeof(buf));

    recv(msgsock, buf, 1024, 0);
    printf("内核返回: %s \n", buf);

    char send_buffer[1024] = { 0 };
    memset(send_buffer, 0, 1024);
    strcpy(send_buffer, "Hi, RO !");
    send(msgsock, send_buffer, strlen(send_buffer), 0);
    closesocket(msgsock);
}

closesocket(sock);
WSACleanup();
return 0;
}

```

驱动程序代码如下，`RecvThreadProc` 主要负责数据接收，`SendThreadData` 负责数据发送。

```

#include "LySocket.hpp"

typedef struct _MY_DATA
{
    PDEVICE_OBJECT pTdiAddressDevObj;
    PFILE_OBJECT pTdiEndPointFileObject;
    HANDLE hTdiAddress;
    HANDLE hTdiEndPoint;
}MY_DATA, *PMY_DATA;

// 接收信息多线程
VOID RcvThreadProc(_In_ PVOID StartContext)
{
    PMY_DATA pMyData = (PMY_DATA)StartContext;
    NTSTATUS status = STATUS_SUCCESS;
    char szRecvData[1024] = { 0 };
    ULONG ulRecvDataLenngth = 1024;
    RtlZeroMemory(szRecvData, ulRecvDataLenngth);

    // TDI TCP接收信息
    do
    {
        ulRecvDataLenngth = TdiRecv(pMyData->pTdiAddressDevobj, pMyData->pTdiEndPointFileObject,
        szRecvData, ulRecvDataLenngth);
        if (0 < ulRecvDataLenngth)
        {
            DbgPrint("线程句柄:%x --> 接收数据包: %s\n", pMyData->hTdiEndPoint, szRecvData);
            break;;
        }
    } while (TRUE);
}

```

```
// 释放
Tdiclose(pMyData->pTdiEndPointFileObject, pMyData->hTdiAddress, pMyData->hTdiEndPoint);
ExFreePool(pMyData);
}

// 多线程发送
NTSTATUS SendThreadData()
{
    NTSTATUS status = STATUS_SUCCESS;
    HANDLE hTdiAddress = NULL;
    HANDLE hTdiEndPoint = NULL;
    PDEVICE_OBJECT pTdiAddressDevObj = NULL;
    PFILE_OBJECT pTdiEndPointFileObject = NULL;
    LONG pServerIp[4] = { 127, 0, 0, 1 };
    LONG lServerPort = 8888;
    UCHAR szSendData[] = "hello lyshark";
    ULONG ulSendDataLength = 1 + strlen(szSendData);
    HANDLE hThread = NULL;

    // TDI初始化
    status = TdiOpen(&pTdiAddressDevObj, &pTdiEndPointFileObject, &hTdiAddress,
&hTdiEndPoint);
    if (!NT_SUCCESS(status))
    {
        return STATUS_SUCCESS;
    }

    // TDI TCP连接服务器
    status = TdiConnection(pTdiAddressDevObj, pTdiEndPointFileObject, pServerIp, lServerPort);
    if (!NT_SUCCESS(status))
    {
        return STATUS_SUCCESS;
    }

    // TDI TCP发送信息
    status = Tdisend(pTdiAddressDevObj, pTdiEndPointFileObject, szSendData, ulSendDataLength);
    if (!NT_SUCCESS(status))
    {
        return STATUS_SUCCESS;
    }
    DbgPrint("发送 %s\n", szSendData);

    // 创建接收信息多线程，循环接收信息
    PMY_DATA pMyData = ExAllocatePool(NonPagedPool, sizeof(MY_DATA));
    pMyData->pTdiAddressDevObj = pTdiAddressDevObj;
    pMyData->pTdiEndPointFileObject = pTdiEndPointFileObject;
    pMyData->hTdiAddress = hTdiAddress;
    pMyData->hTdiEndPoint = hTdiEndPoint;

    PsCreateSystemThread(&hThread, 0, NULL, NtCurrentProcess(), NULL, RecvThreadProc,
pMyData);
}
```

```

VOID UnDriver(PDRIVER_OBJECT driver)
{
    DbgPrint("驱动卸载成功 \n");
}

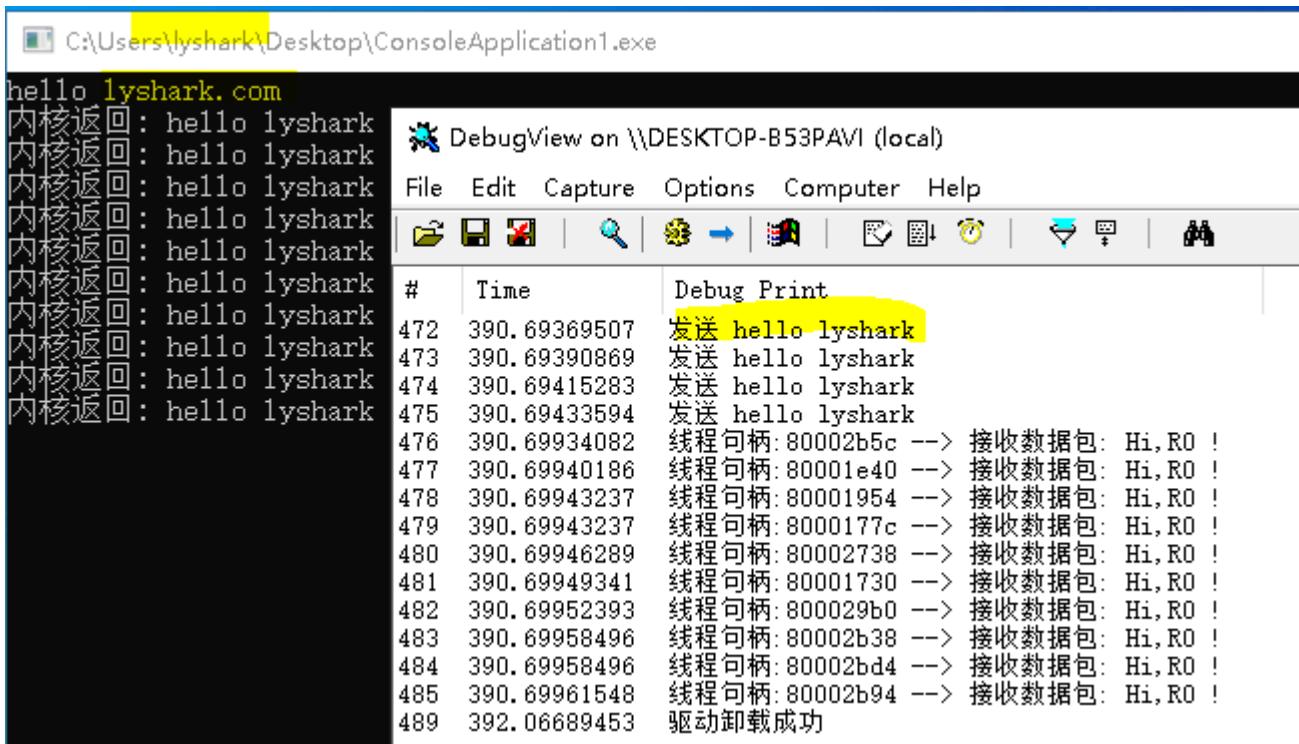
NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark.com \n");

    for (int x = 0; x < 10; x++)
    {
        SendThreadData();
    }

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}

```

运行应用层服务端等待侦听，运行驱动程序输出如下效果；



传输结构认证

驱动内部发送结构体给应用层，应用层验证结构体成员，此功能可实现对驱动程序的控制机制，例如是否允许驱动加载卸载等，通常用于驱动辅助认证。

应用层代码

```

#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <winsock2.h>

#pragma comment(lib,"ws2_32.lib")

```

```
#define PORT 8888

// 传输结构体
typedef struct
{
    int uuid;
    char username[256];
    char password[256];
}SocketData;

int main(int argc, char *argv[])
{
    printf("hello lyshark.com \n");

    WSADATA WSADATA;
    SOCKET sock, msgsock;
    struct sockaddr_in ServerAddr;

    if (WSAStartup(MAKEWORD(2, 0), &WSADATA) != SOCKET_ERROR)
    {
        ServerAddr.sin_family = AF_INET;
        ServerAddr.sin_port = htons(PORT);
        ServerAddr.sin_addr.s_addr = INADDR_ANY;

        sock = socket(AF_INET, SOCK_STREAM, 0);
        int BindRet = bind(sock, (LPSOCKADDR)&ServerAddr, sizeof(ServerAddr));
        int LinsRet = listen(sock, 10);
    }

    while (1)
    {
        char buf[8192] = { 0 };
        msgsock = accept(sock, (LPSOCKADDR)0, (int *)0);
        memset(buf, 0, sizeof(buf));

        // 接收返回数据
        recv(msgsock, buf, sizeof(SocketData), 0);

        // 强转结构体
        SocketData* msg = (SocketData*)buf;

        printf("UUID = %d \n", msg->uuid);
        printf("名字 = %s \n", msg->username);
        printf("密码 = %s \n", msg->password);

        // 验证通过则继续使用
        if ((strcmp(msg->username, "lyshark") == 0) && (strcmp(msg->password, "123") == 0))
        {
            char send_buffer[8192] = { 0 };
            memset(send_buffer, 0, 8192);
            strcpy(send_buffer, "success");
            send(msgsock, send_buffer, strlen(send_buffer), 0);
            closesocket(msgsock);
        }
    }
}
```

```

    }
    // 不通过则禁止驱动加载
    else
    {
        char send_buffer[8192] = { 0 };
        memset(send_buffer, 0, 8192);
        strcpy(send_buffer, "error");
        send(msgsock, send_buffer, strlen(send_buffer), 0);
        closesocket(msgsock);
    }
}
closesocket(sock);
WSACleanup();
return 0;
}

```

驱动层代码

```

#include "LySocket.hpp"

// 传输结构体
typedef struct
{
    int uuid;
    char username[256];
    char password[256];
} SocketData;

// 验证账号密码是否正确
BOOLEAN CheckDriver()
{
    NTSTATUS status = STATUS_SUCCESS;
    HANDLE hTdiAddress = NULL;
    HANDLE hTdiEndPoint = NULL;
    PDEVICE_OBJECT pTdiAddressDevObj = NULL;
    PFILE_OBJECT pTdiEndPointFileObject = NULL;
    LONG pServerIp[4] = { 127, 0, 0, 1 };
    LONG lServerPort = 8888;

    // TDI初始化
    status = TdiOpen(&pTdiAddressDevObj, &pTdiEndPointFileObject, &hTdiAddress,
&hTdiEndPoint);
    if (!NT_SUCCESS(status))
    {
        return STATUS_SUCCESS;
    }

    // TDI TCP连接服务器
    status = TdiConnection(pTdiAddressDevObj, pTdiEndPointFileObject, pServerIp, lServerPort);
    if (!NT_SUCCESS(status))
    {
        return STATUS_SUCCESS;
    }
}

```

```
SocketData ptr;

RtlZeroMemory(&ptr, sizeof(SocketData));

// 填充结构
ptr.uuid = 1001;
RtlCopyMemory(ptr.username, "lyshark", strlen("xxxxxx"));
RtlCopyMemory(ptr.password, "123123", strlen("xxxxxx"));

// TDI TCP发送信息
status = TdiSend(pTdiAddressDevObj, pTdiEndPointFileObject, &ptr, sizeof(SocketData));
if (!NT_SUCCESS(status))
{
    return STATUS_SUCCESS;
}

// 创建接收信息多线程，循环接收信息
char szRecvData[8192] = { 0 };
ULONG ulRecvDataLenngth = 8192;
RtlZeroMemory(szRecvData, ulRecvDataLenngth);

// TDI TCP接收信息
do
{
    ulRecvDataLenngth = TdiRecv(pTdiAddressDevObj, pTdiEndPointFileObject, szRecvData,
ulRecvDataLenngth);
    if (0 < ulRecvDataLenngth)
    {
        DbgPrint("接收数据: %s\n", szRecvData);

        if (strcmp(szRecvData, "success", 7) == 0)
        {
            // 释放
            Tdiclose(pTdiEndPointFileObject, hTdiAddress, hTdiEndPoint);
            return TRUE;
        }
        else if (strcmp(szRecvData, "error", 5) == 0)
        {
            // 释放
            Tdiclose(pTdiEndPointFileObject, hTdiAddress, hTdiEndPoint);
            return FALSE;
        }
        break;;
    }
} while (TRUE);

// 释放
Tdiclose(pTdiEndPointFileObject, hTdiAddress, hTdiEndPoint);
return STATUS_SUCCESS;
}

VOID UnDriver(PDRIVER_OBJECT driver)
```

```

{
    DbgPrint("驱动卸载成功 \n");
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
    DbgPrint("hello lyshark.com \n");

    BOOLEAN ref = CheckDriver();

    if (ref == FALSE)
    {
        DbgPrint("[LyShark.com] 驱动已过期,无法加载 \n");
        Driver->DriverUnload = UnDriver;
        return STATUS_SUCCESS;
    }

    DbgPrint("[*] 驱动正常使用 \n");
    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}

```

运行应用层服务端，并运行驱动程序，则会验证该驱动是否合法，如果合法则加载不合法则拒绝；

