**《寒江独钓-Windows内核安全编程》**

**《Windows驱动开发技术详解》读书笔记**

# 0概述

作者：keenjin（金才）

时间：2017/9/5

计划阅读：**3**个月

截止日期：2017/12/5

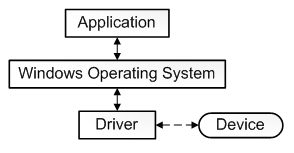
# 1驱动原理简介

源自微软官方文档《Windows驱动程序入门》：<https://docs.microsoft.com/zh-cn/windows-hardware/drivers/gettingstarted/index>

## 1.1什么是驱动程序

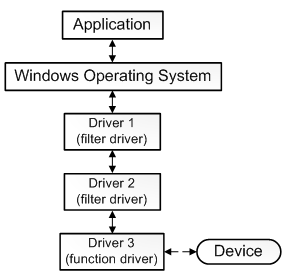
### 1.1.1函数型驱动

应用程序与设备通信，通常由硬件厂商完成设计制作，如显卡、CPU主板等。但是，有一些是硬件统一规范的，由微软实现，如硬盘、键盘等



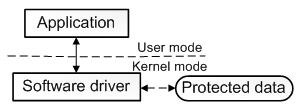
### 1.1.2过滤型驱动（筛选器驱动）

并不与设备通信，只作为堆栈分层，执行辅助处理前置过滤处理的作用，如磁盘过滤驱动、文件系统过滤驱动、网络过滤驱动等



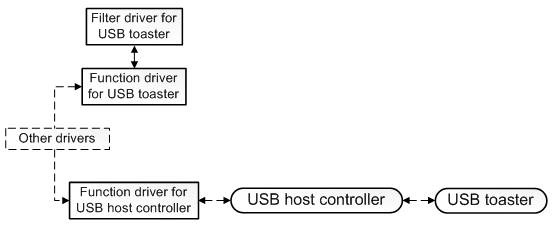
### 1.1.3软件驱动

与设备无关，它一般用于访问核心操作系统数据结构，且这些数据结构只能由内核模式下运行的代码访问，如读写MBR等



### 1.1.4总线驱动

设备并未直接连接到PCI总线，而是通过中转连接，如USB驱动



## 1.2是否需要编写驱动程序

是否需要编写，取决于微软是否已经支持。

### 1.2.1 USB类驱动程序

详情见：<https://msdn.microsoft.com/library/windows/hardware/ff538820>

|  |  |  |
| --- | --- | --- |
| **USB-IF类代码** | **设备安装程序类** | **Microsoft 提供的驱动程序和 INF** |
| **音频 (01h)** | **媒体**  {4d36e96c-e325-11ce-bfc1-08002be10318} | Usbaudio.sys  Wdma\_usb.inf |
| **通信和 CDC 控件 (02h)** | **端口**  {4D36E978-E325-11CE-BFC1-08002BE10318} | Usbser.sys  Usbser.inf |
| **调制解调器**  {4D36E96D-E325-11CE-BFC1-08002BE10318}  注意  支持子类 02h (ACM) | Usbser.sys  自定义引用 mdmcpq.inf 的 INF |
| **Net**  {4d36e972-e325-11ce-bfc1-08002be10318}  注意  支持子类 0Eh (MBIM) | wmbclass.sys  Netwmbclass.inf |
| **HID（人体学接口设备） (03h)** | **HIDClass**  {745a17a0-74d3-11d0-b6fe-00a0c90f57da} | Hidclass.sys  Hidusb.sys  Input.inf |
| **物理设备 (05h)** |  | 建议的驱动程序：WinUSB (Winusb.sys) |
| **图像 (06h)** | **图像**  {6bdd1fc6-810f-11d0-bec7-08002be2092f} | Usbscan.sys  Sti.inf |
| **打印机 (07h)** | **USB**  注意  Usbprint.sys 在设备安装程序类下枚举打印机设备：Printer{4d36e979-e325-11ce-bfc1-08002be10318}。 | Usbprint.sys  Usbprint.inf |
| **大容量存储 (08h)** | **USB** | Usbstor.sys |
| **SCSIAdapter**  {4d36e97b-e325-11ce-bfc1-08002be10318} | 子类 (06) 和协议 (62)  Uaspstor.sys  Uaspstor.inf |
| **集线器 (09h)** | **USB**  {36fc9e60-c465-11cf-8056-444553540000} | Usbhub.sys  Usb.inf  Usbhub3.sys  Usbhub3.inf |
| **CDC 数据 (0Ah)** |  | 建议的驱动程序：WinUSB (Winusb.sys) |
| **智能卡 (0Bh)** | **SmartCardReader**  {50dd5230-ba8a-11d1-bf5d-0000f805f530} | Usbccid.sys（已过时）  WUDFUsbccidDriver.dll  WUDFUsbccidDriver.inf |
| **内容安全性 (0Dh)** |  | 建议的驱动程序：USB 通用父驱动程序 (Usbccgp.sys) |
| **视频 (0Eh)** | **图像**  {6bdd1fc6-810f-11d0-bec7-08002be2092f} | Usbvideo.sys  Usbvideo.inf |
| **个人医疗 (0Fh)** |  | 建议的驱动程序：[WinUSB (Winusb.sys)](https://msdn.microsoft.com/zh-cn/library/windows/hardware/ff540196) |
| **音频/视频设备 (10h)** |  |  |
| **诊断设备 (DCh)** |  | 建议的驱动程序：[WinUSB (Winusb.sys)](https://msdn.microsoft.com/zh-cn/library/windows/hardware/ff540196) |
| **无线控制器 (E0h)**  **注意  支持子类 01h 和协议 01h** | **蓝牙**  {e0cbf06c-cd8b-4647-bb8a-263b43f0f974} | Bthusb.sys  Bth.inf |
| **其他 (EFh)** | **Net**  {4d36e972-e325-11ce-bfc1-08002be10318}  注意  支持子类 04h 和协议 01h | Rndismp.sys  Rndismp.inf |
| **应用程序特定 (FEh)** |  | 建议的驱动程序：[WinUSB (Winusb.sys)](https://msdn.microsoft.com/zh-cn/library/windows/hardware/ff540196) |
| **供应商特定 (FFh)** |  | 建议的驱动程序：[WinUSB (Winusb.sys)](https://msdn.microsoft.com/zh-cn/library/windows/hardware/ff540196) |

### 1.2.2其他内置驱动程序

|  |  |
| --- | --- |
| **设备技术和驱动程序** | **内置驱动程序** |
| **ACPI：ACPI 驱动程序** | Acpi.sys |
| **音频：Microsoft 音频类驱动程序** | PortCls.sys |
| **总线：本机 SD 总线驱动程序、本机 SD 存储类驱动程序和存储微型端口驱动程序** | sdbus.sys、sffdisk.sys、sffp\_sd.sys |
| **HID：HID I2C 驱动程序** | HIDI2C.sys |
| **HID：传统游戏端口驱动程序** | HidGame.sys、Gameenum.sys |
| **HID：传统键盘类驱动程序** | Kbdclass.sys |
| **HID：传统鼠标类驱动程序** | Mouclass.sys |
| **HID：PS/2 (i8042prt) 驱动程序** | I8042prt.sys |
| **映像：设备的 Web 服务 (WSD) 扫描类驱动程序** | WSDScan.sys |
| **打印：Microsoft 绘图仪驱动程序** | Msplot |
| **打印：Microsoft PostScript 打印机驱动程序** | Pscript |
| **打印：Microsoft 通用打印机驱动程序** | Unidrv |
| **打印：Microsoft 第 4 版打印机驱动程序** |  |
| **打印：Microsoft XPS 打印机驱动程序** | XPSDrv |
| **传感器：传感器 HID 类驱动程序** | SensorsHIDClassDriver.dll |
| **触摸：Windows 指针设备驱动程序** |  |
| **WPD：媒体传输协议类驱动程序** | WpdMtpDr.dll、WpdMtp.dll、WpdMtpUs.dll、WpdConns.dll 和 WpdUsb.sys |

## 1.3选择驱动程序模型

### 1.3.1驱动模型

#### 1.3.1.1 UMDF驱动模型

**基于COM思想，运行于用户模式的基于WDF驱动框架的驱动程序。**

* 文件名为\*.dll
* 基于COM思想，引入接口机制
* 运行在RING3的驱动，稳定性高，驱动崩溃不会导致bugcheck（蓝屏）
* 在受限的用户身份下运行，不是受信任的系统内核模块
* 可以使用Win32 API
* 驱动程序不需要强制签名
* 直接单机调试

#### 1.3.1.2 KMDF驱动模型

**基于WDF驱动框架的内核层驱动**

* 文件名为\*.sys
* 运行在RING0的驱动，异常会破坏系统进程地址空间
* 不能使用Win32 API
* 使用双击调试

#### 1.3.1.3 WDM驱动模型

**有即插即用功能框架的NT式驱动，分层设计的概念**

#### 1.3.1.4 NT式驱动

**纯粹的Windows原生态驱动**

### 1.3.2不同类别的驱动程序模型选择

#### 1.3.2.1为设备筛选器驱动程序选择驱动程序模型

以下链接详细罗列了设备驱动程序不同类型：

<https://msdn.microsoft.com/library/windows/hardware/ff557557>

选择依据：

1. 特定技术文档中有介绍，则使用介绍的模型
2. 没有介绍，则通过以下顺序考虑：

* UMDF
* KMDF
* WDM
* NT

#### 1.3.2.2为软件驱动程序选择驱动程序模型

* 内核模式Windows NT模型
* KMDF

#### 1.3.2.3为文件系统筛选器驱动程序选择驱动程序模型

#### 1.3.2.4为文件系统驱动程序选择驱动程序模型

这里（3）和（4）需要参考特定的微型驱动程序模型

# 2内核上机指导

## 2.1下载和使用WDK

### 2.1.1下载安装WDK

下载地址：<https://developer.microsoft.com/en-us/windows/hardware/windows-driver-kit>

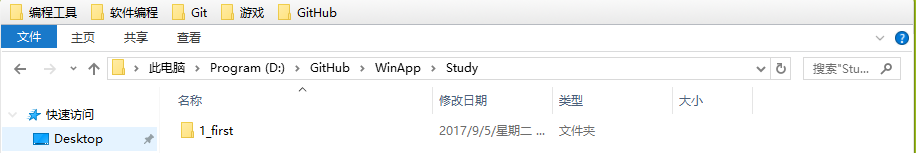
WDK 7600版本，目前测试Win10已经安装不上了。WDK10版本，集成在VisualStudio2015中，我的机器是Win10 14393版本，因此下载**VisualStudio2015 + WDK10\_15063+SDK10\_15063**组件，来编写驱动代码。

**注意：**

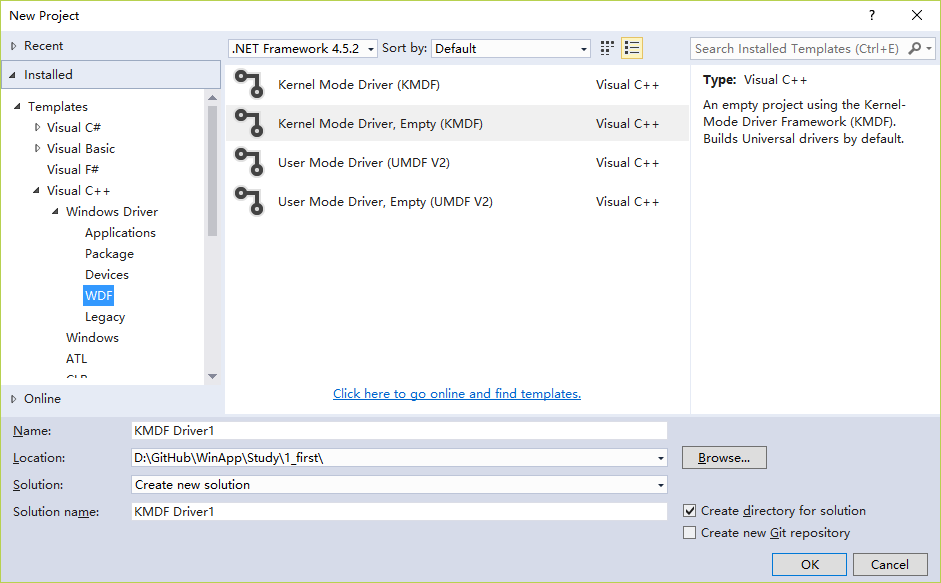
1. SDK10和WDK10由于上述下载地址国内很难连接上，因此使用了公司的香港网络。安装包保存在“**H:\专业软件包\编程工具\微软资源\WDK\**”
2. WDK10开发的驱动，只能应用于Win7及其以上版本，XP不支持。
3. WDK10需要配合SDK10一起，否则使用VS2015开发的时候，会出现编译错误。先安装SDK10，再安装WDK10
4. 如果WDK10和SDK10的版本号不匹配，也会出现一堆问题

### 2.1.2编写第一个C文件

这里为了管理方便以及保存可用代码，用**Git**来管理，建立Git学习章节。



开发使用VS2015，因为WDK已经集成在IDE里面，可以方便使用VS2015直接建立工程开发编译，如下：



编写的第一个程序如下：

|  |
| --- |
| **First.c** |
| #include <ntddk.h>  void DriverUnload(PDRIVER\_OBJECT driver)  {  UNREFERENCED\_PARAMETER(driver);  DbgPrint("Enter First.sys DriverUnload\r\n");  }  NTSTATUS DriverEntry(PDRIVER\_OBJECT driver, PUNICODE\_STRING reg\_path)  {  UNREFERENCED\_PARAMETER(reg\_path);  DbgPrint("Enter First.sys DriverEntry\r\n");  driver->DriverUnload = DriverUnload;  return STATUS\_SUCCESS;  } |

原理说明：

1. DriverEntry是每个内核模块的入口，在加载这个模块时由System进程调用一次
2. 设置了DriverUnload，则内核模块可以被动态卸载；不设置的话，一旦系统开机启动加载之后，是不会被卸载的
3. VS2015默认安全等级编译选项为LEVEL4，未使用的参数会报错，所以这里增加了UNREFERENCED\_PARAMETER宏
4. 工程属性——>驱动设置——>通用，设置Target OS Version和Target Platform，我这里分别选择Windows7，Destop

## 2.2双机调试环境

本节参考博客：<http://lib.csdn.net/article/dotnet/41373>

这里以KMDF更复杂的驱动程序为例，看如何部署双击调试环境。

运行调试器的称为“主机计算机”，运行驱动程序的称为“目标计算机”。这里环境部署的计算机为虚拟机，使用的虚拟机是Oracle VM VirtualBox 5.1.22

### 2.2.1联通主计算机和目标计算机

**（1）关闭目标计算机防火墙**

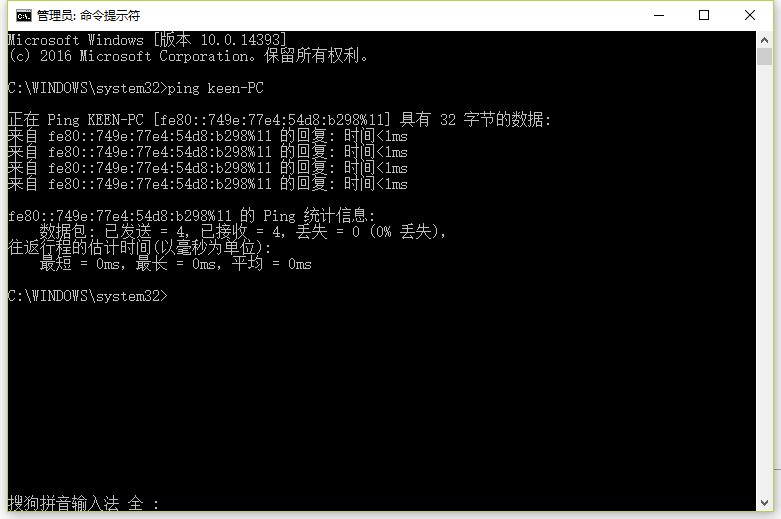
控制面板 ——> 系统和安全 ——> Windows防火墙



**（2）查看目标计算机名**

右键计算机 – 属性，即可查看

**（3）主计算机PING一下目标计算机**



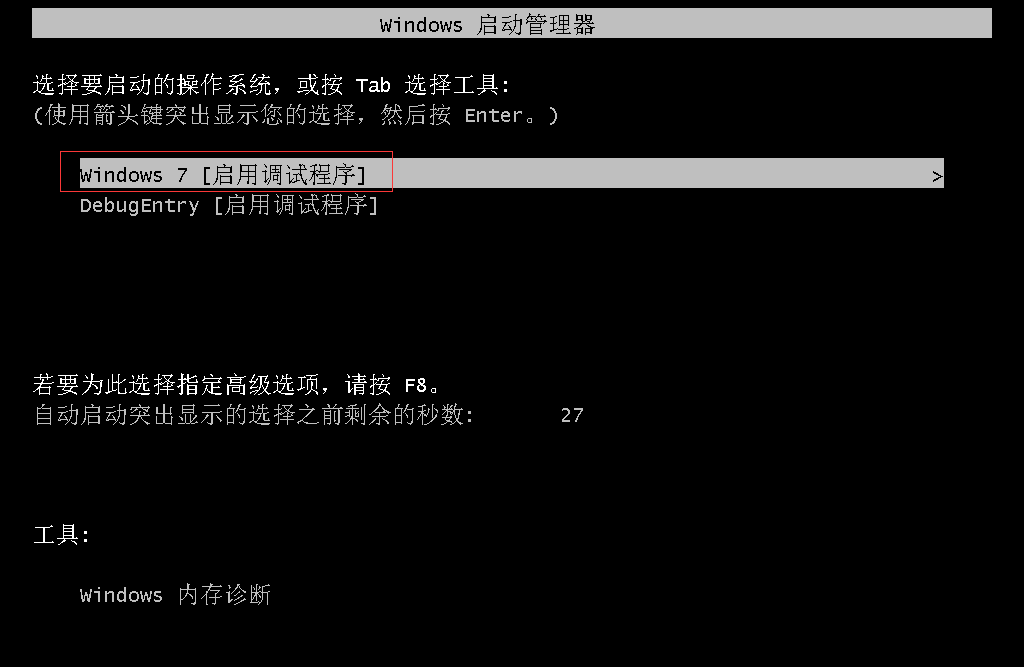
那么这就确认连接ok。如果连接不上，需要让虚拟机网络为桥接模式。

### 2.2.2目标计算机允许内核调试

**目标计算机用系统管理员权限打开CMD，输入下述命令：**

|  |
| --- |
| C:\> bcdedit /bootdebug ON  C:\> bcdedit /set {default} DEBUG YES  C:\> bcdedit /set TESTSIGNING ON  C:\> bcdedit /debug ON |

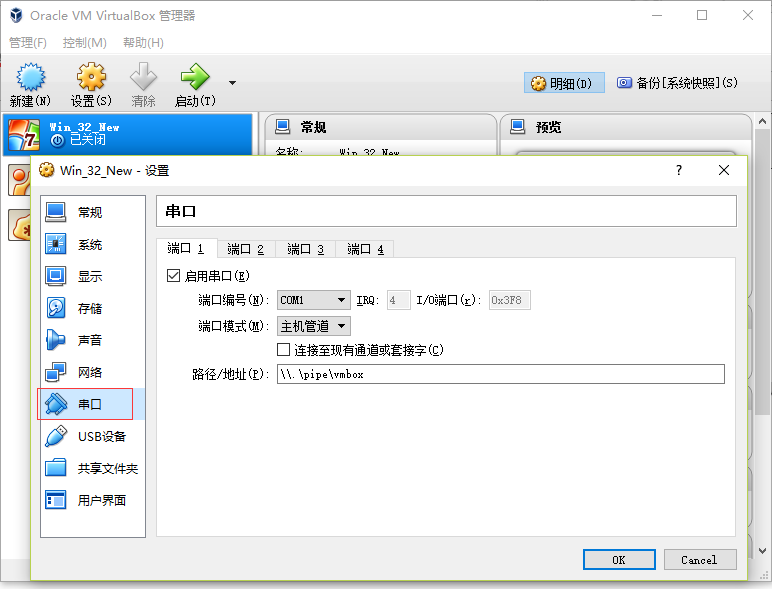
重启电脑生效



### 2.2.3目标计算机串口设置

由于Win7 VBox中不支持网络接口调试，因此采用串口进行调试。

（1）关闭虚拟机中的操作系统（注意必须是关机，不能只是快速休眠），打开虚拟机设置窗口：



采用管道方式的串口通信。管道路径地址格式：[\\.\pipe\](file:///\\.\pipe\){名字}，这里我将管道命名为vmbox

（2）目标计算机设置串口调试

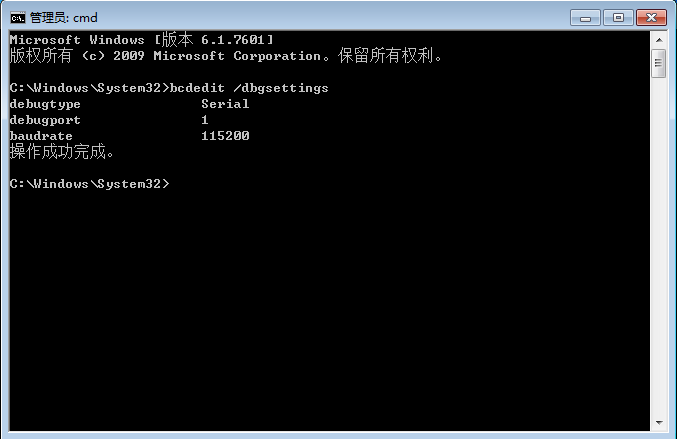
在目标计算机上，通过系统管理员权限打开CMD，输入以下命令：

|  |
| --- |
| C:\> bcdedit /debug on  C:\> bcdedit /dbgsettings serial debugport:1 baudrate:115200 |

其中，debugport:1表示使用的是COM1，波特率最典型的就是115200。

重启计算机生效。重启后，用系统管理员权限打开CMD，输入以下命令查看设置是否成功：

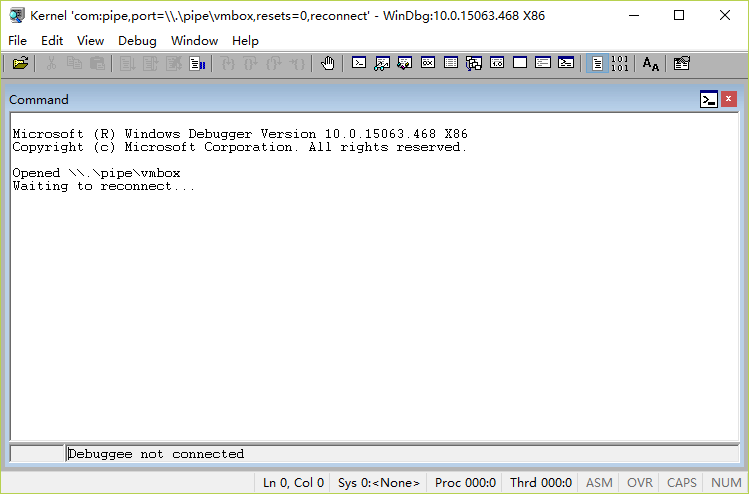
|  |
| --- |
| C:\> bcdedit /dbgsettings |



### 2.2.4主机使用Windbg调试

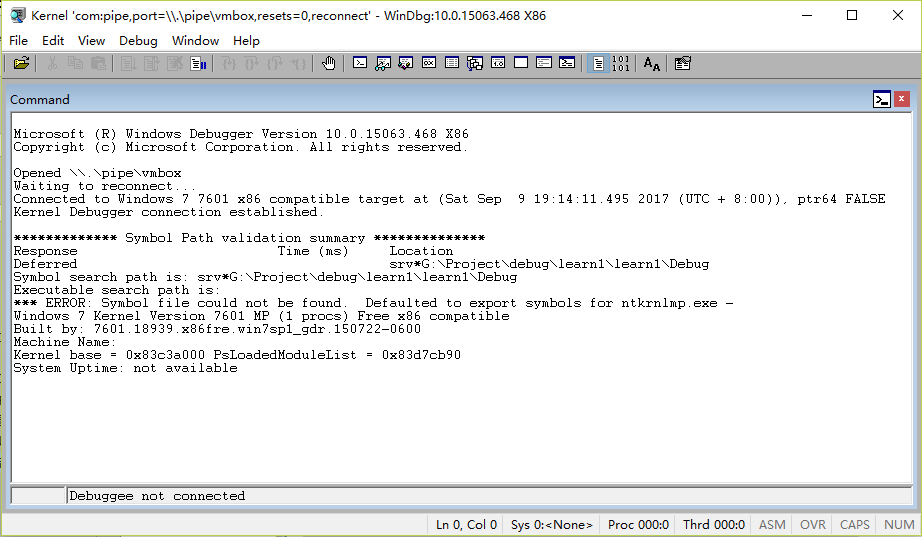
（1）在主计算机打开CMD，进入Windbg文件夹，输入以下命令启动Windbg：

|  |
| --- |
| C:\>cd C:\Program Files\Windows Kits\10\Debuggers\x86  C:\> windbg -k com:pipe,port=\\.\pipe\vmbox,resets=0,reconnect |



Windbg此时处于等待状态

（2）重启虚拟机中的操作系统，之后Windbg将连接上该系统



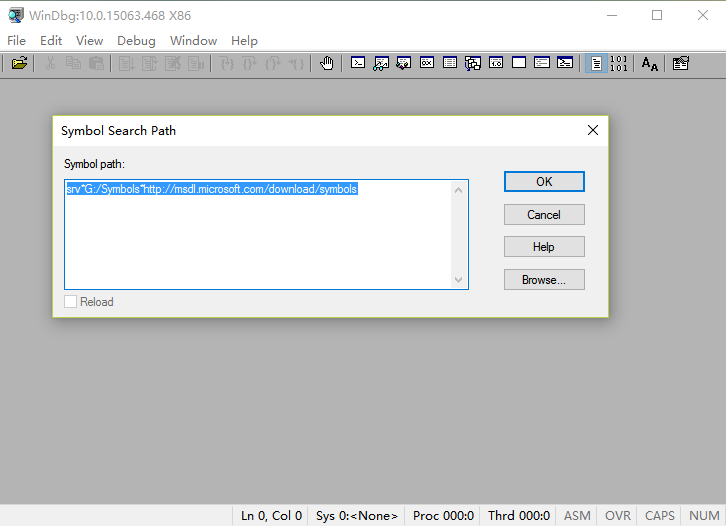
按下Ctrl+Break，可以断下，此时目标计算机将处于卡死状态。

（3）Windbg符号pdb设置

为了在调试的时候，方便windbg导出有用的信息，通常需要加载微软的各个模块的pdb，设置如下：

File——> Symbol File Path

|  |
| --- |
| srv\*G:/Symbols\*http://msdl.microsoft.com/download/symbols |

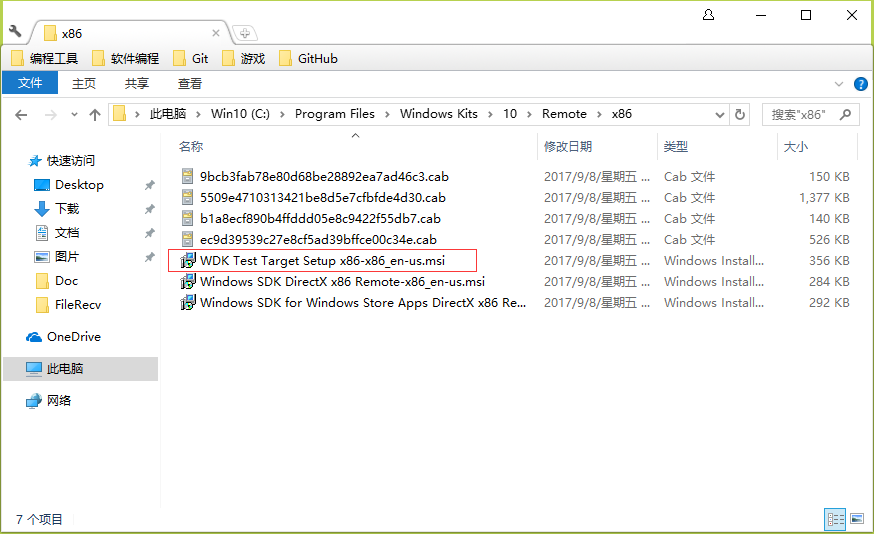


### 2.2.5主机使用VS2015调试

**首先说明一下：我使用此方法失败，可能是机器的原因，死活都无法搭建成功**

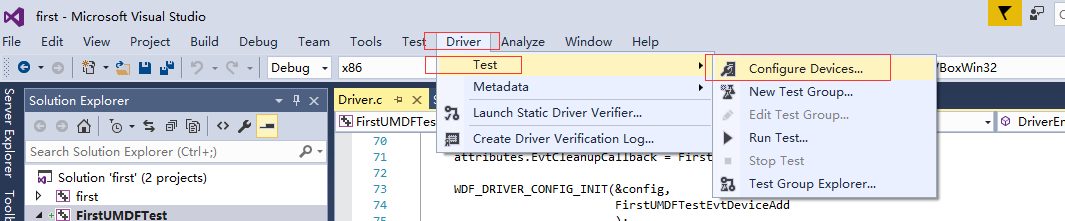
（1）目标计算机设置为WDKRemoteUser

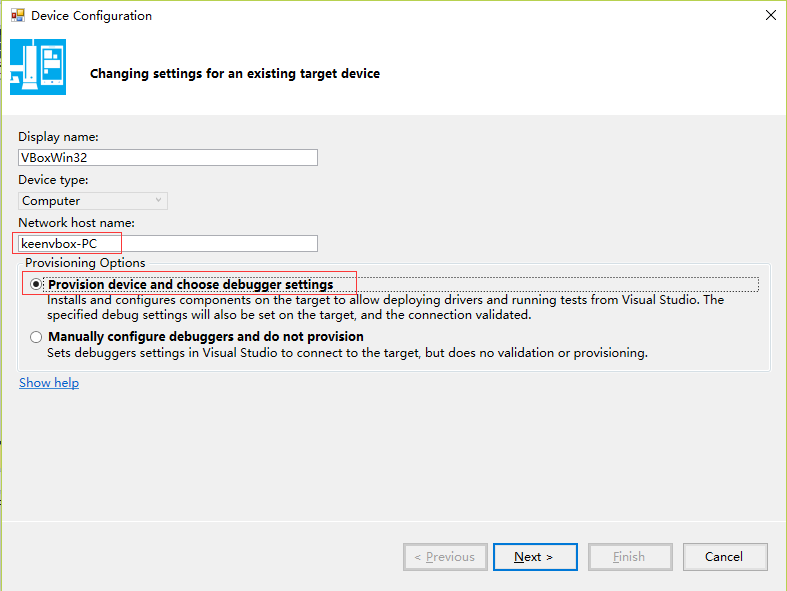
从主计算机的WDK10中，找到符合目标平台（X86或者X64）的WDKRemoteUser程序，本机位置如下：



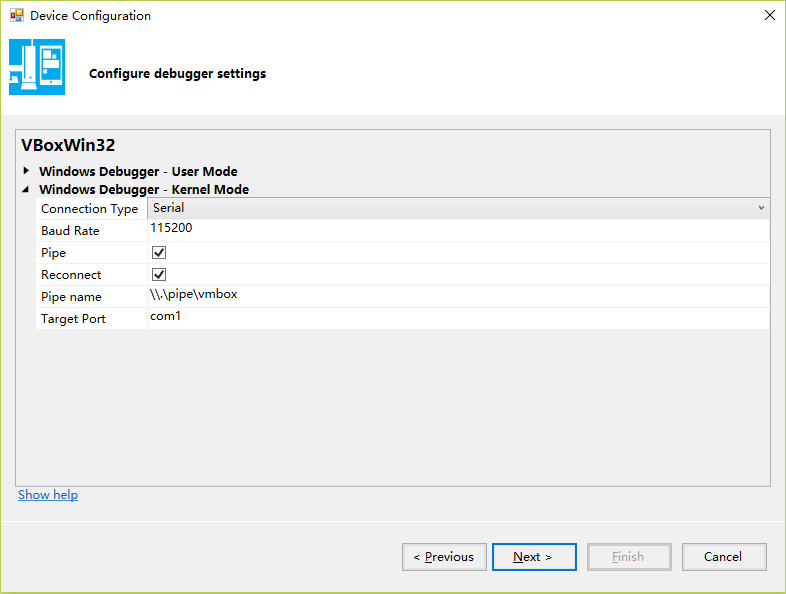
复制到目标计算机中，并且安装

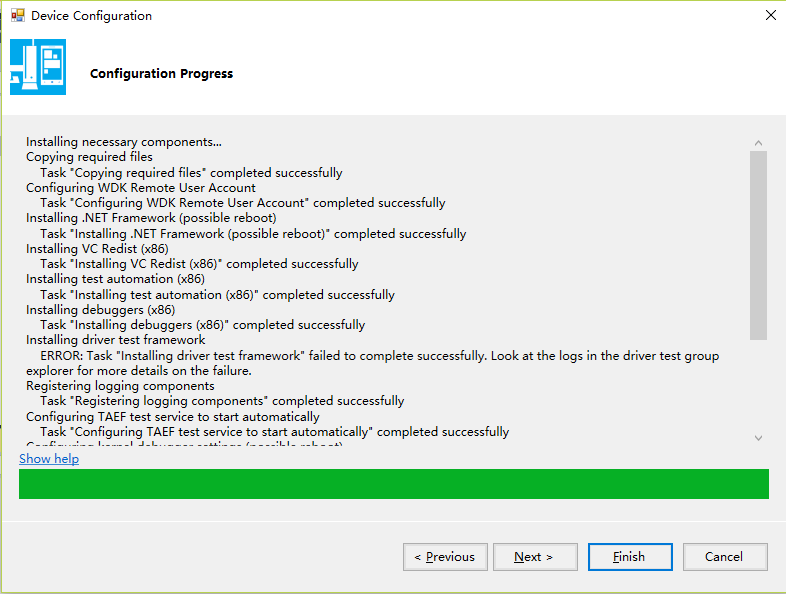
（2）主计算机中设置VS2015调试设备

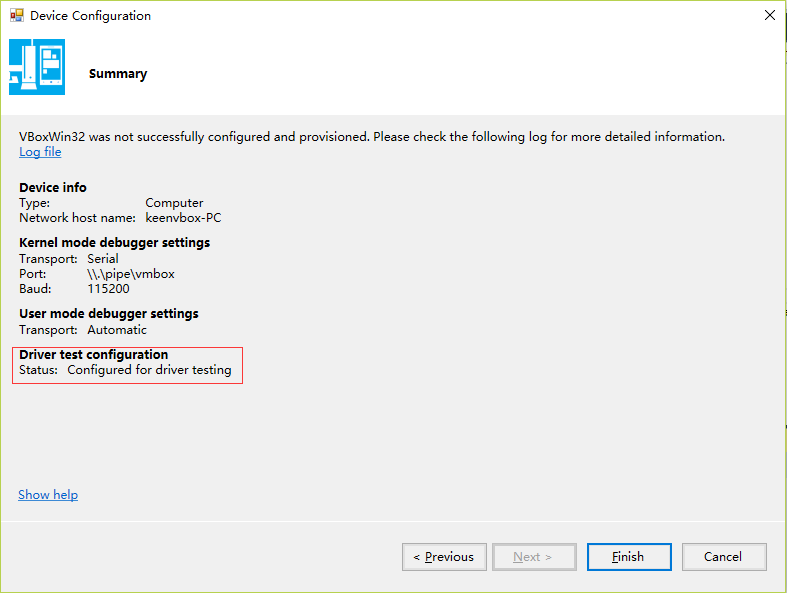




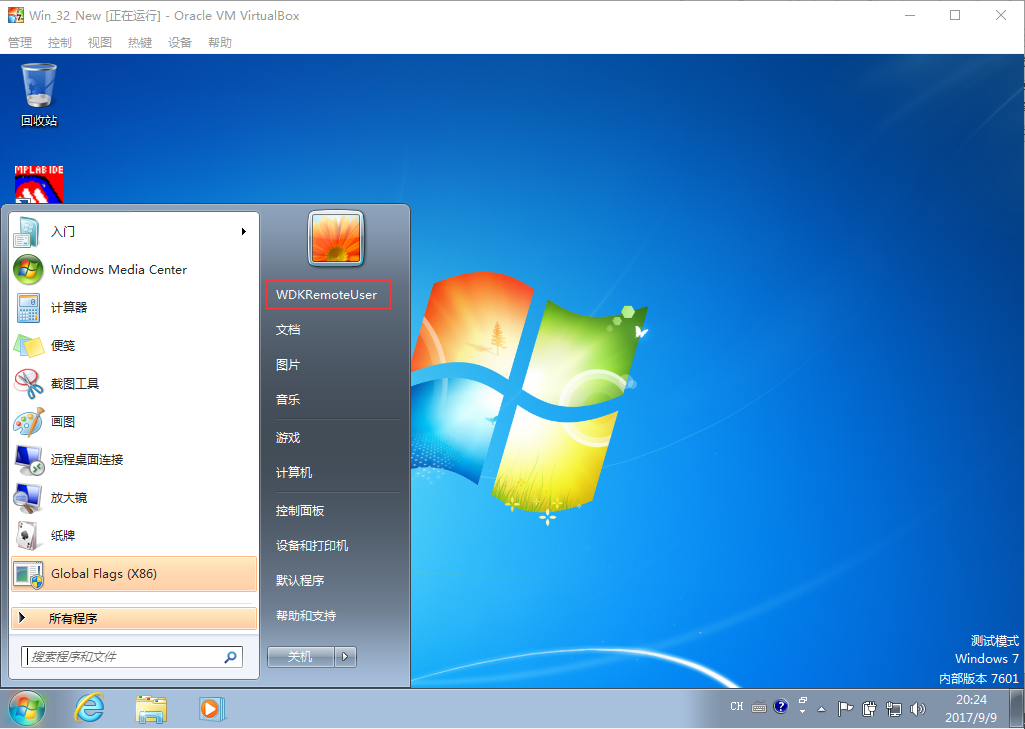
注意，这一步我之前死活都不成功，原因是我的主机的计算机名跟目标计算机名重复了，都是keen-PC，所以会失败。这里确保一下两个的名字不一样，我这里修改了目标计算机名为keenvbox-PC。



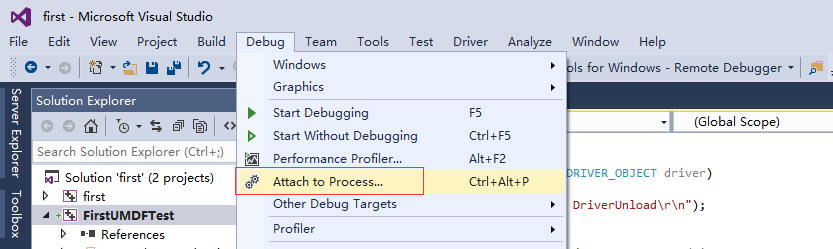


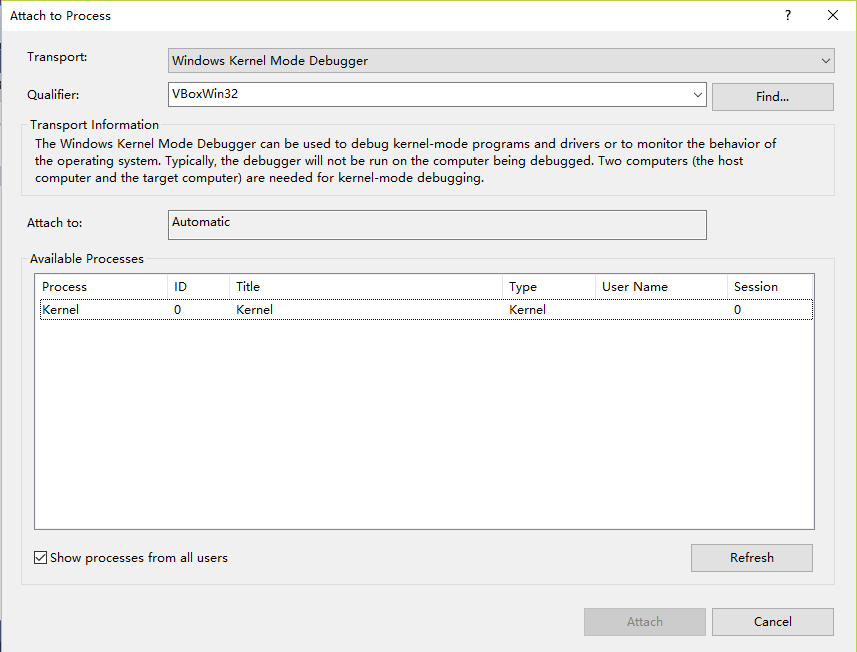


（3）安装完成后，目标计算机会经过几轮重启，切换成WDKRemoteUser



（4）在VS2015里面，开始调试

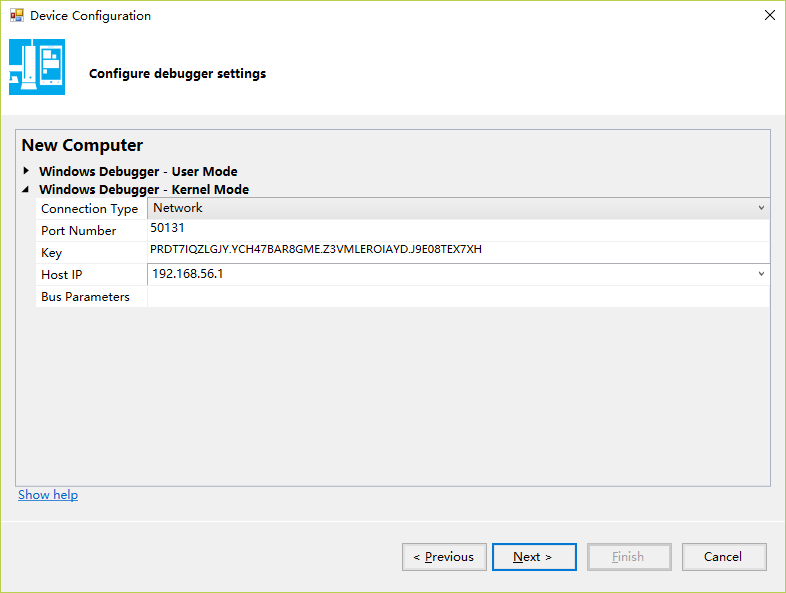


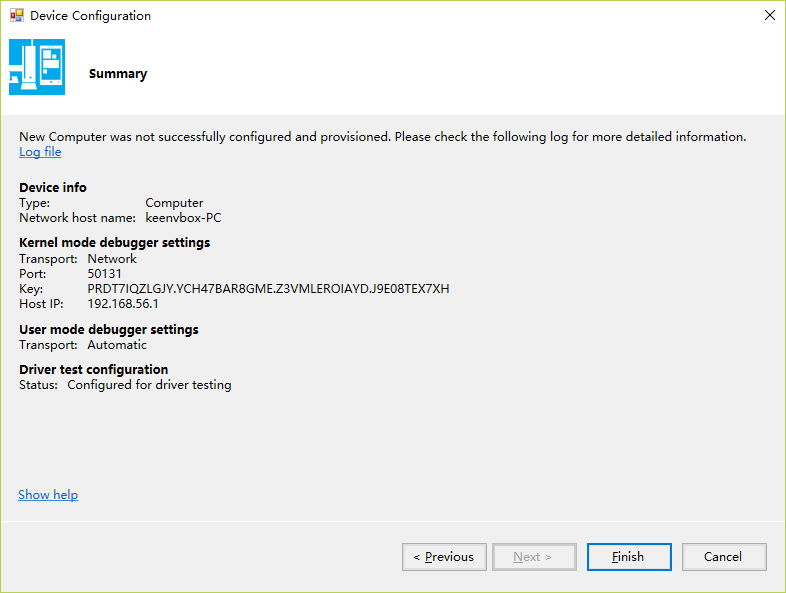


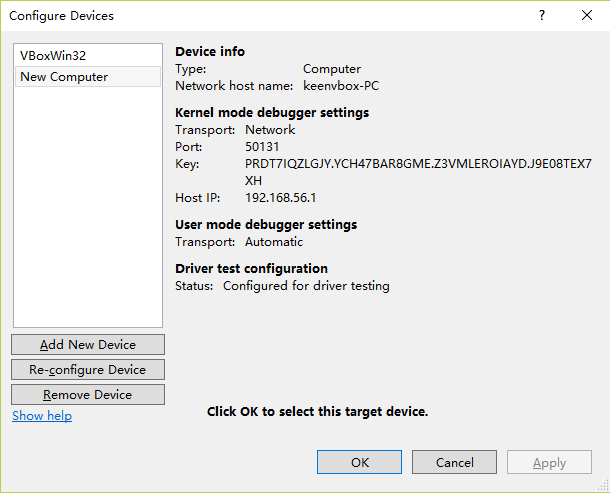
选用内核调试，并附加到刚刚建立的虚拟调试设备VBoxWin32上，选中需要调试的进程，即可开始调试。

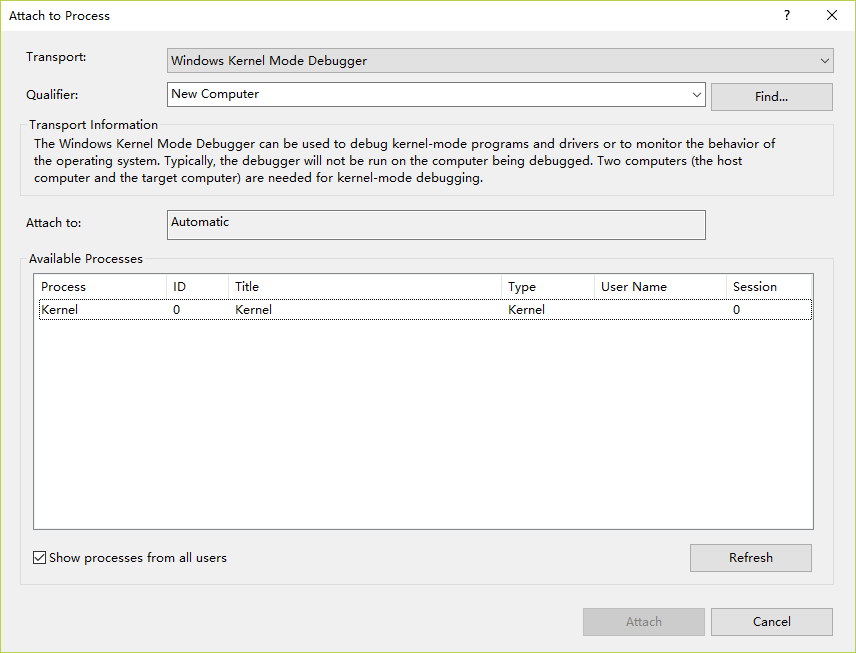
（5）主机使用VS2015调试 – 网络方式

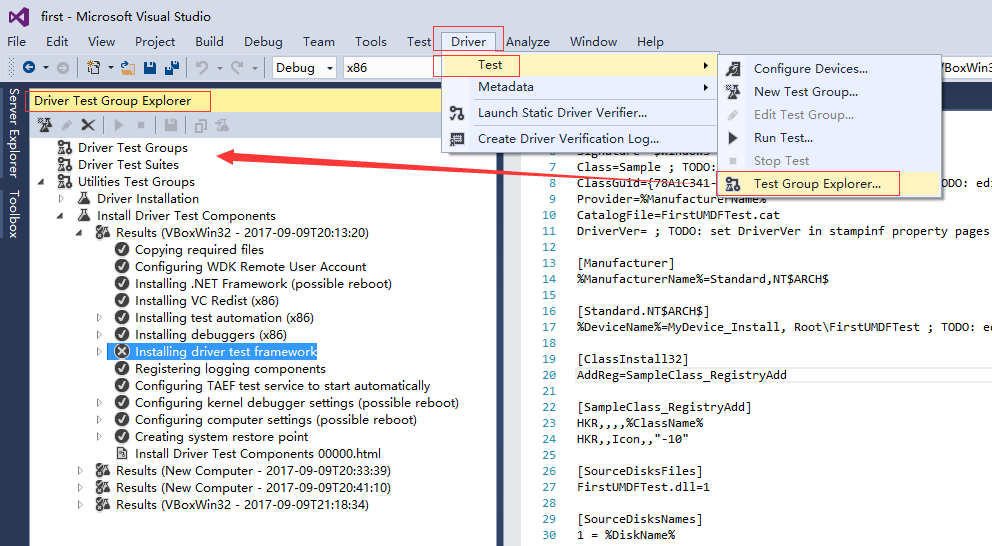
尽管虚拟机无法设置网络调试给windbg使用，但是却可以直接供vs2015使用网络接口方式调试。所有的步骤，均同2.2.4，唯一不同的，就是在VS2015增加驱动设备时，设置的连接方式是网络方式：







（6）Driver Test Group Explorer



出现任何错误，都可以使用它进行查看

## 2.3驱动安装与卸载

### 2.3.1 VS2015直接部署

由于我失败案例，这里就不展开了。正常情况下，使用工程属性里面的Deploy可以直接部署。

### 2.3.2使用工具srvinstw.exe安装驱动

前面已经编写制作了第一个驱动程序First.sys，这里使用srvinstw.exe安装工具，将其安装到虚拟机中，工具存放位置：

|  |
| --- |
| H:\专业软件包\编程工具\srvinstw\srvinstw.exe  D:\Program Files\SysinternalsSuite\Dbgview.exe |

1. 拷贝first.sys、first.inf、srvinstw.exe、dbgview.exe到虚拟机
2. 打开srvinstw.exe，开始安装驱动程序

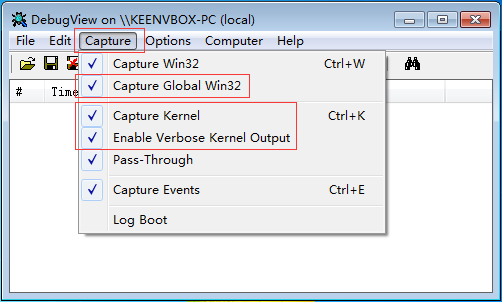
  

注意：

* 这里我测试通过的只有本机计算机安装，即把srvinstw.exe拷贝到虚拟机，在虚拟机里面运行srvinstw.exe，然后安装驱动程序。远程计算机这个，我测试没通过。
* 服务名称，不一定要与文件名相同，也可以不同。
* 程序路径，这里使用“浏览”按钮只能选择exe，因此我们直接输入路径。
* 安装的种类，选择设备驱动
* NT驱动器目标名，不需要输入任何内容
* 选择启动类型为“手动”，可以方便手动进行启动和停止。

1. 使用Dbgview查看输出信息

代码里面调用DbgPrint，这个信息可以用DbgView.exe捕捉到。



1. 启动/停止服务，查看输出日志

输入以下命令：

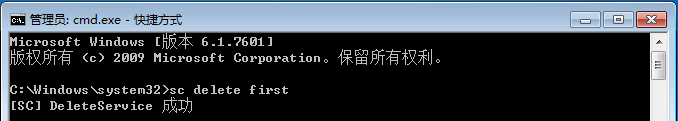
|  |
| --- |
| C:\> net start first  C:\> net stop first |



### 2.3.3驱动卸载

使用srvinstw.exe可以安装驱动，但是在Win7下，无法卸载我们安装的驱动。可以使用如下命令行卸载：

|  |
| --- |
| C:\> sc delete first |

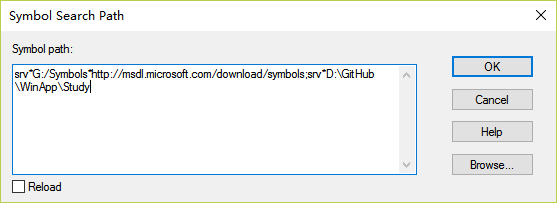


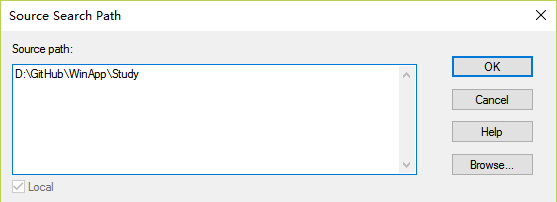
## 2.4 Windbg调试实战

### 2.4.1 Windbg符号设置

前面有讲过设置符号表，设置的是微软的符号表，这里增加我们自己程序的符号表和源码，方便我们定位程序的执行。

|  |
| --- |
| 符号表：  srv\*G:/Symbols\*http://msdl.microsoft.com/download/symbols;srv\*D:\GitHub\WinApp\Study |
| 源码路径：  D:\GitHub\WinApp\Study |

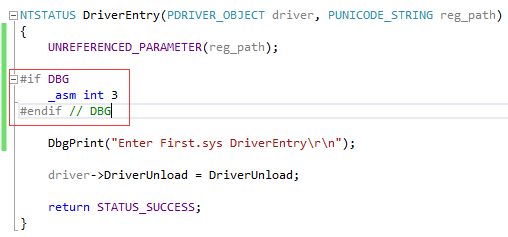




### 2.4.2驱动入口断点技巧

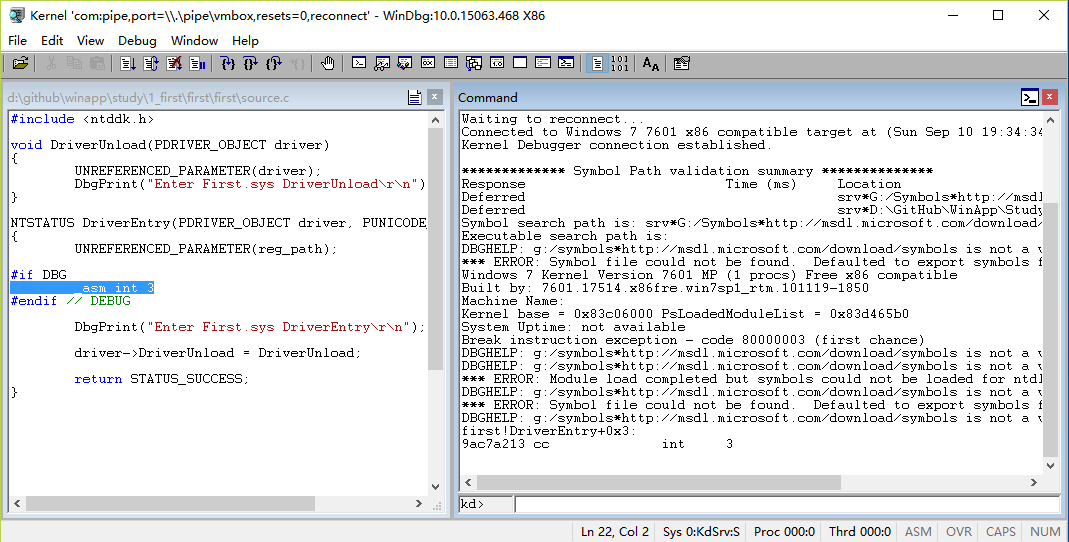
设置int 3断点，可以让驱动启动的时候，断下来

|  |
| --- |
| #if DBG  \_asm int 3  #endif // DBG |



### 2.4.3开始调试

只要让Windbg与虚拟机处于连接状态，那么这个时候，驱动随时都可以更替。用cmd的net start first命令，启动服务，这个时候，会走到DriverEntry入口，遇到int 3中断指令，windbg会断下来，且由于设置了源码，也会自动跳转到源码位置，如下图：



## 2.5总结

部署的一些问题，总结如下：

|  |
| --- |
| D:\GitHub\WinApp\Doc\寒江独钓-windows内核安全编程\内核上级指导总结.mmap |



# 3内核编程环境及其特殊性

## 3.1内核编程环境

### 3.1.1 Windows系统架构



除了特殊的System进程之外（运行在内核模式），以上列出的进程\*.exe，都运行在用户模式，可以看到很多系统的进程，比如会话管理器smss.exe，也是运行在用户模式。

设备及文件系统驱动程序，如图所示，它只能调用内核和直接硬件抽象层的接口，不能使用Windows API（Windows子系统DLL中的接口）。

### 3.1.2隔离空间

（1）32位系统，内核空间是高2GB，用户空间是低2GB。

（2）内核空间，一部分用来给操作系统内核使用，一部分就是给我们的各种驱动程序使用。

（3）用户程序只能使用用户空间，如果想要使用内核空间，需要通过sysenter指令，调用内核空间内容。通常sysenter指令封装在内核提供给R3层使用的接口中

（4）用户空间，可以虚拟出很多应用程序，比如smss.exe、explorer.exe等，他们各自保存一套寄存器，相互独立。内核空间却是只有一份，为所有用户程序共有

（5）内核代码并不一定一直运行于内核空间。如果被应用程序通过接口调用，内核代码也会运行于用户空间。这里就说明一个问题，内核函数具有良好的可重入性。DriverEnter一定是由System进程调用，一定运行于内核空间。

### 3.1.3微内核与单一内核

微内核的概念和单一内核的概念是相互对立的。

**单一内核**：将系统的主要核心组件全部在内核实现。例如：内存管理器、进程管理器、IO管理器等。Linux操作系统就是典型例子。

**微内核**：操作系统的主要组件（例如：内存管理器、进程管理器、IO管理器）各自都运行在独自进程中，除了少量的例如CPU线程切换、中断等，在内核中完成。

Windows NT设计之初，曾考虑设计成一个纯粹的微内核操作系统，Windows的所有程序，全部依赖Win32核心子系统的三大模块：Kernel32.dll、User32.dll（负责窗口消息的分发处理）、Gdi32.dll（负责窗口的图形操作）。以前，User32.dll和Gdi32.dll是放在用户模式下实现的，Windows NT4.0之后，这俩实现放到了Win32k.sys，同时用户模式下留一个空壳的调用转发，这么做是为了提高绘制图形的效率。

但是总的来说，Windows设计中，真正的内核部分，只处理线程调度、中断等核心功能，其他的都放在了执行体组件中（也是在内核模式，但是分层在内核之上，虽然实际内容都是在ntoskernel.exe中，但是逻辑上是分离的），算是一个微内核操作系统。

### 3.1.4 Windows设计理念

#### 3.1.4.1可移植性

**（1）目标**：只需要少量的修改，就可以在尽可能多的硬件平台上运行。（Windows最初，应该是想广纳百川，运行在一切硬件设备上，想法太好了）。

**（2）实际努力**：分层体系结构。和硬件紧密相关的只有硬件抽象层（HAL），所有跟硬件打交道的，都通过统一的硬件抽象层接口来调用（这也提示我们，不要使用跟特殊平台相关的汇编指令）。移植到不同平台，只需要修改可能有变化的HAL即可。实际上，Windows从32位CPU移植到64位CPU的时候，只改动了相当少的代码。

#### 3.1.4.2兼容性

**（1）目标**：让应用程序运行在尽可能多的各个系统版本上。

**（2）实际努力**：实现了一套统一的Win32 API接口，虽然实现不一样，但是保证了兼容；引入环境子系统，不同的环境子系统，实现了各自一套API支持，这些API都是基于Win32 API。（就像是模拟器虚拟机一样，甚至只要实现了android环境子系统的api，就可以跑起来android应用，只要实现了Linux环境子系统的api，就可以跑起来linux系统上的应用）

#### 3.1.4.3健壮性和可靠性

**（1）目标**：健壮可靠，不至于各种蓝屏

**（2）实际努力**：划分用户模式和内核模式。应用程序运行在权限级别低的用户模式，所有的错误操作，都可以被操作系统侦测到，并给予提示，不至于让错误继续蔓延。这里也是由于另外一个点，所有的应用程序的操作，都是基于Win32 API的，这一层，操作系统做了很多检测代码，包括参数异常、空指针检测等。

#### 3.1.4.4可扩展性

**（1）目标**：易于增加新的功能和支持新的硬件，并且对已有的代码的影响达到最小。

**（2）实际努力**：内核和各个核心执行体组件分离（微内核思想），同时各个执行体组件模块化。比如驱动加载有一套框架，都由system.exe来生产，IO管理器控制，可以方便支持新的硬件接入

#### 3.1.4.5性能

**（1）目标**：速度尽可能快

**（2）实际努力**：分层会导致性能降低，但是通过其他地方补回来。例如：硬件抽象层的调用是宏调用，而非函数。另外，IO设计为异步，无需等待，提高IO吞吐能力，无需让CPU等待。同时，有多进程多线程的设计。

### 3.1.5 Windows操作系统分层结构详解

#### 3.1.5.1简化版Windows架构图



#### 3.1.5.2应用程序与Win32子系统

由于其他子系统很少被用到，慢慢成为废弃状态（虽然当初设计理念非常好，但是阻挡不住流行的步伐），因此只介绍Win32子系统。

1. 每个应用程序在编译链接时，需要指明属于哪个子系统（例如：/subsystem:windows是指这个应用程序是Win32子系统的应用程序，也就是说调用的是Win32 API）
2. Win32环境子系统对应的三个dll：Kernel.dll、User32.dll、Gdi32.dll，是应用程序实现的基础。其中Kernel.dll的内核实现，是依靠执行体组件Ntoskrnl.exe完成，后两者也放到了内核模式Win32k.sys，提高绘制效率。



#### 3.1.5.3 Native API

此API都是在ntdll.dll中实现的。

Native API是从用户模式进入内核模式的大门，进入方式是**软件中断**方式。一般不推荐程序员直接使用Native API，因为在各个版本的Windows中会有所不同。

#### 3.1.5.4系统服务

Native API从用户模式穿越进入内核模式，调用系统服务。这里通过sysenter指令，触发软中断，同时将Native API中的参数和系统服务号一通传进内核模式。不同的Native API会对应不同的系统服务号。在系统服务组件中，有一个系统服务描述符表（System Service Descriptor Table，简称SSDT），记录了系统服务号到对应服务函数地址的映射关系。



#### 3.1.5.5执行体程序组件

内核和执行体程序组件，都存在于**ntoskrnl.exe**中，逻辑分离。执行体程序组件是内核模式下的一组服务函数，分模块的。

1. **对象管理程序**：Windows提供的所有服务几乎都是以对象的形式存在（包括同步互斥量、驱动对象、设备对象、进程等），这个对象就由对象管理器来控制，包括创建、管理、回收这些对象组件等。
2. **进程管理程序**：每一个进程维护一个专门的地址空间和安全身份，进程管理器负责创建和终止进程。进程包括一个或多个线程，而线程的调度，是由内核负责，不由进程管理器负责。
3. **虚拟内存管理程序**：负责虚拟内存的申请、回收等操作
4. **I/O管理程序**：负责发起I/O请求，并且管理这些请求（IRP）。IRP被传递到具体设备的驱动程序中，驱动程序负责“完成”这些IRP，并将完成的状态原路返回到用户模式下的应用程序中。担任用户模式应用程序与硬件设备之间的接口。
5. **配置管理程序**：管理注册表

#### 3.1.5.6驱动程序

配合I/O管理器，实现应用程序与硬件设备的交互。处理方法有如下几种：

1. 根据IRP的请求，直接操作具体硬件，然后完成此IRP，并返回
2. 将此IRP的请求，转发到更底层的驱动中去，并等待底层驱动的返回
3. 接收到IRP请求后，不是急于完成，而是分配新的IRP发到其他驱动程序中，并等待返回

#### 3.1.5.7内核

内核也处于ntoskrnl.exe中，但是与执行体组件逻辑分离。功能如下：

1. 对内核对象的支持
2. 对线程调度的支持
3. 对多处理器同步的支持
4. 中断处理函数的支持
5. 对错误陷阱的支持
6. 对其他硬件特殊功能的支持

内核执行在最高特权之上，被设计成可以并行运行在多处理器上，内核在调度线程的时候，不能被其他线程所打断，即不允许线程的切换，但是内核可以被更高的中断请求级别（IRQL）所打断

#### 3.1.5.8硬件抽象层

不同硬件平台的操作不尽相同，硬件抽象层提供统一操作硬件的接口。例如：IN、OUT汇编指令，硬件抽象层有READ\_PORT\_BUFFER\_UCHAR、WRITE\_PORT\_BUFFER\_UCHAR函数可用

## 3.2数据类型及数据结构

### 3.2.1编程规范

（1）尽可能使用以下重定义的数据类型

ULONG、UCHAR、UINT、VOID、PULONG、PUCHAR、PUINT、PVOID

（2）函数返回值

绝大部分内核API都会返回一个状态NTSTATUS，写内核函数要非常小心，任何资源分配问题、异常地址等，都需要小心校验，否则会蓝屏。我们写内核函数也要对此负责。

|  |
| --- |
| NTSTATUS TestFunc()  {  NTSTATUS status = STATUS\_SUCCESS;  status = ZwCreateFile(...);  if (!NT\_SUCCESS(status))  {  return status;  }  ...  return status;  } |

状态码都在<ntstatus.h>中，具体含义参考：

<http://blog.csdn.net/qq1841370452/article/details/54766811>

### 3.2.2字符串

驱动中使用的字符串结构如下（举UNICODE的例子，实际使用的还有ASCII）：

|  |
| --- |
| typedef struct \_UNICODE\_STRING {  USHORT Length;  USHORT MaximumLength;  \_Field\_size\_bytes\_part\_opt\_(MaximumLength, Length) PWCH Buffer;  } UNICODE\_STRING;  typedef UNICODE\_STRING \*PUNICODE\_STRING;  typedef const UNICODE\_STRING \*PCUNICODE\_STRING; |

常规使用方法，并不直接使用字符串指针，而是把它当做一个类似wstring的对象使用，内核框架中很多API可直接支持。例如：

|  |
| --- |
| UNICODE\_STRING str = RTL\_CONSTANT\_STRING(L"My first Kernel String!");  DbgPrint("%wZ", str); |

注意，使用的格式化“%wZ”。

其他常用的字符串操作还有拷贝、连接、转换等。

### 3.2.3驱动对象

什么是驱动对象？驱动对象，可以理解成，就是一个内核模块。内核编程，是用c模拟c++，模拟面向对象编程。所有的方法，都以函数指针的方式存储，与相关的对象数据存储在一起。驱动对象DRIVER\_OBJECT即是如此：

|  |
| --- |
| typedef struct \_DRIVER\_OBJECT {  CSHORT Type;  CSHORT Size;  //  // The following links all of the devices created by a single driver  // together on a list, and the Flags word provides an extensible flag  // location for driver objects.  //  PDEVICE\_OBJECT DeviceObject;  ULONG Flags;  //  // The following section describes where the driver is loaded. The count  // field is used to count the number of times the driver has had its  // registered reinitialization routine invoked.  //  PVOID DriverStart;  ULONG DriverSize;  PVOID DriverSection;  PDRIVER\_EXTENSION DriverExtension;  //  // The driver name field is used by the error log thread  // determine the name of the driver that an I/O request is/was bound.  //  UNICODE\_STRING DriverName;  //  // The following section is for registry support. This is a pointer  // to the path to the hardware information in the registry  //  PUNICODE\_STRING HardwareDatabase;  //  // The following section contains the optional pointer to an array of  // alternate entry points to a driver for "fast I/O" support. Fast I/O  // is performed by invoking the driver routine directly with separate  // parameters, rather than using the standard IRP call mechanism. Note  // that these functions may only be used for synchronous I/O, and when  // the file is cached.  //  PFAST\_IO\_DISPATCH FastIoDispatch;  //  // The following section describes the entry points to this particular  // driver. Note that the major function dispatch table must be the last  // field in the object so that it remains extensible.  //  PDRIVER\_INITIALIZE DriverInit;  PDRIVER\_STARTIO DriverStartIo;  PDRIVER\_UNLOAD DriverUnload;  PDRIVER\_DISPATCH MajorFunction[IRP\_MJ\_MAXIMUM\_FUNCTION + 1];  } DRIVER\_OBJECT;  typedef struct \_DRIVER\_OBJECT \*PDRIVER\_OBJECT; |

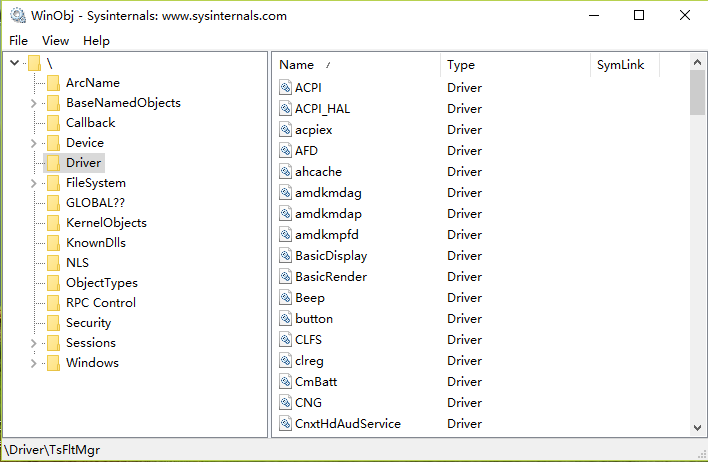
解释下重要的几个字段：

* **DeviceObject**：每个驱动对象都会有一个或多个设备对象（貌似也可以一个也没有，只是用来访问内核），多个设备对象指针互连组成链表。通过这个值，可以遍历所有设备对象。设备对象是由程序员自己创建，在驱动卸载的时候，也需要程序员自己遍历每个设备对象并将其删除。
* **DriverName**：驱动名称，该字符串的命名一般是：\Driver\[服务名]。服务名是由驱动创建时，写入到HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services子项的名称，例如HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services\MyHelloDDK
* **HardwareDatabase**：记录硬件数据库键名。该字符串的一般为：\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM
* **DriverStartIO**：记录StartIO例程的函数地址，用于串行化操作
* **DriverUnload**：指定驱动卸载时所用的回调函数地址
* **MajorFunction**：是一个数组，记录IRP派遣函数指针数组
* **FastIODispatch**：文件驱动中用到的派遣函数



相当于，windows实现了这个“驱动类”，这个“驱动类”的函数都相当于“虚函数”，我们每一个程序员去实现一个特定的对象，并实现具体的“虚函数”。也就是说，我们只需要实现一组回调接口，给Windows调用即可，不需要管完整的调用流程。

Windows有很多组件，都有实现了自己的DRIVER\_OBJECT，比如：所有的硬件驱动程序（硬盘、鼠标等）、所有的类驱动（Disk、CDROM…）、文件系统（NTFS、FastFat…），以及许多其他的内核组件。使用WinObj，可以查看，如下：



至于WinObj为什么可以获取到这些内容，从网上得知的，它可以根据R3层的这几个接口：

ZwOpenDirectoryObject、ZwQueryDirectoryObject获取到这些。

对象内的快速分发函数和普通分发函数，是功能实现的重点。另外，如果我们可以找到这些内核对象地址，就可以截胡这些分发函数，这就是所谓的**分发函数Hook技术**。

### 3.2.4设备对象

什么是设备对象？它可以是一个硬盘（硬盘读写直接驱动的对象），接收读写请求，然后传递数据；也可以是一个非硬件相关，如管道，也是传递数据。

设备对象类似于Windows GUI里面的窗口，是用来接收“消息”的实体。内核世界里，消息大多以IRP请求的方式传递，而这些请求都是发送给某个设备对象。比如一边是读请求，一边是写请求，这样建立起通信。一个设备对象，总是属于一个驱动对象（因为设备对象打开，必然是在驱动对象中实现的）。设备对象是一个DEVICE\_OBJECT的结构：

|  |
| --- |
| typedef struct DECLSPEC\_ALIGN(MEMORY\_ALLOCATION\_ALIGNMENT) \_DEVICE\_OBJECT {  CSHORT Type;  USHORT Size;  LONG ReferenceCount;  struct \_DRIVER\_OBJECT \*DriverObject;  struct \_DEVICE\_OBJECT \*NextDevice;  struct \_DEVICE\_OBJECT \*AttachedDevice;  struct \_IRP \*CurrentIrp;  PIO\_TIMER Timer;  ULONG Flags; // See above: DO\_...  ULONG Characteristics; // See ntioapi: FILE\_...  \_\_volatile PVPB Vpb;  PVOID DeviceExtension;  DEVICE\_TYPE DeviceType;  CCHAR StackSize;  union {  LIST\_ENTRY ListEntry;  WAIT\_CONTEXT\_BLOCK Wcb;  } Queue;  ULONG AlignmentRequirement;  KDEVICE\_QUEUE DeviceQueue;  KDPC Dpc;  //  // The following field is for exclusive use by the filesystem to keep  // track of the number of Fsp threads currently using the device  //  ULONG ActiveThreadCount;  PSECURITY\_DESCRIPTOR SecurityDescriptor;  KEVENT DeviceLock;  USHORT SectorSize;  USHORT Spare1;  struct \_DEVOBJ\_EXTENSION \*DeviceObjectExtension;  PVOID Reserved;  } DEVICE\_OBJECT;  typedef struct \_DEVICE\_OBJECT \*PDEVICE\_OBJECT; |

解释下重要字段：

* **DriverObject**：指向驱动程序中的驱动对象。也就是由一个驱动对象，可以遍历得到设备对象；而由一个设备对象，直接可以得知驱动对象，相互访问通畅。
* **NextDevice**：指向下一个设备对象。指的是同一个驱动对象DriverObject的设备栈的下一个设备对象。
* **AttachedDevice**：指向下一个设备对象。这里指的是，如果有更高一层的驱动附加到这个驱动的时候，AttachedDevice就指向那个更高一层的驱动
* **CurrentIrp**：在使用StartIO例程的时候，指向当前IRP结构
* **Flags**：是一个按位的值，每一位有具体的含义，主要位如下：

|  |  |
| --- | --- |
| 标志 | 描述 |
| DO\_BUFFERRED\_IO | 读写操作使用缓冲方式（系统复制缓冲区）访问用户模式数据 |
| DO\_EXCLUSIVE | 一次只允许一个县城打开设备句柄，独占方式 |
| DO\_DIRECT\_IO | 读写操作使用直接方式（内存描述符表）访问用户模式数据 |
| DO\_DEVICE\_INITIALIZING | 设备对象正在初始化 |
| DO\_POWER\_PAGABLE | 必须再PASSIVE\_LEVEL级上处理IRP\_MJ\_PNP请求 |
| DO\_POWER\_INRUSH | 设备上电期间需要大电流 |

* **DeviceExtension**：设备扩展，由程序员自己定义的结构，通常将全局变量放到这里传递，避免全局变量同步问题
* **DeviceType**：设备类型。有很多种，参见MSDN。这里虚拟设备的设备类型是FILE\_DEVICE\_UNKNOWN类型
* **StackSize**：多层驱动情况下，驱动与驱动之间会形成堆栈的结构，IRP会依次从最高层传递到最底层，StackSize就是这个层数。
* **AlignmentRequirement**：设备在大容量的时候，需要内存对齐，保证传输速度



设备对象常规的用法可以如图表示：



请求方想向一个设备对象发送数据，通常是被内核对象的某个分发函数捕获，进而来处理的。至于请求怎么到内核对象的分发函数，则是系统内核分发服务处理完成的。常规分发函数的声明如下：

|  |
| --- |
| NTSTATUS MyDispatch(PDEVICE\_OBJECT device, PIRP irp); |

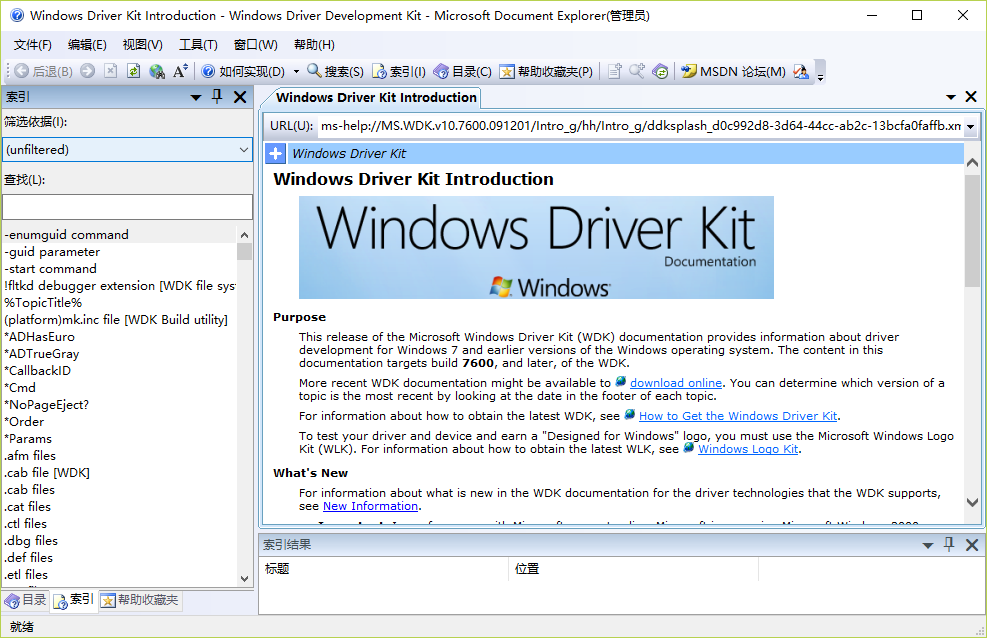
### 3.2.5请求

请求类似于Windows消息，是发送给设备对象的数据传递方式。一个IRP常常要经过n个设备，才能得以完成。所以IRP请求，会有一个IRP栈空间，每次经过一个设备，都保留一部分数据。定义如下：

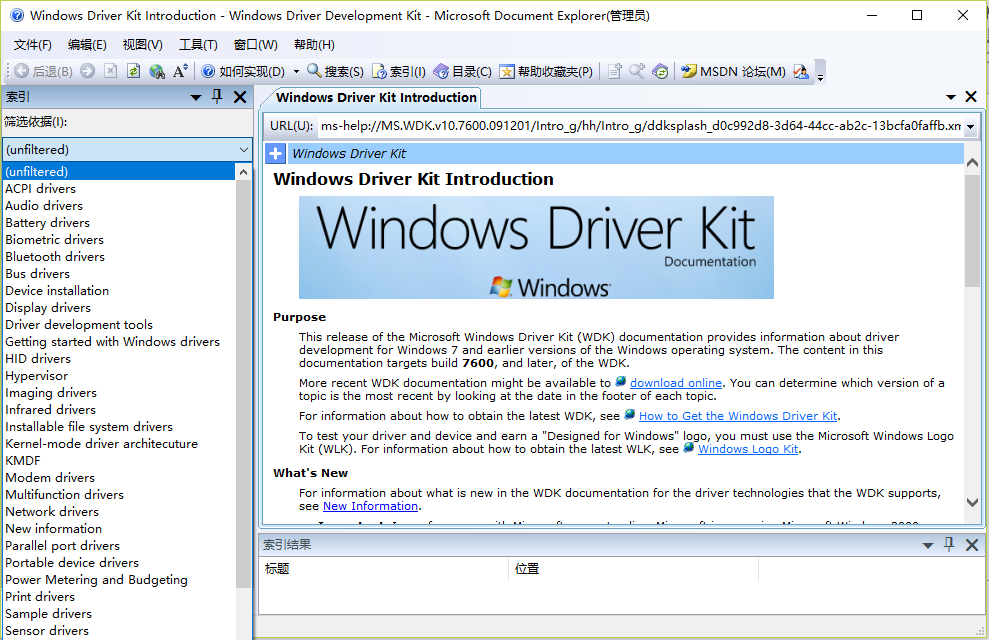
|  |
| --- |
| //  // I/O Request Packet (IRP) definition  //  typedef struct DECLSPEC\_ALIGN(MEMORY\_ALLOCATION\_ALIGNMENT) \_IRP {  CSHORT Type;  USHORT Size;  //  // Define the common fields used to control the IRP.  //  //  // Define a pointer to the Memory Descriptor List (MDL) for this I/O  // request. This field is only used if the I/O is "direct I/O".  //  PMDL MdlAddress;  //  // Flags word - used to remember various flags.  //  ULONG Flags;  //  // The following union is used for one of three purposes:  //  // 1. This IRP is an associated IRP. The field is a pointer to a master  // IRP.  //  // 2. This is the master IRP. The field is the count of the number of  // IRPs which must complete (associated IRPs) before the master can  // complete.  //  // 3. This operation is being buffered and the field is the address of  // the system space buffer.  //  union {  struct \_IRP \*MasterIrp;  \_\_volatile LONG IrpCount;  PVOID SystemBuffer;  } AssociatedIrp;  //  // Thread list entry - allows queuing the IRP to the thread pending I/O  // request packet list.  //  LIST\_ENTRY ThreadListEntry;  //  // I/O status - final status of operation.  //  IO\_STATUS\_BLOCK IoStatus;  //  // Requester mode - mode of the original requester of this operation.  //  KPROCESSOR\_MODE RequestorMode;  //  // Pending returned - TRUE if pending was initially returned as the  // status for this packet.  //  BOOLEAN PendingReturned;  //  // Stack state information.  //  CHAR StackCount;  CHAR CurrentLocation;  //  // Cancel - packet has been canceled.  //  BOOLEAN Cancel;  //  // Cancel Irql - Irql at which the cancel spinlock was acquired.  //  KIRQL CancelIrql;  //  // ApcEnvironment - Used to save the APC environment at the time that the  // packet was initialized.  //  CCHAR ApcEnvironment;  //  // Allocation control flags.  //  UCHAR AllocationFlags;  //  // User parameters.  //  PIO\_STATUS\_BLOCK UserIosb;  PKEVENT UserEvent;  union {  struct {  union {  PIO\_APC\_ROUTINE UserApcRoutine;  PVOID IssuingProcess;  };  PVOID UserApcContext;  } AsynchronousParameters;  LARGE\_INTEGER AllocationSize;  } Overlay;  //  // CancelRoutine - Used to contain the address of a cancel routine supplied  // by a device driver when the IRP is in a cancelable state.  //  \_\_volatile PDRIVER\_CANCEL CancelRoutine;  //  // Note that the UserBuffer parameter is outside of the stack so that I/O  // completion can copy data back into the user's address space without  // having to know exactly which service was being invoked. The length  // of the copy is stored in the second half of the I/O status block. If  // the UserBuffer field is NULL, then no copy is performed.  //  PVOID UserBuffer;  //  // Kernel structures  //  // The following section contains kernel structures which the IRP needs  // in order to place various work information in kernel controller system  // queues. Because the size and alignment cannot be controlled, they are  // placed here at the end so they just hang off and do not affect the  // alignment of other fields in the IRP.  //  union {  struct {  union {  //  // DeviceQueueEntry - The device queue entry field is used to  // queue the IRP to the device driver device queue.  //  KDEVICE\_QUEUE\_ENTRY DeviceQueueEntry;  struct {  //  // The following are available to the driver to use in  // whatever manner is desired, while the driver owns the  // packet.  //  PVOID DriverContext[4];  } ;  } ;  //  // Thread - pointer to caller's Thread Control Block.  //  PETHREAD Thread;  //  // Auxiliary buffer - pointer to any auxiliary buffer that is  // required to pass information to a driver that is not contained  // in a normal buffer.  //  PCHAR AuxiliaryBuffer;  //  // The following unnamed structure must be exactly identical  // to the unnamed structure used in the minipacket header used  // for completion queue entries.  //  struct {  //  // List entry - used to queue the packet to completion queue, among  // others.  //  LIST\_ENTRY ListEntry;  union {  //  // Current stack location - contains a pointer to the current  // IO\_STACK\_LOCATION structure in the IRP stack. This field  // should never be directly accessed by drivers. They should  // use the standard functions.  //  struct \_IO\_STACK\_LOCATION \*CurrentStackLocation;  //  // Minipacket type.  //  ULONG PacketType;  };  };  //  // Original file object - pointer to the original file object  // that was used to open the file. This field is owned by the  // I/O system and should not be used by any other drivers.  //  PFILE\_OBJECT OriginalFileObject;  } Overlay;  //  // APC - This APC control block is used for the special kernel APC as  // well as for the caller's APC, if one was specified in the original  // argument list. If so, then the APC is reused for the normal APC for  // whatever mode the caller was in and the "special" routine that is  // invoked before the APC gets control simply deallocates the IRP.  //  KAPC Apc;  //  // CompletionKey - This is the key that is used to distinguish  // individual I/O operations initiated on a single file handle.  //  PVOID CompletionKey;  } Tail;  } IRP;  typedef IRP \*PIRP; |

## 3.4函数调用

### 3.4.1查阅帮助



安装WDK Document，可以方便的使用WDK自带的帮助文档。因为WDK10都是直接使用的网页，不太好使，所以我安装的WDK 7600里面的WDK Document（解压WDK 7600后可以单独安装）。



从筛选里面，可以看出，这个帮助文档里面，只包含内核API相关。我们如果使用MSDN的帮助文档，也是可以看到内核API，但是因为与应用层混淆，不太好区分。

### 3.4.2函数分类

Ex开头，多是内存分配相关

Zw开头，多是文件注册表操作相关

Rtl开头，多是字符串操作相关

Io开头，多是IO管理器相关，处理IRP请求常用到的

Ps开头，多是进程线程相关

Ndis开头，是NDIS网络驱动相关

一些比如c运行时库的函数，最好不要使用，比如malloc等，这些使用会有问题。

## 3.5 WDK编程中的特殊点

### 3.5.1内核编程的主要调用源

单线程用户态程序只有一个调用源（main等入口函数），内核编程中，一个函数往往有多个调用源。

1. 入口函数DriverEntry和卸载函数DriverUnload
2. 各种分发函数（包括普通分发和快速分发函数）
3. 处理请求时设置的完成函数（被系统回调的函数）
4. 其他回调函数

经常需要回溯调用源，去查看一个函数的安全性和可访问性，比如多线程安全性、可重入性、中断级。

### 3.5.2内核编程特殊性

（1）需要非常关心函数的多线程安全性

（2）需要关心函数所处的中断级，因为高中断级无法调用低中断级，是被隔离的。

（3）常用中断级有Passive级、Dispatch级（Dispatch级高于Passive级），下述有几大调用源的中断级说明

|  |  |
| --- | --- |
| 调用源 | 一般的运行中断级 |
| DriverEntry和DriverUnload | Passive级 |
| 各种分发函数 | Passive级 |
| 完成函数 | Dispatch级 |
| 各种NDIS回调函数 | Dispatch级 |

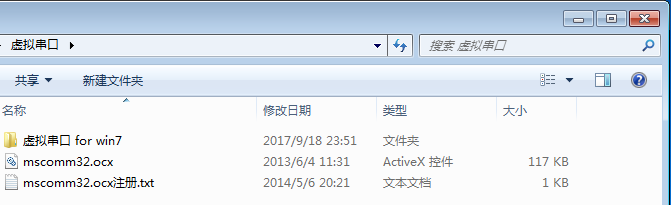
# 4串口的过滤

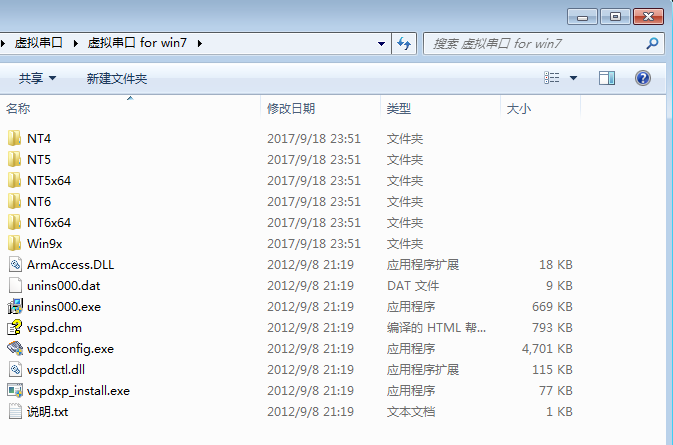
## 4.1准备串口环境

我使用的电脑是联想笔记本，实体机没有串口。我们测试使用的是虚拟机，可以模拟串口行为。

### 4.1.1使用虚拟串口软件（不可行）

首先想到使用虚拟串口软件，看看模拟串口是否可以。虚拟串口软件，我使用的是：

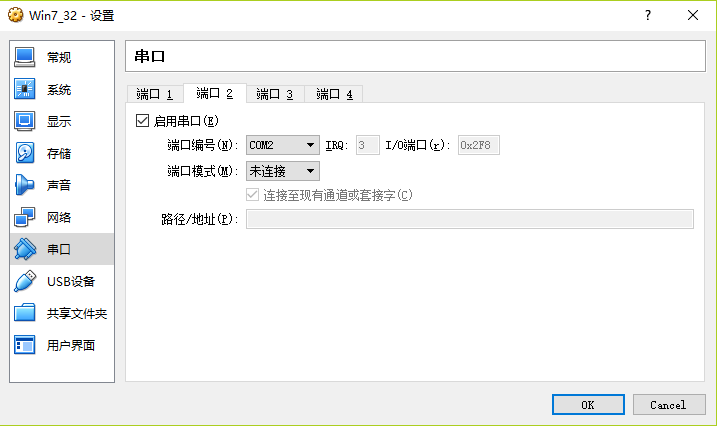




使用前，先注册mscomm32.ocx即可。但是**最终实践证明，这个是不可取的**。

### 4.1.2直接使用虚拟机的串口模拟

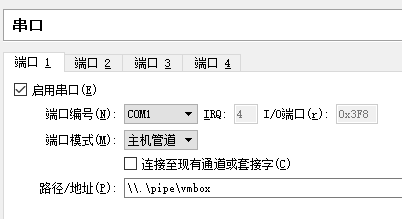
**虚拟机串口设置界面：**



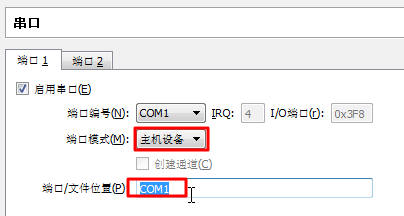
COM1已经被我用于管道，给windbg调试使用。这里使用COM2。

注意：VBox端口模式，一共有四种，如下：

1. 未连接：虚拟机内部模拟的串口硬件，但是实际没有跟外面实体机的串口硬件连接起来
2. 主机管道：实体机与虚拟机之间的通信方式（管道），此时路径地址填写管道地址。

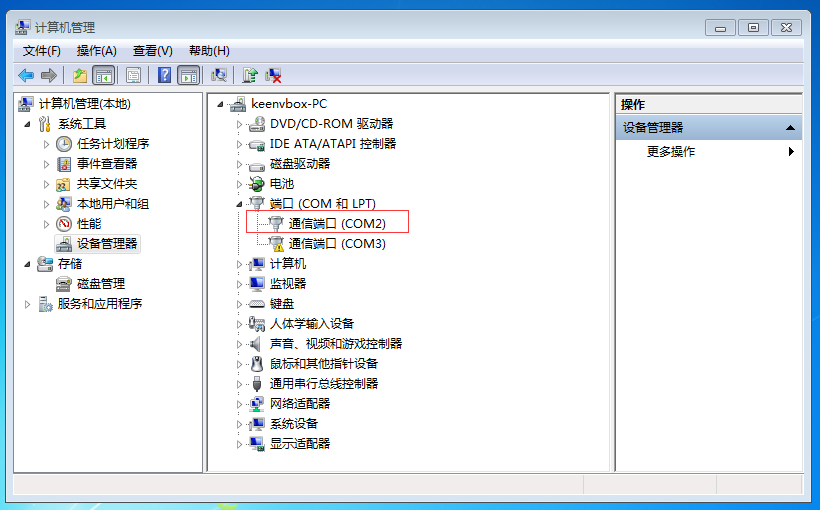


1. 主机设备：虚拟机内部模拟的串口硬件，并且与实体机的串口硬件连接在一起。此时路径/地址，填写对应的COM接口



1. 裸文件：它的功能，是把串口的输出放到一个文件中去
2. Tcp：如果勾选了“连接到现有套接字”，那么它就是作为tcp client。如果没有勾选，就是作为tcp server。如果作为tcp client，意味着它要提供hostname:port；如果它作为tcp server，则只提供一个端口号即可

**最终虚拟机内设备管理器的效果如下：**



## 4.2串口设备通信原理

操作系统串口设备通信整体框架图如下：



### 4.2.1串口编程框架

串口编程框架如下：



### 4.2.2设备绑定



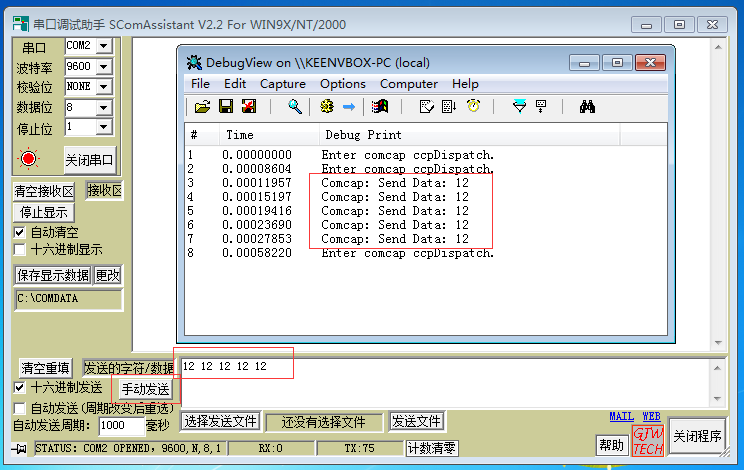
### 4.2.3 IRP传递



## 4.3完整代码

|  |
| --- |
| #include <ntddk.h>  #include <ntstrsafe.h>  #define CCP\_MAX\_COM\_ID 32  #define DELAY\_ONE\_MICROSECOND (-10)  #define DELAY\_ONE\_MILLISECOND (DELAY\_ONE\_MICROSECOND \* 1000)  #define DELAY\_ONE\_SECOND (DELAY\_ONE\_MILLISECOND \* 1000)  static PDEVICE\_OBJECT s\_fltobj[CCP\_MAX\_COM\_ID] = { 0 };  static PDEVICE\_OBJECT s\_nextobj[CCP\_MAX\_COM\_ID] = { 0 };  NTSTATUS ccpAttachDevice(PDRIVER\_OBJECT driver, PDEVICE\_OBJECT oldobj, PDEVICE\_OBJECT \*fltobj, PDEVICE\_OBJECT \*next){  DbgPrint("Enter comcap ccpAttachDevice.\n");  NTSTATUS status = IoCreateDevice(driver, 0, NULL, oldobj->DeviceType, 0, FALSE, fltobj);  if (status != STATUS\_SUCCESS) {  return status;  }  if (oldobj->Flags & DO\_BUFFERED\_IO) {  (\*fltobj)->Flags |= DO\_BUFFERED\_IO;  }  if (oldobj->Flags & DO\_DIRECT\_IO) {  (\*fltobj)->Flags |= DO\_DIRECT\_IO;  }  if (oldobj->Characteristics & FILE\_DEVICE\_SECURE\_OPEN) {  (\*fltobj)->Characteristics |= FILE\_DEVICE\_SECURE\_OPEN;  }  (\*fltobj)->Characteristics |= DO\_POWER\_PAGABLE;  PDEVICE\_OBJECT topdev = IoAttachDeviceToDeviceStack(\*fltobj, oldobj);  if (topdev == NULL) {  // 绑定失败  IoDeleteDevice(\*fltobj);  \*fltobj = NULL;  status = STATUS\_UNSUCCESSFUL;  return status;  }  \*next = topdev;  (\*fltobj)->Flags = (\*fltobj)->Flags & ~DO\_DEVICE\_INITIALIZING;  return STATUS\_SUCCESS;  }  PDEVICE\_OBJECT ccpOpenCom(ULONG id, NTSTATUS\* status){  DbgPrint("Enter comcap ccpOpenCom.\n");  UNICODE\_STRING name\_str;  static WCHAR name[32] = {0};  PFILE\_OBJECT fileobj = NULL;  PDEVICE\_OBJECT devobj = NULL;  memset(name, 0, sizeof(name));  RtlStringCchPrintfW(name, 32, L"\\Device\\Serial%d", id);  RtlInitUnicodeString(&name\_str, name);  \*status = IoGetDeviceObjectPointer(&name\_str, FILE\_ALL\_ACCESS, &fileobj, &devobj);  if (\*status == STATUS\_SUCCESS) {  ObDereferenceObject(fileobj);  }  return devobj;  }  void ccpAttachAllComs(PDRIVER\_OBJECT driver){  DbgPrint("Enter comcap ccpAttachAllComs.\n");  NTSTATUS status;  for (size\_t i = 0; i < CCP\_MAX\_COM\_ID; i++) {  PDEVICE\_OBJECT com\_ob = ccpOpenCom(i, &status);  if (com\_ob == NULL) {  continue;  }  ccpAttachDevice(driver, com\_ob, &s\_fltobj[i], &s\_nextobj[i]);  }  }  NTSTATUS ccpDispatch(PDEVICE\_OBJECT device, PIRP irp){  DbgPrint("Enter comcap ccpDispatch.\n");  PIO\_STACK\_LOCATION irpsp = IoGetCurrentIrpStackLocation(irp);  for (size\_t i = 0; i < CCP\_MAX\_COM\_ID; i++) {  if (s\_fltobj[i] == device) {  // 电源操作，都放过  if (irpsp->MajorFunction == IRP\_MJ\_POWER) {  PoStartNextPowerIrp(irp);  IoSkipCurrentIrpStackLocation(irp);  return PoCallDriver(s\_nextobj[i], irp);  }  // 过滤写请求  if (irpsp->MajorFunction == IRP\_MJ\_WRITE) {  ULONG len = irpsp->Parameters.Write.Length;  // 获取缓冲区  PUCHAR buf = NULL;  if (irp->MdlAddress != NULL) {  buf = (PUCHAR)MmGetSystemAddressForMdlSafe(irp->MdlAddress, NormalPagePriority);  }  else {  buf = (PUCHAR)irp->UserBuffer;  }  if (buf == NULL) {  buf = (PUCHAR)irp->AssociatedIrp.SystemBuffer;  }  for (size\_t j = 0; j < len; j++) {  DbgPrint("Comcap: Send Data: %2x ", buf[j]);  }  }  IoSkipCurrentIrpStackLocation(irp);  return IoCallDriver(s\_nextobj[i], irp);  }  }  return STATUS\_SUCCESS;  }  void ccpUnload(PDRIVER\_OBJECT drv){  UNREFERENCED\_PARAMETER(drv);  // 先一个个解除绑定  for (size\_t i = 0; i < CCP\_MAX\_COM\_ID; i++) {  if (s\_nextobj[i] != NULL) {  IoDetachDevice(s\_nextobj[i]);  }  }  // 睡眠5s，等待所有IRP结束  LARGE\_INTEGER interval;  interval.QuadPart = 5 \* 1000 \* DELAY\_ONE\_MILLISECOND;  KeDelayExecutionThread(KernelMode, FALSE, &interval);  // 删除这些设备  for (size\_t i = 0; i < CCP\_MAX\_COM\_ID; i++) {  if (s\_fltobj[i] != NULL) {  IoDeleteDevice(s\_fltobj[i]);  }  }  }  NTSTATUS DriverEntry(PDRIVER\_OBJECT driver, PUNICODE\_STRING reg\_path){  UNREFERENCED\_PARAMETER(reg\_path);  DbgPrint("Enter comcap.\n");  for (size\_t i = 0; i < IRP\_MJ\_MAXIMUM\_FUNCTION; i++) {  driver->MajorFunction[i] = ccpDispatch;  }  driver->DriverUnload = ccpUnload;  ccpAttachAllComs(driver);  return STATUS\_SUCCESS;  } |

## 4.4实践效果



使用串口调试助手，连接我们的COM2接口（因为前面只有COM2接口设置是正常的），然后模拟串口发数据。

使用DbgView.exe，来接收数据，观察收到的数据都是什么。

# 5键盘的过滤

## 5.1键盘设备原理



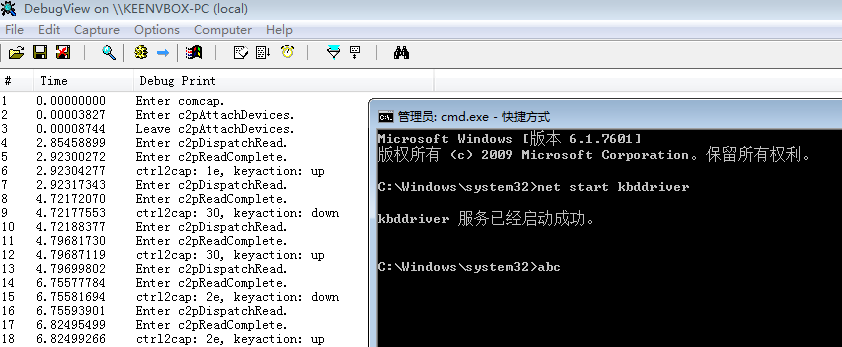
## 5.2键盘过滤框架



## 5.3键盘过滤完整代码

|  |
| --- |
| #include <ntddk.h>  #include <ntstrsafe.h>  extern POBJECT\_TYPE\* IoDriverObjectType;  #define KBD\_DRIVER\_NAME L"\\Driver\\Kbdclass"  #define DELAY\_ONE\_MICROSECOND (-10)  #define DELAY\_ONE\_MILLISECOND (DELAY\_ONE\_MICROSECOND \* 1000)  #define DELAY\_ONE\_SECOND (DELAY\_ONE\_MILLISECOND \* 1000)  ULONG gc2pKeyCount;  // 声明非公开函数  NTSTATUS ObReferenceObjectByName(PUNICODE\_STRING ObjectName,  ULONG Attributes,  PACCESS\_STATE AccessState,  ACCESS\_MASK DesiredAccess,  POBJECT\_TYPE ObjectType,  KPROCESSOR\_MODE AccessMode,  PVOID ParseContext,  PVOID \*Object);  // 自定义设备扩展结构  typedef struct \_C2P\_DEV\_EXT{  ULONG NodeSize; // 这个结构大小  PDEVICE\_OBJECT pFilterDeviceObject; // 过滤设备对象  KSPIN\_LOCK IoRequestSpinLock; // 同时调用时的保护锁  KEVENT IoInProgressEvent; // 进程间同步处理  PDEVICE\_OBJECT TargetDeviceObject; // 绑定的设备对象  PDEVICE\_OBJECT LowerDeviceObject; // 绑定前底层设备对象  }C2P\_DEV\_EXT, \*PC2P\_DEV\_EXT;  NTSTATUS c2pDevExtInit(PC2P\_DEV\_EXT devExt,  PDEVICE\_OBJECT pFilterOb,  PDEVICE\_OBJECT pTargetOb,  PDEVICE\_OBJECT pLowerOb){  memset(devExt, 0, sizeof(C2P\_DEV\_EXT));  devExt->NodeSize = sizeof(C2P\_DEV\_EXT);  devExt->pFilterDeviceObject = pFilterOb;  devExt->TargetDeviceObject = pTargetOb;  devExt->LowerDeviceObject = pLowerOb;  KeInitializeSpinLock(&(devExt->IoRequestSpinLock));  KeInitializeEvent(&(devExt->IoInProgressEvent), NotificationEvent, FALSE);  return STATUS\_SUCCESS;  }  NTSTATUS c2pAttachDevices(PDRIVER\_OBJECT driver, PUNICODE\_STRING reg\_path){  UNREFERENCED\_PARAMETER(reg\_path);  DbgPrint("Enter c2pAttachDevices.\n");  UNICODE\_STRING uniNtNameString;  PDRIVER\_OBJECT kbdDriverObject = NULL;  // 初始化kdbClass驱动的名字为一个字符串  // 打开驱动对象  RtlInitUnicodeString(&uniNtNameString, KBD\_DRIVER\_NAME);  NTSTATUS status = ObReferenceObjectByName(  &uniNtNameString,  OBJ\_CASE\_INSENSITIVE,  NULL,  0,  \*IoDriverObjectType,  KernelMode,  NULL,  &kbdDriverObject  );  if (!NT\_SUCCESS(status)) {  DbgPrint("ObReferenceObjectByName error. %d", status);  return status;  }  else {  // 会导致对驱动对象引用计数+1，需要解引用  ObReferenceObject(kbdDriverObject);  }  // 找到设备链中第一个设备，开始遍历  PDEVICE\_OBJECT pFilterDeviceObject = NULL;  PDEVICE\_OBJECT pTargetDeviceObject = kbdDriverObject->DeviceObject;  while (pTargetDeviceObject) {  // 生成虚拟过滤设备  status = IoCreateDevice(driver, sizeof(C2P\_DEV\_EXT), NULL, pTargetDeviceObject->DeviceType, 0, FALSE, &pFilterDeviceObject);  if (!NT\_SUCCESS(status)) {  DbgPrint("IoCreateDevice error.");  return status;  }  PDEVICE\_OBJECT pLowerDeviceObject = IoAttachDeviceToDeviceStack(pFilterDeviceObject, pTargetDeviceObject);  if (!pLowerDeviceObject) {  DbgPrint("IoAttachDeviceToDeviceStack error.");  IoDeleteDevice(pFilterDeviceObject);  pFilterDeviceObject = NULL;  return status;  }  // 设备扩展  PC2P\_DEV\_EXT devExt = (PC2P\_DEV\_EXT)(pFilterDeviceObject->DeviceExtension);  c2pDevExtInit(devExt, pFilterDeviceObject, pTargetDeviceObject, pLowerDeviceObject);  pFilterDeviceObject->DeviceType = pLowerDeviceObject->DeviceType;  pFilterDeviceObject->Characteristics = pLowerDeviceObject->Characteristics;  pFilterDeviceObject->StackSize = pLowerDeviceObject->StackSize + 1;  pFilterDeviceObject->Flags |= pLowerDeviceObject->Flags & (DO\_BUFFERED\_IO | DO\_DIRECT\_IO | DO\_POWER\_PAGABLE);  pTargetDeviceObject = pTargetDeviceObject->NextDevice;  }  DbgPrint("Leave c2pAttachDevices.\n");  return status;  }  VOID c2pUnload(PDRIVER\_OBJECT driver)  {  UNREFERENCED\_PARAMETER(driver);  DbgPrint("Enter c2pUnload.\n");  // 当前线程设置为低实模式，以便让它的运行尽量少影响其他程序  PRKTHREAD CurrentThread = KeGetCurrentThread();  KeSetPriorityThread(CurrentThread, LOW\_REALTIME\_PRIORITY);  PDEVICE\_OBJECT pOb = driver->DeviceObject;  while (pOb) {  IoDetachDevice(((PC2P\_DEV\_EXT)(pOb->DeviceExtension))->TargetDeviceObject);  IoDeleteDevice(pOb);  pOb = pOb->NextDevice;  }  ASSERT(NULL == driver->DeviceObject);  LARGE\_INTEGER lDelay = RtlConvertLongToLargeInteger(100 \* DELAY\_ONE\_MILLISECOND);  while (gc2pKeyCount) {  KeDelayExecutionThread(KernelMode, FALSE, &lDelay);  }  DbgPrint("Leave c2pUnload.\n");  }  NTSTATUS c2pDispatchGeneral(PDEVICE\_OBJECT device, PIRP irp){  UNREFERENCED\_PARAMETER(device);  DbgPrint("Enter c2pDispatchGeneral.\n");  // 跳过当前过滤设备，直接交付给真实设备  IoSkipCurrentIrpStackLocation(irp);  return IoCallDriver(((PC2P\_DEV\_EXT)(device->DeviceExtension))->LowerDeviceObject, irp);  }  NTSTATUS c2pReadComplete(PDEVICE\_OBJECT device, PIRP irp, PVOID context){  UNREFERENCED\_PARAMETER(device);  UNREFERENCED\_PARAMETER(context);  DbgPrint("Enter c2pReadComplete.\n");  if (NT\_SUCCESS(irp->IoStatus.Status)) {  PKEYBOARD\_INPUT\_DATA keyData = (PKEYBOARD\_INPUT\_DATA)irp->AssociatedIrp.SystemBuffer;  ULONG numKeys = irp->IoStatus.Information / sizeof(KEYBOARD\_INPUT\_DATA);  for (size\_t i = 0; i < numKeys; i++) {  DbgPrint("ctrl2cap: %2x, keyaction: %s",  keyData[i].MakeCode, keyData[i].Flags ? "up" : "down");  }  }  gc2pKeyCount--;  if (irp->PendingReturned) {  IoMarkIrpPending(irp);  }  return irp->IoStatus.Status;  }  NTSTATUS c2pDispatchRead(PDEVICE\_OBJECT device, PIRP irp){  DbgPrint("Enter c2pDispatchRead.\n");  // write是已知源数据，过滤驱动是知道该数据是什么，只需要针对该数据做处理  // read是不知道源数据，过滤驱动必须把请求交给真实设备驱动，等真实设备驱动完成请求后，方能拿到对应数据  KEVENT waitEvent;  KeInitializeEvent(&waitEvent, NotificationEvent, FALSE);  if (irp->CurrentLocation == 1) {  NTSTATUS status = STATUS\_INVALID\_DEVICE\_REQUEST;  irp->IoStatus.Status = status;  irp->IoStatus.Information = 0;  IoCompleteRequest(irp, IO\_NO\_INCREMENT);  return status;  }  gc2pKeyCount++;  IoCopyCurrentIrpStackLocationToNext(irp); // 针对当前irp，做了修改，所以需要使用Copy而不是Skip  IoSetCompletionRoutine(irp, c2pReadComplete, device, TRUE, TRUE, TRUE);  return IoCallDriver(((PC2P\_DEV\_EXT)(device->DeviceExtension))->LowerDeviceObject, irp);  }  NTSTATUS c2pPower(PDEVICE\_OBJECT device, PIRP irp){  DbgPrint("Enter c2pPower.\n");  PoStartNextPowerIrp(irp);  IoSkipCurrentIrpStackLocation(irp);  return PoCallDriver(((PC2P\_DEV\_EXT)(device->DeviceExtension))->LowerDeviceObject, irp);  }  NTSTATUS c2pPnp(PDEVICE\_OBJECT device, PIRP irp){  DbgPrint("Enter c2pPnp.\n");  NTSTATUS status;  // 即插即用，当设备拔出时，需要解除绑定  PIO\_STACK\_LOCATION irpStack = IoGetCurrentIrpStackLocation(irp);  switch (irpStack->MinorFunction)  {  case IRP\_MN\_REMOVE\_DEVICE:  DbgPrint("IRP\_MN\_REMOVE\_DEVICE.\n");  // 首先把请求传递下去，然后解除绑定  IoSkipCurrentIrpStackLocation(irp);  status = IoCallDriver(((PC2P\_DEV\_EXT)(device->DeviceExtension))->LowerDeviceObject, irp);  IoDetachDevice(((PC2P\_DEV\_EXT)(device->DeviceExtension))->LowerDeviceObject);  IoDeleteDevice(device);  break;  default:  IoSkipCurrentIrpStackLocation(irp);  status = IoCallDriver(((PC2P\_DEV\_EXT)(device->DeviceExtension))->LowerDeviceObject, irp);  break;  }  DbgPrint("Leave c2pPnp.\n");  return status;  }  NTSTATUS DriverEntry(PDRIVER\_OBJECT driver, PUNICODE\_STRING reg\_path)  {  UNREFERENCED\_PARAMETER(reg\_path);  DbgPrint("Enter comcap.\n");  for (size\_t i = 0; i < IRP\_MJ\_MAXIMUM\_FUNCTION; i++) {  driver->MajorFunction[i] = c2pDispatchGeneral;  }  driver->MajorFunction[IRP\_MJ\_READ] = c2pDispatchRead;  driver->MajorFunction[IRP\_MJ\_POWER] = c2pPower;  driver->MajorFunction[IRP\_MJ\_PNP] = c2pPnp;  driver->DriverUnload = c2pUnload;  c2pAttachDevices(driver, reg\_path);  return STATUS\_SUCCESS;  } |

## 5.4实践效果



## 5.5键盘扫描码对应表

|  |
| --- |
| Keyboard Scan Codes (Numerical Order)  ---------------+---------------+---------------+---------------+---------------  HEX DEC keys |HEX DEC keys |HEX DEC keys |HEX DEC keys |HEX DEC keys  ---------------+---------------+---------------+---------------+---------------  |10 16 Q |20 32 D |30 48 B |40 64 F6  01 1 ESC |11 17 W |21 33 F |31 49 N |41 65 F7  02 2 1 |12 18 E |22 34 G |32 50 M |42 66 F8  03 3 2 |13 19 R |23 35 H |33 51 , |43 67 F9  04 4 3 |14 20 T |24 36 J |34 52 . |44 68 F10  05 5 4 |15 21 Y |25 37 K |35 53 / |45 69 Num  06 6 5 |16 22 U |26 38 L |36 54 R Shift|46 70 Scroll  07 7 6 |17 23 I |27 39 ; |37 55 PrtSc |47 71 Home  08 8 7 |18 24 O |28 40 ' |38 56 Alt |48 72 Up  09 9 8 |19 25 P |29 41 ` |39 57 Space |49 73 PgUp  0A 10 9 |1A 26 [ |2A 42 L Shift|3A 58 Caps |4A 74 -  0B 11 0 |1B 27 ] |2B 43 \ |3B 59 F1 |4B 75 Left  0C 12 - |1C 28 |2C 44 Z |3C 60 F2 |4C 76 Center  0D 13 = |1D 29 CTRL |2D 45 X |3D 61 F3 |4D 77 Right  0E 14 bs |1E 30 A |2E 46 C |3E 62 F4 |4E 78 +  0F 15 Tab |1F 31 S |2F 47 V |3F 63 F5 |4F 79 End  ---------------+---------------+---------------+---------------+---------------  50 80 Down | | | |  51 81 PgDn | | | |  52 82 Ins | | | |  53 83 Del | | | |  ---------------+---------------+---------------+---------------+--------------- |

## 5.6 Hook

### 5.6.1 Hook类驱动分发函数完整代码

前面讲解了，可以获取一个kbdClass类驱动对象DriverObject的指针，那么就可以修改它的分发函数，修改为我们自己的函数。要知道的是，内核对象，都是在同一个内存空间，所以其他驱动是可以调用我们自己的驱动对象的函数的。

完整代码实现如下：

|  |
| --- |
| #include <ntddk.h>  #include <ntstrsafe.h>  #include <ntddkbd.h>  #pragma warning(disable: 4054)  #define KBD\_DRIVER\_NAME L"\\Driver\\Kbdclass"  #define DELAY\_ONE\_MICROSECOND (-10)  #define DELAY\_ONE\_MILLISECOND (DELAY\_ONE\_MICROSECOND \* 1000)  #define DELAY\_ONE\_SECOND (DELAY\_ONE\_MILLISECOND \* 1000)  ULONG gc2pKeyCount;  PDRIVER\_DISPATCH gOldDispatchFunc;  // 声明非公开函数  NTSTATUS ObReferenceObjectByName(PUNICODE\_STRING ObjectName,  ULONG Attributes,  PACCESS\_STATE AccessState,  ACCESS\_MASK DesiredAccess,  POBJECT\_TYPE ObjectType,  KPROCESSOR\_MODE AccessMode,  PVOID ParseContext,  PVOID \*Object);  PDRIVER\_OBJECT OpenKbdClass(){  // 找到kbdclass类驱动对象地址  UNICODE\_STRING uniNtNameString;  PDRIVER\_OBJECT kbdDriverObject = NULL;  // 初始化kdbClass驱动的名字为一个字符串  // 打开驱动对象  RtlInitUnicodeString(&uniNtNameString, KBD\_DRIVER\_NAME);  NTSTATUS status = ObReferenceObjectByName(  &uniNtNameString,  OBJ\_CASE\_INSENSITIVE,  NULL,  0,  \*IoDriverObjectType,  KernelMode,  NULL,  &kbdDriverObject  );  if (!NT\_SUCCESS(status)) {  DbgPrint("ObReferenceObjectByName error. %d", status);  return NULL;  }  else {  // 会导致驱动对象引用计数+1，需要解引用  ObReferenceObject(kbdDriverObject);  }  return kbdDriverObject;  }  VOID c2pUnload(PDRIVER\_OBJECT driver){  UNREFERENCED\_PARAMETER(driver);  DbgPrint("Enter c2pUnload.\n");  // 当前线程设置为低实模式，以便让它的运行尽量少影响其他程序  PRKTHREAD CurrentThread = KeGetCurrentThread();  KeSetPriorityThread(CurrentThread, LOW\_REALTIME\_PRIORITY);  // 还原分发函数  PDRIVER\_OBJECT pDrv = OpenKbdClass();  if (pDrv != NULL) {  pDrv->MajorFunction[IRP\_MJ\_READ] = gOldDispatchFunc;  }  ASSERT(NULL == driver->DeviceObject);  LARGE\_INTEGER lDelay = RtlConvertLongToLargeInteger(100 \* DELAY\_ONE\_MILLISECOND);  while (gc2pKeyCount) {  KeDelayExecutionThread(KernelMode, FALSE, &lDelay);  }  DbgPrint("Leave c2pUnload.\n");  }  NTSTATUS c2pReadComplete(PDEVICE\_OBJECT device, PIRP irp, PVOID context)  {  UNREFERENCED\_PARAMETER(device);  UNREFERENCED\_PARAMETER(context);  DbgPrint("Enter c2pReadComplete.\n");  if (NT\_SUCCESS(irp->IoStatus.Status)) {  PKEYBOARD\_INPUT\_DATA keyData = (PKEYBOARD\_INPUT\_DATA)irp->AssociatedIrp.SystemBuffer;  ULONG numKeys = irp->IoStatus.Information / sizeof(KEYBOARD\_INPUT\_DATA);  for (size\_t i = 0; i < numKeys; i++)  {  DbgPrint("hook kbdclass: %2x, keyaction: %s",  keyData[i].MakeCode, keyData[i].Flags ? "up" : "down");  }  }  gc2pKeyCount--;  if (irp->PendingReturned) {  IoMarkIrpPending(irp);  }  return irp->IoStatus.Status;  }  NTSTATUS c2pDispatchRead(PDEVICE\_OBJECT device, PIRP irp){  DbgPrint("Enter c2pDispatchRead.\n");  if (irp->CurrentLocation == 1) {  NTSTATUS status = STATUS\_INVALID\_DEVICE\_REQUEST;  irp->IoStatus.Status = status;  irp->IoStatus.Information = 0;  IoCompleteRequest(irp, IO\_NO\_INCREMENT);  return status;  }  PIO\_STACK\_LOCATION irpSp = IoGetCurrentIrpStackLocation(irp);  irpSp->Control = SL\_INVOKE\_ON\_SUCCESS | SL\_INVOKE\_ON\_ERROR | SL\_INVOKE\_ON\_CANCEL;  //保留原来的完成函数，如果有的话  irpSp->Context = (PVOID)irpSp->CompletionRoutine;  irpSp->CompletionRoutine = (PIO\_COMPLETION\_ROUTINE)c2pReadComplete;  gc2pKeyCount++;  return gOldDispatchFunc(device, irp);  }  NTSTATUS DriverEntry(PDRIVER\_OBJECT driver, PUNICODE\_STRING reg\_path){  UNREFERENCED\_PARAMETER(reg\_path);  DbgPrint("Enter kbddriver hook class.\n");  driver->DriverUnload = c2pUnload;  PDRIVER\_OBJECT pDrv = OpenKbdClass();  if (pDrv != NULL) {  // 只替换Read分发函数  gOldDispatchFunc = pDrv->MajorFunction[IRP\_MJ\_READ];  // 进行原子交换  InterlockedExchangePointer((volatile PVOID\*)&(pDrv->MajorFunction[IRP\_MJ\_READ]), (PVOID)c2pDispatchRead);  }  return STATUS\_SUCCESS;  } |

### 5.6.2 Hook端口驱动

#### 5.6.2.1键盘端口驱动原理



#### 5.6.2.2键盘端口Hook完整代码

|  |
| --- |
| #include <ntddk.h>  #include <ntstrsafe.h>  #include <ntddkbd.h>  #pragma warning(disable: 4054)  extern POBJECT\_TYPE\* IoDriverObjectType;  #define KBD\_DRIVER\_NAME L"\\Driver\\Kbdclass"  #define USBKBD\_DRIVER\_NAME L"\\Driver\\kbdhid"  #define PS2KBD\_DRIVER\_NAME L"\\Driver\\i8042prt"  // 声明非公开函数  NTSTATUS ObReferenceObjectByName(PUNICODE\_STRING ObjectName,  ULONG Attributes,  PACCESS\_STATE AccessState,  ACCESS\_MASK DesiredAccess,  POBJECT\_TYPE ObjectType,  KPROCESSOR\_MODE AccessMode,  PVOID ParseContext,  PVOID \*Object);  // 要搜索是回调函数  typedef VOID(\_stdcall \*KEYBOARDCLASSSERVICECALLBACK)(  IN PDEVICE\_OBJECT DeviceObject,  IN PKEYBOARD\_INPUT\_DATA InputDataStart,  IN PKEYBOARD\_INPUT\_DATA InputDataEnd,  IN OUT PULONG InputDataConsumed);  // 自定义结构保存回调  typedef struct \_KBD\_CALLBACK{  PDEVICE\_OBJECT classDeviceObject;  KEYBOARDCLASSSERVICECALLBACK serviceCallBack;  }KBD\_CALLBACK, \*PKBD\_CALLBACK;  KBD\_CALLBACK gKbdCallBack = { 0 };  void MyKeyboardClassServiceCallBack(  IN PDEVICE\_OBJECT DeviceObject,  IN PKEYBOARD\_INPUT\_DATA InputDataStart,  IN PKEYBOARD\_INPUT\_DATA InputDataEnd,  IN OUT PULONG InputDataConsumed){  DbgPrint("Enter MyKeyboardClassServiceCallBack");  for (PKEYBOARD\_INPUT\_DATA keyData = InputDataStart; keyData <= InputDataEnd; keyData++) {  DbgPrint("hook kbdport: %2x, keyaction: %s",  keyData->MakeCode, keyData->Flags ? "up" : "down");  }  gKbdCallBack.serviceCallBack(DeviceObject, InputDataStart, InputDataEnd, InputDataConsumed);  }  PDRIVER\_OBJECT OpenKbdDriver(PCWSTR driverName){  // 找到kbdclass类驱动对象地址  UNICODE\_STRING uniNtNameString;  PDRIVER\_OBJECT kbdDriverObject = NULL;  // 初始化kdbClass驱动的名字为一个字符串  // 打开驱动对象  RtlInitUnicodeString(&uniNtNameString, driverName);  DbgPrint("%wZ\n", uniNtNameString);  NTSTATUS status = ObReferenceObjectByName(  &uniNtNameString,  OBJ\_CASE\_INSENSITIVE,  NULL,  0,  \*IoDriverObjectType,  KernelMode,  NULL,  &kbdDriverObject  );  if (!NT\_SUCCESS(status)) {  DbgPrint("OpenKbdDriver ObReferenceObjectByName error. %d\n", status);  return NULL;  }  else {  // 会导致驱动对象引用计数+1，需要解引用  ObReferenceObject(kbdDriverObject);  }  return kbdDriverObject;  }  NTSTATUS SearchServiceCallBack(IN PDRIVER\_OBJECT DriverObject){  DbgPrint("Enter SearchServiceCallBack.\n");  UNREFERENCED\_PARAMETER(DriverObject);  NTSTATUS status = STATUS\_UNSUCCESSFUL;  // 分别打开USB键盘端口驱动和PS2键盘驱动，看看哪个能打开，说明当前正在使用的驱动对象就是谁  PDRIVER\_OBJECT KbdhidDriverObject = OpenKbdDriver(USBKBD\_DRIVER\_NAME);  PDRIVER\_OBJECT Kbdi8042DriverObject = OpenKbdDriver(PS2KBD\_DRIVER\_NAME);  // 两种键盘同时存在，这种极端情况不考虑  if (KbdhidDriverObject && Kbdi8042DriverObject) {  DbgPrint("SearchServiceCallBack two kbd exist.\n");  return STATUS\_UNSUCCESSFUL;  }  // 两种键盘同时不存在，直接返回失败  if (!KbdhidDriverObject && !Kbdi8042DriverObject) {  DbgPrint("SearchServiceCallBack two kbd not exist.\n");  return STATUS\_UNSUCCESSFUL;  }  PDRIVER\_OBJECT UsingKbdDriverObject = KbdhidDriverObject ? KbdhidDriverObject : Kbdi8042DriverObject;  PDEVICE\_OBJECT UsingDeviceObject = UsingKbdDriverObject->DeviceObject;  // 上述UsingDeviceExt设备扩展里，应该有一个函数指针，是存在于KbdClass中的  // KbdClass的一个设备对象的指针，也保存在UsingDeviceExt设备扩展里  PDRIVER\_OBJECT KbdClassObject = OpenKbdDriver(KBD\_DRIVER\_NAME);  if (!KbdClassObject) {  DbgPrint("SearchServiceCallBack KbdClass not exist.\n");  return STATUS\_UNSUCCESSFUL;  }  PVOID KbdClassDriverStart = KbdClassObject->DriverStart;  ULONG KbdClassDriverSize = KbdClassObject->DriverSize;  PVOID\* AddrServiceCallBack = NULL;  while (UsingDeviceObject) {  PVOID UsingDeviceExt = UsingDeviceObject->DeviceExtension;  // 遍历KbdClass的设备对象，确认是否在UsingDeviceExt中存在  PDEVICE\_OBJECT pTargetDeviceObject = KbdClassObject->DeviceObject;  while (pTargetDeviceObject) {  PUCHAR pDeviceExt = (PUCHAR)UsingDeviceExt;  for (size\_t i = 0; i < 4096; i++, pDeviceExt += sizeof(PUCHAR)){  if (!MmIsAddressValid(pDeviceExt)){  break;  }  if (gKbdCallBack.classDeviceObject && gKbdCallBack.serviceCallBack){  status = STATUS\_SUCCESS;  break;  }  PVOID tmp = \*(PVOID\*)pDeviceExt;  if (tmp == pTargetDeviceObject){  // 找到这里，说明在端口驱动的设备扩展里，保存了一个Kbdclass类驱动的设备对象  gKbdCallBack.classDeviceObject = (PDEVICE\_OBJECT)tmp;  DbgPrint("finded classDeviceObject %8x\n", tmp);  continue;  }  if ((tmp > KbdClassDriverStart) &&  (tmp < (PVOID)((PUCHAR)KbdClassDriverStart + KbdClassDriverSize)) &&  MmIsAddressValid(tmp)){  gKbdCallBack.serviceCallBack = (KEYBOARDCLASSSERVICECALLBACK)tmp;  AddrServiceCallBack = (PVOID\*)pDeviceExt;  DbgPrint("SearchServiceCallBack servicecallback : %8x\naddrServiceCallBack : %8x\n", tmp, AddrServiceCallBack);  }  }  pTargetDeviceObject = pTargetDeviceObject->NextDevice;  }  UsingDeviceObject = UsingDeviceObject->NextDevice;  }  // 成功找到了回调地址，就替换成我们自己的  if (AddrServiceCallBack && gKbdCallBack.serviceCallBack) {  DbgPrint("SearchServiceCallBack Hook KeyboardClassServiceCallBack\n");  \*AddrServiceCallBack = MyKeyboardClassServiceCallBack;  }  else {  DbgPrint("SearchServiceCallBack error search.\n");  }  return status;  }  VOID c2pUnload(PDRIVER\_OBJECT driver){  UNREFERENCED\_PARAMETER(driver);  DbgPrint("Leave c2pUnload.\n");  }  NTSTATUS DriverEntry(PDRIVER\_OBJECT driver, PUNICODE\_STRING reg\_path){  UNREFERENCED\_PARAMETER(reg\_path);  DbgPrint("Enter kbddriver hook class.\n");  driver->DriverUnload = c2pUnload;  SearchServiceCallBack(driver);  return STATUS\_SUCCESS;  } |

**注意：*经测试，此代码行不通。***

**网上找到的一种说法是**：*就算成功Hook掉这个回调函数,系统也不会去调用我们的函数,当系统在加载驱动的时候,i8042prt就记住了该回调函数真实地址,在回调的时候,不需要再去查设备扩展里的这个回调函数地址了。*

不过，找到这个回调地址，我们可以进行模拟键盘点击，跳过游戏的模拟键盘保护

## 5.7反Hook – 中断反过滤

如果键盘输入不想被驱动截胡，那么根据前面的方法，自己的过滤驱动，需要置于更前期获取原始数据，通过一些加密手段，才能避免被盗号木马获取。但是，这个方法，只适用于盗号木马也使用过滤驱动的类型，而针对采用Hook分发函数直接改写的，是无法避免的，因为别人都直接Hook了更底层的东西（端口驱动）。

那么我们只能Hook比端口驱动更底层的——Hook键盘中断。虽然黑客也能使用Hook键盘中断，但是由于中断永远是最后一个生效，只要保证在键盘输入的时机Hook，我们就可以保证自己是最后一个。因为我们自己的产品在输入密码的时候，我们是知道的。

### 5.7.1中断原理



每个CPU核心有自己的IDT。IDT在内存中的地址不固定，可以使用sidt指令，可以获取到idt的内存地址。

### 5.7.1中断服务例程重定向

|  |
| --- |
| #include <ntddk.h>  #include <ntstrsafe.h>  #include <ntddkbd.h>  #pragma warning(disable: 4054)  typedef unsigned char P2C\_U8;  typedef unsigned short P2C\_U16;  typedef unsigned long P2C\_U32;  // 从sidt指令获得如下一个结构  #pragma pack(push, 1)  typedef struct \_P2C\_IDTR {  P2C\_U16 limit;  P2C\_U32 base;  }P2C\_IDTR, \*PP2C\_IDTR;  #pragma pack(pop)  void \*p2cGetIdt(){  P2C\_IDTR idtr;  \_asm sidt idtr  return (void\*)idtr.base;  }  // IDT表单元  // 跳转地址保存在offset\_low和offset\_high中  #pragma pack(push, 1)  typedef struct \_P2C\_IDT\_ENTRY {  P2C\_U16 offset\_low;  P2C\_U16 selector;  P2C\_U8 reserved;  P2C\_U8 type : 4;  P2C\_U8 always0 : 1;  P2C\_U8 dpl : 2;  P2C\_U8 present : 1;  P2C\_U16 offset\_high;  }P2C\_IDTENTRY, \*PP2C\_IDTENTRY;  #pragma pack(pop)  #define P2C\_MAKELONG(low, high) ( ((P2C\_U32)(low) & 0xffff) | (((P2C\_U32)(high) & 0xffff) << 16) )  #define P2C\_LOW16\_OF\_32(data) ( (P2C\_U16)((P2C\_U32)(data) & 0xffff) )  #define P2C\_HIGH16\_OF\_32(data) ( (P2C\_U16)((P2C\_U32)(data) >> 16) )  void\* g\_p2c\_old = NULL;  #define OBUFFER\_FULL 0x02  #define IBUFFER\_FULL 0x01  ULONG p2cWaitForKdRead(){  int i = 100;  P2C\_U8 mychar;  do {  \_asm in al,0x64  \_asm mov mychar,al  KeStallExecutionProcessor(50);  if (!(mychar & OBUFFER\_FULL))  {  break;  }  } while (i--);  if (i) return TRUE;  return FALSE;  }  ULONG p2cWaitForKdWrite(){  int i = 100;  P2C\_U8 mychar;  do {  \_asm in al, 0x64  \_asm mov mychar, al  KeStallExecutionProcessor(50);  if (!(mychar & IBUFFER\_FULL))  {  break;  }  } while (i--);  if (i) return TRUE;  return FALSE;  }  void p2cUserFilter(){  DbgPrint("Enter p2cUserFilter\r\n");  static P2C\_U8 sch\_pre = 0;  P2C\_U8 sch;  p2cWaitForKdRead();  \_asm in al,0x60  \_asm mov sch,al  // 端口的数据一旦被读出来，就没有了，这个时候，需要强制回写  if (sch\_pre != sch) {  DbgPrint("p2c: scan code = %2x\r\n", sch);  sch\_pre = sch;  \_asm mov al, 0xd2;  \_asm out 0x64,al  p2cWaitForKdWrite();  \_asm mov al, sch  \_asm out 0x60,al  }  }  \_\_declspec(naked) p2cInterruptProc(){  \_\_asm {  pushad // 保存所有的通用寄存器  pushfd // 保存标志寄存器  call p2cUserFilter // 调用我们自己的一个函数，这个函数将实现我们自己的一个功能  popfd  popad  jmp g\_p2c\_old // 跳转到原来的中断服务程序  }  }  void p2cHookInt93(BOOLEAN hook\_or\_unhook){  PP2C\_IDTENTRY idt\_addr = (PP2C\_IDTENTRY)p2cGetIdt();  idt\_addr += 0x93; // 第0x93个中断索引  DbgPrint("p2c: the pre address = %x.\r\n", (void\*)P2C\_MAKELONG(idt\_addr->offset\_low, idt\_addr->offset\_high));  if (hook\_or\_unhook) {  DbgPrint("p2c: try to hook interrupt.\r\n");  g\_p2c\_old = (void\*)P2C\_MAKELONG(idt\_addr->offset\_low, idt\_addr->offset\_high);  idt\_addr->offset\_low = P2C\_LOW16\_OF\_32(p2cInterruptProc);  idt\_addr->offset\_high = P2C\_HIGH16\_OF\_32(p2cInterruptProc);  }  else {  DbgPrint("p2c: try to recovery interrupt.\r\n");  idt\_addr->offset\_low = P2C\_LOW16\_OF\_32(g\_p2c\_old);  idt\_addr->offset\_high = P2C\_HIGH16\_OF\_32(g\_p2c\_old);  g\_p2c\_old = NULL;  }  DbgPrint("p2c: the current address = %x.\r\n", (void\*)P2C\_MAKELONG(idt\_addr->offset\_low, idt\_addr->offset\_high));  }  VOID c2pUnload(PDRIVER\_OBJECT driver){  UNREFERENCED\_PARAMETER(driver);  p2cHookInt93(FALSE);  LARGE\_INTEGER interval;  interval.QuadPart = 5 \* 1000 \* DELAY\_ONE\_MILLISECOND;  KeDelayExecutionThread(KernelMode, FALSE, &interval);  DbgPrint("Leave c2pUnload.\n");  }  NTSTATUS DriverEntry(PDRIVER\_OBJECT driver, PUNICODE\_STRING reg\_path){  UNREFERENCED\_PARAMETER(reg\_path);  DbgPrint("Enter kbddriver hook class.\n");  driver->DriverUnload = c2pUnload;  p2cHookInt93(TRUE);  return STATUS\_SUCCESS;  } |

**注意：经实测，在虚拟机中又行不通，怀疑中断向量表被修改了**

### 5.7.2利用IOAPIC重定位中断处理函数

|  |
| --- |
| #include <ntddk.h>  #include <ntstrsafe.h>  #include <ntddkbd.h>  #pragma warning(disable: 4054)  #define DELAY\_ONE\_MICROSECOND (-10)  #define DELAY\_ONE\_MILLISECOND (DELAY\_ONE\_MICROSECOND \* 1000)  #define DELAY\_ONE\_SECOND (DELAY\_ONE\_MILLISECOND \* 1000)  typedef unsigned char P2C\_U8;  typedef unsigned short P2C\_U16;  typedef unsigned long P2C\_U32;  // 从sidt指令获得如下一个结构  #pragma pack(push, 1)  typedef struct \_P2C\_IDTR {  P2C\_U16 limit;  P2C\_U32 base;  }P2C\_IDTR, \*PP2C\_IDTR;  #pragma pack(pop)  void \*p2cGetIdt(){  P2C\_IDTR idtr;  \_asm sidt idtr  return (void\*)idtr.base;  }  // IDT表单元  // 跳转地址保存在offset\_low和offset\_high中  #pragma pack(push, 1)  typedef struct \_P2C\_IDT\_ENTRY {  P2C\_U16 offset\_low;  P2C\_U16 selector;  P2C\_U8 reserved;  P2C\_U8 type : 4;  P2C\_U8 always0 : 1;  P2C\_U8 dpl : 2;  P2C\_U8 present : 1;  P2C\_U16 offset\_high;  }P2C\_IDTENTRY, \*PP2C\_IDTENTRY;  #pragma pack(pop)  #define P2C\_MAKELONG(low, high) ( ((P2C\_U32)(low) & 0xffff) | (((P2C\_U32)(high) & 0xffff) << 16) )  #define P2C\_LOW16\_OF\_32(data) ( (P2C\_U16)((P2C\_U32)(data) & 0xffff) )  #define P2C\_HIGH16\_OF\_32(data) ( (P2C\_U16)((P2C\_U32)(data) >> 16) )  void\* g\_p2c\_old = NULL;  #define OBUFFER\_FULL 0x02  #define IBUFFER\_FULL 0x01  ULONG p2cWaitForKdRead(){  int i = 100;  P2C\_U8 mychar;  do  {  \_asm in al, 0x64  \_asm mov mychar, al  KeStallExecutionProcessor(50);  if (!(mychar & OBUFFER\_FULL))  {  break;  }  } while (i--);  if (i) return TRUE;  return FALSE;  }  ULONG p2cWaitForKdWrite(){  int i = 100;  P2C\_U8 mychar;  do {  \_asm in al, 0x64  \_asm mov mychar, al  KeStallExecutionProcessor(50);  if (!(mychar & IBUFFER\_FULL)) {  break;  }  } while (i--);  if (i) return TRUE;  return FALSE;  }  void p2cUserFilter(){  DbgPrint("Enter p2cUserFilter\r\n");  static P2C\_U8 sch\_pre = 0;  P2C\_U8 sch;  p2cWaitForKdRead();  \_asm in al, 0x60  \_asm mov sch, al  // 端口的数据一旦被读出来，就没有了，这个时候，需要强制回写  if (sch\_pre != sch) {  DbgPrint("p2c: scan code = %2x\r\n", sch);  sch\_pre = sch;  \_asm mov al, 0xd2;  \_asm out 0x64, al  p2cWaitForKdWrite();  \_asm mov al, sch  \_asm out 0x60, al  }  }  \_\_declspec(naked) p2cInterruptProc(){  \_\_asm {  pushad // 保存所有的通用寄存器  pushfd // 保存标志寄存器  call p2cUserFilter // 调用我们自己的一个函数，这个函数将实现我们自己的一个功能  popfd  popad  jmp g\_p2c\_old // 跳转到原来的中断服务程序  }  }  // 修改IOAPIC重定向表  P2C\_U8 p2cSearchOrSetIrql(P2C\_U8 new\_ch){  DbgPrint("Enter p2cSearchOrSetIrql");  // 定义一个物理地址0xfec00000，正是IOAPIC寄存器组在Windows上的开始地址  PHYSICAL\_ADDRESS phys;  RtlZeroMemory(&phys, sizeof(PHYSICAL\_ADDRESS));  phys.u.LowPart = 0xfec00000;  // 物理地址是不能直接读写，需要映射为虚拟地址  PVOID paddr = MmMapIoSpace(phys, 0x14, MmNonCached);  if (!MmIsAddressValid(paddr)) {  return 0;  }  // 选择寄存器，是寄存器组偏移为0的寄存器。选择寄存器虽然是32位的，但是只用了8位偏移  P2C\_U8 \*io\_reg\_sel = (P2C\_U8 \*)paddr;  // 窗口寄存器偏移为0x10  P2C\_U32 \*io\_win = (P2C\_U32\*)((P2C\_U8 \*)paddr + 0x10);  // 选择0x12项，对应的IRQ1，键盘中断  \*io\_reg\_sel = 0x12;  P2C\_U32 ch = \*io\_win;  P2C\_U32 ch1;  // 设置新值，返回旧值  if (new\_ch != 0) {  ch1 = \*io\_win;  ch1 &= 0xffffff00;  ch1 |= (P2C\_U32)new\_ch;  \*io\_win = ch1;  DbgPrint("p2cSearchOrSetIrql: set %2x to irql.\r\n", new\_ch);  }  ch &= 0xff;  MmUnmapIoSpace(paddr, 0x14);  DbgPrint("p2cSearchOrSetIrql: the old vec of irq1 is %2x to irql.\r\n", (P2C\_U8)ch);  return (P2C\_U8)ch;  }  P2C\_U8 p2cGetIdleIdtVec(){  PP2C\_IDTENTRY idt\_addr = (PP2C\_IDTENTRY)p2cGetIdt();  for (P2C\_U8 i = 0x20; i < 0x2a; i++) {  if (idt\_addr[i].type == 0) {  return i;  }  }  return 0;  }  P2C\_U8 p2cCopyANewIdt93(P2C\_U8 id, void\* interrupt\_proc){  PP2C\_IDTENTRY idt\_addr = (PP2C\_IDTENTRY)p2cGetIdt();  idt\_addr[id] = idt\_addr[0x93];  idt\_addr->offset\_low = P2C\_LOW16\_OF\_32(interrupt\_proc);  idt\_addr->offset\_high = P2C\_HIGH16\_OF\_32(interrupt\_proc);  return id;  }  void p2cResetIoApic(BOOLEAN set\_or\_recovery){  static P2C\_U8 idle\_id = 0;  P2C\_U8 old\_id = 0;  PP2C\_IDTENTRY idt\_addr = (PP2C\_IDTENTRY)p2cGetIdt();  DbgPrint("p2c: the pre address = %x.\r\n", (void\*)P2C\_MAKELONG(idt\_addr->offset\_low, idt\_addr->offset\_high));  if (set\_or\_recovery) {  DbgPrint("p2c: try to set IoApic.\r\n");  idt\_addr += 0x93;  g\_p2c\_old = (void\*)P2C\_MAKELONG(idt\_addr->offset\_low, idt\_addr->offset\_high);  idle\_id = p2cGetIdleIdtVec();  if (idle\_id != 0) {  p2cCopyANewIdt93(idle\_id, p2cInterruptProc);  // 再重新定位到这个中断  old\_id = p2cSearchOrSetIrql(idle\_id);  // 默认定位到0x93  ASSERT(old\_id == 0x93);  }  }  else {  DbgPrint("p2c: try to recovery IoApic.\r\n");  old\_id = p2cSearchOrSetIrql(0x93);  ASSERT(old\_id == idle\_id);  idt\_addr[old\_id].type = 0;  }  DbgPrint("p2c: the current address = %x.\r\n", (void\*)P2C\_MAKELONG(idt\_addr->offset\_low, idt\_addr->offset\_high));  }  VOID c2pUnload(PDRIVER\_OBJECT driver){  UNREFERENCED\_PARAMETER(driver);  p2cResetIoApic(FALSE);  LARGE\_INTEGER interval;  interval.QuadPart = 5 \* 1000 \* DELAY\_ONE\_MILLISECOND;  KeDelayExecutionThread(KernelMode, FALSE, &interval);  DbgPrint("Leave c2pUnload.\n");  }  NTSTATUS DriverEntry(PDRIVER\_OBJECT driver, PUNICODE\_STRING reg\_path){  UNREFERENCED\_PARAMETER(reg\_path);  DbgPrint("Enter kbddriver hook class.\n");  driver->DriverUnload = c2pUnload;  p2cResetIoApic(TRUE);  return STATUS\_SUCCESS;  } |

**注意：经实测，在虚拟机中又行不通，怀疑跟虚拟硬件实现有关**

# 6 NT式驱动和WDM式驱动详解

## 6.1NT式驱动原理

前面介绍了两种驱动示例，但是没有深入讲解对应的原理，只说了特定硬件的大概。这里开始，深入驱动程序内部原理。

### 6.1.1完整代码示例

|  |
| --- |
| Driver.h |
| #pragma once  // 因为这里采用C++编写，而ntddk的实现，是C编写。  // 这里为了防止ntddk的函数符号表导入错误  #ifdef \_\_cplusplus  extern "C"  {  #endif  #include <ntddk.h>  #ifdef \_\_cplusplus  }  #endif  #define PAGECODE code\_seq("PAGE") // 定义分页标记  #define LOCKEDCODE code\_seq() // 定义非分页标记  #define INITCODE code\_seq("INIT") // 定义初始化内存块。  // 函数只是在加载的时候载入内存，当驱动成功加载后，就可以从内存中卸载掉  #define PAGEDATA data\_seq("PAGE")  #define LOCKEDDATA data\_seq()  #define INITDATA data\_seq("INIT")  #define arraysize(p) (sizeof(p) / sizeof((p)[0]))  // 自定义设备扩展  typedef struct \_DEVICE\_EXTENSION {  PDEVICE\_OBJECT pDevice;  UNICODE\_STRING ustrDeviceName; // 设备名，可以在设备管理器中显示的名字  UNICODE\_STRING ustrSymLinkName; // 符号链接名，用于ring3层访问设备对象的  } DEVICE\_EXTENSION, \*PDEVICE\_EXTENSION;  // 函数声明  NTSTATUS CreateDevice(IN PDRIVER\_OBJECT pDriverObject);  VOID HelloDDKUnload(IN PDRIVER\_OBJECT pDriverObject);  NTSTATUS HelloDDKDispatchRoutine(IN PDEVICE\_OBJECT pDevObj, IN PIRP pIrp); |
| Driver.cpp |
| #include "Driver.h"  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：DriverEntry  \* 功能描述：初始化驱动程序，定位和申请硬件资源，创建内核对象  \* 参数列表：  pDriverObject：从I/O管理器中传进来的驱动对象，由执行体组件-对象管理器创建和管理  pRegistryPath：驱动程序在注册表中的路径，由执行体组件-配置管理器创建和管理  \* 返回值：返回初始化驱动状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma INITCODE // 指明DriverEntry是加载到INIT内存区域中，驱动成功卸载后，可以退出内存  extern "C" NTSTATUS DriverEntry( // 特定的回调函数，都需要编译成C的函数命名  IN PDRIVER\_OBJECT pDriverObject,  IN PUNICODE\_STRING pRegistryPath)  {  UNREFERENCED\_PARAMETER(pRegistryPath);  KdPrint(("Enter DriverEntry\n")); // 使用KdPrint宏，可以在调试版本(Checked/Debug)打日志，发行版本(Free/Release)不打日志  // 注册其他驱动调用函数入口  pDriverObject->DriverUnload = HelloDDKUnload;  pDriverObject->MajorFunction[IRP\_MJ\_CREATE] = HelloDDKDispatchRoutine;  pDriverObject->MajorFunction[IRP\_MJ\_CLOSE] = HelloDDKDispatchRoutine;  pDriverObject->MajorFunction[IRP\_MJ\_WRITE] = HelloDDKDispatchRoutine;  pDriverObject->MajorFunction[IRP\_MJ\_READ] = HelloDDKDispatchRoutine;  // 创建驱动设备对象  NTSTATUS status = CreateDevice(pDriverObject);  KdPrint(("DriverEntry end\n"));  return status;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：CreateDevice  \* 功能描述：初始化设备对象  \* 参数列表：  pDriverObject：从I/O管理器中传进来的驱动对象，由执行体组件-对象管理器创建和管理  \* 返回值：返回初始化状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma INITCODE  NTSTATUS CreateDevice(  IN PDRIVER\_OBJECT pDriverObject)  {  // 创建设备名称  UNICODE\_STRING devName;  RtlInitUnicodeString(&devName, L"\\Device\\KeenDDKDevice");  // 创建设备。一个驱动，有一个默认的设备对象，大概这个就是一个标准吧  PDEVICE\_OBJECT pDevObj = NULL;  NTSTATUS status = IoCreateDevice(pDriverObject,  sizeof(DEVICE\_EXTENSION),  &devName,  FILE\_DEVICE\_UNKNOWN, // 虚拟设备，必须设置为FILE\_DEVICE\_UNKNOWN  0,  TRUE, // 指定此设备为独占设备，不能被其他应用程序所使用  &pDevObj); // 对象管理器来管理的，这里同时也会添加到驱动对象pDriverObject相应的参数中  if (!NT\_SUCCESS(status)) {  return status;  }  pDevObj->Flags |= DO\_BUFFERED\_IO; // 内存操作方式。DO\_BUFFERED\_IO表示读写操作使用系统复制缓冲区访问用户模式数据；  // 如果是DO\_DIRECT\_IO则表示读写操作使用内存描述符表访问用户模式数据  PDEVICE\_EXTENSION pDevExt = (PDEVICE\_EXTENSION)pDevObj->DeviceExtension;  pDevExt->pDevice = pDevObj; // 使用设备扩展的目的是，避免使用全局变量（有各种同步问题）  pDevExt->ustrDeviceName = devName;  // 创建符号链接名  UNICODE\_STRING symLinkName;  RtlInitUnicodeString(&symLinkName, L"\\??\\SymKeenHelloDDK");  pDevExt->ustrSymLinkName = symLinkName; // 使用符号链接的目的是，方便用户模式应用程序可以访问设备。因为设备名只能用于内核访问  status = IoCreateSymbolicLink(&symLinkName, &devName);  if (!NT\_SUCCESS(status)) {  IoDeleteDevice(pDevObj);  return status;  }  return STATUS\_SUCCESS;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：HelloDDKUnload  \* 功能描述：负责驱动程序的卸载操作。会卸载整个设备链，删除设备对象和符号链接  \* 参数列表：  pDriverObject：从I/O管理器中传进来的驱动对象，由执行体组件-对象管理器创建和管理  \* 返回值：返回状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma PAGECODE  VOID HelloDDKUnload(IN PDRIVER\_OBJECT pDriverObject)  {  KdPrint(("Enter DriverUnload\n"));  PDEVICE\_OBJECT pNextObj = pDriverObject->DeviceObject;  while (pNextObj != NULL) {  PDEVICE\_EXTENSION pDevExt = (PDEVICE\_EXTENSION)pNextObj->DeviceExtension;  // 删除符号链接  UNICODE\_STRING linkName = pDevExt->ustrSymLinkName;  IoDeleteSymbolicLink(&linkName);  pNextObj = pNextObj->NextDevice;  IoDeleteDevice(pDevExt->pDevice);  }  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：HelloDDKDispatchRoutine  \* 功能描述：对IRP进行处理  \* 参数列表：  pDevObj：功能设备对象  pIrp:从I/0请求包  \* 返回值：返回状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma PAGECODE  NTSTATUS HelloDDKDispatchRoutine(IN PDEVICE\_OBJECT pDevObj,  IN PIRP pIrp)  {  UNREFERENCED\_PARAMETER(pDevObj);  KdPrint(("Enter HelloDDKDispatchRoutine\n"));  NTSTATUS status = STATUS\_SUCCESS;  // 完成IRP  pIrp->IoStatus.Status = status;  pIrp->IoStatus.Information = 0;  IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);  KdPrint(("Leave HelloDDKDispatchRoutine\n"));  return status;  } |

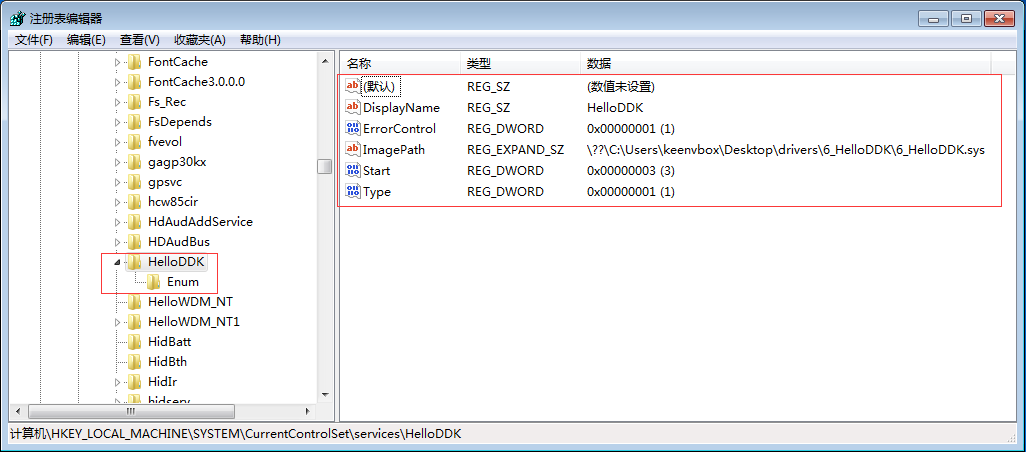
说明：

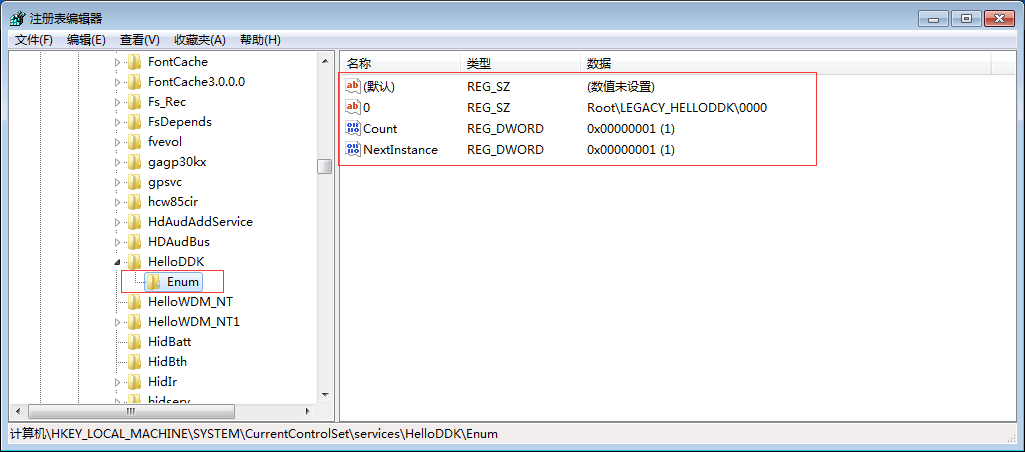
* 创建设备对象中，设备名称一般为：\Device\[设备名]。例如C盘，命名为：\Device\HarddiskVolume1
* 如果创建设备中不给设备名称，则I/O管理器会自动分配一个数字作为设备名，例如：\Device\00000001
* 设备名只能由内核层访问，用户层想访问，有两种方法：通过符号链接；通过设备接口。NT式驱动很少通过设备接口直接访问，WDM式驱动才使用此方式

### 6.1.2驱动安装

#### 6.1.2.1手动安装

NT式驱动可以直接导入注册表，可以实现，注册表类似如下：





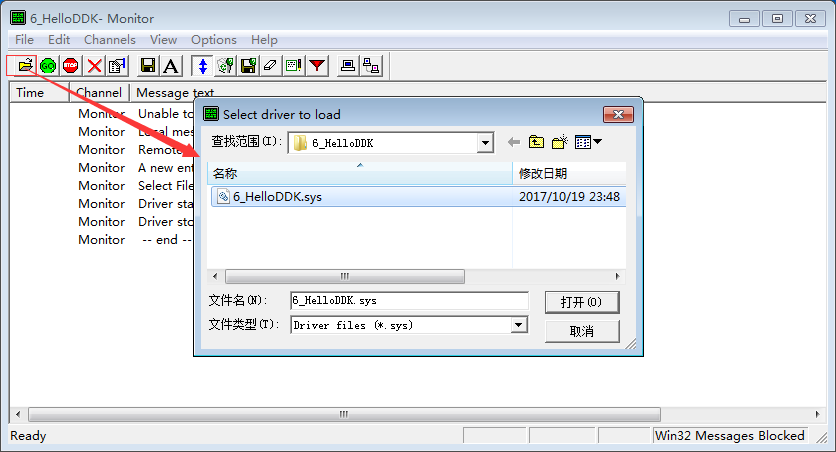
#### 6.1.2.2使用SRVINSTW安装

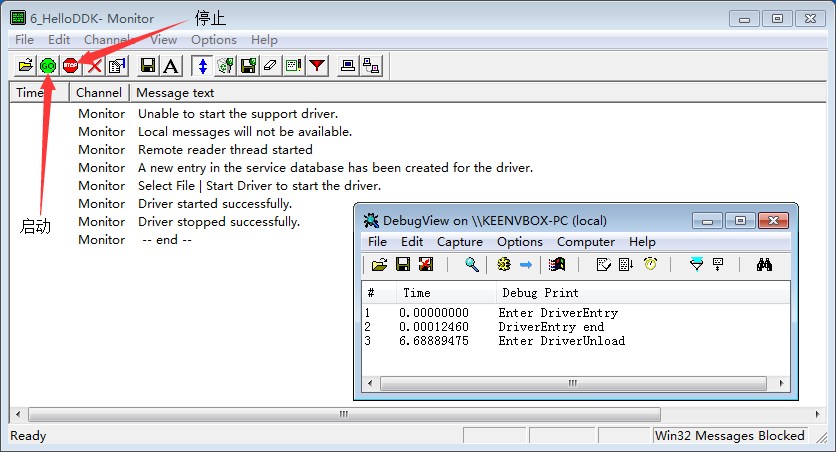
前面已介绍，参考[2.3.2使用工具srvinstw.exe安装驱动](#_2.3.2使用工具srvinstw.exe安装驱动)

#### 6.1.2.3使用DriverMonitor安装

这个是一个新工具，它比SRVINSTW要方便简单多了，后续NT式驱动，将使用它作为默认安装工具。

使用截图，如下：



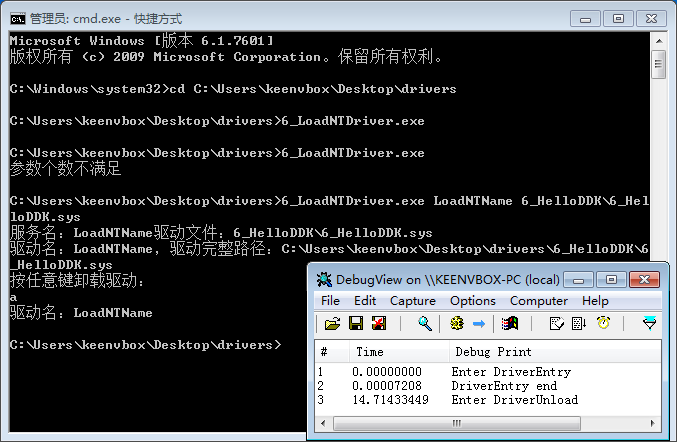


#### 6.1.2.4使用SCM程序安装

**（1）完整代码**

|  |
| --- |
| // 6\_LoadNTDriver.cpp : Defines the entry point for the console application.  //  #include "stdafx.h"  #include <iostream>  #include <atlpath.h>  // 函数声明  BOOL LoadNTDriver(LPCSTR lpstrDriverName, LPCSTR lpstrDriverPath);  BOOL UnloadNTDriver(LPCSTR lpstrDriverName);  // 命令行格式：  // LoadNTDriver.exe [drivername] [driverpath]  int main(int argc, char\* argv[])  {  if (argc != 3) {  std::cout << "参数个数不满足" << std::endl;  return -1;  }  LPCSTR lpstrDriverName = argv[1];  LPCSTR lpstrDriverPath = argv[2];  if (!lpstrDriverName || !lpstrDriverPath) {  std::cout << "参数错误" << std::endl;  return -1;  }  std::cout << "服务名：" << lpstrDriverName << "驱动文件：" << lpstrDriverPath << std::endl;  if (!ATLPath::FileExists(lpstrDriverPath) || ATLPath::IsDirectory(lpstrDriverPath)) {  std::cout << "驱动文件不存在" << std::endl;  return -1;  }  // 加载驱动  BOOL bRet = LoadNTDriver(lpstrDriverName, lpstrDriverPath);  if (!bRet) {  std::cout << "驱动加载失败" << std::endl;  return -1;  }  std::cout << "按任意键卸载驱动：";  char ch;  std::cin >> ch;  // 卸载驱动  bRet = UnloadNTDriver(lpstrDriverName);  if (!bRet) {  std::cout << "驱动卸载失败" << std::endl;  return -1;  }  return 0;  }  BOOL LoadNTDriver(LPCSTR lpstrDriverName, LPCSTR lpstrDriverPath)  {  // 得到完整的驱动路径  char szDriverImagePath[MAX\_PATH] = { 0 };  GetFullPathNameA(lpstrDriverPath, MAX\_PATH, szDriverImagePath, NULL);  std::cout << "驱动名：" << lpstrDriverName << "，驱动完整路径：" << szDriverImagePath << std::endl;  BOOL bRet = FALSE;  SC\_HANDLE hServiceMgr = NULL;  SC\_HANDLE hServiceDDK = NULL;  do  {  // 打开服务控制管理器  hServiceMgr = OpenSCManager(NULL, NULL, SC\_MANAGER\_ALL\_ACCESS);  if (!hServiceMgr) {  std::cout << "打开服务控制管理器失败" << std::endl;  break;  }  hServiceDDK = CreateServiceA(hServiceMgr, // 服务管理器  lpstrDriverName, // 服务名  lpstrDriverName, // 显示名称  SERVICE\_ALL\_ACCESS, // 加载驱动程序的访问权限  SERVICE\_KERNEL\_DRIVER, // 表示加载的服务是驱动程序  SERVICE\_DEMAND\_START, // 手动方式启动驱动  SERVICE\_ERROR\_IGNORE,  szDriverImagePath,  NULL,NULL,NULL,NULL,NULL);  if (!hServiceDDK) {  DWORD dwErrorCode = GetLastError();  if (dwErrorCode != ERROR\_IO\_PENDING && dwErrorCode != ERROR\_SERVICE\_EXISTS) {  std::cout << "创建服务失败，错误码：" << dwErrorCode << std::endl;  break;  }  std::cout << "服务已存在" << std::endl;  hServiceDDK = OpenServiceA(hServiceMgr, lpstrDriverName, SERVICE\_ALL\_ACCESS);  if (!hServiceDDK) {  std::cout << "服务存在，但打开服务失败，错误码：" << GetLastError() << std::endl;  break;  }  }  // 开启此服务  bRet = StartServiceA(hServiceDDK, NULL, NULL);  if (!bRet) {  DWORD dwErrorCode = GetLastError();  if (dwErrorCode != ERROR\_IO\_PENDING && dwErrorCode != ERROR\_SERVICE\_ALREADY\_RUNNING) {  std::cout << "启动服务失败，错误码：" << dwErrorCode << std::endl;  break;  }    if (dwErrorCode != ERROR\_IO\_PENDING){  std::cout << "设备被挂起" << std::endl;  break;  }  std::cout << "服务已启动" << std::endl;  }  bRet = TRUE;  } while (FALSE);  if (hServiceDDK){  CloseServiceHandle(hServiceDDK);  hServiceDDK = NULL;  }  if (hServiceMgr){  CloseServiceHandle(hServiceMgr);  hServiceMgr = NULL;  }  return bRet;  }  BOOL UnloadNTDriver(LPCSTR lpstrDriverName)  {  std::cout << "驱动名：" << lpstrDriverName << std::endl;  BOOL bRet = FALSE;  SC\_HANDLE hServiceMgr = NULL;  SC\_HANDLE hServiceDDK = NULL;  do  {  // 打开服务控制管理器  hServiceMgr = OpenSCManager(NULL, NULL, SC\_MANAGER\_ALL\_ACCESS);  if (!hServiceMgr){  std::cout << "打开服务控制管理器失败" << std::endl;  break;  }  hServiceDDK = OpenServiceA(hServiceMgr, lpstrDriverName, SERVICE\_ALL\_ACCESS);  if (!hServiceDDK){  std::cout << "打开服务失败，错误码：" << GetLastError() << std::endl;  break;  }  // 停止驱动程序  SERVICE\_STATUS srvStatus;  bRet = ControlService(hServiceDDK, SERVICE\_CONTROL\_STOP, &srvStatus);  if (!bRet) {  std::cout << "停止服务失败，错误码：" << GetLastError() << std::endl;  }  // 卸载驱动程序  bRet = DeleteService(hServiceDDK);  if (!bRet) {  std::cout << "卸载驱动失败，错误码：" << GetLastError() << std::endl;  break;  }  bRet = TRUE;  } while (FALSE);  if (hServiceDDK){  CloseServiceHandle(hServiceDDK);  hServiceDDK = NULL;  }  if (hServiceMgr){  CloseServiceHandle(hServiceMgr);  hServiceMgr = NULL;  }  return bRet;  } |

**（2）实践效果**

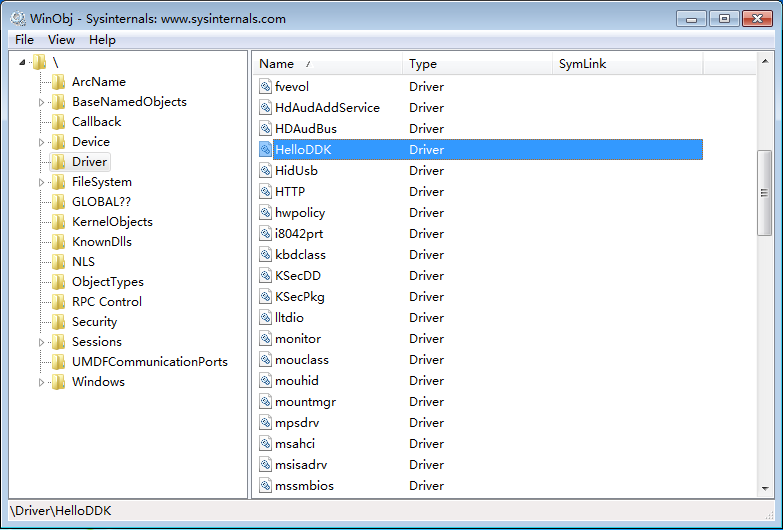


### 6.1.3设备驱动查看工具

#### 6.1.3.1WinObj工具查看

WinObj是查看当前驱动对象的存在与否，必须等待驱动加载到内存才能查看到（若驱动是手动加载方式安装的，必须Start该驱动才能看到）

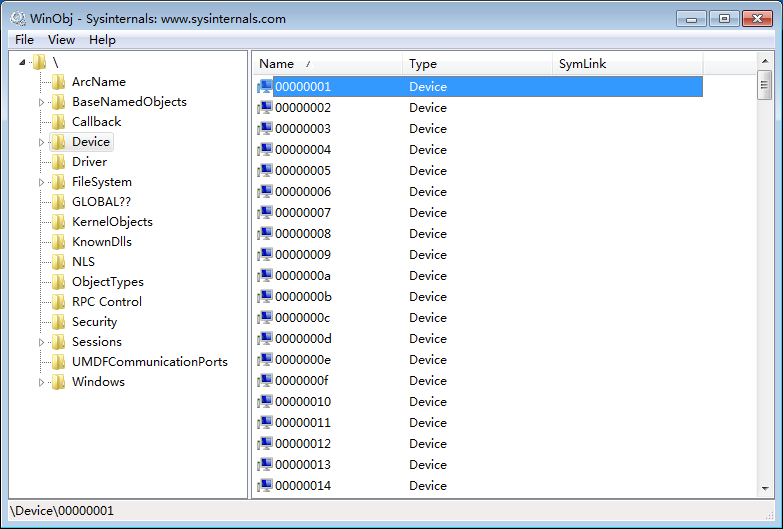
（1）驱动对象



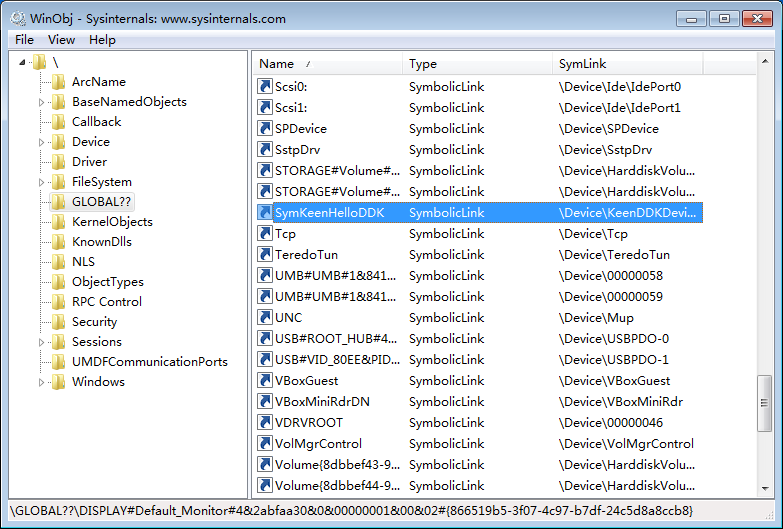
（2）设备对象



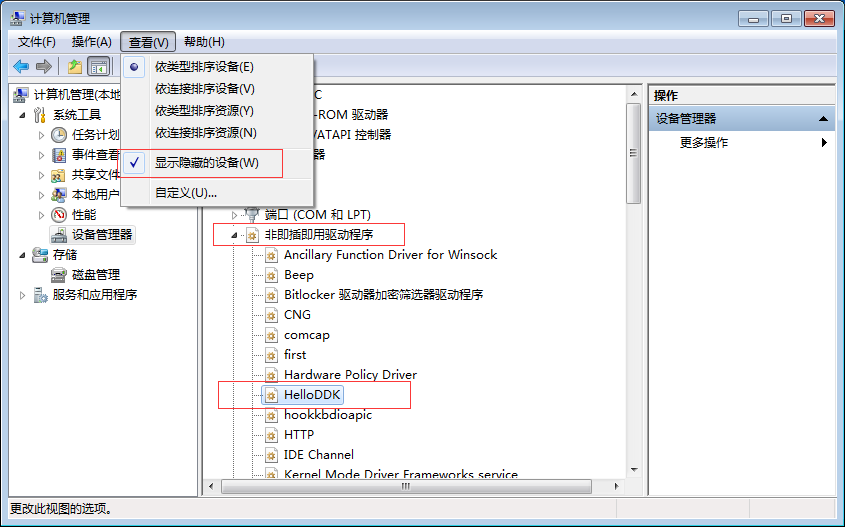
这里特别说明下，如果设备对象在创建的时候，**未命名**，将会以00000001这种数字名称递增展示。如下：



（3）符号链接对象



#### 6.1.3.2设备管理器查看



## 6.2WDM式驱动原理

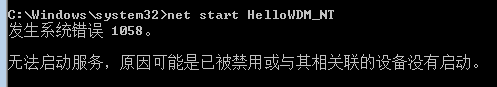
### 6.2.1完整代码示例

|  |
| --- |
| Driver.h |
| #pragma once  // 因为这里采用C++编写，而ntddk的实现，是C编写。  // 这里为了防止ntddk的函数符号表导入错误  #ifdef \_\_cplusplus  extern "C"  {  #endif  #include <wdm.h>  #ifdef \_\_cplusplus  }  #endif  #define PAGEDCODE code\_seq("PAGE") // 定义分页标记  #define LOCKEDCODE code\_seq() // 定义非分页标记  #define INITCODE code\_seq("INIT") // 定义初始化内存块。  // 函数只是在加载的时候载入内存，当驱动成功加载后，就可以从内存中卸载掉  #define PAGEDATA data\_seq("PAGE")  #define LOCKEDDATA data\_seq()  #define INITDATA data\_seq("INIT")  #define arraysize(p) (sizeof(p) / sizeof((p)[0]))  // 自定义设备扩展  typedef struct \_DEVICE\_EXTENSION {  PDEVICE\_OBJECT fdo;  PDEVICE\_OBJECT NextStackDevice;  UNICODE\_STRING ustrDeviceName; // 设备名，可以在设备管理器中显示的名字  UNICODE\_STRING ustrSymLinkName; // 符号链接名，用于ring3层访问设备对象的  } DEVICE\_EXTENSION, \*PDEVICE\_EXTENSION;  // 函数声明  NTSTATUS HelloWDMAddDevice(IN PDRIVER\_OBJECT pDriverObject,  IN PDEVICE\_OBJECT pPhysicalDeviceObject);  NTSTATUS HelloWDMPnp(IN PDEVICE\_OBJECT fdo, IN PIRP pIrp);  NTSTATUS DefaultPnpHandler(IN PDEVICE\_EXTENSION pdx, IN PIRP pIrp);  NTSTATUS HandleRemoveDevice(IN PDEVICE\_EXTENSION pdx, IN PIRP pIrp);  VOID HelloWDMUnload(IN PDRIVER\_OBJECT pDriverObject);  NTSTATUS HelloWDMDispatchRoutine(IN PDEVICE\_OBJECT pDevObj, IN PIRP pIrp); |
| Driver.cpp |
| #include "Driver.h"  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：DriverEntry  \* 功能描述：初始化驱动程序，定位和申请硬件资源，创建内核对象  \* 参数列表：  pDriverObject：从I/O管理器中传进来的驱动对象，由执行体组件-对象管理器创建和管理  pRegistryPath：驱动程序在注册表中的路径，由执行体组件-配置管理器创建和管理  \* 返回值：返回初始化驱动状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma INITCODE // 指明DriverEntry是加载到INIT内存区域中，驱动成功卸载后，可以退出内存  extern "C" NTSTATUS DriverEntry( // 特定的回调函数，都需要编译成C的函数命名  IN PDRIVER\_OBJECT pDriverObject,  IN PUNICODE\_STRING pRegistryPath)  {  UNREFERENCED\_PARAMETER(pRegistryPath);  KdPrint(("Enter DriverEntry\n")); // 使用KdPrint宏，可以在调试版本(Checked/Debug)打日志，发行版本(Free/Release)不打日志  // 注册其他驱动调用函数入口  pDriverObject->DriverExtension->AddDevice = HelloWDMAddDevice; // 即插即用的支持  pDriverObject->MajorFunction[IRP\_MJ\_PNP] = HelloWDMPnp;  pDriverObject->MajorFunction[IRP\_MJ\_CREATE] = HelloWDMDispatchRoutine;  pDriverObject->MajorFunction[IRP\_MJ\_CLOSE] = HelloWDMDispatchRoutine;  pDriverObject->MajorFunction[IRP\_MJ\_WRITE] = HelloWDMDispatchRoutine;  pDriverObject->MajorFunction[IRP\_MJ\_READ] = HelloWDMDispatchRoutine;  pDriverObject->DriverUnload = HelloWDMUnload;  KdPrint(("DriverEntry end\n"));  return STATUS\_SUCCESS;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：HelloWDMAddDevice  \* 功能描述：添加新设备。由PNP管理器负责调用  \* 参数列表：  pDriverObject：从I/O管理器中传进来的驱动对象，由执行体组件-对象管理器创建和管理  \* 返回值：返回初始化状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma PAGEDCODE // 分页内存  NTSTATUS HelloWDMAddDevice(IN PDRIVER\_OBJECT pDriverObject,  IN PDEVICE\_OBJECT pPhysicalDeviceObject)  {  PAGED\_CODE(); // 中断请求级别的校验，超过APC\_LEVEL时，会产生一个断言  KdPrint(("Enter HelloWDMAddDevice\n"));  // 创建设备名称  UNICODE\_STRING devName;  RtlInitUnicodeString(&devName, L"\\Device\\KeenWDMDevice");  // 创建设备。一个驱动，有一个默认的设备对象，大概这个就是一个标准吧  PDEVICE\_OBJECT fdo = NULL;  NTSTATUS status = IoCreateDevice(pDriverObject,  sizeof(DEVICE\_EXTENSION),  &devName,  FILE\_DEVICE\_UNKNOWN, // 虚拟设备，必须设置为FILE\_DEVICE\_UNKNOWN  0,  TRUE, // 指定此设备为独占设备，不能被其他应用程序所使用  &fdo); // 对象管理器来管理的，这里同时也会添加到驱动对象pDriverObject相应的参数中  if (!NT\_SUCCESS(status)) {  return status;  }  PDEVICE\_EXTENSION pDevExt = (PDEVICE\_EXTENSION)fdo->DeviceExtension;  pDevExt->fdo = fdo; // 使用设备扩展的目的是，避免使用全局变量（有各种同步问题）  pDevExt->NextStackDevice = IoAttachDeviceToDeviceStack(fdo, pPhysicalDeviceObject); // 将此设备挂接到设备栈（这里是物理设备），返回值是下一层堆栈的设备对象  pDevExt->ustrDeviceName = devName;  // 创建符号链接名  UNICODE\_STRING symLinkName;  RtlInitUnicodeString(&symLinkName, L"\\DosDevices\\SymKeenHelloWDM"); // 这里既可以使用\\DosDevices\\，也可以使用\\??\\  pDevExt->ustrSymLinkName = symLinkName; // 使用符号链接的目的是，方便用户模式应用程序可以访问设备。因为设备名只能用于内核访问  status = IoCreateSymbolicLink(&symLinkName, &devName);  if (!NT\_SUCCESS(status)) {  IoDeleteDevice(fdo);  return status;  }  fdo->Flags |= DO\_BUFFERED\_IO | DO\_POWER\_PAGABLE; // 内存操作方式。DO\_BUFFERED\_IO表示读写操作使用系统复制缓冲区访问用户模式数据；  // 如果是DO\_DIRECT\_IO则表示读写操作使用内存描述符表访问用户模式数据  fdo->Flags &= ~DO\_DEVICE\_INITIALIZING;  KdPrint(("Leave HelloWDMAddDevice\n"));  return STATUS\_SUCCESS;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：HelloWDMPnp  \* 功能描述：对即插即用IRP进行处理  \* 参数列表：  fdo：功能设备对象  pIrp：从I/O请求包  \* 返回值：返回状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma PAGEDCODE  NTSTATUS HelloWDMPnp(IN PDEVICE\_OBJECT fdo, IN PIRP pIrp)  {  PAGED\_CODE(); // 确保该函数运行在低于APC\_LEVEL的中断优先级  KdPrint(("Enter HelloWDMPnp\n"));  NTSTATUS status = STATUS\_SUCCESS;  PDEVICE\_EXTENSION pdx = (PDEVICE\_EXTENSION)fdo->DeviceExtension;  PIO\_STACK\_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);  // 不同IRP的处理例程  static NTSTATUS(\*fcntab[])(PDEVICE\_EXTENSION pdx, PIRP pIrp) = {  DefaultPnpHandler, // IRP\_MN\_START\_DEVICE  DefaultPnpHandler, // IRP\_MN\_QUERY\_REMOVE\_DEVICE  HandleRemoveDevice, // IRP\_MN\_REMOVE\_DEVICE  DefaultPnpHandler, // IRP\_MN\_CANCEL\_REMOVE\_DEVICE  DefaultPnpHandler, // IRP\_MN\_STOP\_DEVICE  DefaultPnpHandler, // IRP\_MN\_QUERY\_STOP\_DEVICE  DefaultPnpHandler, // IRP\_MN\_CANCEL\_STOP\_DEVICE  DefaultPnpHandler, // IRP\_MN\_QUERY\_DEVICE\_RELATIONS  DefaultPnpHandler, // IRP\_MN\_QUERY\_INTERFACE  DefaultPnpHandler, // IRP\_MN\_QUERY\_CAPABILITIES  DefaultPnpHandler, // IRP\_MN\_QUERY\_RESOURCES  DefaultPnpHandler, // IRP\_MN\_QUERY\_RESOURCE\_REQUIREMENTS  DefaultPnpHandler, // IRP\_MN\_QUERY\_DEVICE\_TEXT  DefaultPnpHandler, // IRP\_MN\_FILTER\_RESOURCE\_REQUIREMENTS  DefaultPnpHandler,  DefaultPnpHandler, // IRP\_MN\_READ\_CONFIG  DefaultPnpHandler, // IRP\_MN\_WRITE\_CONFIG  DefaultPnpHandler, // IRP\_MN\_EJECT  DefaultPnpHandler, // IRP\_MN\_SET\_LOCK  DefaultPnpHandler, // IRP\_MN\_QUERY\_ID  DefaultPnpHandler, // IRP\_MN\_QUERY\_PNP\_DEVICE\_STATE  DefaultPnpHandler, // IRP\_MN\_QUERY\_BUS\_INFORMATION  DefaultPnpHandler, // IRP\_MN\_DEVICE\_USAGE\_NOTIFICATION  DefaultPnpHandler, // IRP\_MN\_SURPRISE\_REMOVAL  };  ULONG fcn = stack->MinorFunction; // 当前IRP辅类型号  // 未知IRP的处理  if (fcn >= arraysize(fcntab)) {  status = DefaultPnpHandler(pdx, pIrp);  return status;  }  status = (\*fcntab[fcn])(pdx, pIrp);  KdPrint(("Leave HelloWDMPnp\n"));  return status;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：DefaultPnpHandler  \* 功能描述：对PNP IRP进行默认处理  \* 参数列表：  pdx：设备对象的扩展  pIrp：从I/O请求包  \* 返回值：返回状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma PAGEDCODE  NTSTATUS DefaultPnpHandler(IN PDEVICE\_EXTENSION pdx, IN PIRP pIrp)  {  PAGED\_CODE();  KdPrint(("Enter DefaultPnpHandler\n"));  IoSkipCurrentIrpStackLocation(pIrp); // 略过当前IRP栈，直接跳转到下一层处理  KdPrint(("Leave DefaultPnpHandler\n"));  return IoCallDriver(pdx->NextStackDevice, pIrp);  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：HandleRemoveDevice  \* 功能描述：对IRP\_MN\_REMOVE\_DEVICE进行处理，也就是设备卸载  \* 参数列表：  pdx：设备对象的扩展  pIrp：从I/O请求包  \* 返回值：返回状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma PAGEDCODE  NTSTATUS HandleRemoveDevice(IN PDEVICE\_EXTENSION pdx, IN PIRP pIrp)  {  PAGED\_CODE();  KdPrint(("Enter HandleRemoveDevice\n"));  pIrp->IoStatus.Status = STATUS\_SUCCESS; // 表明此IRP的状态为顺利完成  NTSTATUS status = DefaultPnpHandler(pdx, pIrp);  IoDeleteSymbolicLink(&pdx->ustrSymLinkName);  // 把fdo从设备栈中脱离  if (pdx->NextStackDevice) {  IoDetachDevice(pdx->NextStackDevice); // 卸载设备对象  }  // 删除fdo  IoDeleteDevice(pdx->fdo);  KdPrint(("Leave HandleRemoveDevice\n"));  return status;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：HelloWDMDispatchRoutine  \* 功能描述：对IRP进行处理  \* 参数列表：  fdo：功能设备对象  pIrp:从I/0请求包  \* 返回值：返回状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma PAGEDCODE  NTSTATUS HelloWDMDispatchRoutine(IN PDEVICE\_OBJECT fdo,  IN PIRP pIrp)  {  UNREFERENCED\_PARAMETER(fdo);  PAGED\_CODE();  KdPrint(("Enter HelloWDMDispatchRoutine\n"));  NTSTATUS status = STATUS\_SUCCESS;  // 完成IRP  pIrp->IoStatus.Status = status;  pIrp->IoStatus.Information = 0;  IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);  KdPrint(("Leave HelloWDMDispatchRoutine\n"));  return status;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：HelloWDMUnload  \* 功能描述：负责驱动程序的卸载操作。会卸载整个设备链，删除设备对象和符号链接  \* 参数列表：  pDriverObject：从I/O管理器中传进来的驱动对象，由执行体组件-对象管理器创建和管理  \* 返回值：返回状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma PAGEDCODE  VOID HelloWDMUnload(IN PDRIVER\_OBJECT pDriverObject)  {  UNREFERENCED\_PARAMETER(pDriverObject);  PAGED\_CODE();  KdPrint(("Enter HelloWDMUnload\n"));  KdPrint(("Leave HelloWDMUnload\n"));  } |

### 6.2.2驱动安装

#### 6.2.2.1尝试使用SRVINSTW/DriverMonitor安装 —— 失败

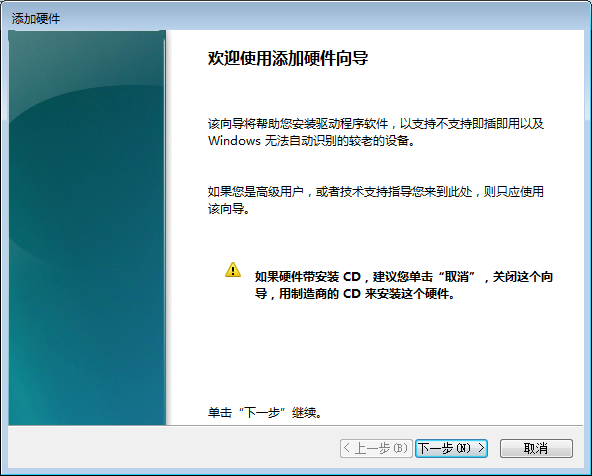
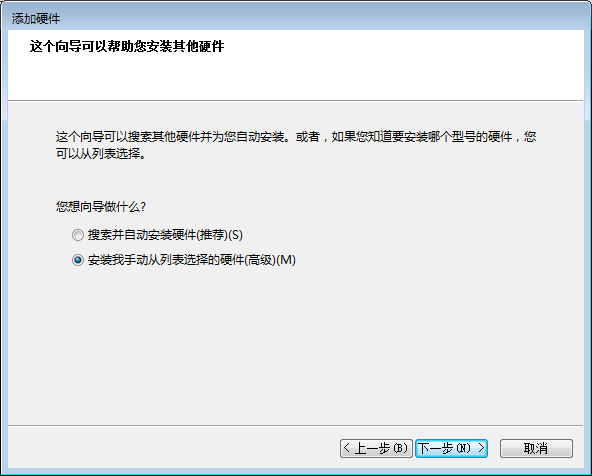
即插即用驱动设备安装，跟以前的NT式驱动设备安装还不一样，我先尝试使用NT式驱动设备安装看看，结果如下：

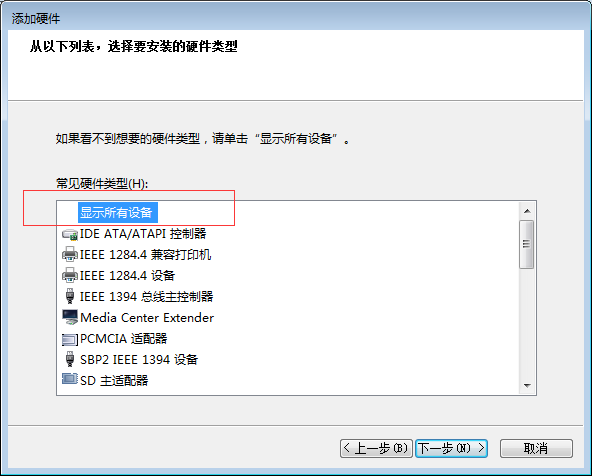
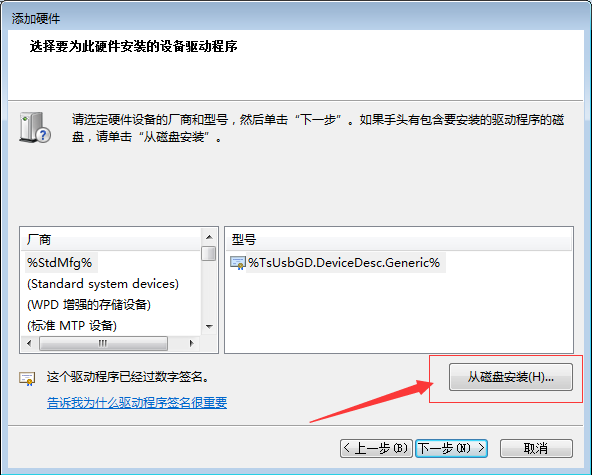


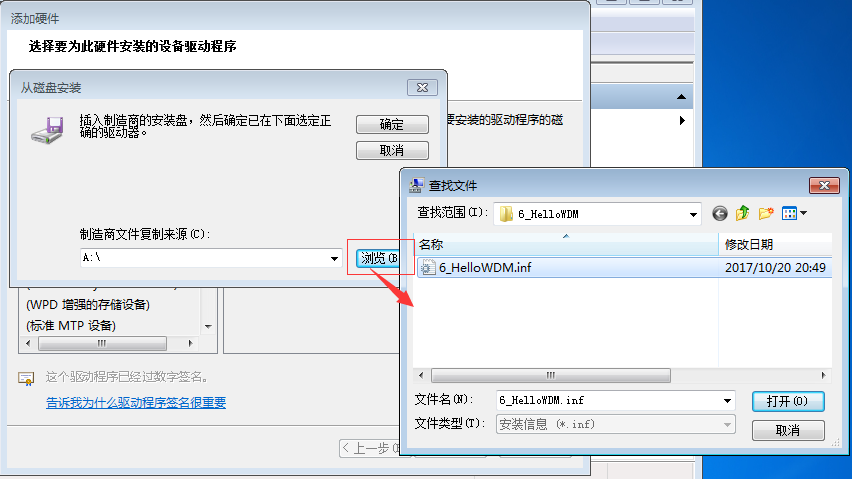
说明此方式，并不适合WDM式驱动。

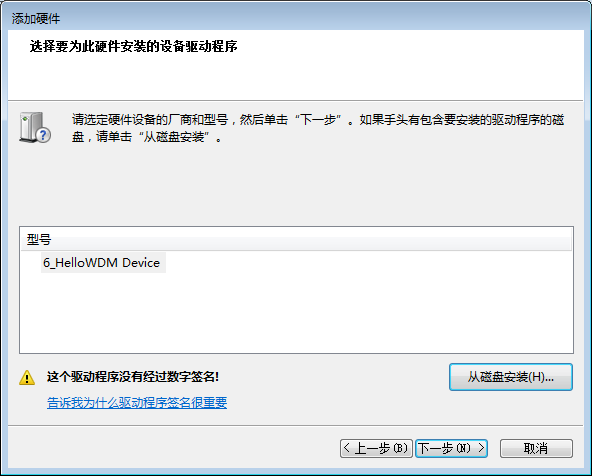
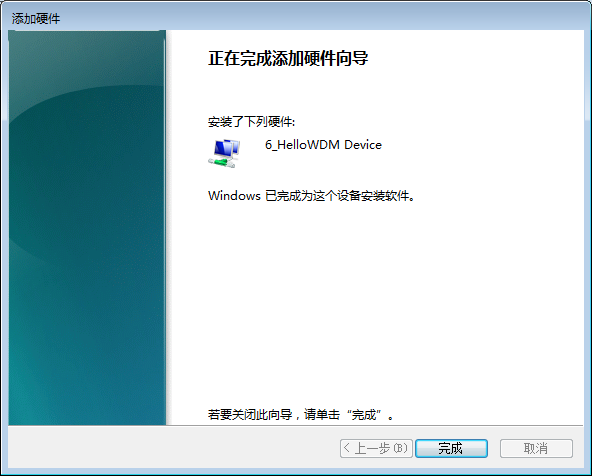
#### 6.2.2.2使用INF安装方式



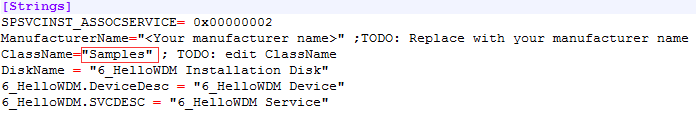
 



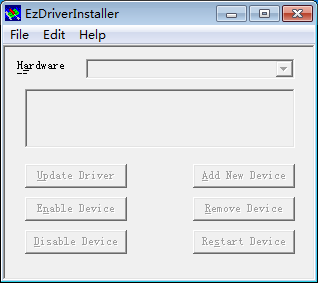
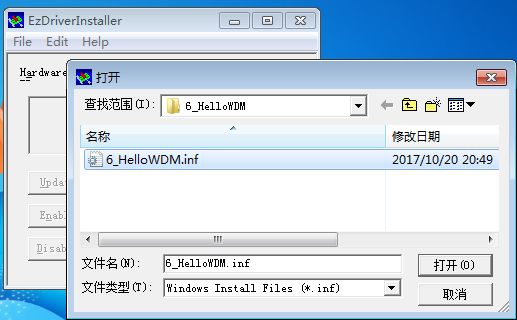
 

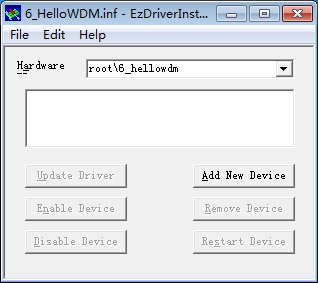
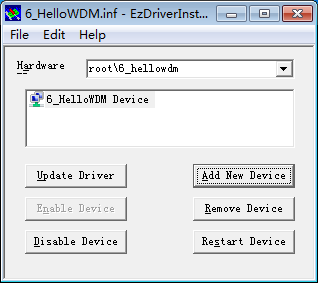


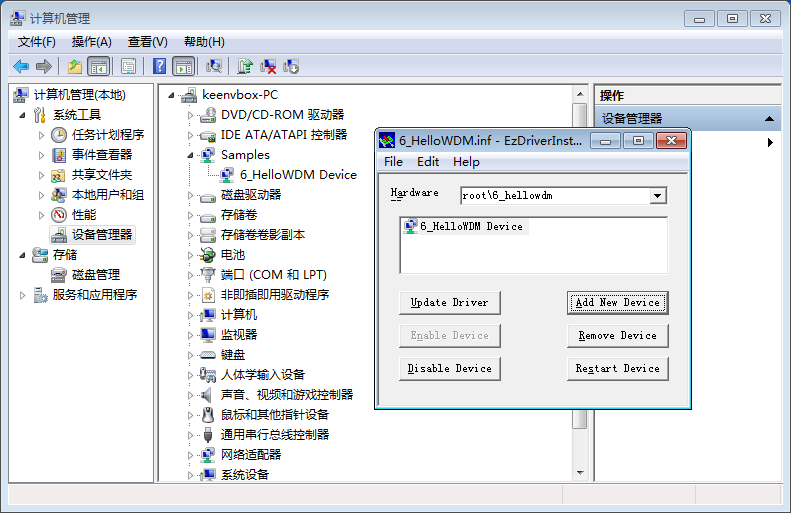
在Samples类型下，之所以属于这个类别，是因为我们INF文件中所写：



#### 6.2.2.3使用EzDriverInstaller安装



## 6.3驱动结构

不管是NT式驱动，还是WDM式驱动，都可以分层。不过WDM本身就是基于分层结构的。这里以WDM为例说明。

在WDM式驱动中，完成一个设备，至少需要两个设备对象，一个是物理设备对象（简称PDO），一个是功能设备对象（简称FDO），其中，FDO是附加到PDO上的（也可以说PDO的被附加对象是FDO，所以PDO的AttachedDevice中存储的是FDO）。当PC插入某个设备时，PDO会自动创建（实际是由总线驱动创建），PDO不能单独操作设备，需要配合FDO一起使用。当插入硬件的时候，会提示检测到新设备，安装的就是WDM驱动程序，由它创建的就是FDO。PDO被称作底层驱动或者下层驱动，FDO被称作高层或者上层驱动。这里是“上”层指的是接近发出I/O请求的地方，而“下”层指的是靠近物理设备的地方。PDO和FDO简单关系如下：



它俩还可能存在复杂一点的关系，比如FDO和PDO之间存在过滤驱动，如下：



程序中，附加操作，是调用IoAttachDeviceToDeviceStack，我们这里的HelloWDM是将我们HelloWDM的第一个设备对象FDO，附加到PDO，返回值是下一层的驱动设备对象（如果中间没有过滤层，下一层就是PDO；如果中间有过滤层，下一层就是Low FDO）。

当设备初始化完毕，需要我们自己告知上层管理器对象，设备已创建完成。告知方式，就是将FDO的flag，设置~DO\_DEVICE\_INITIALIZING。

一个PDO，附加到它之上的驱动，可能有很多个，这些直接附加到PDO之上的驱动，为一个层级关系，他们之间用NextDevice互连。

## 6.4总结

（1）NT式驱动，更像是一个软件服务，运行于内核中；WDM式驱动则更像是一个设备驱动，需要安装特定的INF文件

（2）WDM支持即插即用，NT不支持

# 7内核常用函数及原理

## 7.1内存管理

### 7.1.1虚拟内存



7.2字符串操作

7.3文件操作

7.4注册表操作

# 8派遣函数和IRP原理

## 8.1 IRP重要说明

### 8.1.1 IRP定义

* IRP中的MajorFunction和MinorFunction分别记录IRP的主类型和子类型。主类型是一个大的分类，子类型是某个分类下的子分类，比如IRP\_MJ\_PNP下有很多字类型，包括IRP\_MN\_REMOVE\_DEVICE等；
* 一个IRP对应一个派遣函数，NT式驱动和WDM式驱动设置IRP到派遣函数的映射关系，一般都在DriverEntry中；
* 系统默认的IRP的派遣函数是\_IopInvalidDeviceRequest

### 8.1.2 IRP类型

IRP的Major类型并不多，罗列如下：

|  |  |
| --- | --- |
| **IRP类型** | **来源** |
| IRP\_MJ\_CREATE | 创建设备，CreateFile会产生此IRP |
| IRP\_MJ\_CLOSE | 关闭设备，CloseHandle会产生此IRP |
| IRP\_MJ\_CLEANUP | 清除工作，CloseHandle会产生此IRP |
| IRP\_MJ\_DEVICE\_CONTROL | DeviceIoControl会产生此IRP |
| IRP\_MJ\_PNP | 即插即用消息，只有WDM驱动才支持 |
| IRP\_MJ\_POWER | 操作系统处理电源消息时，产生此IRP |
| IRP\_MJ\_QUERY\_INFORMATION | 获取文件长度，GetFileSize会产生此IRP |
| IRP\_MJ\_READ | 读取设备内容，ReadFile会产生此IRP |
| IRP\_MJ\_SET\_INFORMATION | 设置文件长度，GetFileSize会产生此IRP |
| IRP\_MJ\_SHUTDOWN | 关闭系统前会产生此IRP |
| IRP\_MJ\_SYSTEM\_CONTROL | 系统内部产生的控制信息，类似于内核调用DeviceIoControl |
| IRP\_MJ\_WRITE | 对设备进行WriteFile时会产生此IRP |
| IRP\_MJ\_DIRECTORY\_CONTROL | 读取目录变更，ReadDirectoryChanges会产生此IRP |
| IRP\_MJ\_FILE\_SYSTEM\_CONTROL | DeviceIoControl发出FSCTL时产生此IRP |
| IRP\_MJ\_FLUSH\_BUFFERS | 刷新缓冲区数据到磁盘，FlushFileBuffers产生此IRP |
| IRP\_MJ\_INTERNAL\_DEVICE\_CONTROL | 驱动内部的通信，不能用于用户层 |
| IRP\_MJ\_LOCK\_CONTROL | 锁文件和解锁文件，LockFile和UnlockFile产生IRP |
| IRP\_MJ\_QUERY\_EA | 读取文件扩展属性 |
| IRP\_MJ\_SET\_EA | 设置文件扩展属性 |
| IRP\_MJ\_SET\_QUOTA | 设置磁盘配额，IDiskQuotaControl::SetQuotaState产生此IRP |
| IRP\_MJ\_SET\_SECURITY | 设置安全级别，SetSecurityInfo产生此IRP |
| IRP\_MJ\_SET\_VOLUME\_INFORMATION | 设置磁盘卷等，SetVolumeLabel会产生此IRP |

所有的IRP，均可以通过MSDN上的详细介绍，得知其使用方式和具体参数含义

### 8.1.3 IRPTrace工具使用

工具无法使用，不知道为啥

## 8.2派遣函数使用方式

### 8.2.1最简单派遣函数处理

大部分IRP源于文件I/O处理的Win32API，如CreateFile、ReadFile、WriteFile等，处理这些IRP最简单的方式，就是：

* **将IRP的状态设置为成功**；—— 发起IRP请求的API（如WriteFile）将会返回TRUE
* **结束IRP的请求**；—— 调用IoCompleteRequest。在这之前，上层Win32API都是出于等待状态（线程睡眠，或者说处于Pending状态），IoCompleteRequest会设置等待事件，唤醒等待线程
* **让派遣函数返回成功**；

代码参考：

|  |
| --- |
| NTSTATUS HelloDDKDispatchRoutine(IN PDEVICE\_OBJECT pDevObj,  IN PIRP pIrp)  {  UNREFERENCED\_PARAMETER(pDevObj);  KdPrint(("Enter HelloDDKDispatchRoutine\n"));  NTSTATUS status = STATUS\_SUCCESS;  // 完成IRP  pIrp->IoStatus.Status = status;  pIrp->IoStatus.Information = 0;  IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);  KdPrint(("Leave HelloDDKDispatchRoutine\n"));  return status;  } |

这里要说明一下：

1. IoStatus.Status表示返回状态成功or失败，IoStatus.Information通常表示读取或者写入多少字节（ReadFile或者WriteFile中的参数）
2. 对于IoCompleteRequest第二个参数，表示上述待恢复的线程的优先级，有如下：

|  |  |
| --- | --- |
| **优先级** | **说明** |
| IO\_NO\_INCREMENT | 不增加优先级 |
| IO\_CD\_ROM\_INCREMENT | 光驱设备增加的优先级 |
| IO\_DISK\_INCREMENT | 磁盘设备增加的优先级 |
| IO\_KEYBOARD\_INCREMENT | 键盘设备增加的优先级 |
| IO\_MOUSE\_INCREMENT | 鼠标设备增加的优先级 |
| IO\_NAMED\_PIPE\_INCREMENT | 命名管道增加的优先级 |
| IO\_NETWORK\_INCREMENT | 网络设备增加的优先级 |
| IO\_PARALLEL\_INCREMENT | 并口设备增加的优先级 |
| IO\_SERIAL\_INCREMENT | 串口设备增加的优先级 |
| IO\_SOUND\_INCREMENT | 声卡设备增加的优先级 |
| IO\_VIDEO\_INCREMENT | 视频设备增加的优先级 |
| SEMAPHORE\_INCREMENT | 信号灯增加的优先级 |

原则是，对于像键盘、鼠标等输入设备，需要更快的反应，就需要提升线程优先级。

这里以ReadFile为例，说明整个流程：



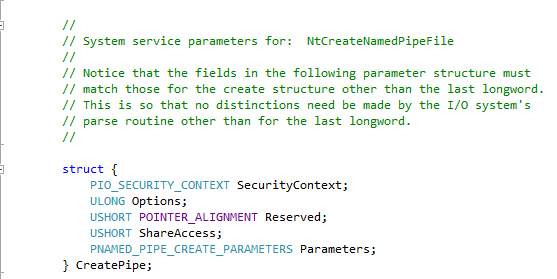
### 8.2.2更通用的派遣函数

我们的驱动，通常不止一层，是多层次结构，每一层有一个驱动对象，针对IRP进行处理，如[6.3驱动结构](#_6.3驱动结构)里面描述的。IO\_STACK\_LOCATION就是为了这个层次结构准备的，辅助IRP传递的一个结构，它记录了IRP在每一层中的一些操作信息，通过IoGetCurrentIrpStackLocation可以获取本层设备对应的IO\_STACK\_LOCATION。

下面的例子，说明了IO\_STACK\_LOCATION存储当前IRP的具体请求信息：

|  |
| --- |
| NTSTATUS HelloDDKDispatchRoutine(IN PDEVICE\_OBJECT pDevObj,  IN PIRP pIrp)  {  UNREFERENCED\_PARAMETER(pDevObj);  KdPrint(("Enter HelloDDKDispatchRoutine\n"));  PIO\_STACK\_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);  static char\* irpname[] = {  "IRP\_MJ\_CREATE",  "IRP\_MJ\_CREATE\_NAMED\_PIPE",  "IRP\_MJ\_CLOSE",  "IRP\_MJ\_READ",  "IRP\_MJ\_WRITE",  "IRP\_MJ\_QUERY\_INFORMATION",  "IRP\_MJ\_SET\_INFORMATION",  "IRP\_MJ\_QUERY\_EA",  "IRP\_MJ\_SET\_EA",  "IRP\_MJ\_FLUSH\_BUFFERS",  "IRP\_MJ\_QUERY\_VOLUME\_INFORMATION",  "IRP\_MJ\_SET\_VOLUME\_INFORMATION",  "IRP\_MJ\_DIRECTORY\_CONTROL",  "IRP\_MJ\_FILE\_SYSTEM\_CONTROL",  "IRP\_MJ\_DEVICE\_CONTROL",  "IRP\_MJ\_INTERNAL\_DEVICE\_CONTROL",  "IRP\_MJ\_SHUTDOWN",  "IRP\_MJ\_LOCK\_CONTROL",  "IRP\_MJ\_CLEANUP",  "IRP\_MJ\_CREATE\_MAILSLOT",  "IRP\_MJ\_QUERY\_SECURITY",  "IRP\_MJ\_SET\_SECURITY",  "IRP\_MJ\_POWER",  "IRP\_MJ\_SYSTEM\_CONTROL",  "IRP\_MJ\_DEVICE\_CHANGE",  "IRP\_MJ\_QUERY\_QUOTA",  "IRP\_MJ\_SET\_QUOTA",  "IRP\_MJ\_PNP"  };  UCHAR type = stack->MajorFunction;  if (type >= arraysize(irpname)) {  KdPrint(("Unknown irp, major type %x\n", type));  }  else {  KdPrint(("irpname: %s\n", irpname[type]));  }  NTSTATUS status = STATUS\_SUCCESS;  // 完成IRP  pIrp->IoStatus.Status = status;  pIrp->IoStatus.Information = 0; // 设置IRP操作字节数  IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);  KdPrint(("Leave HelloDDKDispatchRoutine\n"));  return status;  } |

IO\_STACK\_LOCATION这个结构太大，这里简单截图看看，如下：

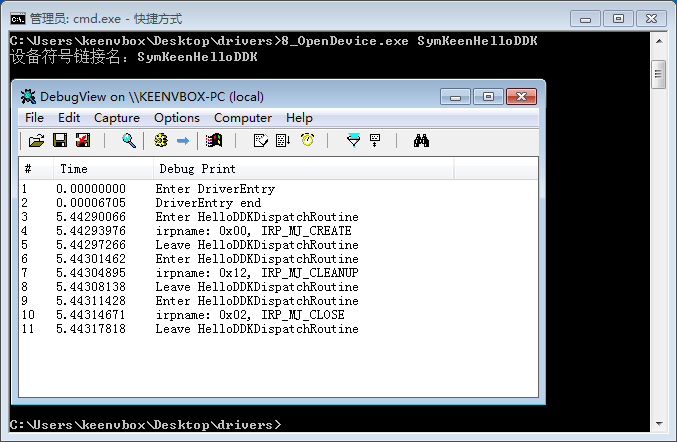


基本针对每一个IRP类别，都有一个结构与之关联，保存对应的信息，在各个设备对象之间传递。

## 8.3与设备通信——系统IRP

### 8.3.1通过设备链接打开设备

|  |
| --- |
| #include "stdafx.h"  #include <Windows.h>  #include <iostream>  // 命令行：OpenDevice.exe [设备符号链接名]  int main(int argc, char\* argv[])  {  if (argc != 2) {  std::cout << "参数个数不满足" << std::endl;  return -1;  }  LPCSTR lpstrDeviceSymName = argv[1];  if (!lpstrDeviceSymName) {  std::cout << "参数错误" << std::endl;  return -1;  }  std::cout << "设备符号链接名：" << lpstrDeviceSymName << std::endl;  CAtlStringA strFullDeviceSymName;  strFullDeviceSymName.Format("\\\\.\\%s", lpstrDeviceSymName);  HANDLE hDevice = CreateFileA(strFullDeviceSymName,  GENERIC\_WRITE | GENERIC\_READ,  0, // 不共享，也就是独占  NULL,  OPEN\_EXISTING,  FILE\_ATTRIBUTE\_NORMAL,  NULL);  if (hDevice == INVALID\_HANDLE\_VALUE) {  std::cout << "无法打开设备" << std::endl;  return -1;  }  CloseHandle(hDevice);  return 0;  } |



驱动部分，使用的是[6.1NT式驱动完整代码](#_6.1.1完整代码示例)以及[8.2.2通用派遣函数](#_8.2.2更通用的派遣函数)。

### 8.3.2缓冲区方式读写操作

**主要问题**：**驱动程序直接操作用户模式内存地址是非常危险的，因为用户进程可能随时被切换**，把用户模式地址传递给驱动，此时的驱动很可能已经被切换了。（例如：ReadFile会传递一个用户的缓冲区，来保存数据，这个缓冲区首地址会跟随IRP一起传递给驱动）

**解决方式之一**：采用缓冲区，**系统自动将用户地址空间内存拷贝到内核地址空间**。

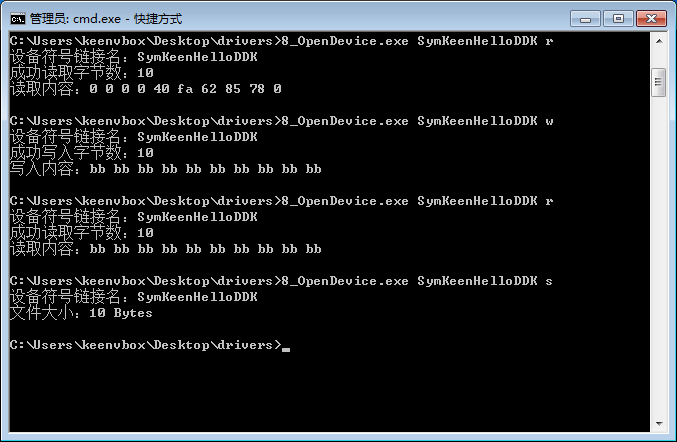
1. 系统自动分配ReadFile/WriteFile指定的字节数的内核内存，并且IRP.AssociatedIrp.SystemBuffer会记录这个内核内存地址。
2. 这段内核内存，由操作系统创建和回收
3. IO\_STACK\_LOCATION.Parameters.Read.Length记录ReadFile向驱动请求读取多少字节；IO\_STACK\_LOCATION.Parameters.Write.Length记录WriteFile请求写入多少字节。
4. IRP.IoStatus.Information记录实际读取或者写入的字节数，ReadFile/WriteFile通过第四个参数获取这个值

**优点**：比较简单的解决地址传入问题

**缺点**：需要在用户模式和内核模式之间拷贝内存数据，影响效率

|  |
| --- |
| 驱动程序Driver.h |
| #pragma once  // 因为这里采用C++编写，而ntddk的实现，是C编写。  // 这里为了防止ntddk的函数符号表导入错误  #ifdef \_\_cplusplus  extern "C"  {  #endif  #include <ntddk.h>  #ifdef \_\_cplusplus  }  #endif  #define PAGECODE code\_seq("PAGE") // 定义分页标记  #define LOCKEDCODE code\_seq() // 定义非分页标记  #define INITCODE code\_seq("INIT") // 定义初始化内存块。  // 函数只是在加载的时候载入内存，当驱动成功加载后，就可以从内存中卸载掉  #define PAGEDATA data\_seq("PAGE")  #define LOCKEDDATA data\_seq()  #define INITDATA data\_seq("INIT")  #define arraysize(p) (sizeof(p) / sizeof((p)[0]))  // 自定义设备扩展  #define MAX\_FILE\_LENGTH 1024  typedef struct \_DEVICE\_EXTENSION {  PDEVICE\_OBJECT pDevice;  UNICODE\_STRING ustrDeviceName; // 设备名，可以在设备管理器中显示的名字  UNICODE\_STRING ustrSymLinkName; // 符号链接名，用于ring3层访问设备对象的  PUCHAR buffer;//缓冲区  ULONG file\_length;//模拟的文件长度，必须小于MAX\_FILE\_LENGTH  } DEVICE\_EXTENSION, \*PDEVICE\_EXTENSION;  // 函数声明  NTSTATUS CreateDevice(IN PDRIVER\_OBJECT pDriverObject);  VOID HelloDDKUnload(IN PDRIVER\_OBJECT pDriverObject);  NTSTATUS HelloDDKDispatchRoutine(IN PDEVICE\_OBJECT pDevObj, IN PIRP pIrp);  NTSTATUS HelloDDKRead(IN PDEVICE\_OBJECT pDevObj, IN PIRP pIrp);  NTSTATUS HelloDDKWrite(IN PDEVICE\_OBJECT pDevObj, IN PIRP pIrp);  NTSTATUS HelloDDKQueryInformation(IN PDEVICE\_OBJECT pDevObj, IN PIRP pIrp); |
| 驱动程序Driver.cpp |
| #include "Driver.h"  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：DriverEntry  \* 功能描述：初始化驱动程序，定位和申请硬件资源，创建内核对象  \* 参数列表：  pDriverObject：从I/O管理器中传进来的驱动对象，由执行体组件-对象管理器创建和管理  pRegistryPath：驱动程序在注册表中的路径，由执行体组件-配置管理器创建和管理  \* 返回值：返回初始化驱动状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma INITCODE // 指明DriverEntry是加载到INIT内存区域中，驱动成功卸载后，可以退出内存  extern "C" NTSTATUS DriverEntry( // 特定的回调函数，都需要编译成C的函数命名  IN PDRIVER\_OBJECT pDriverObject,  IN PUNICODE\_STRING pRegistryPath)  {  UNREFERENCED\_PARAMETER(pRegistryPath);  KdPrint(("Enter DriverEntry\n")); // 使用KdPrint宏，可以在调试版本(Checked/Debug)打日志，发行版本(Free/Release)不打日志  // 注册其他驱动调用函数入口  pDriverObject->DriverUnload = HelloDDKUnload;  pDriverObject->MajorFunction[IRP\_MJ\_CREATE] = HelloDDKDispatchRoutine;  pDriverObject->MajorFunction[IRP\_MJ\_CLOSE] = HelloDDKDispatchRoutine;  pDriverObject->MajorFunction[IRP\_MJ\_WRITE] = HelloDDKWrite;  pDriverObject->MajorFunction[IRP\_MJ\_READ] = HelloDDKRead;  pDriverObject->MajorFunction[IRP\_MJ\_QUERY\_INFORMATION] = HelloDDKQueryInformation;  pDriverObject->MajorFunction[IRP\_MJ\_CLEANUP] = HelloDDKDispatchRoutine;  pDriverObject->MajorFunction[IRP\_MJ\_DEVICE\_CONTROL] = HelloDDKDispatchRoutine;  pDriverObject->MajorFunction[IRP\_MJ\_SET\_INFORMATION] = HelloDDKDispatchRoutine;  pDriverObject->MajorFunction[IRP\_MJ\_SHUTDOWN] = HelloDDKDispatchRoutine;  pDriverObject->MajorFunction[IRP\_MJ\_SYSTEM\_CONTROL] = HelloDDKDispatchRoutine;  // 创建驱动设备对象  NTSTATUS status = CreateDevice(pDriverObject);  KdPrint(("DriverEntry end\n"));  return status;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：CreateDevice  \* 功能描述：初始化设备对象  \* 参数列表：  pDriverObject：从I/O管理器中传进来的驱动对象，由执行体组件-对象管理器创建和管理  \* 返回值：返回初始化状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma INITCODE  NTSTATUS CreateDevice(  IN PDRIVER\_OBJECT pDriverObject)  {  // 创建设备名称  UNICODE\_STRING devName;  RtlInitUnicodeString(&devName, L"\\Device\\KeenDDKDevice");  // 创建设备。一个驱动，有一个默认的设备对象，大概这个就是一个标准吧  PDEVICE\_OBJECT pDevObj = NULL;  NTSTATUS status = IoCreateDevice(pDriverObject,  sizeof(DEVICE\_EXTENSION),  &devName,  FILE\_DEVICE\_UNKNOWN, // 虚拟设备，必须设置为FILE\_DEVICE\_UNKNOWN  0,  TRUE, // 指定此设备为独占设备，不能被其他应用程序所使用  &pDevObj); // 对象管理器来管理的，这里同时也会添加到驱动对象pDriverObject相应的参数中  if (!NT\_SUCCESS(status)) {  return status;  }  pDevObj->Flags |= DO\_BUFFERED\_IO; // 内存操作方式。DO\_BUFFERED\_IO表示读写操作使用系统复制缓冲区访问用户模式数据；  // 如果是DO\_DIRECT\_IO则表示读写操作使用内存描述符表访问用户模式数据  PDEVICE\_EXTENSION pDevExt = (PDEVICE\_EXTENSION)pDevObj->DeviceExtension;  pDevExt->pDevice = pDevObj; // 使用设备扩展的目的是，避免使用全局变量（有各种同步问题）  pDevExt->ustrDeviceName = devName;  //申请模拟文件的缓冲区  pDevExt->buffer = (PUCHAR)ExAllocatePool(PagedPool, MAX\_FILE\_LENGTH);  //设置模拟文件大小  pDevExt->file\_length = 0;  // 创建符号链接名  UNICODE\_STRING symLinkName;  RtlInitUnicodeString(&symLinkName, L"\\??\\SymKeenHelloDDK");  pDevExt->ustrSymLinkName = symLinkName; // 使用符号链接的目的是，方便用户模式应用程序可以访问设备。因为设备名只能用于内核访问  status = IoCreateSymbolicLink(&symLinkName, &devName);  if (!NT\_SUCCESS(status)) {  IoDeleteDevice(pDevObj);  return status;  }  return STATUS\_SUCCESS;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：HelloDDKUnload  \* 功能描述：负责驱动程序的卸载操作。会卸载整个设备链，删除设备对象和符号链接  \* 参数列表：  pDriverObject：从I/O管理器中传进来的驱动对象，由执行体组件-对象管理器创建和管理  \* 返回值：返回状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma PAGECODE  VOID HelloDDKUnload(IN PDRIVER\_OBJECT pDriverObject)  {  KdPrint(("Enter DriverUnload\n"));  PDEVICE\_OBJECT pNextObj = pDriverObject->DeviceObject;  while (pNextObj != NULL) {  PDEVICE\_EXTENSION pDevExt = (PDEVICE\_EXTENSION)pNextObj->DeviceExtension;  if (pDevExt->buffer) {  ExFreePool(pDevExt->buffer);  pDevExt->buffer = NULL;  }  // 删除符号链接  UNICODE\_STRING linkName = pDevExt->ustrSymLinkName;  IoDeleteSymbolicLink(&linkName);  pNextObj = pNextObj->NextDevice;  IoDeleteDevice(pDevExt->pDevice);  }  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：HelloDDKDispatchRoutine  \* 功能描述：对IRP进行处理  \* 参数列表：  pDevObj：功能设备对象  pIrp:从I/0请求包  \* 返回值：返回状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma PAGECODE  NTSTATUS HelloDDKDispatchRoutine(IN PDEVICE\_OBJECT pDevObj,  IN PIRP pIrp)  {  UNREFERENCED\_PARAMETER(pDevObj);  KdPrint(("Enter HelloDDKDispatchRoutine\n"));  PIO\_STACK\_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);  static char\* irpname[] = {  "IRP\_MJ\_CREATE",  "IRP\_MJ\_CREATE\_NAMED\_PIPE",  "IRP\_MJ\_CLOSE",  "IRP\_MJ\_READ",  "IRP\_MJ\_WRITE",  "IRP\_MJ\_QUERY\_INFORMATION",  "IRP\_MJ\_SET\_INFORMATION",  "IRP\_MJ\_QUERY\_EA",  "IRP\_MJ\_SET\_EA",  "IRP\_MJ\_FLUSH\_BUFFERS",  "IRP\_MJ\_QUERY\_VOLUME\_INFORMATION",  "IRP\_MJ\_SET\_VOLUME\_INFORMATION",  "IRP\_MJ\_DIRECTORY\_CONTROL",  "IRP\_MJ\_FILE\_SYSTEM\_CONTROL",  "IRP\_MJ\_DEVICE\_CONTROL",  "IRP\_MJ\_INTERNAL\_DEVICE\_CONTROL",  "IRP\_MJ\_SHUTDOWN",  "IRP\_MJ\_LOCK\_CONTROL",  "IRP\_MJ\_CLEANUP",  "IRP\_MJ\_CREATE\_MAILSLOT",  "IRP\_MJ\_QUERY\_SECURITY",  "IRP\_MJ\_SET\_SECURITY",  "IRP\_MJ\_POWER",  "IRP\_MJ\_SYSTEM\_CONTROL",  "IRP\_MJ\_DEVICE\_CHANGE",  "IRP\_MJ\_QUERY\_QUOTA",  "IRP\_MJ\_SET\_QUOTA",  "IRP\_MJ\_PNP"  };  UCHAR type = stack->MajorFunction;  if (type >= arraysize(irpname)) {  KdPrint(("Unknown irp, major type 0x%02x\n", type));  }else {  KdPrint(("irpname: 0x%02x, %s\n", type, irpname[type]));  }  NTSTATUS status = STATUS\_SUCCESS;  // 完成IRP  pIrp->IoStatus.Status = status;  pIrp->IoStatus.Information = 0; // 设置IRP操作字节数  IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);  KdPrint(("Leave HelloDDKDispatchRoutine\n"));  return status;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：HelloDDKRead  \* 功能描述：对读IRP进行处理  \* 参数列表：  pDevObj：功能设备对象  pIrp:从I/0请求包  \* 返回值：返回状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  NTSTATUS HelloDDKRead(IN PDEVICE\_OBJECT pDevObj,  IN PIRP pIrp)  {  UNREFERENCED\_PARAMETER(pDevObj);  KdPrint(("Enter HelloDDKRead\n"));  NTSTATUS status = STATUS\_SUCCESS;  // 获取需要读设备的字节数  PIO\_STACK\_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);  ULONG ulReadLength = stack->Parameters.Read.Length;  ULONG ulReadOffset = (ULONG)stack->Parameters.Read.ByteOffset.QuadPart;  PDEVICE\_EXTENSION pDevExt = (PDEVICE\_EXTENSION)pDevObj->DeviceExtension;  if (ulReadLength + ulReadOffset > MAX\_FILE\_LENGTH) {  status = STATUS\_FILE\_INVALID;  ulReadLength = 0;  }else {  // 将数据保存在缓冲区，以便后续使用。如果ReadFile传递进来的一个指针参数pBuffer（用户模式地址空间）的话，操作系统会自动将SystemBuffer（内核模式地址空间）的内容，拷贝到pBuffer  memcpy(pIrp->AssociatedIrp.SystemBuffer, pDevExt->buffer + ulReadOffset, ulReadLength);  status = STATUS\_SUCCESS;  }  // 设置IRP完成状态  pIrp->IoStatus.Status = status;  pIrp->IoStatus.Information = ulReadLength; // 设置实际读取的字节数    IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);  KdPrint(("Leave HelloDDKRead\n"));  return status;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：HelloDDKWrite  \* 功能描述：对写IRP进行处理  \* 参数列表：  pDevObj：功能设备对象  pIrp:从I/0请求包  \* 返回值：返回状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  NTSTATUS HelloDDKWrite(IN PDEVICE\_OBJECT pDevObj, IN PIRP pIrp)  {  KdPrint(("Enter HelloDDKWrite\n"));  NTSTATUS status = STATUS\_SUCCESS;  // 获取需要读设备的字节数  PIO\_STACK\_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);  ULONG ulWriteLength = stack->Parameters.Write.Length;  ULONG ulWriteOffset = (ULONG)stack->Parameters.Write.ByteOffset.QuadPart;  // 将写入的数据，存储在缓冲区内  PDEVICE\_EXTENSION pDevExt = (PDEVICE\_EXTENSION)pDevObj->DeviceExtension;  if (ulWriteOffset+ ulWriteLength > MAX\_FILE\_LENGTH) {  // 存储长度大于缓冲区长度  status = STATUS\_FILE\_INVALID;  ulWriteLength = 0;  }else {  // 将写入的数据，存储在缓冲区。如果WriteFile传递进来的一个指针参数pBuffer（用户模式地址空间），操作系统会自动将pBuffer的内容拷贝到SystemBuffer（内核模式地址空间）中  memcpy(pDevExt->buffer + ulWriteOffset, pIrp->AssociatedIrp.SystemBuffer, ulWriteLength);  status = STATUS\_SUCCESS;  if (ulWriteLength + ulWriteOffset > pDevExt->file\_length) {  pDevExt->file\_length = ulWriteLength + ulWriteOffset;  }  }  // 设置IRP完成状态  pIrp->IoStatus.Status = status;  pIrp->IoStatus.Information = ulWriteLength; // 设置实际读取的字节数  IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);  KdPrint(("Leave HelloDDKWrite\n"));  return status;  }  NTSTATUS HelloDDKQueryInformation(IN PDEVICE\_OBJECT pDevObj, IN PIRP pIrp)  {  KdPrint(("Enter HelloDDKQueryInformation\n"));  NTSTATUS status = STATUS\_SUCCESS;  // 获取需要读设备的字节数  PIO\_STACK\_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);  FILE\_INFORMATION\_CLASS info = stack->Parameters.QueryFile.FileInformationClass;  // 将写入的数据，存储在缓冲区内  PDEVICE\_EXTENSION pDevExt = (PDEVICE\_EXTENSION)pDevObj->DeviceExtension;  if (info == FileStandardInformation) {  // 标准文件  KdPrint(("FileStandardInformation\n"));  PFILE\_STANDARD\_INFORMATION file\_info = (PFILE\_STANDARD\_INFORMATION)pIrp->AssociatedIrp.SystemBuffer;  file\_info->EndOfFile = RtlConvertLongToLargeInteger(pDevExt->file\_length);  }  // 设置IRP完成状态  pIrp->IoStatus.Status = status;  pIrp->IoStatus.Information = stack->Parameters.QueryFile.Length; // 设置实际读取的字节数  IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);  KdPrint(("Leave HelloDDKQueryInformation\n"));  return status;  } |
| 测试应用程序 |
| // 8\_OpenDevice.cpp : Defines the entry point for the console application.  //  #include "stdafx.h"  #include <Windows.h>  #include <iostream>  #include <iomanip>  // 命令行：OpenDevice.exe [设备符号链接名] [r/w]  int main(int argc, char\* argv[])  {  if (argc < 2) {  std::cout << "参数个数不满足" << std::endl;  return -1;  }  LPCSTR lpstrDeviceSymName = argv[1];  if (!lpstrDeviceSymName) {  std::cout << "参数错误" << std::endl;  return -1;  }  std::cout << "设备符号链接名：" << lpstrDeviceSymName << std::endl;  CAtlStringA strFullDeviceSymName;  strFullDeviceSymName.Format("\\\\.\\%s", lpstrDeviceSymName);  HANDLE hDevice = CreateFileA(strFullDeviceSymName,  GENERIC\_WRITE | GENERIC\_READ,  0, // 不共享，也就是独占  NULL,  OPEN\_EXISTING,  FILE\_ATTRIBUTE\_NORMAL,  NULL);  if (hDevice == INVALID\_HANDLE\_VALUE) {  std::cout << "无法打开设备" << std::endl;  return -1;  }  if (argc == 3) {  CHAR\* pCh = argv[2];  if (\*pCh == 'r') {  UCHAR buffer[10] = { 0 };  ULONG ulReaded = 0;  BOOL bRet = ReadFile(hDevice, buffer, 10, &ulReaded, NULL);  if (bRet) {  std::cout << "成功读取字节数：" << ulReaded << std::endl;  std::cout << "读取内容：";  for (size\_t i = 0; i < ulReaded; i++) {  std::cout << std::hex << (int)buffer[i] << " ";  }  std::cout << std::endl;  }else {  std::cout << "读取失败" << std::endl;  }  }else if (\*pCh == 'w') {  UCHAR buffer[10] = { 0 };  ULONG ulWrited = 0;  memset(buffer, 0xBB, 10);  BOOL bRet = WriteFile(hDevice, buffer, 10, &ulWrited, NULL);  if (bRet) {  std::cout << "成功写入字节数：" << ulWrited << std::endl;  std::cout << "写入内容：";  for (size\_t i = 0; i < ulWrited; i++) {  std::cout << std::hex << (int)buffer[i] << " ";  }  std::cout << std::endl;  }else {  std::cout << "写入失败" << std::endl;  }  }else if (\*pCh == 's') {  LARGE\_INTEGER llFileSize;  llFileSize.QuadPart = 0;  llFileSize.LowPart = GetFileSize(hDevice, (LPDWORD)&llFileSize.HighPart);  std::cout << "文件大小：" << llFileSize.QuadPart << " Bytes" << std::endl;  }  }  CloseHandle(hDevice);  return 0;  } |

实际效果如下：

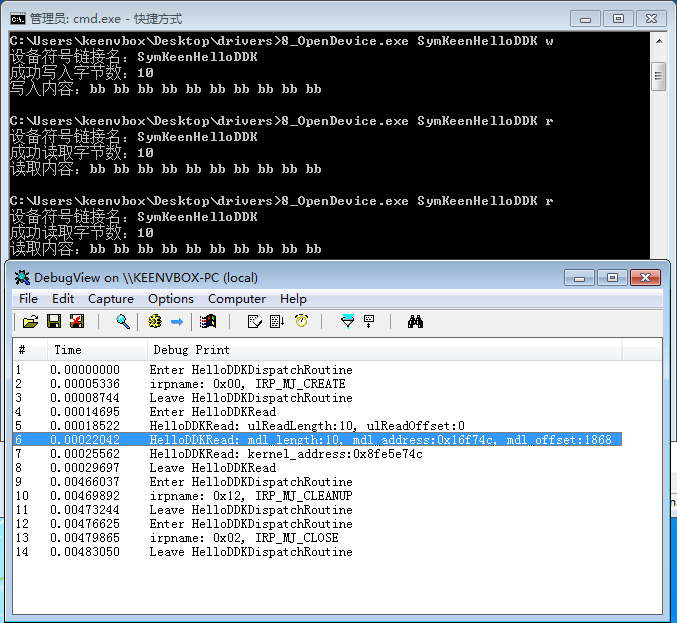


### 8.3.3直接方式读写操作

**划重点**：直接方式读写设备，**操作系统会自动将用户模式下的缓冲区锁住，然后自动将这段缓冲区在内核模式地址重新映射一遍，这样用户模式的缓冲区和内核模式的缓冲区指向的是同一区域物理内存**。操作系统将用户模式下的缓冲区锁住后，用内存描述符表（MDL数据结构）记录这段内存。

|  |
| --- |
| Driver.cpp |
| #include "Driver.h"  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：DriverEntry  \* 功能描述：初始化驱动程序，定位和申请硬件资源，创建内核对象  \* 参数列表：  pDriverObject：从I/O管理器中传进来的驱动对象，由执行体组件-对象管理器创建和管理  pRegistryPath：驱动程序在注册表中的路径，由执行体组件-配置管理器创建和管理  \* 返回值：返回初始化驱动状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma INITCODE // 指明DriverEntry是加载到INIT内存区域中，驱动成功卸载后，可以退出内存  extern "C" NTSTATUS DriverEntry( // 特定的回调函数，都需要编译成C的函数命名  IN PDRIVER\_OBJECT pDriverObject,  IN PUNICODE\_STRING pRegistryPath)  {  UNREFERENCED\_PARAMETER(pRegistryPath);  KdPrint(("Enter DriverEntry\n")); // 使用KdPrint宏，可以在调试版本(Checked/Debug)打日志，发行版本(Free/Release)不打日志  // 注册其他驱动调用函数入口  pDriverObject->DriverUnload = HelloDDKUnload;  pDriverObject->MajorFunction[IRP\_MJ\_CREATE] = HelloDDKDispatchRoutine;  pDriverObject->MajorFunction[IRP\_MJ\_CLOSE] = HelloDDKDispatchRoutine;  pDriverObject->MajorFunction[IRP\_MJ\_WRITE] = HelloDDKWrite;  pDriverObject->MajorFunction[IRP\_MJ\_READ] = HelloDDKRead;  pDriverObject->MajorFunction[IRP\_MJ\_QUERY\_INFORMATION] = HelloDDKQueryInformation;  pDriverObject->MajorFunction[IRP\_MJ\_CLEANUP] = HelloDDKDispatchRoutine;  pDriverObject->MajorFunction[IRP\_MJ\_DEVICE\_CONTROL] = HelloDDKDispatchRoutine;  pDriverObject->MajorFunction[IRP\_MJ\_SET\_INFORMATION] = HelloDDKDispatchRoutine;  pDriverObject->MajorFunction[IRP\_MJ\_SHUTDOWN] = HelloDDKDispatchRoutine;  pDriverObject->MajorFunction[IRP\_MJ\_SYSTEM\_CONTROL] = HelloDDKDispatchRoutine;  // 创建驱动设备对象  NTSTATUS status = CreateDevice(pDriverObject);  KdPrint(("DriverEntry end\n"));  return status;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：CreateDevice  \* 功能描述：初始化设备对象  \* 参数列表：  pDriverObject：从I/O管理器中传进来的驱动对象，由执行体组件-对象管理器创建和管理  \* 返回值：返回初始化状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma INITCODE  NTSTATUS CreateDevice(  IN PDRIVER\_OBJECT pDriverObject)  {  // 创建设备名称  UNICODE\_STRING devName;  RtlInitUnicodeString(&devName, L"\\Device\\KeenDDKDevice");  // 创建设备。一个驱动，有一个默认的设备对象，大概这个就是一个标准吧  PDEVICE\_OBJECT pDevObj = NULL;  NTSTATUS status = IoCreateDevice(pDriverObject,  sizeof(DEVICE\_EXTENSION),  &devName,  FILE\_DEVICE\_UNKNOWN, // 虚拟设备，必须设置为FILE\_DEVICE\_UNKNOWN  0,  TRUE, // 指定此设备为独占设备，不能被其他应用程序所使用  &pDevObj); // 对象管理器来管理的，这里同时也会添加到驱动对象pDriverObject相应的参数中  if (!NT\_SUCCESS(status)) {  return status;  }  pDevObj->Flags |= DO\_DIRECT\_IO; // 内存操作方式。DO\_BUFFERED\_IO表示读写操作使用系统复制缓冲区访问用户模式数据；  // 如果是DO\_DIRECT\_IO则表示读写操作使用内存描述符表访问用户模式数据  PDEVICE\_EXTENSION pDevExt = (PDEVICE\_EXTENSION)pDevObj->DeviceExtension;  pDevExt->pDevice = pDevObj; // 使用设备扩展的目的是，避免使用全局变量（有各种同步问题）  pDevExt->ustrDeviceName = devName;  //申请模拟文件的缓冲区  pDevExt->buffer = (PUCHAR)ExAllocatePool(PagedPool, MAX\_FILE\_LENGTH);  //设置模拟文件大小  pDevExt->file\_length = 0;  // 创建符号链接名  UNICODE\_STRING symLinkName;  RtlInitUnicodeString(&symLinkName, L"\\??\\SymKeenHelloDDK");  pDevExt->ustrSymLinkName = symLinkName; // 使用符号链接的目的是，方便用户模式应用程序可以访问设备。因为设备名只能用于内核访问  status = IoCreateSymbolicLink(&symLinkName, &devName);  if (!NT\_SUCCESS(status)) {  IoDeleteDevice(pDevObj);  return status;  }  return STATUS\_SUCCESS;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：HelloDDKUnload  \* 功能描述：负责驱动程序的卸载操作。会卸载整个设备链，删除设备对象和符号链接  \* 参数列表：  pDriverObject：从I/O管理器中传进来的驱动对象，由执行体组件-对象管理器创建和管理  \* 返回值：返回状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma PAGECODE  VOID HelloDDKUnload(IN PDRIVER\_OBJECT pDriverObject)  {  KdPrint(("Enter DriverUnload\n"));  PDEVICE\_OBJECT pNextObj = pDriverObject->DeviceObject;  while (pNextObj != NULL) {  PDEVICE\_EXTENSION pDevExt = (PDEVICE\_EXTENSION)pNextObj->DeviceExtension;  if (pDevExt->buffer) {  ExFreePool(pDevExt->buffer);  pDevExt->buffer = NULL;  }  // 删除符号链接  UNICODE\_STRING linkName = pDevExt->ustrSymLinkName;  IoDeleteSymbolicLink(&linkName);  pNextObj = pNextObj->NextDevice;  IoDeleteDevice(pDevExt->pDevice);  }  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：HelloDDKDispatchRoutine  \* 功能描述：对IRP进行处理  \* 参数列表：  pDevObj：功能设备对象  pIrp:从I/0请求包  \* 返回值：返回状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #pragma PAGECODE  NTSTATUS HelloDDKDispatchRoutine(IN PDEVICE\_OBJECT pDevObj,  IN PIRP pIrp)  {  UNREFERENCED\_PARAMETER(pDevObj);  KdPrint(("Enter HelloDDKDispatchRoutine\n"));  PIO\_STACK\_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);  static char\* irpname[] = {  "IRP\_MJ\_CREATE",  "IRP\_MJ\_CREATE\_NAMED\_PIPE",  "IRP\_MJ\_CLOSE",  "IRP\_MJ\_READ",  "IRP\_MJ\_WRITE",  "IRP\_MJ\_QUERY\_INFORMATION",  "IRP\_MJ\_SET\_INFORMATION",  "IRP\_MJ\_QUERY\_EA",  "IRP\_MJ\_SET\_EA",  "IRP\_MJ\_FLUSH\_BUFFERS",  "IRP\_MJ\_QUERY\_VOLUME\_INFORMATION",  "IRP\_MJ\_SET\_VOLUME\_INFORMATION",  "IRP\_MJ\_DIRECTORY\_CONTROL",  "IRP\_MJ\_FILE\_SYSTEM\_CONTROL",  "IRP\_MJ\_DEVICE\_CONTROL",  "IRP\_MJ\_INTERNAL\_DEVICE\_CONTROL",  "IRP\_MJ\_SHUTDOWN",  "IRP\_MJ\_LOCK\_CONTROL",  "IRP\_MJ\_CLEANUP",  "IRP\_MJ\_CREATE\_MAILSLOT",  "IRP\_MJ\_QUERY\_SECURITY",  "IRP\_MJ\_SET\_SECURITY",  "IRP\_MJ\_POWER",  "IRP\_MJ\_SYSTEM\_CONTROL",  "IRP\_MJ\_DEVICE\_CHANGE",  "IRP\_MJ\_QUERY\_QUOTA",  "IRP\_MJ\_SET\_QUOTA",  "IRP\_MJ\_PNP"  };  UCHAR type = stack->MajorFunction;  if (type >= arraysize(irpname)) {  KdPrint(("Unknown irp, major type 0x%02x\n", type));  }else {  KdPrint(("irpname: 0x%02x, %s\n", type, irpname[type]));  }  NTSTATUS status = STATUS\_SUCCESS;  // 完成IRP  pIrp->IoStatus.Status = status;  pIrp->IoStatus.Information = 0; // 设置IRP操作字节数  IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);  KdPrint(("Leave HelloDDKDispatchRoutine\n"));  return status;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：HelloDDKRead  \* 功能描述：对读IRP进行处理  \* 参数列表：  pDevObj：功能设备对象  pIrp:从I/0请求包  \* 返回值：返回状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  NTSTATUS HelloDDKRead(IN PDEVICE\_OBJECT pDevObj,  IN PIRP pIrp)  {  UNREFERENCED\_PARAMETER(pDevObj);  KdPrint(("Enter HelloDDKRead\n"));  NTSTATUS status = STATUS\_SUCCESS;  // 获取需要读设备的字节数  PIO\_STACK\_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);  ULONG ulReadLength = stack->Parameters.Read.Length;  ULONG ulReadOffset = (ULONG)stack->Parameters.Read.ByteOffset.QuadPart;  KdPrint(("HelloDDKRead: ulReadLength:%d, ulReadOffset:%d", ulReadLength, ulReadOffset));  PDEVICE\_EXTENSION pDevExt = (PDEVICE\_EXTENSION)pDevObj->DeviceExtension;  // 得到锁定缓冲区信息  ULONG mdl\_length = MmGetMdlByteCount(pIrp->MdlAddress);  PVOID mdl\_address = MmGetMdlVirtualAddress(pIrp->MdlAddress);  ULONG mdl\_offset = MmGetMdlByteOffset(pIrp->MdlAddress);  KdPrint(("HelloDDKRead: mdl\_length:%d, mdl\_address:0x%x, mdl\_offset:%d", mdl\_length, mdl\_address, mdl\_offset));  if ((mdl\_length != ulReadLength) || (ulReadLength + ulReadOffset > MAX\_FILE\_LENGTH)) {  status = STATUS\_FILE\_INVALID;  ulReadLength = 0;  }else {  // 得到内核模式下的映射，将设备里面的数据，读入内核模式地址，相当于映射给了外界设备  PVOID kernel\_address = MmGetSystemAddressForMdlSafe(pIrp->MdlAddress, NormalPagePriority);  KdPrint(("HelloDDKRead: kernel\_address:0x%x", kernel\_address));  memcpy(kernel\_address, pDevExt->buffer + ulReadOffset, ulReadLength);  status = STATUS\_SUCCESS;  }  // 设置IRP完成状态  pIrp->IoStatus.Status = status;  pIrp->IoStatus.Information = ulReadLength; // 设置实际读取的字节数  IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);  KdPrint(("Leave HelloDDKRead\n"));  return status;  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 函数名称：HelloDDKWrite  \* 功能描述：对写IRP进行处理  \* 参数列表：  pDevObj：功能设备对象  pIrp:从I/0请求包  \* 返回值：返回状态  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  NTSTATUS HelloDDKWrite(IN PDEVICE\_OBJECT pDevObj, IN PIRP pIrp)  {  KdPrint(("Enter HelloDDKWrite\n"));  NTSTATUS status = STATUS\_SUCCESS;  // 获取需要读设备的字节数  PIO\_STACK\_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);  ULONG ulWriteLength = stack->Parameters.Write.Length;  ULONG ulWriteOffset = (ULONG)stack->Parameters.Write.ByteOffset.QuadPart;  // 将写入的数据，存储在缓冲区内  PDEVICE\_EXTENSION pDevExt = (PDEVICE\_EXTENSION)pDevObj->DeviceExtension;  KdPrint(("HelloDDKWrite: ulWriteLength:%d, ulWriteOffset:%d", ulWriteLength, ulWriteOffset));  // 得到锁定缓冲区信息  ULONG mdl\_length = MmGetMdlByteCount(pIrp->MdlAddress);  PVOID mdl\_address = MmGetMdlVirtualAddress(pIrp->MdlAddress);  ULONG mdl\_offset = MmGetMdlByteOffset(pIrp->MdlAddress);  KdPrint(("HelloDDKRead: mdl\_length:%d, mdl\_address:0x%x, mdl\_offset:%d", mdl\_length, mdl\_address, mdl\_offset));  if ((mdl\_length != ulWriteLength) || (ulWriteOffset + ulWriteLength > MAX\_FILE\_LENGTH)){  status = STATUS\_FILE\_INVALID;  ulWriteLength = 0;  }else {  PVOID kernel\_address = MmGetSystemAddressForMdlSafe(pIrp->MdlAddress, NormalPagePriority);  KdPrint(("HelloDDKRead: kernel\_address:0x%x", kernel\_address));  memcpy(pDevExt->buffer + ulWriteOffset, kernel\_address, ulWriteLength);  status = STATUS\_SUCCESS;  if (ulWriteLength + ulWriteOffset > pDevExt->file\_length) {  pDevExt->file\_length = ulWriteLength + ulWriteOffset;  }  }  // 设置IRP完成状态  pIrp->IoStatus.Status = status;  pIrp->IoStatus.Information = ulWriteLength; // 设置实际读取的字节数  IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);  KdPrint(("Leave HelloDDKWrite\n"));  return status;  }  NTSTATUS HelloDDKQueryInformation(IN PDEVICE\_OBJECT pDevObj, IN PIRP pIrp)  {  KdPrint(("Enter HelloDDKQueryInformation\n"));  NTSTATUS status = STATUS\_SUCCESS;  // 获取需要读设备的字节数  PIO\_STACK\_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);  FILE\_INFORMATION\_CLASS info = stack->Parameters.QueryFile.FileInformationClass;  // 将写入的数据，存储在缓冲区内  PDEVICE\_EXTENSION pDevExt = (PDEVICE\_EXTENSION)pDevObj->DeviceExtension;  if (info == FileStandardInformation) {  // 标准文件  KdPrint(("FileStandardInformation\n"));  PFILE\_STANDARD\_INFORMATION file\_info = (PFILE\_STANDARD\_INFORMATION)pIrp->AssociatedIrp.SystemBuffer;  file\_info->EndOfFile = RtlConvertLongToLargeInteger(pDevExt->file\_length);  }  // 设置IRP完成状态  pIrp->IoStatus.Status = status;  pIrp->IoStatus.Information = stack->Parameters.QueryFile.Length; // 设置实际读取的字节数  IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);  KdPrint(("Leave HelloDDKQueryInformation\n"));  return status;  } |

实现效果如下：



### 8.3.4其他方式读写操作

**划重点**：此种方式，是在CreateDevice之时，内存Flag标记置为0，这种方式操作内存是危险的，**仅当驱动程序与应用程序运行在相同线程上下文的情况**，才能使用。

|  |
| --- |
| NTSTATUS HelloDDKRead(IN PDEVICE\_OBJECT pDevObj,  IN PIRP pIrp)  {  UNREFERENCED\_PARAMETER(pDevObj);  KdPrint(("Enter HelloDDKRead\n"));  NTSTATUS status = STATUS\_SUCCESS;  // 获取需要读设备的字节数  PIO\_STACK\_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);  ULONG ulReadLength = stack->Parameters.Read.Length;  ULONG ulReadOffset = (ULONG)stack->Parameters.Read.ByteOffset.QuadPart;  KdPrint(("HelloDDKRead: ulReadLength:%d, ulReadOffset:%d", ulReadLength, ulReadOffset));  PDEVICE\_EXTENSION pDevExt = (PDEVICE\_EXTENSION)pDevObj->DeviceExtension;  PVOID user\_address = pIrp->UserBuffer;  \_\_try {  ProbeForWrite(user\_address, ulReadLength, 4);  memset(user\_address, 0xAA, ulReadLength);  }  \_\_except (EXCEPTION\_EXECUTE\_HANDLER){  status = STATUS\_UNSUCCESSFUL;  }  // 设置IRP完成状态  pIrp->IoStatus.Status = status;  pIrp->IoStatus.Information = ulReadLength; // 设置实际读取的字节数  IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);  KdPrint(("Leave HelloDDKRead\n"));  return status;  } |

## 8.4与设备通信——自定义IRP

### 8.4.1 DeviceIoControl与驱动交互

通过自定义IOCTRL，可以实现应用层与驱动层交互。

|  |
| --- |
| // 8\_OpenDevice.cpp : Defines the entry point for the console application.  //  #include "stdafx.h"  #include <Windows.h>  #include <iostream>  #include <iomanip>  #include <winioctl.h>  #include "../6\_HelloDDK/define.h"  // 命令行：OpenDevice.exe [设备符号链接名] [r/w]  int main(int argc, char\* argv[])  {  if (argc < 2) {  std::cout << "参数个数不满足" << std::endl;  return -1;  }  LPCSTR lpstrDeviceSymName = argv[1];  if (!lpstrDeviceSymName) {  std::cout << "参数错误" << std::endl;  return -1;  }  std::cout << "设备符号链接名：" << lpstrDeviceSymName << std::endl;  CAtlStringA strFullDeviceSymName;  strFullDeviceSymName.Format("\\\\.\\%s", lpstrDeviceSymName);  HANDLE hDevice = CreateFileA(strFullDeviceSymName,  GENERIC\_WRITE | GENERIC\_READ,  0, // 不共享，也就是独占  NULL,  OPEN\_EXISTING,  FILE\_ATTRIBUTE\_NORMAL,  NULL);  if (hDevice == INVALID\_HANDLE\_VALUE) {  std::cout << "无法打开设备" << std::endl;  return -1;  }  if (argc == 3) {  CHAR\* pCh = argv[2];  if (\*pCh == 'r') {  UCHAR buffer[10] = { 0 };  ULONG ulReaded = 0;  BOOL bRet = ReadFile(hDevice, buffer, 10, &ulReaded, NULL);  if (bRet) {  std::cout << "成功读取字节数：" << ulReaded << std::endl;  std::cout << "读取内容：";  for (size\_t i = 0; i < ulReaded; i++) {  std::cout << std::hex << (int)buffer[i] << " ";  }  std::cout << std::endl;  }else {  std::cout << "读取失败" << std::endl;  }  }else if (\*pCh == 'w'){  UCHAR buffer[10] = { 0 };  ULONG ulWrited = 0;  memset(buffer, 0xBB, 10);  BOOL bRet = WriteFile(hDevice, buffer, 10, &ulWrited, NULL);  if (bRet) {  std::cout << "成功写入字节数：" << ulWrited << std::endl;  std::cout << "写入内容：";  for (size\_t i = 0; i < ulWrited; i++){  std::cout << std::hex << (int)buffer[i] << " ";  }  std::cout << std::endl;  }else {  std::cout << "写入失败" << std::endl;  }  }else if (\*pCh == 's'){  LARGE\_INTEGER llFileSize;  llFileSize.QuadPart = 0;  llFileSize.LowPart = GetFileSize(hDevice, (LPDWORD)&llFileSize.HighPart);  std::cout << "文件大小：" << llFileSize.QuadPart << " Bytes" << std::endl;  }else if (\*pCh == 'd'){  UCHAR bufferIn[10] = { 0 };  UCHAR bufferOut[10] = { 0 };  DWORD dwRet = 0;  // IOCTL\_TEST1  memset(bufferIn, 0xBB, 10);  BOOL bRet = DeviceIoControl(hDevice, IOCTL\_TEST1, bufferIn, 10, bufferOut, 10, &dwRet, NULL);  if (bRet) {  for (size\_t i = 0; i < dwRet; i++) {  std::cout << std::hex << (int)bufferOut[i] << " ";  }  std::cout << std::endl;  }else {  std::cout << "IOCTL\_TEST1通信失败" << std::endl;  }  // IOCTL\_TEST2  memset(bufferIn, 0xCC, 10);  dwRet = 0;  bRet = DeviceIoControl(hDevice, IOCTL\_TEST2, bufferIn, 10, bufferOut, 10, &dwRet, NULL);  if (bRet) {  std::cout << "IOCTL\_TEST2通信成功" << std::endl;  for (size\_t i = 0; i < dwRet; i++) {  std::cout << std::hex << (int)bufferOut[i] << " ";  }  std::cout << std::endl;  }else {  std::cout << "IOCTL\_TEST2通信失败" << std::endl;  }  // IOCTL\_TEST3  memset(bufferIn, 0xBC, 10);  dwRet = 0;  bRet = DeviceIoControl(hDevice, IOCTL\_TEST3, bufferIn, 10, bufferOut, 10, &dwRet, NULL);  if (bRet) {  std::cout << "IOCTL\_TEST3通信成功" << std::endl;  for (size\_t i = 0; i < dwRet; i++) {  std::cout << std::hex << (int)bufferOut[i] << " ";  }  std::cout << std::endl;  }else {  std::cout << "IOCTL\_TEST3通信失败" << std::endl;  }  }  }  CloseHandle(hDevice);  return 0;  } |

### 8.4.2三种交互方式：缓冲区、直接、其他

（1）**缓冲内存模式IOCTL**：DeviceIoControl自动将输入内容复制到SystemBuffer，它同时作为输出

（2）**直接内存模式IOCTL**：DeviceIoControl自动将输入内容复制到SystemBuffer，但是使用内存描述符Mdl映射同一物理内存

（3）**其他内存模式IOCTL**：危险，DeviceIoControl将输入内容保存到Type3InputBuffer中

|  |
| --- |
| NTSTATUS HelloDDKDeviceControl(IN PDEVICE\_OBJECT pDevObj, IN PIRP pIrp)  {  UNREFERENCED\_PARAMETER(pDevObj);  KdPrint(("Enter HelloDDKDeviceControl\n"));  NTSTATUS status = STATUS\_SUCCESS;  // 获取需要读设备的字节数  PIO\_STACK\_LOCATION stack = IoGetCurrentIrpStackLocation(pIrp);  // 得到输入缓冲区数据  ULONG cbin = stack->Parameters.DeviceIoControl.InputBufferLength;  ULONG cbout = stack->Parameters.DeviceIoControl.OutputBufferLength;  ULONG code = stack->Parameters.DeviceIoControl.IoControlCode;  ULONG length = 0;  switch (code) {  case IOCTL\_TEST1:{  KdPrint(("HelloDDKDeviceControl, IOCTL\_TEST1\n"));  // 输入缓冲区  UCHAR\* bufferIn = (UCHAR\*)pIrp->AssociatedIrp.SystemBuffer;  for (size\_t i = 0; i < cbin; i++) {  KdPrint(("0x%x ", bufferIn[i]));  }  KdPrint(("\n"));  // 输出缓冲区  UCHAR\* bufferOut = (UCHAR\*)pIrp->AssociatedIrp.SystemBuffer;  memset(bufferOut, 0x11, cbout);  length = cbout;  }break;  case IOCTL\_TEST2:{  KdPrint(("HelloDDKDeviceControl, IOCTL\_TEST2\n"));  // 输入缓冲区  UCHAR\* bufferIn = (UCHAR\*)pIrp->AssociatedIrp.SystemBuffer;  for (size\_t i = 0; i < cbin; i++){  KdPrint(("0x%x ", bufferIn[i]));  }  KdPrint(("\n"));  // 输出缓冲区  KdPrint(("user address: 0x%x\n", MmGetMdlVirtualAddress(pIrp->MdlAddress)));  UCHAR\* bufferOut = (UCHAR\*)MmGetSystemAddressForMdlSafe(pIrp->MdlAddress, NormalPagePriority);  KdPrint(("kernel address: 0x%x\n", bufferOut));  memset(bufferOut, 0x22, cbout);  length = cbout;  }break;  case IOCTL\_TEST3:{  KdPrint(("HelloDDKDeviceControl, IOCTL\_TEST3\n"));  // 输入缓冲区  UCHAR\* bufferIn = (UCHAR\*)stack->Parameters.DeviceIoControl.Type3InputBuffer;  KdPrint(("user address in: 0x%x\n", bufferIn));  // 输出缓冲区  UCHAR\* bufferOut = (UCHAR\*)pIrp->UserBuffer;  KdPrint(("user address out: 0x%x\n", bufferOut));  \_\_try {  // 判断指针可读，然后显示输入缓冲区内容  ProbeForRead(bufferIn, cbin, 4);  for (size\_t i = 0; i < cbin; i++){  KdPrint(("0x%x ", bufferIn[i]));  }  KdPrint(("\n"));  // 判断指针可写，然后输出内容到输出缓冲区  ProbeForWrite(bufferOut, cbout, 4);  memset(bufferOut, 0x33, cbout);  length = cbout;  }\_\_except (EXCEPTION\_EXECUTE\_HANDLER) {  status = STATUS\_UNSUCCESSFUL;  }  }  break;  default:  status = STATUS\_INVALID\_VARIANT;  break;  }  // 设置IRP完成状态  pIrp->IoStatus.Status = status;  pIrp->IoStatus.Information = length; // 设置实际读取的字节数  IoCompleteRequest(pIrp, IO\_NO\_INCREMENT);  KdPrint(("Leave HelloDDKDeviceControl\n"));  return status;  } |



# 9驱动程序同步处理