本章主要介绍如何实现内核驱动和应用层之间的通信。通信是驱动开发中不可或缺的一部分，通过通信可以实现驱动和应用层之间的数据交换和控制命令传递等功能。

本章将首先介绍驱动如何与应用层简单通信，包括应用DeviceIoContro通信模板、通过SystemBuf实现通信以及通过ReadFile实现与内核通信等。然后，本章将介绍如何通过PIPE管道与内核通信，以及使用Async异步通信实现更高效的数据传输。

为了实现连续通信，本章还将介绍如何通过MDL映射实现内存映射，以及如何基于事件同步实现反向通信。

通过学习本章的内容，读者将了解如何在驱动程序和应用层之间建立有效的通信，以及如何选择最适合的通信方式。同时，本章所介绍的内容也是深入学习驱动开发必不可少的一部分，为后续章节的内容打下了坚实的基础。

先来简单介绍一下 `IRP(I/O Request Package)` 输入输出请求包，该请求包在Windows内核中是一个非常重要的数据结构，当我们的上层应用与底层的驱动程序通信时，应用程序就会发出 `I/O` 请求，操作系统将该请求转化为相应的 `IRP` 数据，然后会根据不同的请求数据将请求派遣到相应的驱动函数中执行，这一点有点类似于Windows的消息机制。

IRP被用来表示 `输入/输出(I/O)` 请求，并在内核模式下传递给驱动程序进行处理，当应用程序需要进行 `I/O` 操作时，它会向操作系统发出请求。操作系统会将请求转化为IRP数据结构，并在 `内核模式` 下对其进行处理。在处理IRP时，操作系统会根据不同的请求类型将IRP派遣到相应的驱动程序中执行。

在驱动程序中处理IRP时，可以使用一系列的回调函数来对其进行处理。这些回调函数包括以下几个：

- DriverEntry：驱动程序的入口点，它会在驱动程序加载时被调用。
- AddDevice：用于为驱动程序的每个设备对象添加一个设备扩展。
- DispatchXxx：一组用于处理不同类型I/O请求的回调函数，其中Xxx代表请求类型，例如读请求是DispatchRead，写请求是DispatchWrite等。
- Unload：驱动程序卸载时被调用。

通过使用这些回调函数，驱动程序可以对不同类型的 `I/O` 请求进行处理，并执行相应的操作，例如读写数据，打开或关闭设备等。

DispatchXxx 派遣函数 `(Dispatch Function)` 是一组函数，它们负责处理 `I/O` 请求，并将请求转发给相应的处理程序。

当应用程序需要进行 `I/O` 操作时，它会向操作系统发出请求。操作系统会将请求转化为一个 `IRP(I/O Request Packet)` 数据结构，并在内核模式下对其进行处理。在处理IRP时，操作系统会根据不同的请求类型将IRP派遣到相应的派遣函数中执行。

派遣函数是驱动程序中用来处理 `I/O` 请求的核心函数，一般情况下，驱动程序通过实现一组派遣函数来处理 `I/O` 请求。这些派遣函数包括：

- IRP_MJ_CREATE：用于处理创建请求。
- IRP_MJ_READ：用于处理读请求。
- IRP_MJ_WRITE：用于处理写请求。
- IRP_MJ_DEVICE_CONTROL：用于处理设备控制请求等。

当操作系统将IRP派遣到相应的派遣函数中执行时，驱动程序会根据请求类型调用相应的派遣函数来处理请求。派遣函数可以通过访问IRP中的数据结构来获取请求的信息，并执行相应的操作。例如，当收到读请求时，派遣函数可以从IRP中获取读取的数据长度和读取的起始位置，并将相应的数据从设备中读取出来。

需要注意的是，派遣函数在执行时需要遵循内核模式的安全性规则，避免在驱动程序中出现安全漏洞等。同时，由于派遣函数是驱动程序中的核心函数，因此在设计和实现时需要考虑性能和可维护性等方面的问题。

那么该如何创建一个具有通信能力的驱动程序呢，下面是一个简单的驱动程序创建流程：

- 1.使用`IoCreateDevice`函数创建设备对象。设备对象用于表示驱动程序控制的硬件设备，并提供与设备的通信。函数需要传入设备对象的名称、设备类型、设备特征等参数，并返回一个设备对象指针。
- 2.对于某些驱动程序，需要创建符号链接对象。符号链接对象用于允许应用程序通过名称访问设备对象。使用`IoCreateSymbolicLink`函数创建符号链接对象，该函数需要传入符号链接的名称和设备对象指针。
- 3.在驱动程序的`DriverEntry`函数中，使用`IoRegisterShutdownNotification`函数注册一个回调函数，以便在系统关闭时处理必要的清理操作。
- 4.在驱动程序的`DriverUnload`函数中，释放设备对象和符号链接对象，并进行必要的资源清理操作。

如下代码我们简单的创建实现了一个带有通信功能的驱动程序，并注册了创建和关闭派遣函数。下面对代码进行分析：

- 驱动程序入口函数`DriverEntry`中，调用了`CreateDriverObject`函数创建设备对象和符号链接对象。然后，注册了`IRP_MJ_CREATE`和`IRP_MJ_CLOSE`派遣函数，分别对应设备的创建和关闭。最后，将驱动卸载函数设置为`UnDriver`。
- CreateDriverObject函数中，使用`IoCreateDevice`函数创建设备对象，并使用`IoCreateSymbolicLink`函数创建符号链接对象。然后将设备对象标记为`DO_BUFFERED_IO`，这意味着将使用缓冲区读写数据。
- DispatchCreate和DispatchClose函数分别对应设备的创建和关闭，这里简单地设置返回状态为`STATUS_SUCCESS`，并使用`DbgPrint`函数输出调试信息。然后，使用`IoCompleteRequest`函数指示完成此IRP。
- UnDriver函数是卸载驱动程序时执行的函数。在这里，首先使用`IoDeleteDevice`函数删除设备对象，然后使用`IoDeleteSymbolicLink`函数删除符号链接对象。最后，使用`DbgPrint`函数输出调试信息。

```
#include <ntddk.h>

VOID UnDriver(PDRIVER_OBJECT pDriver)
{
    PDEVICE_OBJECT pDev;          // 用来取得要删除设备对象
    UNICODE_STRING SymLinkName;   // 局部变量symLinkName

    pDev = pDriver->DeviceObject;
    IoDeleteDevice(pDev);                                          // 调用IoDeleteDevice用于删除
设备
    RtlInitUnicodeString(&SymLinkName, L"\\??\\My_Driver");      // 初始化字符串将symLinkName定
义成需要删除的符号链接名称
    IoDeleteSymbolicLink(&SymLinkName);                            // 调用IoDeleteSymbolicLink删
除符号链接
    DbgPrint("删除设备与符号链接成功...");
}

NTSTATUS DispatchCreate(PDEVICE_OBJECT pDevObj, PIRP pIrp)
{
    pIrp->IoStatus.Status = STATUS_SUCCESS;           // 返回成功
    DbgPrint("派遣函数 IRP_MJ_CREATE 成功执行 !\n");
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);         // 指示完成此IRP
    return STATUS_SUCCESS;                            // 返回成功
```

```
}

NTSTATUS DispatchClose(PDEVICE_OBJECT pDevObj, PIRP pIrp)
{
    pIrp->IoStatus.Status = STATUS_SUCCESS;           // 返回成功
    DbgPrint("派遣函数 IRP_MJ_CLOSE 成功执行 !\n");
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);         // 指示完成此IRP
    return STATUS_SUCCESS;                            // 返回成功

}

NTSTATUS CreateDriverObject(IN PDRIVER_OBJECT pDriver)
{
    NTSTATUS Status;
    PDEVICE_OBJECT pDevObj;
    UNICODE_STRING DriverName;
    UNICODE_STRING SymLinkName;

    RtlInitUnicodeString(&DriverName, L"\\Device\\My_Device");
    Status = IoCreateDevice(pDriver, 0, &DriverName, FILE_DEVICE_UNKNOWN, 0, TRUE,
&pDevObj);
    DbgPrint("命令 IoCreateDevice 状态: %d", Status);

    // DO_BUFFERED_IO 设置读写方式 Flags的三个不同的值分别为：DO_BUFFERED_IO、DO_DIRECT_IO和0
    pDevObj->Flags |= DO_BUFFERED_IO;
    RtlInitUnicodeString(&SymLinkName, L"\\??\\My_Device");
    Status = IoCreateSymbolicLink(&SymLinkName, &DriverName);
    DbgPrint("当前命令IoCreateSymbolicLink状态: %d", Status);
    return STATUS_SUCCESS;
}

NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING RegistryPath)
{
    CreateDriverObject(pDriver);            // 调用创建设备子过程
    // 注册两个派遣函数,分别对应创建与关闭,派遣函数名可自定义
    pDriver->MajorFunction[IRP_MJ_CREATE] = DispatchCreate;    // 创建成功派遣函数
    pDriver->MajorFunction[IRP_MJ_CLOSE] = DispatchClose;      // 关闭派遣函数

    DbgPrint("驱动加载完成...");
    pDriver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

需要注意的是，在实际开发中，驱动程序需要更多的代码来处理设备的读写、控制和中断处理等问题，此处只是提供一个简单的框架供参考。同时，驱动程序的开发需要具备一定的底层编程和操作系统知识，需要小心谨慎地进行开发和调试。

对于客户端而言，用户需要对驱动进行控制并传递参数，如下程序调用`CreateFile`函数来打开设备句柄。其中参数`\\.\My_Device`表示设备的符号链接名，`GENERIC_READ`和`GENERIC_WRITE`分别表示读和写的权限，其他参数为`0`、`NULL`和`OPEN_EXISTING`则是常见的默认值。如果`CreateFile`函数返回`INVALID_HANDLE_VALUE`，表示获取设备句柄失败，程序会输出错误信息并等待输入后退出。最后调用`CloseHandle`函数关闭设备句柄，防止资源泄漏。

```
#include <windows.h>
#include <stdio.h>
#include <winioctl.h>

int main()
{
    HANDLE hDevice = CreateFile(L"\\\\.\\My_Device", GENERIC_READ | GENERIC_WRITE, 0, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hDevice == INVALID_HANDLE_VALUE) //判断hDevice返回值是否为空
    {
        printf("获取驱动句柄失败!错误: %d\n", GetLastError());
        getchar();
    }

    getchar();
    CloseHandle(hDevice);
    return 0;
}
```

需要注意的是，这里的设备名`\\.\My_Device`需要与驱动程序中的设备名保持一致，否则无法打开设备句柄。此外，程序只打开了设备句柄并关闭，没有进行读写操作。如果需要进行读写操作，可以调用`ReadFile`和`WriteFile`函数。

我们以实现读取内核缓冲区中的数据为例，实现一个简单的数据通信案例，回到程序中当驱动通过`CreateDriverObject`创建好对象以后，下一步则通过调用`pDriver->MajorFunction[IRP_MJ_READ] = DispatchRead;`来初始化一个派遣函数，当接收到客户端的`IRP_MJ_READ`读取请求时，则使用`DispatchRead`函数对该请求进行处理，而这个函数中所执行的操作可以概括为如下五个步骤；

1.首先，函数获取了 `IRP` 栈的当前位置，这是一个包含 `IRP` 相关参数和操作的数据结构。

```
PIO_STACK_LOCATION Stack = IoGetCurrentIrpStackLocation(pIrp);
```

2.然后，获取了应用程序请求读取的数据长度。

```
ULONG ulReadLength = Stack->Parameters.Read.Length;
```

3.接下来，函数将读取操作的状态设置为成功，并将读取的数据长度作为信息存储在 `IRP` 的 `I/O` 状态块中，以供稍后处理。

```
NTSTATUS Status = STATUS_SUCCESS;
pIrp->IoStatus.Status = Status;
pIrp->IoStatus.Information = ulReadLength;
```

4.代码使用 `memset()` 函数将 `IRP` 中的关联缓冲区全部填充为 `0x68`，以便演示读取的效果。

```
memset(pIrp->AssociatedIrp.SystemBuffer, 0x68, ulReadLength);
```

5.最后，代码调用 `IoCompleteRequest()` 函数，表示读取操作已完成，并将 `IRP` 返回给操作系统。

```
IoCompleteRequest(pIrp, IO_NO_INCREMENT);
return Status;
```

总体来说，这个函数的作用是响应应用程序的读取请求并向应用程序返回一定数量的填充数据，这段驱动程序完整代码如下所示；

```
#include <ntddk.h>

VOID UnDriver(PDRIVER_OBJECT pDriver)
{
    PDEVICE_OBJECT pDev;          // 用来取得要删除设备对象
    UNICODE_STRING SymLinkName;   // 局部变量symLinkName
    pDev = pDriver->DeviceObject;
    IoDeleteDevice(pDev);                                          // 调用IoDeleteDevice用于删除设备
    RtlInitUnicodeString(&SymLinkName, L"\\??\\My_Driver");      // 初始化字符串将symLinkName定义成需要删除的符号链接名称
    IoDeleteSymbolicLink(&SymLinkName);                            // 调用IoDeleteSymbolicLink删除符号链接
    DbgPrint("删除设备与符号链接成功...");
}
NTSTATUS DispatchCreate(PDEVICE_OBJECT pDevObj, PIRP pIrp)
{
    pIrp->IoStatus.Status = STATUS_SUCCESS;           // 返回成功
    DbgPrint("派遣函数 IRP_MJ_CREATE 成功执行 !\n");
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);         // 指示完成此IRP
    return STATUS_SUCCESS;                            // 返回成功
}
NTSTATUS DispatchClose(PDEVICE_OBJECT pDevObj, PIRP pIrp)
{
    pIrp->IoStatus.Status = STATUS_SUCCESS;           // 返回成功
    DbgPrint("派遣函数 IRP_MJ_CLOSE 成功执行 !\n");
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);         // 指示完成此IRP
    return STATUS_SUCCESS;                            // 返回成功
}

NTSTATUS DispatchRead(PDEVICE_OBJECT pDevObj, PIRP pIrp)
{
    NTSTATUS Status = STATUS_SUCCESS;
    PIO_STACK_LOCATION Stack = IoGetCurrentIrpStackLocation(pIrp);
    ULONG ulReadLength = Stack->Parameters.Read.Length;
    pIrp->IoStatus.Status = Status;
    pIrp->IoStatus.Information = ulReadLength;
    DbgPrint("应用要读取的长度：%d\n", ulReadLength);

    // 将内核中的缓冲区全部填充为0x68 方便演示读取的效果
    memset(pIrp->AssociatedIrp.SystemBuffer, 0x68, ulReadLength);
    IoCompleteRequest(pIrp, IO_NO_INCREMENT);
    return Status;
}

NTSTATUS CreateDriverObject(IN PDRIVER_OBJECT pDriver)
```

```
{
    NTSTATUS Status;
    PDEVICE_OBJECT pDevObj;
    UNICODE_STRING DriverName;
    UNICODE_STRING SymLinkName;

    RtlInitUnicodeString(&DriverName, L"\\Device\\My_Device");
    Status = IoCreateDevice(pDriver, 0, &DriverName, FILE_DEVICE_UNKNOWN, 0, TRUE,
&pDevObj);
    DbgPrint("命令 IoCreateDevice 状态: %d", Status);
    pDevObj->Flags |= DO_BUFFERED_IO;
    RtlInitUnicodeString(&SymLinkName, L"\\??\\My_Device");
    Status = IoCreateSymbolicLink(&SymLinkName, &DriverName);
    DbgPrint("当前命令IoCreateSymbolicLink状态: %d", Status);
    return STATUS_SUCCESS;
}

NTSTATUS DriverEntry(PDRIVER_OBJECT pDriver, PUNICODE_STRING RegistryPath)
{
    CreateDriverObject(pDriver);                                    // 调用创建设备
    pDriver->MajorFunction[IRP_MJ_CREATE] = DispatchCreate;     // 创建成功派遣函数
    pDriver->MajorFunction[IRP_MJ_CLOSE] = DispatchClose;       // 关闭派遣函数
    pDriver->MajorFunction[IRP_MJ_READ] = DispatchRead;

    DbgPrint("驱动加载完成...");
    pDriver->DriverUnload = UnDriver;
    return STATUS_SUCCESS;
}
```

客户端代码的实现与前面所提到的通信方式一致，此处我们依然使用`CreateFile`创建一个通信句柄，当需要读取数据时客户端只需要调用 `ReadFile` 函数读取从驱动程序设备对象发送的数据。函数的第一个参数是之前创建的句柄，第二个参数是缓冲区地址，第三个参数是要读取的字节数，第四个参数是实际读取的字节数，第五个参数为NULL表示不使用 `overlapped` 结构体。如果读取失败， `GetLastError()` 函数将返回错误代码。

```
#include <windows.h>
#include <stdio.h>
#include <winioctl.h>

int main()
{
    HANDLE hDevice = CreateFile(L"\\\\.\\My_Device", GENERIC_READ | GENERIC_WRITE, 0, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hDevice == INVALID_HANDLE_VALUE)
    {
        printf("获取驱动句柄失败: %d\n", GetLastError());
        getchar();
    }

    UCHAR buffer[10];
    ULONG ulRead;

    ReadFile(hDevice, buffer, 10, &ulRead, 0);
```

```
    for (int i = 0; i < (int)ulRead; i++)
    {
        printf("%02X", buffer[i]);
    }
    getchar();
    CloseHandle(hDevice);
    return 0;
}
```