本章`LyShark`将带大家学习如何在内核中使用标准的`Socket`套接字通信接口，我们都知道`Windows`应用层下可直接调用`WinSocket`来实现网络通信，但在内核模式下应用层API接口无法使用，内核模式下有一套专有的`WSK`通信接口，我们对WSK进行封装，让其与应用层调用规范保持一致，并实现内核与内核直接通过`Socket`通信的案例。

WSK（Windows Sockets Kernel）是一个Windows内核级别的网络通信API，用于在内核级别实现网络通信功能。WSK提供了一个抽象的网络协议栈，使开发者可以在内核中实现网络通信应用程序。

主要特点包括：

- 高性能：WSK在内核级别实现，避免了内核态和用户态之间的频繁切换，因此具有较高的性能。
- 可扩展性：WSK支持多个协议，包括TCP、UDP、IPv4和IPv6等，同时还支持自定义协议的实现。
- 安全性：WSK具有内核级别的权限，可以访问系统资源并执行一些高权限的操作，因此需要遵循严格的安全规范。
- 灵活性：WSK支持基于流和报文的数据传输，同时还提供了一些高级功能，例如安全传输、多播和多队列等。

WSK使用的编程模型与Winsock编程模型类似，开发者可以通过WSK API来创建、连接、发送和接收网络数据。同时，WSK还提供了一些高级功能，例如QoS（Quality of Service）和RSS（Receive Side Scaling）等。

该框架在网络安全、流量监控、网络加速和高性能网络通信等领域有着广泛的应用。例如，安全软件可以利用WSK在内核中实现网络过滤和流量监控功能；高性能服务器可以利用WSK在内核中实现网络通信，以提高网络传输效率和响应速度。

当然在早期如果需要实现网络通信一般都会采用`TDI`框架，但在新版本`Windows10`系统上虽然依然可以使用TDI接口，但是`LyShark`并不推荐使用，因为微软已经对接口搁置了，为了使WSK通信更加易用，我们需要封装内核层中的通信API，新建`MyWSK.hpp`头文件，该文件中封装了WSK通信API接口，其封装格式与应用层接口保持了高度一致，当需要在内核中使用Socket通信时可直接引入本文件。

我们需要使用`WDM`驱动程序，并配置以下参数。

- 配置属性 -> 连接器 -> 输入-> 附加依赖 -> $(DDK_LIB_PATH)\Netio.lib
- 配置属性 -> C/C++ -> 常规 -> 设置 警告等级2级 (警告视为错误关闭)

配置好以后，我们就开始吧，首先我们需要封装内核层中的通信API，我们新建`MyWSK.hpp`头文件，该文件中封装了WSK通信API接口，其封装格式与应用层接口保持了高度一致，当需要在内核中使用Socket通信时可直接引入本文件。

```
// 配置属性 -> 连接器 -> 输入-> 附加依赖 -> $(DDK_LIB_PATH)\Netio.lib
// 配置属性 -> C/C++ -> 常规 -> 设置 警告等级2级 (警告视为错误关闭)

#define NDIS_SUPPORT_NDIS6 1
#define SOCKET_ERROR -1
#define HTON_SHORT(n) (((((unsigned short)(n) & 0xFFu  )) << 8) | (((unsigned short)(n) & 0xFF00u) >> 8))
#define HTON_LONG(x)  (((((x)& 0xff)<<24) | ((x)>>24) & 0xff) | (((x) & 0xff0000)>>8) | (((x) & 0xff00)<<8))

#include <ndis.h>
#include <fwpmk.h>
#include <fwpsk.h>
#include <wsk.h>

static WSK_REGISTRATION     g_WskRegistration;
static WSK_PROVIDER_NPI     g_WskProvider;
static WSK_CLIENT_DISPATCH  g_WskDispatch = { MAKE_WSK_VERSION(1, 0), 0, NULL };

enum { DEINITIALIZED, DEINITIALIZING, INITIALIZING, INITIALIZED };
static LONG g_SocketsState = DEINITIALIZED;

static NTSTATUS NTAPI CompletionRoutine(__in PDEVICE_OBJECT DeviceObject, __in PIRP Irp,
__in PKEVENT CompletionEvent)
{
  ASSERT(CompletionEvent);

  UNREFERENCED_PARAMETER(Irp);
  UNREFERENCED_PARAMETER(DeviceObject);

  KeSetEvent(CompletionEvent, IO_NO_INCREMENT, FALSE);
  return STATUS_MORE_PROCESSING_REQUIRED;
}

static NTSTATUS InitWskData(__out PIRP* pIrp, __out PKEVENT CompletionEvent)
{
  ASSERT(pIrp);
  ASSERT(CompletionEvent);

  *pIrp = IoAllocateIrp(1, FALSE);
  if (!*pIrp)
  {
    return STATUS_INSUFFICIENT_RESOURCES;
  }

  KeInitializeEvent(CompletionEvent, SynchronizationEvent, FALSE);
  IoSetCompletionRoutine(*pIrp, (PIO_COMPLETION_ROUTINE)CompletionRoutine, CompletionEvent,
TRUE, TRUE, TRUE);
  return STATUS_SUCCESS;
}

static NTSTATUS InitWskBuffer(__in  PVOID Buffer, __in ULONG BufferSize, __out PWSK_BUF
WskBuffer)
{
  NTSTATUS Status = STATUS_SUCCESS;

  ASSERT(Buffer);
  ASSERT(BufferSize);
  ASSERT(WskBuffer);

  WskBuffer->Offset = 0;
  WskBuffer->Length = BufferSize;

  WskBuffer->Mdl = IoAllocateMdl(Buffer, BufferSize, FALSE, FALSE, NULL);
  if (!WskBuffer->Mdl)
  {
    return STATUS_INSUFFICIENT_RESOURCES;
  }
  __try
  {
    MmProbeAndLockPages(WskBuffer->Mdl, KernelMode, IoWriteAccess);
  }
  __except (EXCEPTION_EXECUTE_HANDLER)
  {
    IoFreeMdl(WskBuffer->Mdl);
    Status = STATUS_ACCESS_VIOLATION;
  }

  return Status;
}

static VOID FreeWskBuffer(__in PWSK_BUF WskBuffer)
{
  ASSERT(WskBuffer);
  MmUnlockPages(WskBuffer->Mdl);
  IoFreeMdl(WskBuffer->Mdl);
}

// 初始化网络通信库
NTSTATUS WSKStartup()
{
  WSK_CLIENT_NPI WskClient = { 0 };
  NTSTATUS Status = STATUS_UNSUCCESSFUL;

  if (InterlockedCompareExchange(&g_SocketsState, INITIALIZING, DEINITIALIZED) != DEINITIALIZED)
    return STATUS_ALREADY_REGISTERED;

  WskClient.ClientContext = NULL;
  WskClient.Dispatch = &g_WskDispatch;

  Status = WskRegister(&WskClient, &g_WskRegistration);

  // 注册失败则关闭
  if (!NT_SUCCESS(Status))
  {
    InterlockedExchange(&g_SocketsState, DEINITIALIZED);
    return Status;
  }

  Status = WskCaptureProviderNPI(&g_WskRegistration, WSK_NO_WAIT, &g_WskProvider);
  if (!NT_SUCCESS(Status))
  {
    WskDeregister(&g_WskRegistration);
    InterlockedExchange(&g_SocketsState, DEINITIALIZED);
    return Status;
  }

  InterlockedExchange(&g_SocketsState, INITIALIZED);
  return STATUS_SUCCESS;
}
```

```
// 释放网络通信库
VOID WSKCleanup()
{
  if (InterlockedCompareExchange(&g_SocketsState, INITIALIZED, DEINITIALIZING) !=
INITIALIZED)
    return;

  WskReleaseProviderNPI(&g_WskRegistration);
  WskDeregister(&g_WskRegistration);

  InterlockedExchange(&g_SocketsState, DEINITIALIZED);
}

// 创建套接字
PWSK_SOCKET NTAPI CreateSocket(__in ADDRESS_FAMILY AddressFamily, __in USHORT SocketType,
__in ULONG Protocol, __in ULONG Flags)
{
  KEVENT CompletionEvent = { 0 };
  PIRP Irp = NULL;
  PWSK_SOCKET WskSocket = NULL;
  NTSTATUS Status = STATUS_UNSUCCESSFUL;

  if (g_SocketsState != INITIALIZED)
  {
    return NULL;
  }

  Status = InitWskData(&Irp, &CompletionEvent);
  if (!NT_SUCCESS(Status))
  {
    return NULL;
  }

  Status = g_WskProvider.Dispatch->WskSocket(g_WskProvider.Client, AddressFamily,
SocketType, Protocol, Flags, NULL, NULL, NULL, NULL, NULL, Irp);

  if (Status == STATUS_PENDING)
  {
    KeWaitForSingleObject(&CompletionEvent, Executive, KernelMode, FALSE, NULL);
    Status = Irp->IoStatus.Status;
  }

  WskSocket = NT_SUCCESS(Status) ? (PWSK_SOCKET)Irp->IoStatus.Information : NULL;
  IoFreeIrp(Irp);
  return (PWSK_SOCKET)WskSocket;
}

// 关闭套接字
NTSTATUS NTAPI CloseSocket(__in PWSK_SOCKET WskSocket)
{
  KEVENT CompletionEvent = { 0 };
  PIRP Irp = NULL;
  NTSTATUS Status = STATUS_UNSUCCESSFUL;

  if (g_SocketsState != INITIALIZED || !WskSocket)
    return STATUS_INVALID_PARAMETER;

  Status = InitWskData(&Irp, &CompletionEvent);
  if (!NT_SUCCESS(Status))
  {
    return Status;
  }

  Status = ((PWSK_PROVIDER_BASIC_DISPATCH)WskSocket->Dispatch)->WskCloseSocket(WskSocket,
Irp);
  if (Status == STATUS_PENDING)
  {
    KeWaitForSingleObject(&CompletionEvent, Executive, KernelMode, FALSE, NULL);
    Status = Irp->IoStatus.Status;
  }

  IoFreeIrp(Irp);
  return Status;
}

// 连接套接字
NTSTATUS NTAPI Connect(__in PWSK_SOCKET WskSocket, __in PSOCKADDR RemoteAddress)
{
  KEVENT CompletionEvent = { 0 };
  PIRP Irp = NULL;
  NTSTATUS Status = STATUS_UNSUCCESSFUL;

  if (g_SocketsState != INITIALIZED || !WskSocket || !RemoteAddress)
    return STATUS_INVALID_PARAMETER;

  Status = InitWskData(&Irp, &CompletionEvent);
  if (!NT_SUCCESS(Status))
  {
    return Status;
  }

  Status = ((PWSK_PROVIDER_CONNECTION_DISPATCH)WskSocket->Dispatch)->WskConnect(WskSocket,
RemoteAddress, 0, Irp);
  if (Status == STATUS_PENDING)
  {
    KeWaitForSingleObject(&CompletionEvent, Executive, KernelMode, FALSE, NULL);
    Status = Irp->IoStatus.Status;
  }

  IoFreeIrp(Irp);
  return Status;
}

// 连接套接字
```

```
PWSK_SOCKET NTAPI SocketConnect(__in USHORT SocketType, __in ULONG Protocol, __in PSOCKADDR
RemoteAddress, __in PSOCKADDR LocalAddress)
{
  KEVENT CompletionEvent = { 0 };
  PIRP Irp = NULL;
  NTSTATUS Status = STATUS_UNSUCCESSFUL;
  PWSK_SOCKET WskSocket = NULL;

  if (g_SocketsState != INITIALIZED || !RemoteAddress || !LocalAddress)
    return NULL;

  Status = InitWskData(&Irp, &CompletionEvent);
  if (!NT_SUCCESS(Status))
  {
    return NULL;
  }

  Status = g_WskProvider.Dispatch->WskSocketConnect(g_WskProvider.Client, SocketType,
Protocol, LocalAddress, RemoteAddress, 0, NULL, NULL, NULL, NULL, NULL, Irp);

  if (Status == STATUS_PENDING)
  {
    KeWaitForSingleObject(&CompletionEvent, Executive, KernelMode, FALSE, NULL);
    Status = Irp->IoStatus.Status;
  }

  WskSocket = NT_SUCCESS(Status) ? (PWSK_SOCKET)Irp->IoStatus.Information : NULL;
  IoFreeIrp(Irp);
  return WskSocket;
}

// 发送数据
LONG NTAPI Send(__in PWSK_SOCKET WskSocket, __in PVOID Buffer, __in ULONG BufferSize, __in
ULONG Flags)
{
  KEVENT CompletionEvent = { 0 };
  PIRP Irp = NULL;
  WSK_BUF WskBuffer = { 0 };
  LONG BytesSent = SOCKET_ERROR;
  NTSTATUS Status = STATUS_UNSUCCESSFUL;

  if (g_SocketsState != INITIALIZED || !WskSocket || !Buffer || !BufferSize)
    return SOCKET_ERROR;

  Status = InitWskBuffer(Buffer, BufferSize, &WskBuffer);
  if (!NT_SUCCESS(Status))
  {
    return SOCKET_ERROR;
  }

  Status = InitWskData(&Irp, &CompletionEvent);
  if (!NT_SUCCESS(Status))
  {
    FreeWskBuffer(&WskBuffer);
    return SOCKET_ERROR;
  }

  Status = ((PWSK_PROVIDER_CONNECTION_DISPATCH)WskSocket->Dispatch)->WskSend(WskSocket,
&WskBuffer, Flags, Irp);
  if (Status == STATUS_PENDING)
  {
    KeWaitForSingleObject(&CompletionEvent, Executive, KernelMode, FALSE, NULL);
    Status = Irp->IoStatus.Status;
  }

  BytesSent = NT_SUCCESS(Status) ? (LONG)Irp->IoStatus.Information : SOCKET_ERROR;

  IoFreeIrp(Irp);
  FreeWskBuffer(&WskBuffer);
  return BytesSent;
}

// 发送全部数据
LONG NTAPI SendTo(__in PWSK_SOCKET WskSocket, __in PVOID Buffer, __in ULONG BufferSize,
__in_opt PSOCKADDR RemoteAddress)
{
  KEVENT CompletionEvent = { 0 };
  PIRP Irp = NULL;
  WSK_BUF WskBuffer = { 0 };
  LONG BytesSent = SOCKET_ERROR;
  NTSTATUS Status = STATUS_UNSUCCESSFUL;

  if (g_SocketsState != INITIALIZED || !WskSocket || !Buffer || !BufferSize)
    return SOCKET_ERROR;

  Status = InitWskBuffer(Buffer, BufferSize, &WskBuffer);
  if (!NT_SUCCESS(Status))
  {
    return SOCKET_ERROR;
  }

  Status = InitWskData(&Irp, &CompletionEvent);
  if (!NT_SUCCESS(Status))
  {
    FreeWskBuffer(&WskBuffer);
    return SOCKET_ERROR;
  }

  Status = ((PWSK_PROVIDER_DATAGRAM_DISPATCH)WskSocket->Dispatch)->WskSendTo(WskSocket,
&WskBuffer, 0, RemoteAddress, 0, NULL, Irp);
  if (Status == STATUS_PENDING)
  {
    KeWaitForSingleObject(&CompletionEvent, Executive, KernelMode, FALSE, NULL);
    Status = Irp->IoStatus.Status;
  }
  BytesSent = NT_SUCCESS(Status) ? (LONG)Irp->IoStatus.Information : SOCKET_ERROR;

  IoFreeIrp(Irp);
  FreeWskBuffer(&WskBuffer);
  return BytesSent;
}

// 接收数据
LONG NTAPI Receive(__in PWSK_SOCKET WskSocket, __out PVOID Buffer, __in ULONG BufferSize, __in ULONG Flags)
{
  KEVENT CompletionEvent = { 0 };
  PIRP Irp = NULL;
  WSK_BUF WskBuffer = { 0 };
  LONG BytesReceived = SOCKET_ERROR;
  NTSTATUS Status = STATUS_UNSUCCESSFUL;

  if (g_SocketsState != INITIALIZED || !WskSocket || !Buffer || !BufferSize)
    return SOCKET_ERROR;

  Status = InitWskBuffer(Buffer, BufferSize, &WskBuffer);
  if (!NT_SUCCESS(Status))
  {
    return SOCKET_ERROR;
  }

  Status = InitWskData(&Irp, &CompletionEvent);
  if (!NT_SUCCESS(Status))
  {
    FreeWskBuffer(&WskBuffer);
    return SOCKET_ERROR;
  }

  Status = ((PWSK_PROVIDER_CONNECTION_DISPATCH)WskSocket->Dispatch)->WskReceive(WskSocket, &WskBuffer, Flags, Irp);
  if (Status == STATUS_PENDING)
  {
    KeWaitForSingleObject(&CompletionEvent, Executive, KernelMode, FALSE, NULL);
    Status = Irp->IoStatus.Status;
  }

  BytesReceived = NT_SUCCESS(Status) ? (LONG)Irp->IoStatus.Information : SOCKET_ERROR;

  IoFreeIrp(Irp);
  FreeWskBuffer(&WskBuffer);
  return BytesReceived;
}

// 接受全部数据
LONG NTAPI ReceiveFrom(__in PWSK_SOCKET WskSocket, __out PVOID Buffer, __in ULONG BufferSize, __out_opt PSOCKADDR RemoteAddress, __out_opt PULONG ControlFlags)
{
  KEVENT CompletionEvent = { 0 };
  PIRP Irp = NULL;
  WSK_BUF WskBuffer = { 0 };
  LONG BytesReceived = SOCKET_ERROR;
  NTSTATUS Status = STATUS_UNSUCCESSFUL;

  if (g_SocketsState != INITIALIZED || !WskSocket || !Buffer || !BufferSize)
    return SOCKET_ERROR;

  Status = InitWskBuffer(Buffer, BufferSize, &WskBuffer);
  if (!NT_SUCCESS(Status))
  {
    return SOCKET_ERROR;
  }

  Status = InitWskData(&Irp, &CompletionEvent);
  if (!NT_SUCCESS(Status))
  {
    FreeWskBuffer(&WskBuffer);
    return SOCKET_ERROR;
  }

  Status = ((PWSK_PROVIDER_DATAGRAM_DISPATCH)WskSocket->Dispatch)->WskReceiveFrom(WskSocket, &WskBuffer, 0, RemoteAddress, 0, NULL, ControlFlags, Irp);
  if (Status == STATUS_PENDING)
  {
    KeWaitForSingleObject(&CompletionEvent, Executive, KernelMode, FALSE, NULL);
    Status = Irp->IoStatus.Status;
  }

  BytesReceived = NT_SUCCESS(Status) ? (LONG)Irp->IoStatus.Information : SOCKET_ERROR;

  IoFreeIrp(Irp);
  FreeWskBuffer(&WskBuffer);
  return BytesReceived;
}

// 绑定套接字
NTSTATUS NTAPI Bind(__in PWSK_SOCKET WskSocket, __in PSOCKADDR LocalAddress)
{
  KEVENT CompletionEvent = { 0 };
  PIRP Irp = NULL;
  NTSTATUS Status = STATUS_UNSUCCESSFUL;

  if (g_SocketsState != INITIALIZED || !WskSocket || !LocalAddress)
    return STATUS_INVALID_PARAMETER;

  Status = InitWskData(&Irp, &CompletionEvent);
  if (!NT_SUCCESS(Status))
  {
    return Status;
  }
  Status = ((PWSK_PROVIDER_CONNECTION_DISPATCH)WskSocket->Dispatch)->WskBind(WskSocket,
LocalAddress, 0, Irp);
  if (Status == STATUS_PENDING)
  {
    KeWaitForSingleObject(&CompletionEvent, Executive, KernelMode, FALSE, NULL);
    Status = Irp->IoStatus.Status;
  }

  IoFreeIrp(Irp);
  return Status;
}

// 等待响应
PWSK_SOCKET NTAPI Accept(__in PWSK_SOCKET WskSocket, __out_opt PSOCKADDR LocalAddress,
__out_opt PSOCKADDR RemoteAddress)
{
  KEVENT CompletionEvent = { 0 };
  PIRP Irp = NULL;
  NTSTATUS Status = STATUS_UNSUCCESSFUL;
  PWSK_SOCKET AcceptedSocket = NULL;

  if (g_SocketsState != INITIALIZED || !WskSocket)
    return NULL;

  Status = InitWskData(&Irp, &CompletionEvent);
  if (!NT_SUCCESS(Status))
  {
    return NULL;
  }

  Status = ((PWSK_PROVIDER_LISTEN_DISPATCH)WskSocket->Dispatch)->WskAccept(WskSocket, 0,
NULL, NULL, LocalAddress, RemoteAddress, Irp);
  if (Status == STATUS_PENDING)
  {
    KeWaitForSingleObject(&CompletionEvent, Executive, KernelMode, FALSE, NULL);
    Status = Irp->IoStatus.Status;
  }

  AcceptedSocket = NT_SUCCESS(Status) ? (PWSK_SOCKET)Irp->IoStatus.Information : NULL;

  IoFreeIrp(Irp);
  return AcceptedSocket;
}

// IP地址转为网络字节序
long change_uint(long a, long b, long c, long d)
{
  long address = 0;
  address |= d << 24;
  address |= c << 16;
  address |= b << 8;
  address |= a;
  return address;
```

```
}
```

对于`服务端`来说，驱动通信必须保证服务端开启多线程来处理异步请求，不然驱动加载后系统会处于等待状态，而一旦等待则系统将会卡死，那么对于服务端`DriverEntry`入口说我们不能让其等待，必须使用`PsCreateSystemThread`来启用系统线程，该函数属于WDM的一部分，官方定义如下；

```
NTSTATUS PsCreateSystemThread(
  [out]           PHANDLE            ThreadHandle,
  [in]            ULONG              DesiredAccess,
  [in, optional]  POBJECT_ATTRIBUTES ObjectAttributes,
  [in, optional]  HANDLE             ProcessHandle,
  [out, optional] PCLIENT_ID         ClientId,
  [in]            PKSTART_ROUTINE    StartRoutine,
  [in, optional]  PVOID              StartContext
);
```

我们使用`PsCreateSystemThread`函数开辟线程`TcpListenWorker`在线程内部执行如下流程启动驱动服务端，由于我们自己封装实现了标准接口组，所以使用起来几乎与应用层无任何差异了。

- CreateSocket 创建套接字
- Bind 绑定套接字
- Accept 等待接收请求
- Receive 用于接收返回值
- Send 用于发送返回值

```
#include <MyWSK.hpp>

PETHREAD m_EThread = NULL;

// 线程函数
VOID TcpListenWorker(PVOID Context)
{
  WSK_SOCKET* paccept_socket = NULL;
  SOCKADDR_IN LocalAddress = { 0 };
  SOCKADDR_IN RemoteAddress = { 0 };
  NTSTATUS status = STATUS_UNSUCCESSFUL;

  // 创建套接字
  PWSK_SOCKET TcpSocket = CreateSocket(AF_INET, SOCK_STREAM, IPPROTO_TCP,
WSK_FLAG_LISTEN_SOCKET);
  if (TcpSocket == NULL)
  {
    return;
  }

  // 设置绑定地址
  LocalAddress.sin_family = AF_INET;
  LocalAddress.sin_addr.s_addr = INADDR_ANY;
  LocalAddress.sin_port = HTON_SHORT(8888);
  status = Bind(TcpSocket, (PSOCKADDR)&LocalAddress);
  if (!NT_SUCCESS(status))
  {
    return;
  }

  // 循环接收
  while (1)
  {
    CHAR* read_buffer = (CHAR*)ExAllocatePoolWithTag(NonPagedPool, 2048, "read");
    paccept_socket = Accept(TcpSocket, (PSOCKADDR)&LocalAddress, (PSOCKADDR)&RemoteAddress);
    if (paccept_socket == NULL)
    {
      continue;
    }

    // 接收数据
    memset(read_buffer, 0, 2048);
    int read_len = Receive(paccept_socket, read_buffer, 2048, 0);
    if (read_len != 0)
    {
      DbgPrint("[内核A] => %s \n", read_buffer);

      // 发送数据
      char send_buffer[2048] = "Hi, Kernel B";
      Send(paccept_socket, send_buffer, strlen(send_buffer), 0);

      // 接收确认包
      memset(read_buffer, 0, 2048);
      Receive(paccept_socket, read_buffer, 2, 0);
    }

    // 清理堆
    if (read_buffer != NULL)
    {
      ExFreePool(read_buffer);
    }

    // 关闭当前套接字
    if (paccept_socket)
    {
      CloseSocket(paccept_socket);
    }
  }

  if (TcpSocket)
  {
    CloseSocket(TcpSocket);
  }
  PsTerminateSystemThread(STATUS_SUCCESS);
  return;
}
```

```
// 关闭套接字
VOID UnDriver(PDRIVER_OBJECT driver)
{
  WSKCleanup();
  KeWaitForSingleObject(m_EThread, Executive, KernelMode, FALSE, NULL);
  if (m_EThread != NULL)
  {
    ObDereferenceObject(m_EThread);
  }
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
  // 初始化
  WSKStartup();

  HANDLE hThread = NULL;
  NTSTATUS status = STATUS_UNSUCCESSFUL;

  // 创建系统线程
  status = PsCreateSystemThread(&hThread, THREAD_ALL_ACCESS, NULL, NULL, NULL,
TcpListenWorker, NULL);
  if (!NT_SUCCESS(status))
  {
    return status;
  }

  // 获取线程EProcess结构
  status = ObReferenceObjectByHandle(hThread, THREAD_ALL_ACCESS, NULL, KernelMode,
(PVOID*)&m_EThread, NULL);
  if (NT_SUCCESS(status) == FALSE)
  {
    return status;
  }

  ZwClose(hThread);
  Driver->DriverUnload = UnDriver;
  return STATUS_SUCCESS;
}
```

对于客户端来说，只需要创建套接字并连接到指定地址即可，这个过程大体上可以总结为如下；

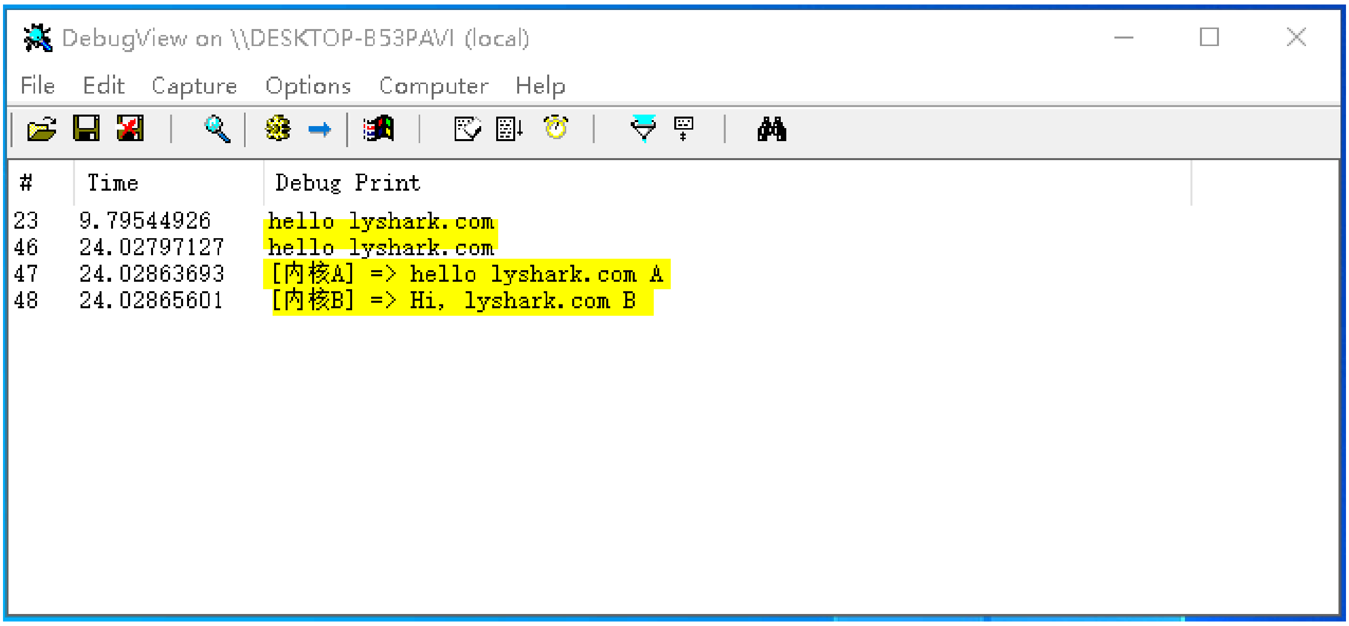
- CreateSocket 创建套接字
- Bind 绑定套接字
- Connect 链接服务端驱动
- Send 发送数据到服务端
- Receive 接收数据到服务端

```
#include <MyWSK.hpp>

VOID UnDriver(PDRIVER_OBJECT driver)
```

```
{
  // 卸载并关闭Socket库
  WSKCleanup();
}

NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver, PUNICODE_STRING RegistryPath)
{
  // 初始化
  WSKStartup();

  NTSTATUS      status = STATUS_SUCCESS;
  SOCKADDR_IN   LocalAddress = { 0, };
  SOCKADDR_IN   RemoteAddress = { 0, };

  // 创建套接字
  PWSK_SOCKET TcpSocket = CreateSocket(AF_INET, SOCK_STREAM, IPPROTO_TCP,
WSK_FLAG_CONNECTION_SOCKET);
  if (TcpSocket == NULL)
  {
    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
  }

  LocalAddress.sin_family = AF_INET;
  LocalAddress.sin_addr.s_addr = INADDR_ANY;
  status = Bind(TcpSocket, (PSOCKADDR)&LocalAddress);

  // 绑定失败则关闭驱动
  if (!NT_SUCCESS(status))
  {
    CloseSocket(TcpSocket);

    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
  }

  // 初始化服务端地址与端口信息
  ULONG address[4] = { 127, 0, 0, 1 };

  RemoteAddress.sin_family = AF_INET;
  RemoteAddress.sin_addr.s_addr = change_uint(address[0], address[1], address[2],
address[3]);
  RemoteAddress.sin_port = HTON_SHORT(8888);

  status = Connect(TcpSocket, (PSOCKADDR)&RemoteAddress);

  // 连接服务端,如果失败则关闭驱动
  if (!NT_SUCCESS(status))
  {
    CloseSocket(TcpSocket);
    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
  }
```

```
    // 发送数据
    char send_buffer[2048] = "hello Kernel A";
    Send(TcpSocket, send_buffer, strlen(send_buffer), 0);

    // 接收数据
    CHAR* read_buffer = (CHAR*)ExAllocatePoolWithTag(NonPagedPool, 2048, "read");

    memset(read_buffer, 0, 1024);
    Receive(TcpSocket, read_buffer, 2048, 0);
    DbgPrint("[内核B] => %s \n", read_buffer);

    // 发送确认包
    Send(TcpSocket, "ok", 2, 0);

    // 释放内存
    ExFreePool(read_buffer);
    CloseSocket(TcpSocket);
    Driver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

编译两个驱动程序，首先运行`server.sys`驱动，运行后该驱动会在后台等待客户端连接，接着运行`client.sys`屏幕上可输出如下提示，说明通信已经建立了。


DebugView on \\DESKTOP-B53PAVI (local)
File Edit Capture Options Computer Help

```
#   Time         Debug Print
23  9.79544926   hello lyshark.com
46  24.02797127  hello lyshark.com
47  24.02863693  [内核A] => hello lyshark.com A
48  24.02865601  [内核B] => Hi, lyshark.com B
```