在笔者上一篇文章 《驱动开发：应用DeviceIoContro开发模板》 简单为大家介绍了如何使用

# DeviceIoContro 模板快速创建一个驱动开发通信案例，但是该案例过于简单也无法独立加载运行，本章将继续延申这个知识点，通过封装一套标准通用模板来实现驱动通信中的常用传递方式，这其中包括 了如何传递字符串，传递整数，传递数组，传递结构体等方法。可以说如果你能掌握本章模板精讲的内 容基本上市面上的功能都可以使用本方法进行通信。

首先定义驱动功能号和名字，提供接口给应用程序调用。

METHOD\_BUFFERED, FILE\_ANY\_ACCESS)

CTL\_CODE(FILE\_DEVICE\_UNKNOWN, 0x804,

#define IOCTL\_IO\_String

METHOD\_BUFFERED, FILE\_ANY\_ACCESS)

CTL\_CODE(FILE\_DEVICE\_UNKNOWN, 0x803,

#define IOCTL\_IO\_STRUCT

METHOD\_BUFFERED, FILE\_ANY\_ACCESS)

CTL\_CODE(FILE\_DEVICE\_UNKNOWN, 0x802,

#define IOCTL\_IO\_Array

METHOD\_BUFFERED, FILE\_ANY\_ACCESS)

CTL\_CODE(FILE\_DEVICE\_UNKNOWN, 0x801,

#define IOCTL\_IO\_TEST

METHOD\_BUFFERED, FILE\_ANY\_ACCESS)

CTL\_CODE(FILE\_DEVICE\_UNKNOWN, 0x800,

#define IOCTL\_IO\_Msg

**IOCTL\_IO\_TEST 传递整数：** 派遣例程用于传递整数类型，首先客户端通过 DriveControl.IoControl 将数据通过变量 input 传入到内核中，并等待返回，如果返回了结果则 outpot 里存储的就是返回结果， ref\_len 则是返回长度。

# 看看客户端如何接收这个数据的传递。

// 传入x参数,返回到y中,返回长度为z

DWORD input = 100, output = 0, ref\_len = 0; DriveControl.IoControl(IOCTL\_IO\_TEST, &input, sizeof(input), &output, sizeof(output), &ref\_len);

std::cout << "传入参数: " << input << std::endl; std::cout << "输出参数: " << output << std::endl; std::cout << "参数长度: " << ref\_len << std::endl;

对于驱动中我们不需要做任何操作只需要通过 memcpy(&dw, pIoBuffer, sizeof(DWORD)) 得到缓冲区内的数据，对该数据 dw++ 递增，最后通过 memcpy(pIoBuffer, &dw, sizeof(DWORD)) 再将数据写回到应用层。

case IOCTL\_IO\_TEST:

{

DWORD dw = 0;

// 得到输入参数

memcpy(&dw, pIoBuffer, sizeof(DWORD));

// 对输入参数进行处理

dw++;

// 设置输出参数

memcpy(pIoBuffer, &dw, sizeof(DWORD));

// 返回通信状态

status = STATUS\_SUCCESS;

break;

}

**IOCTL\_IO\_Array 传递数组：** 派遣例程用于传递数组类型，首先定义数组 MyArray 将数组 首地址 以及 数

组长度 传递到内核中，内核收到首地址以及长度后通过 uInSize / sizeof(int) 得到每一个元素的长

# 度，最后循环输出元素即可。



// --------------------------------------------------------------------------

// 应用层

// --------------------------------------------------------------------------

// 传入数组

int MyArray[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };

int array\_count = 10 sizeof(int);

DriveControl.IoControl(IOCTL\_IO\_Array, &MyArray, array\_count, 0, 0, 0);

// --------------------------------------------------------------------------

// 内核层

// --------------------------------------------------------------------------

// 输出一个数组

case IOCTL\_IO\_Array:

{

int ArrayPtr = (int )pIoBuffer; int count = uInSize / sizeof(int);

for (int x = 0; x < count; x++)

{

DbgPrint("计数器: %d \n", ArrayPtr[x]);

}

status = STATUS\_SUCCESS; break;

}

**IOCTL\_IO\_STRUCT 传递结构：** 结构体的传输与数组类似，仅仅只是在接收到数据后对其进行一定的转换即可，应用层只需要 DriveControl.IoControl 发送 send\_ptr 指针并等待回传recv\_ptr 即可，最后将

得到的结果直接输出，而内核层收到数据后仅仅也只需要 (MyData \*)pIoBuffer 将数据转为一个指针即可操作这片区域，当操作完成时以同样的方式 memcpy(pIoBuffer, &send\_data, sizeof(MyData))`返回一个结构体给

# 应用层。

// --------------------------------------------------------------------------

// 应用层

// --------------------------------------------------------------------------

// 传入一个结构体,返回结构体MyData send\_ptr, recv\_ptr; DWORD dwSize = 0;

memset(send\_ptr.szUname, 0, 1024);

memset(recv\_ptr.szUname, 0, 1024);

send\_ptr.uuid = 1001; strcpy(send\_ptr.szUname, "lyshark");



// 派遣命令

DriveControl.IoControl(IOCTL\_IO\_STRUCT, &send\_ptr, sizeof(send\_ptr), &recv\_ptr, sizeof(recv\_ptr), &dwSize);

// DeviceIoControl(hDevice, IOCTL\_IO\_STRUCT, &send\_ptr, sizeof(send\_ptr), (LPVOID)&recv\_ptr, sizeof(recv\_ptr), &dwSize, 0);

std::cout << "内核返回数据: " << recv\_ptr.uuid << std::endl; std::cout << "内核返回数据: " << recv\_ptr.szUname << std::endl; std::cout << "内核返回长度: " << dwSize << std::endl;

// --------------------------------------------------------------------------

// 内核层

// --------------------------------------------------------------------------

// 测试传递结构体

case IOCTL\_IO\_STRUCT:

{

MyData recv\_data, send\_data;

// 获取到应用层传入的数据

/

MyData ptr = (MyData )pIoBuffer;

DbgPrint("获取序列号: %d \n", ptr->uid);

DbgPrint("获取名字: %s \n", ptr->szBuf);

/

// 获取数据

memcpy(&recv\_data, pIoBuffer, sizeof(MyData));

DbgPrint("[驱动读入]: UID: %d --> Name: %s \n", recv\_data.uuid, recv\_data.szUname);

// 发送数据

memset(send\_data.szUname, 0, 1024);

send\_data.uuid = 1002; strcpy(send\_data.szUname, "lyshark"); memcpy(pIoBuffer, &send\_data, sizeof(MyData));

status = STATUS\_SUCCESS; break;

}

如上就是内核层与应用层的部分代码功能分析，至于传递字符串也很简单大家自己学习下即可掌握，接 下来我将完整代码分享出来，大家可以自行测试效果。

驱动程序 WinDDK.sys 完整代码；

// 署名权

// right to sign one's name on a piece of work

// PowerBy: LyShark

// Email: [me@lyshark.com](mailto:me@lyshark.com)

#define \_CRT\_SECURE\_NO\_WARNINGS #include <ntifs.h>

#include <windef.h>

// 定义符号链接，一般来说修改为驱动的名字即可

#define DEVICE\_NAME L"\\Device\\WinDDK" #define LINK\_NAME L"\\DosDevices\\WinDDK"

#define LINK\_GLOBAL\_NAME L"\\DosDevices\\Global\\WinDDK"

// 定义驱动功能号和名字，提供接口给应用程序调用

#define IOCTL\_IO\_Msg CTL\_CODE(FILE\_DEVICE\_UNKNOWN, 0x800, METHOD\_BUFFERED, FILE\_ANY\_ACCESS)

#define IOCTL\_IO\_TEST CTL\_CODE(FILE\_DEVICE\_UNKNOWN, 0x801, METHOD\_BUFFERED, FILE\_ANY\_ACCESS)

#define IOCTL\_IO\_Array CTL\_CODE(FILE\_DEVICE\_UNKNOWN, 0x802,

METHOD\_BUFFERED, FILE\_ANY\_ACCESS)

#define IOCTL\_IO\_STRUCT CTL\_CODE(FILE\_DEVICE\_UNKNOWN, 0x803, METHOD\_BUFFERED, FILE\_ANY\_ACCESS)

#define IOCTL\_IO\_String CTL\_CODE(FILE\_DEVICE\_UNKNOWN, 0x804,

METHOD\_BUFFERED, FILE\_ANY\_ACCESS)

// 保存一段非分页内存,用于给全局变量使用

#define FILE\_DEVICE\_EXTENSION 4096

// 定义传递结构体

typedef struct

{

int uuid;

char szUname[1024];

}MyData;

// 驱动绑定默认派遣函数

NTSTATUS DefaultDispatch(PDEVICE\_OBJECT \_pDeviceObject, PIRP \_pIrp)

{

\_pIrp->IoStatus.Status = STATUS\_NOT\_SUPPORTED;

\_pIrp->IoStatus.Information = 0; IoCompleteRequest(\_pIrp, IO\_NO\_INCREMENT); return \_pIrp->IoStatus.Status;

}

// 驱动卸载的处理例程

VOID DriverUnload(PDRIVER\_OBJECT pDriverObj)

{

if (pDriverObj->DeviceObject)

{

UNICODE\_STRING strLink;

// 删除符号连接和设备

RtlInitUnicodeString(&strLink, LINK\_NAME); IoDeleteSymbolicLink(&strLink); IoDeleteDevice(pDriverObj->DeviceObject);

DbgPrint("[kernel] # 驱动已卸载 \n");

}

}

// IRP\_MJ\_CREATE 对应的处理例程，一般不用管它

NTSTATUS DispatchCreate(PDEVICE\_OBJECT pDevObj, PIRP pIrp)

{

DbgPrint("[kernel] # 驱动处理例程载入 \n"); pIrp->IoStatus.Status = STATUS\_SUCCESS; pIrp->IoStatus.Information = 0;

IoCompleteRequest(pIrp, IO\_NO\_INCREMENT); return STATUS\_SUCCESS;

}

// IRP\_MJ\_CLOSE 对应的处理例程，一般不用管它

NTSTATUS DispatchClose(PDEVICE\_OBJECT pDevObj, PIRP pIrp)

{

DbgPrint("[kernel] # 关闭派遣 \n");

pIrp->IoStatus.Status = STATUS\_SUCCESS; pIrp->IoStatus.Information = 0; IoCompleteRequest(pIrp, IO\_NO\_INCREMENT); return STATUS\_SUCCESS;

}

// IRP\_MJ\_DEVICE\_CONTROL 对应的处理例程，驱动最重要的函数

NTSTATUS DispatchIoctl(PDEVICE\_OBJECT pDevObj, PIRP pIrp)

{

NTSTATUS status = STATUS\_INVALID\_DEVICE\_REQUEST;

PIO\_STACK\_LOCATION pIrpStack; ULONG uIoControlCode;

PVOID pIoBuffer;

ULONG uInSize; ULONG uOutSize;

// 获得IRP里的关键数据

pIrpStack = IoGetCurrentIrpStackLocation(pIrp);

// 获取控制码

uIoControlCode = pIrpStack->Parameters.DeviceIoControl.IoControlCode;

// 输入和输出的缓冲区（DeviceIoControl的InBuffer和OutBuffer都是它）

pIoBuffer = pIrp->AssociatedIrp.SystemBuffer;

// EXE发送传入数据的BUFFER长度（DeviceIoControl的nInBufferSize） uInSize = pIrpStack->Parameters.DeviceIoControl.InputBufferLength;

// EXE接收传出数据的BUFFER长度（DeviceIoControl的nOutBufferSize） uOutSize = pIrpStack->Parameters.DeviceIoControl.OutputBufferLength;

// 对不同控制信号的处理流程

switch (uIoControlCode)

{

// 输出一段话

case IOCTL\_IO\_Msg:

{

DbgPrint("[Kernel] --> hello lyshark \n"); status = STATUS\_SUCCESS;

break;

}

// 测试单一参数传递

case IOCTL\_IO\_TEST:

{

DWORD dw = 0;

// 得到输入参数

memcpy(&dw, pIoBuffer, sizeof(DWORD));



// 对输入参数进行处理

dw++;

// 设置输出参数

memcpy(pIoBuffer, &dw, sizeof(DWORD));

// 返回通信状态

status = STATUS\_SUCCESS; break;

}

// 输出一个数组

case IOCTL\_IO\_Array:

{

int ArrayPtr = (int )pIoBuffer; int count = uInSize / sizeof(int);

for (int x = 0; x < count; x++)

{

DbgPrint("计数器: %d \n", ArrayPtr[x]);

}

status = STATUS\_SUCCESS; break;

}

// 测试传递结构体

case IOCTL\_IO\_STRUCT:

{

MyData recv\_data, send\_data;

// 获取到应用层传入的数据

/

MyData ptr = (MyData )pIoBuffer;

DbgPrint("获取序列号: %d \n", ptr->uid);

DbgPrint("获取名字: %s \n", ptr->szBuf);

/

// 获取数据

memcpy(&recv\_data, pIoBuffer, sizeof(MyData));

DbgPrint("[驱动读入]: UID: %d --> Name: %s \n", recv\_data.uuid, recv\_data.szUname);

// 发送数据

memset(send\_data.szUname, 0, 1024);

send\_data.uuid = 1002; strcpy(send\_data.szUname, "lyshark"); memcpy(pIoBuffer, &send\_data, sizeof(MyData));

status = STATUS\_SUCCESS; break;

}

// 测试传递字符串

case IOCTL\_IO\_String:

{

char szString[256] = { 0 }; char szSendPtr[256] = { 0 };

// 接收字符串

memcpy(szString, pIoBuffer, sizeof(szString));

DbgPrint("[接收字符串]: %s \n", szString);

// 发送字符串

strcpy(szSendPtr, "hi, R3");

memcpy(pIoBuffer, &szSendPtr, sizeof(szSendPtr));

status = STATUS\_SUCCESS; break;

}

}

// 设定DeviceIoControl的 lpBytesReturned的值（如果通信失败则返回0长度） if (status == STATUS\_SUCCESS)

{

}

else

{

}

pIrp->IoStatus.Information = uOutSize;

pIrp->IoStatus.Information = 0;



// 设定DeviceIoControl的返回值是成功还是失败pIrp->IoStatus.Status = status; IoCompleteRequest(pIrp, IO\_NO\_INCREMENT); return status;

}

// 驱动的初始化工作

NTSTATUS DriverEntry(PDRIVER\_OBJECT pDriverObj, PUNICODE\_STRING pRegistryString)

{

NTSTATUS status = STATUS\_SUCCESS;

UNICODE\_STRING ustrLinkName; UNICODE\_STRING ustrDevName; PDEVICE\_OBJECT pDevObj;

// 初始化其他派遣

for (ULONG i = 0; i < IRP\_MJ\_MAXIMUM\_FUNCTION; i++)

{

// DbgPrint("初始化派遣: %d \n", i);

pDriverObj->MajorFunction[i] = DefaultDispatch;

}

// 设置分发函数和卸载例程

pDriverObj->MajorFunction[IRP\_MJ\_CREATE] = DispatchCreate; pDriverObj->MajorFunction[IRP\_MJ\_CLOSE] = DispatchClose; pDriverObj->MajorFunction[IRP\_MJ\_DEVICE\_CONTROL] = DispatchIoctl; pDriverObj->DriverUnload = DriverUnload;

// 创建一个设备

RtlInitUnicodeString(&ustrDevName, DEVICE\_NAME);

// FILE\_DEVICE\_EXTENSION 创建设备时，指定设备扩展内存的大小，传一个值进去，就会给设备分配一块非页面内存。

status = IoCreateDevice(pDriverObj, sizeof(FILE\_DEVICE\_EXTENSION), &ustrDevName, FILE\_DEVICE\_UNKNOWN, 0, FALSE, &pDevObj);

if (!NT\_SUCCESS(status))

{

return status;

}

// 判断支持的WDM版本，其实这个已经不需要了，纯属WIN9X和WINNT并存时代的残留物

if (IoIsWdmVersionAvailable(1, 0x10))

{

RtlInitUnicodeString(&ustrLinkName, LINK\_GLOBAL\_NAME);

}

else

{

RtlInitUnicodeString(&ustrLinkName, LINK\_NAME);

}

// 创建符号连接

status = IoCreateSymbolicLink(&ustrLinkName, &ustrDevName); if (!NT\_SUCCESS(status))

{

DbgPrint("创建符号链接失败 \n"); IoDeleteDevice(pDevObj); return status;

}

DbgPrint("[ Hello LyShark.com ] # 驱动初始化完毕 \n");

// 返回加载驱动的状态（如果返回失败，驱动讲被清除出内核空间）

return STATUS\_SUCCESS;

}

应用层客户端程序 lyshark.exe 完整代码；

// 署名权

// right to sign one's name on a piece of work

// PowerBy: LyShark

// Email: [me@lyshark.com](mailto:me@lyshark.com)

#include <iostream> #include <Windows.h> #include <vector>

#pragma comment(lib,"user32.lib") #pragma comment(lib,"advapi32.lib")

// 定义驱动功能号和名字，提供接口给应用程序调用

#define IOCTL\_IO\_Msg 0x800

#define IOCTL\_IO\_TEST 0x801

#define IOCTL\_IO\_Array 0x802

#define IOCTL\_IO\_STRUCT 0x803

#define IOCTL\_IO\_String 0x804

class cDrvCtrl

{

public:

cDrvCtrl()

{

m\_pSysPath = NULL; m\_pServiceName = NULL; m\_pDisplayName = NULL; m\_hSCManager = NULL; m\_hService = NULL;

m\_hDriver = INVALID\_HANDLE\_VALUE;

}

~cDrvCtrl()

{

CloseServiceHandle(m\_hService); CloseServiceHandle(m\_hSCManager); CloseHandle(m\_hDriver);

}

// 安装驱动

BOOL Install(PCHAR pSysPath, PCHAR pServiceName, PCHAR pDisplayName)

{

m\_pSysPath = pSysPath; m\_pServiceName = pServiceName; m\_pDisplayName = pDisplayName;

m\_hSCManager = OpenSCManagerA(NULL, NULL, SC\_MANAGER\_ALL\_ACCESS); if (NULL == m\_hSCManager)

{

m\_dwLastError = GetLastError(); return FALSE;

}

m\_hService = CreateServiceA(m\_hSCManager, m\_pServiceName, m\_pDisplayName, SERVICE\_ALL\_ACCESS, SERVICE\_KERNEL\_DRIVER, SERVICE\_DEMAND\_START,

SERVICE\_ERROR\_NORMAL,

m\_pSysPath, NULL, NULL, NULL, NULL, NULL);

if (NULL == m\_hService)

{

m\_dwLastError = GetLastError();

if (ERROR\_SERVICE\_EXISTS == m\_dwLastError)

{

m\_hService = OpenServiceA(m\_hSCManager, m\_pServiceName, SERVICE\_ALL\_ACCESS);

if (NULL == m\_hService)

{

}

}

else

{

CloseServiceHandle(m\_hSCManager); return FALSE;

CloseServiceHandle(m\_hSCManager); return FALSE;

}

}

return TRUE;

}

// 启动驱动

BOOL Start()

{

if (!StartServiceA(m\_hService, NULL, NULL))

{

m\_dwLastError = GetLastError(); return FALSE;

}

return TRUE;

}

// 关闭驱动

BOOL Stop()

{

SERVICE\_STATUS ss;

GetSvcHandle(m\_pServiceName);

if (!ControlService(m\_hService, SERVICE\_CONTROL\_STOP, &ss))

{

m\_dwLastError = GetLastError(); return FALSE;

}

return TRUE;

}

// 移除驱动

BOOL Remove()

{

GetSvcHandle(m\_pServiceName); if (!DeleteService(m\_hService))

{

m\_dwLastError = GetLastError(); return FALSE;

}

return TRUE;

}

// 打开驱动

BOOL Open(PCHAR pLinkName)

{

if (m\_hDriver != INVALID\_HANDLE\_VALUE) return TRUE;

m\_hDriver = CreateFileA(pLinkName, GENERIC\_READ | GENERIC\_WRITE, 0, 0, OPEN\_EXISTING, FILE\_ATTRIBUTE\_NORMAL, 0);

if (m\_hDriver != INVALID\_HANDLE\_VALUE) return TRUE;

else

}

return FALSE;



// 发送控制信号

BOOL IoControl(DWORD dwIoCode, PVOID InBuff, DWORD InBuffLen, PVOID OutBuff, DWORD OutBuffLen, DWORD RealRetBytes)

{

DWORD dw;

BOOL b = DeviceIoControl(m\_hDriver, CTL\_CODE\_GEN(dwIoCode), InBuff, InBuffLen, OutBuff, OutBuffLen, &dw, NULL);

if (RealRetBytes)

RealRetBytes = dw; return b;

}

private:

// 获取服务句柄

BOOL GetSvcHandle(PCHAR pServiceName)

{

m\_pServiceName = pServiceName;

m\_hSCManager = OpenSCManagerA(NULL, NULL, SC\_MANAGER\_ALL\_ACCESS); if (NULL == m\_hSCManager)

{

m\_dwLastError = GetLastError(); return FALSE;

}

m\_hService = OpenServiceA(m\_hSCManager, m\_pServiceName, SERVICE\_ALL\_ACCESS);

if (NULL == m\_hService)

{

}

else

{

}

}

CloseServiceHandle(m\_hSCManager); return FALSE;

return TRUE;



// 获取控制信号对应字符串

DWORD CTL\_CODE\_GEN(DWORD lngFunction)

{

return (FILE\_DEVICE\_UNKNOWN 65536) | (FILE\_ANY\_ACCESS 16384) |

(lngFunction 4) | METHOD\_BUFFERED;

}

public:

DWORD m\_dwLastError; PCHAR m\_pSysPath; PCHAR m\_pServiceName; PCHAR m\_pDisplayName; HANDLE m\_hDriver;

SC\_HANDLE m\_hSCManager; SC\_HANDLE m\_hService;

};

void GetAppPath(char szCurFile)

{

GetModuleFileNameA(0, szCurFile, MAX\_PATH);

for (SIZE\_T i = strlen(szCurFile) - 1; i >= 0; i--)

{

if (szCurFile[i] == '\\')

{

szCurFile[i + 1] = '\0';

break;



}

}

}

// 定义传递结构体

typedef struct

{

int uuid;

char szUname[1024];

}MyData;

int main(int argc, char argv[])

{

cDrvCtrl DriveControl;

// 设置驱动名称

char szSysFile[MAX\_PATH] = { 0 }; char szSvcLnkName[] = "WinDDK";; GetAppPath(szSysFile); strcat(szSysFile, "WinDDK.sys");

// 安装并启动驱动

DriveControl.Install(szSysFile, szSvcLnkName, szSvcLnkName); DriveControl.Start();

// 打开驱动的符号链接

DriveControl.Open("\\\\.\\WinDDK");

// 无参数输出

DriveControl.IoControl(IOCTL\_IO\_Msg, 0, 0, 0, 0, 0);

// 传入x参数,返回到y中,返回长度为z

DWORD input = 100, output = 0, ref\_len = 0; DriveControl.IoControl(IOCTL\_IO\_TEST, &input, sizeof(input), &output,

sizeof(output), &ref\_len);

std::cout << "传入参数: " << input << std::endl; std::cout << "输出参数: " << output << std::endl; std::cout << "参数长度: " << ref\_len << std::endl;

// 传入数组

int MyArray[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };

int array\_count = 10 sizeof(int);

DriveControl.IoControl(IOCTL\_IO\_Array, &MyArray, array\_count, 0, 0, 0);

// 传入一个结构体,返回结构体MyData send\_ptr, recv\_ptr; DWORD dwSize = 0;

memset(send\_ptr.szUname, 0, 1024);

memset(recv\_ptr.szUname, 0, 1024);

send\_ptr.uuid = 1001; strcpy(send\_ptr.szUname, "lyshark");

// 派遣命令

DriveControl.IoControl(IOCTL\_IO\_STRUCT, &send\_ptr, sizeof(send\_ptr), &recv\_ptr, sizeof(recv\_ptr), &dwSize);

// DeviceIoControl(hDevice, IOCTL\_IO\_STRUCT, &send\_ptr, sizeof(send\_ptr), (LPVOID)&recv\_ptr, sizeof(recv\_ptr), &dwSize, 0);

std::cout << "内核返回数据: " << recv\_ptr.uuid << std::endl; std::cout << "内核返回数据: " << recv\_ptr.szUname << std::endl; std::cout << "内核返回长度: " << dwSize << std::endl;

// 传入一个字符串,返回一个字符串char szString[256] = { 0 }; char szRecvPtr[256] = { 0 };

// 派遣命令

strcpy(szString, "hello lyshark"); DriveControl.IoControl(IOCTL\_IO\_String, &szString, sizeof(szString),

&szRecvPtr, sizeof(szRecvPtr), &dwSize);

std::cout << "内核返回数据: " << szRecvPtr << std::endl;

// 关闭符号链接句柄

CloseHandle(DriveControl.m\_hDriver);

// 停止并卸载驱动DriveControl.Stop(); DriveControl.Remove();

system("pause"); return 0;

}

# 手动编译这两个程序，将驱动签名后以管理员身份运行 lyshark.exe 客户端，此时屏幕中即可看到滚动输出效果，如此一来就实现了循环传递参数的目的。

