版权声明

作者信息

作者网名楚狂人。真名谭文。在上海从事 Windows 驱动开发相关的工作。对本书任何内容有任何疑问的读者，可以用下列方式和作者取得联系：

QQ:16191935

MSN:walled_river@hotmail.com

Email:mfc_tan_wen@163.com,walled_river@hotmail.com

前言

本书非常适合熟悉 Windows 应用编程的读者转向驱动开发。所有的内容都从最基础的编程方法入手。介绍相关的内核 API,然后举出示范的例子。这本书只有不到 70 页，是一本非常精简的小册子。所以它并不直接指导读者开发某种特定类型的驱动程序。而是起到一个入门指导的作用。

即使都是使用 C/C++语言的代码，在不同的应用环境中，常常看起来还是大相径庭。比如用 TurboC++编写的 DOS 程序代码和用 VC++编写的 MFC 应用程序的代码，看起来就几乎不像是同一种语言。这是由于它们所依赖的开发包不相同的缘故。

在任何情况下都以写出避免依赖的代码为最佳。这样可以避免重复劳动。但是我们在学习一种开发包的使用时，必须习惯这个环境的编码方式，以便获得充分利用这个开发包的能力。

本书的代码几乎都依赖于 WDK(Windows Driver Kit)。但是不限 WDK 的版本。WDK 还在不断的升级中。这个开发包是由微软公司免费提供的。读者可以在微软的网站上下载。

当然读者必须把 WDK 安装的计算机上并配置好开发环境。具体的安装和配置方法本书没有提供。因为网上已经有非常多的中文文档介绍它们。

读完这本书之后,读者一定可以更轻松的阅读其他专门的驱动程序开发的文档和相关书籍。而不至于看到大量无法理解的代码而中途放弃。如果有任何关于本书的内容的问题，读者可以随时发邮件到 mfc_tan_wen@163.com 或者 walled_river@hotmail.com。能够回答的问题我一般都会答复。

写本书的时候，我和 wowocock 合作的一本名为《天书夜读》(在网上有一个大约 20%内容的缩减电子版本)正在电子工业出版社编辑。预计还有不到一个月左右就会出版。这也是我自己所见的唯一一本中文原创的从汇编和反汇编角度来学习 Windows 内核编程和信息安全软件开发的书。希望读者多多支持。有想购买的读者请发邮件给我。我会在本书出版的第一时间，回复邮件告知购买的方法。

此外我正在写另一本关于 Windows 安全软件的驱动编程的书。但是题目还没有拟好。实际上，读者现在见到的免费版本的《Windows 驱动编程基础教程》是从这本书的第一部分中节选出来的。这本书篇幅比较大，大约有 600-800 页。主要内容如下：

第一章驱动编程基础
第二章磁盘设备驱动
第三章磁盘还原与加密
第四章传统文件系统过滤
第五章小端口文件系统过滤
第六章文件系统保护与加密
第七章协议网络驱动
第八章物理网络驱动
第九章网络防火墙与安全连接
第十章打印机驱动与虚拟打印
第十一章视频驱动与过滤
附录 A WDK 的安装与驱动开发的环境配置
附录 B 用 WinDbg 调试 Windows 驱动程序

这本书还没有完成。但是肯定要付出巨大的精力，所以请读者不要来邮件索取完整的免

费的电子版本。希望读者支持本书的纸版出版。因为没有完成，所以还没有联系出版商。有愿意合作出版本书的读者请发邮件与我联系。

凡是发送邮件给我的读者，我将会发送邮件提供本人作品最新的出版信息，以及最新发布的驱动开发相关的免费电子书。如果不需要这些信息的，请在邮件里注明，或者回复邮件给我来取消订阅。

谭文

2008 年 6 月 9 日

目录

第一章 字符串

1.1 使用字符串结构

常常使用传统 C 语言的程序员比较喜欢用如下的方法定义和使用字符串：

```
char*str = { “my first string” };          // ansi 字符串
wchar_t *wstr = { L”my first string” };    // unicode 字符串
size_t len = strlen(str);                  // ansi 字符串求长度
size_t wlen = wcslen(wstr);              // unicode 字符串求长度
printf(“%s %ws %d %d”,str,wstr,len,wlen); // 打印两种字符串
```

但是实际上这种字符串相当的不安全。很容易导致缓冲溢出漏洞。这是因为没有任何地方确切的表明一个字符串的长度。仅仅用一个’\0’字符来标明这个字符串的结束。一旦碰到根本就没有空结束的字符串（可能是攻击者恶意的输入、或者是编程错误导致的意外），程序就可能陷入崩溃。

使用高级C++特性的编码者则容易忽略这个问题。因为常常使用std::string和CString这样高级的类。不用去担忧字符串的安全性了。

在驱动开发中，一般不再用空来表示一个字符串的结束。而是定义了如下的一个结构：

```
typedef struct _UNICODE_STRING {
    USHORT Length;              // 字符串的长度（字节数）
    USHORT MaximumLength;       // 字符串缓冲区的长度（字节数）
    PWSTR  Buffer;              // 字符串缓冲区
} UNICODE_STRING, *PUNICODE_STRING;
```

以上是 Unicode 字符串，一个字符为双字节。与之对应的还有一个 Ansi 字符串。Ansi 字符串就是 C 语言中常用的单字节表示一个字符的窄字符串。

```
typedef struct _STRING {
    USHORT Length;
    USHORT MaximumLength;
    PSTR Buffer;
} ANSI_STRING, *PANSI_STRING;
```

在驱动开发中四处可见的是 Unicode 字符串。因此可以说：Windows 的内核是使用 Uincode 编码的。ANSI_STRING 仅仅在某些碰到窄字符的场合使用。而且这种场合非常罕见。

UNICODE_STRING 并不保证 Buffer 中的字符串是以空结束的。因此，类似下面的做法都是错误的，可能会会导致内核崩溃：

```
UNICODE_STRING str;
…
len = wcslen(str.Buffer);            // 试图求长度。
DbgPrint(“%ws”,str.Buffer);          // 试图打印 str.Buffer。
```

如果要用以上的方法，必须在编码中保证 Buffer 始终是以空结束。但这又是一个麻烦的问题。所以，使用微软提供的 Rtl 系列函数来操作字符串，才是正确的方法。下文逐步的讲述这个系列的函数的使用。

1.2 字符串的初始化

请回顾之前的 UNICODE_STRING 结构。读者应该可以注意到，这个结构中并不含有字符串缓冲的空间。这是一个初学者常见的出问题的来源。以下的代码是完全错误的，内核会立刻崩溃：

```
UNICODE_STRING str;
wcscpy(str.Buffer,L”my first string!”);
str.Length = str.MaximumLength = wcslen(L”my first string!”) * sizeof(WCHAR);
```

以上的代码定义了一个字符串并试图初始化它的值。但是非常遗憾这样做是不对的。因为 str.Buffer 只是一个未初始化的指针。它并没有指向有意义的空间。相反以下的方法是正确的：

```
// 先定义后，再定义空间
UNICODE_STRING str;
str.Buffer = L”my first string!”;
str.Length = str.MaximumLength = wcslen(L”my first string!”) * sizeof(WCHAR);
… …
```

上面代码的第二行手写的常数字符串在代码中形成了“常数”内存空间。这个空间位于代码段。将被分配于可执行页面上。一般的情况下不可写。为此，要注意的是这个字符串空间一旦初始化就不要再更改。否则可能引发系统的保护异常。实际上更好的写法如下：

```
//请分析一下为何这样写是对的：
UNICODE_STRING str = {
    sizeof(L”my first string!”) - sizeof((L”my first string!”)[0]),
    sizeof(L”my first string!”),
    L”my first_string!” };
```

但是这样定义一个字符串实在太繁琐了。但是在头文件 ntdef.h 中有一个宏方便这种定义。使用这个宏之后，我们就可以简单的定义一个常数字符串如下：

```
#include <ntdef.h>
```

```
UNICODE_STRING str = RTL_CONSTANT_STRING(L“my first string!”);
```

这只能在定义这个字符串的时候使用。为了随时初始化一个字符串，可以使用 RtlInitUnicodeString。示例如下：

```
UNICODE_STRING str;
RtlInitUnicodeString(&str,L”my first string!”);
```

用本小节的方法初始化的字符串，不用担心内存释放方面的问题。因为我们并没有分配任何内存。

1.3 字符串的拷贝

因为字符串不再是空结束的，所以使用 wcscpy 来拷贝字符串是不行的。UNICODE_STRING 可以用 RtlCopyUnicodeString 来进行拷贝。在进行这种拷贝的时候，最需要注意的一点是：拷贝目的字符串的 Buffer 必须有足够的空间。如果 Buffer 的空间不足，字符串会拷贝不完全。这是一个比较隐蔽的错误。

下面举一个例子。

```
UNICODE_STRING dst;          // 目标字符串
WCHAR dst_buf[256];              // 我们现在还不会分配内存，所以先定义缓冲区
UNICODE_STRING src = RTL_CONST_STRING(L”My source string!”);

// 把目标字符串初始化为拥有缓冲区长度为 256 的 UNICODE_STRING 空串。
RtlInitEmptyString(dst,dst_buf,256*sizeof(WCHAR));
RtlCopyUnicodeString(&dst,&src);    // 字符串拷贝！
```

以上这个拷贝之所以可以成功，是因为 256 比 L” My source string!”的长度要大。如果小，则拷贝也不会出现任何明示的错误。但是拷贝结束之后，与使用者的目标不符，字符串实际上被截短了。

我曾经犯过的一个错误是没有调用 RtlInitEmptyString。结果 dst 字符串被初始化认为缓冲区长度为 0。虽然程序没有崩溃，却实际上没有拷贝任何内容。

在拷贝之前，最谨慎的方法是根据源字符串的长度动态分配空间。在 1.2 节“内存与链表”中，读者会看到动态分配内存处理字符串的方法。

1.4 字符串的连接

UNICODE_STRING 不再是简单的字符串。操作这个数据结构往往需要更多的耐心。读者会常常碰到这样的需求：要把两个字符串连接到一起。简单的追加一个字符串并不困难。重要的依然是保证目标字符串的空间大小。下面是范例：

```
NTSTATUS status;
UNICODE_STRING dst;          // 目标字符串
WCHAR dst_buf[256];              // 我们现在还不会分配内存，所以先定义缓冲区
UNICODE_STRING src = RTL_CONST_STRING(L”My source string!”);

// 把目标字符串初始化为拥有缓冲区长度为 256 的 UNICODE_STRING 空串
RtlInitEmptyString(dst,dst_buf,256*sizeof(WCHAR));
RtlCopyUnicodeString(&dst,&src);    // 字符串拷贝!

status = RtlAppendUnicodeToString(
        &dst,L”my second string!”);
if(status != STATUS_SUCCESS)
{
    ……
}
```

NTSTATUS 是常见的返回值类型。如果函数成功，返回 STATUS_SUCCESS。否则的话，是一个错误码。RtlAppendUnicodeToString 在目标字符串空间不足的时候依然可以连接字符串，但是会返回一个警告性的错误 STATUS_BUFFER_TOO_SMALL。

另外一种情况是希望连接两个 UNICODE_STRING，这种情况请调用 RtlAppendUnicodeStringToString。这个函数的第二个参数也是一个 UNICODE_STRING 的指针。

1.5 字符串的打印

字符串的连接另一种常见的情况是字符串和数字的组合。有时数字需要被转换为字符串。有时需要把若干个数字和字符串混合组合起来。这往往用于打印日志的时候。日志中可能含有文件名、时间、和行号，以及其他的信息。

熟悉 C 语言的读者会使用 sprintf。这个函数的宽字符版本为 swprintf。该函数在驱动开发中依然可以使用，但是不安全。微软建议使用 RtlStringCbPrintfW 来代替它。RtlStringCbPrintfW 需要包含头文件 ntstrsafe.h。在连接的时候，还需要连接库 ntsafestr.lib。

下面的代码生成一个字符串，字符串中包含文件的路径，和这个文件的大小。

```
#include <ntstrsafe.h>
// 任何时候，假设文件路径的长度为有限的都是不对的。应该动态的分配
// 内存。但是动态分配内存的方法还没有讲述，所以这里再次把内存空间
// 定义在局部变量中，也就是所谓的“在栈中”
WCHAR buf[512] = { 0 };
UNICODE_STRING dst;
NTSTATUS status;
……
```

```
// 字符串初始化为空串。缓冲区长度为 512*sizeof(WCHAR)
RtlInitEmptyString(dst,dst_buf,512*sizeof(WCHAR));

// 调用 RtlStringCbPrintfW 来进行打印
status = RtlStringCbPrintfW(
    dst->Buffer,L"file path = %wZ file size = %d \r\n",
    &file_path,file_size);
// 这里调用 wcslen 没问题，这是因为 RtlStringCbPrintfW 打印的
// 字符串是以空结束的。
dst->Length = wcslen(dst->Buffer) * sizeof(WCHAR);
```

RtlStringCbPrintfW 在目标缓冲区内存不足的时候依然可以打印，但是多余的部分被截去了。返回的 status 值为 STATUS_BUFFER_OVERFLOW。调用这个函数之前很难知道究竟需要多长的缓冲区。一般都采取倍增尝试。每次都传入一个为前次尝试长度为 2 倍长度的新缓冲区，直到这个函数返回 STATUS_SUCCESS 为止。

值得注意的是 UNICODE_STRING 类型的指针，用%wZ 打印可以打印出字符串。在不能保证字符串为空结束的时候，必须避免使用%ws 或者%s。其他的打印格式字符串与传统 C 语言中的 printf 函数完全相同。可以尽情使用。

另外就是常见的输出打印。printf 函数只有在有控制台输出的情况下才有意义。在驱动中没有控制台。但是 Windows 内核中拥有调试信息输出机制。可以使用特殊的工具查看打印的调试信息（请参阅附录 1“WDK 的安装与驱动开发的环境配置”）。

驱动中可以调用 DbgPrint()函数来打印调试信息。这个函数的使用和 printf 基本相同。但是格式字符串要使用宽字符。DbgPrint()的一个缺点在于，发行版本的驱动程序往往不希望附带任何输出信息，只有调试版本才需要调试信息。但是 DbgPrint()无论是发行版本还是调试版本编译都会有效。为此可以自己定义一个宏：

```
#if DBG
    KdPrint(a)  DbgPrint##a
#else
    KdPrint (a)
#endif
```

不过这样的后果是，由于 KdPrint (a)只支持 1 个参数，因此必须把 DbgPrint 的所有参数都括起来当作一个参数传入。导致 KdPrint 看起来很奇特的用了双重括弧：

```
// 调用 KdPrint 来进行输出调试信息
status = KdPrint ((
    L"file path = %wZ file size = %d \r\n",
    &file_path,file_size));
```

这个宏没有必要自己定义，WDK 包中已有。所以可以直接使用 KdPrint 来代替 DbgPrint 取得更方便的效果。

第二章 内存与链表

2.1 内存的分配与释放

内存泄漏是 C 语言中一个臭名昭著的问题。但是作为内核开发者，读者将有必要自己来面对它。在传统的 C 语言中，分配内存常常使用的函数是 malloc。这个函数的使用非常简单，传入长度参数就得到内存空间。在驱动中使用内存分配，这个函数不再有效。驱动中分配内存，最常用的是调用 ExAllocatePoolWithTag。其他的方法在本章范围内全部忽略。回忆前一小节关于字符串的处理的情况。一个字符串被复制到另一个字符串的时候，最好根据源字符串的空间长度来分配目标字符串的长度。下面的举例，是把一个字符串 src 拷贝到字符串 dst。

```
// 定义一个内存分配标记
#define MEM_TAG  'MyTt'
// 目标字符串，接下来它需要分配空间。
UNICODE_STRING dst = { 0 };
// 分配空间给目标字符串。根据源字符串的长度。
dst.Buffer =
    (PWCHAR)ExAllocatePoolWithTag(NonpagedPool,src->Length,MEM_TAG);
if(dst.Buffer == NULL)
{
    // 错误处理
    status = STATUS_INSUFFICIENT_RESOUCRES;
    ……
}
dst.Length = dst.MaximumLength = src->Length;
status = RtlCopyUnicodeString(&dst,&src);
ASSERT(status == STATUS_SUCCESS);
```

ExAllocatePoolWithTag 的第一个参数 NonpagedPool 表明分配的内存是锁定内存。这些内存永远真实存在于物理内存上。不会被分页交换到硬盘上去。第二个参数是长度。第三个参数是一个所谓的“内存分配标记”。

内存分配标记用于检测内存泄漏。想象一下，我们根据占用越来越多的内存的分配标记，就能大概知道泄漏的来源。一般每个驱动程序定义一个自己的内存标记。也可以在每个模块中定义单独的内存标记。内存标记是随意的 32 位数字。即使冲突也不会有什么问题。

此外也可以分配可分页内存，使用 PagedPool 即可。

ExAllocatePoolWithTag 分配的内存可以使用 ExFreePool 来释放。如果不释放，则永远泄漏。并不像用户进程关闭后自动释放所有分配的空间。即使驱动程序动态卸载，也不能释放空间。唯一的办法是重启计算机。

ExFreePool 只需要提供需要释放的指针即可。举例如下：

```
ExFreePool(dst.Buffer);
dst.Buffer = NULL;
dst.Length = dst.MaximumLength = 0;
```

ExFreePool 不能用来释放一个栈空间的指针。否则系统立刻崩溃。像以下的代码：

```
UNICODE_STRING src = RTL_CONST_STRING(L”My source string!”);
ExFreePool(src.Buffer);
```

会招来立刻蓝屏。所以请务必保持 ExAllocatePoolWithTag 和 ExFreePool 的成对关系。

2.2 使用 LIST_ENTRY

Windows 的内核开发者们自己开发了部分数据结构，比如说 LIST_ENTRY。

LIST_ENTRY 是一个双向链表结构。它总是在使用的时候，被插入到已有的数据结构中。下面举一个例子。我构筑一个链表，这个链表的每个节点，是一个文件名和一个文件大小两个数据成员组成的结构。此外有一个 FILE_OBJECT 的指针对象。在驱动中，这代表一个文件对象。本书后面的章节会详细解释。这个链表的作用是：保存了文件的文件名和长度。只要传入 FILE_OBJECT 的指针，使用者就可以遍历链表找到文件名和文件长度。

```
typedef struct {
    PFILE_OBJECT file_object;
    UNICODE_STRING file_name;
    LARGE_INTEGER file_length;
} MY_FILE_INFOR, *PMY_FILE_INFOR;
```

一些读者会马上注意到文件的长度用 LARGE_INTEGER 表示。这是一个代表长长整型的数据结构。这个结构我们在下一小小节“使用长长整型数据”中介绍。

为了让上面的结构成为链表节点，我必须在里面插入一个 LIST_ENTRY 结构。至于插入的位置并无所谓。可以放在最前，也可以放中间，或者最后面。但是实际上读者很快会发现把 LIST_ENTRY 放在开头是最简单的做法：

```
typedef struct {
    LIST_ENTRY list_entry;
    PFILE_OBJECT file_object;
    UNICODE_STRING file_name;
    LARGE_INTEGER file_length;
} MY_FILE_INFOR, *PMY_FILE_INFOR;
```

list_entry 如果是作为链表的头，在使用之前，必须调用 InitializeListHead 来初始化。下面是示例的代码：

```
// 我们的链表头
```

```
LIST_ENTRY      my_list_head;

// 链表头初始化。一般的说在应该在程序入口处调用一下
void MyFileInforInilt()
{
    InitializeListHead(&my_list_head);
}

// 我们的链表节点。里面保存一个文件名和一个文件长度信息。
typedef struct {
    LIST_ENTRY list_entry;
    PFILE_OBJECT file_object;
    PUNICODE_STRING file_name;
    LARGE_INTEGER file_length;
} MY_FILE_INFOR, *PMY_FILE_INFOR;

// 追加一条信息。也就是增加一个链表节点。请注意 file_name 是外面分配的。
// 内存由使用者管理。本链表并不管理它。
NTSTATUS MyFileInforAppendNode(
    PFILE_OBJECT file_object,
    PUNICODE_STRING file_name,
    PLARGE_INTEGER file_length)
{
    PMY_FILE_INFOR my_file_infor =
        (PMY_FILE_INFOR)ExAllocatePoolWithTag(
            PagedPool,sizeof(MY_FILE_INFOR),MEM_TAG);
    if(my_file_infor == NULL)
        return STATUS_INSUFFICIENT_RESOURES;

    // 填写数据成员。
    my_file_infor->file_object = file_object;
    my_file_infor->file_name = file_name;
    my_file_infor->file_length = file_length;

    // 插入到链表末尾。请注意这里没有使用任何锁。所以，这个函数不是多
    // 多线程安全的。在下面自旋锁的使用中讲解如何保证多线程安全性。
    InsertHeadList(&my_list_head, (PLIST_ENTRY)& my_file_infor);
    return STATUS_SUCCESS;
}
```

以上的代码实现了插入。可以看到 LIST_ENTRY 插入到 MY_FILE_INFOR 结构的头部的好处。这样一来一个 MY_FILE_INFOR 看起来就像一个 LIST_ENTRY。不过糟糕的是并非所有的情况都可以这样。比如 MS 的许多结构喜欢一开头是结构的长度。因此在通过 LIST_ENTRY 结构的地址获取所在的节点的地址的时候，有个地址偏移计算的过程。可以通过下面的一个

典型的遍历链表的示例中看到：

```
for(p = my_list_head.Flink; p != &my_list_head.Flink; p = p->Flink)
{
    PMY_FILE_INFOR elem =
        CONTAINING_RECORD(p,MY_FILE_INFOR, list_entry);
    // 在这里做需要做的事…
    }
}
```

其中的 CONTAINING_RECORD 是一个 WDK 中已经定义的宏，作用是通过一个 LIST_ENTRY 结构的指针，找到这个结构所在的节点的指针。定义如下：

```
#define CONTAINING_RECORD(address, type, field) ((type *)( \
    (PCHAR)(address) - \
    (ULONG_PTR)(&((type *)0)->field)))
```

从上面的代码中可以总结如下的信息：
LIST_ENTRY 中的数据成员 Flink 指向下一个 LIST_ENTRY。
整个链表中的最后一个 LIST_ENTRY 的 Flink 不是空。而是指向头节点。
得到 LIST_ENTRY 之后，要用 CONTAINING_RECORD 来得到链表节点中的数据。

2.3 使用长长整型数据

这里解释前面碰到的 LARGE_INTEGER 结构。与可能的误解不同，64 位数据并非要在 64 位操作系统下才能使用。在 VC 中，64 位数据的类型为__int64。定义写法如下：

```
__int64 file_offset;
```

上面之所以定义的变量名为 file_offset，是因为文件中的偏移量是一种常见的要使用 64 位数据的情况。同时，文件的大小也是如此（回忆上一小节中定义的文件大小）。32 位数据无符号整型只能表示到 4GB。而众所周知，现在超过 4GB 的文件绝对不罕见了。但是实际上__int64 这个类型在驱动开发中很少被使用。基本上被使用到的是一个共用体：LARGE_INTEGER。这个共用体定义如下：

```
typedef __int64 LONGLONG;
typedef union _LARGE_INTEGER {
    struct {
        ULONG LowPart;
        LONG HighPart;
    };
    struct {
        ULONG LowPart;
```

```
            LONG HighPart;
        } u;
        LONGLONG QuadPart;
    } LARGE_INTEGER;
```

这个共用体的方便之处在于，既可以很方便的得到高 32 位，低 32 位，也可以方便的得到整个 64 位。进行运算和比较的时候，使用 QuadPart 即可。

```
    LARGE_INTEGER a,b;
    a.QuadPart = 100;
    a.QuadPart *= 100;
    b.QuadPart = a.QuadPart;
    if(b.QuadPart > 1000)
    {
        KdPrint(“b.QuadPart < 1000, LowPart = %x HighPart = %x”,
b.LowPart,b.HighPart);
    }
```

上面这段代码演示了这种结构的一般用法。在实际编程中，会碰到大量的参数是 LARGE_INTEGER 类型的。

2.4 使用自旋锁

链表之类的结构总是涉及到恼人的多线程同步问题，这时候就必须使用锁。这里只介绍最简单的自选锁。

有些读者可能疑惑锁存在的意义。这和多线程操作有关。在驱动开发的代码中，大多是存在于多线程执行环境的。就是说可能有几个线程在同时调用当前函数。

这样一来，前文 1.2.2 中提及的追加链表节点函数就根本无法使用了。因为 MyFileInforAppendNode 这个函数只是简单的操作链表。如果两个线程同时调用这个函数来操作链表的话：注意这个函数操作的是一个全局变量链表。换句话说，无论有多少个线程同时执行，他们操作的都是同一个链表。这就可能发生，一个线程插入一个节点的同时，另一个线程也同时插入。他们都插入同一个链表节点的后边。这时链表就会发生问题。到底最后插入的是哪一个呢？要么一个丢失了。要么链表被损坏了。

如下的代码初始化获取一个自选锁：

```
    KSPIN_LOCK my_spin_lock;
    KeInitializeSpinLock(&my_spin_lock);
```

KeInitializeSpinLock 这个函数没有返回值。下面的代码展示了如何使用这个 SpinLock。在 KeAcquireSpinLock 和 KeReleaseSpinLock 之间的代码是只有单线程执行的。其他的线程会停留在 KeAcquireSpinLock 等候。直到 KeReleaseSpinLock 被调用。KIRQL 是一个中断级。KeAcquireSpinLock 会提高当前的中断级。但是目前忽略这个问题。中断级在后面讲述。

```
KIRQL irql;
KeAcquireSpinLock(&my_spin_lock,&irql);
// To do something …
KeReleaseSpinLock(&my_spin_lock,irql);
```

初学者要注意的是，像下面写的这样的“加锁”代码是没有意义的，等于没加锁：

```
void MySafeFunction()
{
    KSPIN_LOCK my_spin_lock;
    KIRQL irql;
    KeInitializeSpinLock(&my_spin_lock);
    KeAcquireSpinLock(&my_spin_lock,&irql);
    // 在这里做要做的事情…
    KeReleaseSpinLock(&my_spin_lock,irql);
}
```

原因是 my_spin_lock 在堆栈中。每个线程来执行的时候都会重新初始化一个锁。只有所有的线程共用一个锁，锁才有意义。所以，锁一般不会定义成局部变量。可以使用静态变量、全局变量，或者分配在堆中（见前面的 1.2.1 内存的分配与释放一节）。请读者自己写出正确的方法。

LIST_ENTRY 有一系列的操作。这些操作并不需要使用者自己调用获取与释放锁。只需要为每个链表定义并初始化一个锁即可：

```
LIST_ENTRY      my_list_head;       // 链表头
KSPIN_LOCK  my_list_lock;       // 链表的锁

// 链表初始化函数
void MyFileInforInilt()
{
    InitializeListHead(&my_list_head);
    KeInitializeSpinLock(&my_list_lock);
}
```

链表一旦完成了初始化，之后的可以采用一系列加锁的操作来代替普通的操作。比如插入一个节点，普通的操作代码如下：

```
InsertHeadList(&my_list_head, (PLIST_ENTRY)& my_file_infor);
```

换成加锁的操作方式如下：

```
ExInterlockedInsertHeadList(
    &my_list_head,
```

```
        (PLIST_ENTRY)& my_file_infor,
        &my_list_lock);
```

注意不同之处在于，增加了一个 KSPIN_LOCK 的指针作为参数。在 ExInterlockedInsertHeadList 中，会自动的使用这个 KSPIN_LOCK 进行加锁。类似的还有一个加锁的 Remove 函数，用来移除一个节点，调用如下：

```
    my_file_infor = ExInterlockedRemoveHeadList (
        &my_list_head,
        &my_list_lock);
```

这个函数从链表中移除第一个节点。并返回到 my_file_infor 中。

第三章 文件操作

在内核中不能调用用户层的 Win32 API 函数来操作文件。在这里必须改用一系列与之对应的内核函数。

3.1 使用 OBJECT_ATTRIBUTES

一般的想法是，打开文件应该传入这个文件的路径。但是实际上这个函数并不直接接受一个字符串。使用者必须首先填写一个 OBJECT_ATTRIBUTES 结构。在文档中并没有公开这个 OBJECT_ATTRIBUTES 结构。这个结构总是被 InitializeObjectAttributes 初始化。

下面专门说明 InitializeObjectAttributes。

```
VOID InitializeObjectAttributes(
    OUT POBJECT_ATTRIBUTES InitializedAttributes,
    IN PUNICODE_STRING ObjectName,
    IN ULONG Attributes,
    IN HANDLE RootDirectory,
    IN PSECURITY_DESCRIPTOR SecurityDescriptor);
```

读者需要注意的以下的几点：InitializedAttributes 是要初始化的 OBJECT_ATTRIBUTES 结构的指针。ObjectName 则是对象名字字符串。也就是前文所描述的文件的路径（如果要打开的对象是一个文件的话）。

Attributes 则只需要填写 OBJ_CASE_INSENSITIVE| OBJ_KERNEL_HANDLE 即可（如果读者是想要方便的简洁的打开一个文件的话）。OBJ_CASE_INSENSITIVE 意味着名字字符串是不区分大小写的。由于 Windows 的文件系统本来就不区分字母大小写，所以笔者并没有尝试过如果不设置这个标记会有什么后果。OBJ_KERNEL_HANDLE 表明打开的文件句柄一个“内核句柄”。内核文件句柄比应用层句柄使用更方便，可以不受线程和进程的限制。在任何线程中都可以读写。同时打开内核文件句柄不需要顾及当前进程是否有权限访问该文件的问题（如果是有安全权限限制的文件系统）。如果不使用内核句柄，则有时不得不填写后面的的 SecurityDescriptor 参数。

RootDirectory 用于相对打开的情况。目前省略。请读者传入 NULL 即可。

SecurityDescriptor 用于设置安全描述符。由于笔者总是打开内核句柄，所以很少设置这个参数。

3.2 打开和关闭文件

下面的函数用于打开一个文件：

```
NTSTATUS ZwCreateFile(
```

```
    OUT PHANDLE FileHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttribute,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    IN PLARGE_INTEGER AllocationSize OPTIONAL,
    IN ULONG FileAttributes,
    IN ULONG ShareAccess,
    IN ULONG CreateDisposition,
    IN ULONG createOptions,
    IN PVOID EaBuffer OPTIONAL,
    IN ULONG EaLength);
```

这个函数的参数异常复杂。下面逐个的说明如下：

FileHandle：是一个句柄的指针。如果这个函数调用返回成成功（STATUS_SUCCESS），那就么打开的文件句柄就返回在这个地址内。

DesiredAccess：申请的权限。如果打开写文件内容，请使用 FILE_WRITE_DATA。如果需要读文件内容，请使用 FILE_READ_DATA。如果需要删除文件或者把文件改名，请使用 DELETE。如果想设置文件属性，请使用 FILE_WRITE_ATTRIBUTES。反之，读文件属性则使用 FILE_READ_ATTRIBUTES。这些条件可以用|（位或）来组合。有两个宏分别组合了常用的读权限和常用的写权限。分别为 GENERIC_READ 和 GENERIC_WRITE。此外还有一个宏代表全部权限，是 GENERIC_ALL。此外，如果想同步的打开文件，请加上 SYNCHRONIZE。同步打开文件详见后面对 CreateOptions 的说明。

ObjectAttribute：对象描述。见前一小节。

IoStatusBlock 也是一个结构。这个结构在内核开发中经常使用。它往往用于表示一个操作的结果。这个结构在文档中是公开的，如下：

```
typedef struct _IO_STATUS_BLOCK {
    union {
        NTSTATUS Status;
        PVOID Pointer;
    };
    ULONG_PTR Information;
} IO_STATUS_BLOCK, *PIO_STATUS_BLOCK;
```

实际编程中很少用到 Pointer。一般的说，返回的结果在 Status 中。成功则为 STATUS_SUCCESS。否则则是一个错误码。进一步的信息在 Information 中。不同的情况下返回的 Information 的信息意义不同。针对 ZwCreateFile 调用的情况，Information 的返回值有以下几种可能：

- FILE_CREATED：文件被成功的新建了。
- FILE_OPENED：文件被打开了。
- FILE_OVERWRITTEN：文件被覆盖了。
- FILE_SUPERSEDED：文件被替代了。
- FILE_EXISTS：文件已存在。（因而打开失败了）。
- FILE_DOES_NOT_EXIST：文件不存在。（因而打开失败了）。

这些返回值和打开文件的意图有关(有时希望打开已存在的文件，有时则希望建立新的文件等等。这些意图在本小节稍后的内容中详细说明。

ZwCreateFile 的下一个参数是 AllocationSize。这个参数很少使用，请设置为 NULL。

再接下来的一个参数为 FileAttributes。这个参数控制新建立的文件的属性。一般的说，设置为 FILE_ATTRIBUTE_NORMAL 即可。在实际编程中，笔者没有尝试过其他的值。

ShareAccess 是一个非常容易被人误解的参数。实际上，这是在本代码打开这个文件的时候，允许别的代码同时打开这个文件所持有的权限。所以称为共享访问。一共有三种共享标记可以设置：FILE_SHARE_READ、FILE_SHARE_WRITE、FILE_SHARE_DELETE。这三个标记可以用|（位或）来组合。举例如下：如果本次打开只使用了 FILE_SHARE_READ，那么这个文件在本次打开之后，关闭之前，别次打开试图以读权限打开，则被允许，可以成功打开。如果别次打开试图以写权限打开，则一定失败。返回共享冲突。

同时，如果本次打开只只用了 FILE_SHARE_READ，而之前这个文件已经被另一次打开用写权限打开着。那么本次打开一定失败，返回共享冲突。其中的逻辑关系貌似比较复杂，读者应耐心理解。

CreateDisposition 参数说明了这次打开的意图。可能的选择如下（请注意这些选择不能组合）：

- FILE_CREATE：新建文件。如果文件已经存在，则这个请求失败。
- FILE_OPEN：打开文件。如果文件不存在，则请求失败。
- FILE_OPEN_IF：打开或新建。如果文件存在，则打开。如果不存在，则失败。
- FILE_OVERWRITE：覆盖。如果文件存在，则打开并覆盖其内容。如果文件不存在，这个请求返回失败。
- FILE_OVERWRITE_IF：新建或覆盖。如果要打开的文件已存在，则打开它，并覆盖其内存。如果不存在，则简单的新建新文件。
- FILE_SUPERSEDE：新建或取代。如果要打开的文件已存在。则生成一个新文件替代之。如果不存在，则简单的生成新文件。

请联系上面的 IoStatusBlock 参数中的 Information 的说明。

最后一个重要的参数是 CreateOptions。在惯常的编程中，笔者使用 FILE_NON_DIRECTORY_FILE| FILE_SYNCHRONOUS_IO_NONALERT。此时文件被同步的打开。而且打开的是文件（而不是目录。创建目录请用 FILE_ DIRECTORY_FILE)。所谓同步的打开的意义在于，以后每次操作文件的时候，比如写入文件，调用 ZwWriteFile，在 ZwWriteFile 返回时，文件写操作已经得到了完成。而不会有返回 STATUS_PENDING（未决）的情况。在非同步文件的情况下，返回未决是常见的。此时文件请求没有完成，使用者需要等待事件来等待请求的完成。当然，好处是使用者可以先去做别的事情。

要同步打开，前面的 DesiredAccess 必须含有 SYNCHRONIZE。

此外还有一些其他的情况。比如不通过缓冲操作文件。希望每次读写文件都是直接往磁盘上操作的。此时 CreateOptions 中应该带标记 FILE_NO_INTERMEDIATE_BUFFERING。带了这个标记后，请注意操作文件每次读写都必须以磁盘扇区大小(最常见的是 512 字节)对齐。否则会返回错误。

这个函数是如此的繁琐，以至于再多的文档也不如一个可以利用的例子。早期笔者调用这个函数往往因为参数设置不对而导致打开失败。非常渴望找到一个实际可以使用的参数的范例。下面举例如下：

```
// 要返回的文件句柄
HANDLE file_handle = NULL;
```

```
    // 返回值
    NTSTATUS status;
    // 首先初始化含有文件路径的 OBJECT_ATTRIBUTES
    OBJECT_ATTRIBUTES object_attributes;
    UNICODE_STRING ufile_name = RTL_CONST_STRING(L"\\??\\C:\\a.dat");
    InitializeObjectAttributes(
        &object_attributes,
        &ufile_name,
        OBJ_CASE_INSENSITIVE|OBJ_KERNEL_HANDLE,
        NULL,
        NULL);
    // 以 OPEN_IF 方式打开文件。
    status = ZwCreateFile(
        &file_handle,
        GENERIC_READ | GENERIC_WRITE,
        &object_attributes,
        &io_status,
        NULL,
        FILE_ATTRIBUTE_NORMAL,
        FILE_SHARE_READ,
        FILE_OPEN_IF,
        FILE_NON_DIRECTORY_FILE |
        FILE_RANDOM_ACCESS |
        FILE_SYNCHRONOUS_IO_NONALERT,
        NULL,
        0);
```

值得注意的是路径的写法。并不是像应用层一样直接写“C:\\a.dat”。而是写成了“\\??\\C:\\a.dat”。这是因为 ZwCreateFile 使用的是对象路径。“C:”是一个符号链接对象。符号链接对象一般都在“\\??\\”路径下。

这种文件句柄的关闭非常简单。调用 ZwClose 即可。内核句柄的关闭不需要和打开在同一进程中。示例如下：

```
    ZwClose(file_handle);
```

3.3 文件的读写操作

打开文件之后，最重要的操作是对文件的读写。读与写的方法是对称的。只是参数输入与输出的方向不同。读取文件内容一般用 ZwReadFile，写文件一般使用 ZwWriteFile。

```
    NTSTATUS
    ZwReadFile(
        IN HANDLE  FileHandle,
        IN HANDLE  Event  OPTIONAL,
```

```
    IN PIO_APC_ROUTINE  ApcRoutine  OPTIONAL,
    IN PVOID  ApcContext  OPTIONAL,
    OUT PIO_STATUS_BLOCK  IoStatusBlock,
    OUT PVOID  Buffer,
    IN ULONG  Length,
    IN PLARGE_INTEGER  ByteOffset  OPTIONAL,
    IN PULONG  Key  OPTIONAL);
```

FileHandle：是前面 ZwCreateFile 成功后所得到的 FileHandle。如果是内核句柄，ZwReadFile 和 ZwCreateFile 并不需要在同一个进程中。句柄是各进程通用的。

Event ：一个事件。用于异步完成读时。下面的举例始终用同步读，所以忽略这个参数。请始终填写 NULL。

ApcRoutine Apc：回调例程。用于异步完成读时。下面的举例始终用同步读，所以忽略这个参数。请始终填写 NULL。

IoStatusBlock：返回结果状态。同 ZwCreateFile 中的同名参数。

Buffer：缓冲区。如果读文件的内容成功，则内容被被读到这个缓冲里。

Length：描述缓冲区的长度。这个长度也就是试图读取文件的长度。

ByteOffset：要读取的文件的偏移量。也就是要读取的内容在文件中的位置。一般的说，不要设置为 NULL。文件句柄不一定支持直接读取当前偏移。

Key：读取文件时用的一种附加信息，一般不使用。设置 NULL。

返回值：成功的返回值是 STATUS_SUCCESS。只要读取到任意多个字节（不管是否符合输入的 Length 的要求），返回值都是 STATUS_SUCCESS。即使试图读取的长度范围超出了文件本来的大小。但是，如果仅读取文件长度之外的部分，则返回 STATUS_END_OF_FILE。

ZwWriteFile 的参数与 ZwReadFile 完全相同。当然，除了读写文件外，有的读者可能会问是否提供一个 ZwCopyFile 用来拷贝一个文件。这个要求未能被满足。如果有这个需求，这个函数必须自己来编写。下面是一个例子，用来拷贝一个文件。利用到了 ZwCreateFile，ZwReadFile 和 ZwWrite 这三个函数。不过作为本节的例子，只举出 ZwReadFile 和 ZwWriteFile 的部分：

```
NTSTATUS MyCopyFile(
    PUNICODE_STRING target_path,
    PUNICODE_STRING source_path)
{
    // 源和目标的文件句柄
    HANDLE target = NULL,source = NULL;
    // 用来拷贝的缓冲区
    PVOID buffer = NULL;
    LARGE_INTEGER offset = { 0 };
    IO_STATUS_BLOCK io_status = { 0 };

    do {
        // 这里请用前一小节说到的例子打开 target_path 和 source_path 所对应的
        // 句柄 target 和 source,并为 buffer 分配一个页面也就是 4k 的内存。
        … …
```

```
        // 然后用一个循环来读取文件。每次从源文件中读取 4k 内容，然后往
        // 目标文件中写入 4k，直到拷贝结束为止。
         while(1) {
            length = 4*1024;        // 每次读取 4k。
            // 读取旧文件。注意 status。
            status = ZwReadFile (
                    source,NULL,NULL,NULL,
                    &my_io_status,buffer, length,&offset,
                    NULL);
            if(!NT_SUCCESS(status))
            {
                // 如果状态为 STATUS_END_OF_FILE，则说明文件
                // 的拷贝已经成功的结束了。
                if(status == STATUS_END_OF_FILE)
                    status = STATUS_SUCCESS;
                break;
            }
            // 获得实际读取到的长度。
            length = IoStatus.Information;

            // 现在读取了内容。读出的长度为 length.那么我写入
            // 的长度也应该是 length。写入必须成功。如果失败，
            // 则返回错误。
            status = ZwWriteFile(
                target,NULL,NULL,NULL,
                &my_io_status,
                buffer,length,&offset,
                NULL);
            if(!NT_SUCCESS(status))
                break;

            // offset 移动，然后继续。直到出现 STATUS_END_OF_FILE
            // 的时候才结束。
            offset.QuadPart += length;
        }
    } while(0);

    // 在退出之前，释放资源，关闭所有的句柄。
    if(target != NULL)
        ZwClose(target);
    if(source != NULL)
        ZwClose(source);
    if(buffer != NULL)
        ExFreePool(buffer);
```

```
        return STATUS_SUCCESS;
    }
```

除了读写之外，文件还有很多的操作。比如删除、重新命名、枚举。这些操作将在后面实例中用到时，再详细讲解。

第四章 操作注册表

4.1 注册键的打开操作

和在应用程序中编程的方式类似，注册表是一个巨大的树形结构。操作一般都是打开某个子键。子键下有若干个值可以获得。每一个值有一个名字。值有不同的类型。一般需要查询才能获得其类型。

子键一般用一个路径来表示。和应用程序编程的一点重大不同是这个路径的写法不一样。一般应用编程中需要提供一个根子键的句柄。而驱动中则全部用路径表示。相应的有一张表表示如下：

应用编程中对应的子键	驱动编程中的路径写法
HKEY_LOCAL_MACHINE	\Registry\Machine
HKEY_USERS	\Registry\User
HKEY_CLASSES_ROOT	没有对应的路径
HKEY_CURRENT_USER	没有简单的对应路径，但是可以求得

实际上应用程序和驱动程序很大的一个不同在于应用程序总是由某个“当前用户”启动的。因此可以直接读取 HKEY_CLASSES_ROOT 和 HKEY_CURRENT_USER。而驱动程序和用户无关，所以直接去打开 HKEY_CURRENT_USER 也就不符合逻辑了。

打开注册表键使用函数 ZwOpenKey。新建或者打开则使用 ZwCreateKey。一般在驱动编程中，使用 ZwOpenKey 的情况比较多见。下面以此为例讲解。ZwOpenKey 的原型如下：

```
NTSTATUS
    ZwOpenKey(
        OUT PHANDLE  KeyHandle,
        IN ACCESS_MASK  DesiredAccess,
        IN POBJECT_ATTRIBUTES  ObjectAttributes
    );
```

这个函数和 ZwCreateFile 是类似的。它并不接受直接传入一个字符串来表示一个子键。而是要求输入一个 OBJECT_ATTRIBUTES 的指针。如何初始化一个 OBJECT_ATTRIBUTES 请参考前面的讲解 ZwCreateFile 的章节。

DesiredAccess 支持一系列的组合权限。可以是下表中所有权限的任何组合：

- KEY_QUERY_VALUE：读取键下的值。
- KEY_SET_VALUE：设置键下的值。
- KEY_CREATE_SUB_KEY：生成子键。
- KEY_ENUMERATE_SUB_KEYS：枚举子键。

不过实际上可以用 KEY_READ 来做为通用的读权限组合。这是一个组合宏。此外对应的

有 KEY_WRITE。如果需要获得全部的权限，可以使用 KEY_ALL_ACCESS。

下面是一个例子，这个例子非常的有实用价值。它读取注册表中保存的 Windows 系统目录（指 Windows 目录）的位置。不过这里只涉及打开子键。并不读取值。读取具体的值在后面的小节中再完成。

Windows 目录的位置被称为 SystemRoot，这一值保存在注册表中，路径是“HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion”。当然，请注意注意在驱动编程中的写法有所不同。下面的代码初始化一个 OBJECT_ATTRIBUTES。

```
    HANDLE my_key = NULL;
    NTSTATUS status;

    // 定义要获取的路径
    UNICODE_STRING my_key_path =
        RTL_CONSTANT_STRING(
            L"        \\        Registry\\Machine\\SOFTWARE\\Microsoft\\Windows
NT\\CurrentVersion");
    OBJECT_ATTRIBUTE my_obj_attr = { 0 };

    // 初始化 OBJECT_ATTRIBUTE
    InitializeObjectAttributes(
        &my_obj_attr,
        &my_key_path,
        OBJ_CASE_INSENSITIVE,
        NULL,
        NULL);
    // 接下来是打开 Key
    status = ZwOpenKey(&my_key,KEY_READ,&my_obj_attr);
    if(!NT_SUCCESS(status))
    {
        // 失败处理
        ……
    }
```

上面的代码得到了 my_key。子键已经打开。然后的步骤是读取下面的 SystemRoot 值。这在后面一个小节中讲述。

4.2 注册值的读

一般使用 ZwQueryValueKey 来读取注册表中键的值。要注意的是注册表中的值可能有多种数据类型。而且长度也是没有定数的。为此，在读取过程中，就可能要面对很多种可能的情况。ZwQueryValueKey 这个函数的原型如下：

```
    NTSTATUS ZwQueryValueKey(
        IN HANDLE  KeyHandle,
```

```
    IN PUNICODE_STRING  ValueName,
    IN KEY_VALUE_INFORMATION_CLASS  KeyValueInformationClass,
    OUT PVOID  KeyValueInformation,
    IN ULONG  Length,
    OUT PULONG  ResultLength
    );
```

KeyHandle：这是用 ZwCreateKey 或者 ZwOpenKey 所打开的一个注册表键句柄。

ValueName：要读取的值的名字。

KeyValueInformationClass：本次查询所需要查询的信息类型。这有如下的三种可能。

- KeyValueBasicInformation：获得基础信息，包含值名和类型。
- KeyValueFullInformation：获得完整信息。包含值名、类型和值的数据。
- KeyValuePartialInformation：获得局部信息。包含类型和值数据。

很容易看出实际上名字是已知的，获得基础信息是多此一举。同样获得完整信息也是浪费内存空间。因为调用 ZwQueryValueKey 的目的是为了得到类型和值数据。因此使用 KeyValuePartialInformation 最常见。当采用 KeyValuePartialInformation 的时候，一个类型为 KEY_VALUE_PARTIAL_INFORMATION 的结构将被返回到参数 KeyValueInformation 所指向的内存中。

KeyValueInformation：当 KeyValueInformationClass 被设置为 KeyValuePartialInformation 时，KEY_VALUE_PARTIAL_INFORMATION 结构将被返回到这个指针所指内存中。下面是结构 KEY_VALUE_PARTIAL_INFORMATION 的原型。

```
typedef struct _KEY_VALUE_PARTIAL_INFORMATION {
    ULONG  TitleIndex;          // 请忽略这个成员
    ULONG  Type;                // 数据类型
    ULONG  DataLength;          // 数据长度
    UCHAR  Data[1];             // 可变长度的数据
}KEY_VALUE_PARTIAL_INFORMATION,*PKEY_VALUE_PARTIAL_INFORMATIO;
```

上面的数据类型 Type 有很多种可能，但是最常见的几种如下：

- REG_BINARY：十六进制数据。
- REG_DWORD：四字节整数。
- REG_SZ：以空结束的 Unicode 字符串。

Length：用户传入的输出空间 KeyValueInformation 的长度。

ResultLength：返回实际需要的长度。

返回值：如果说实际需要的长度比 Length 要大，那么返回 STATUS_BUFFER_OVERFLOW 或者是 STATUS_BUFFER_TOO_SMALL。如果成功读出了全部数据，那么返回 STATUS_SUCCESS。其他的情况，返回一个错误码。

下面请读者考虑如何把上一小节的函数写完整。这其中比较常见的一个问题是在读取注册表键下的值之前，往往不知道这个值有多长。所以有些比较偷懒的程序员总是定义一个足够的大小的空间（比如 512 字节）。这样的坏处是浪费内存（一般都是在堆栈中定义，而内核编程中堆栈空间被耗尽又是另一个常见的蓝屏问题）。此外也无法避免值实际上大于该长度的情况。为此应该耐心的首先获取长度，然后不足时再动态分配内存进行读取。下面是示例代码：

```
// 要读取的值的名字
UNICODE_STRING my_key_name =
    RTL_CONSTANT_STRING(L”SystemRoot”);
// 用来试探大小的 key_infor
KEY_VALUE_PARTIAL_INFORMATION key_infor;
// 最后实际用到的 key_infor 指针。内存分配在堆中
PKEY_VALUE_PARTIAL_INFORMATION ac_key_infor;
ULONG ac_length;
……
// 前面已经打开了句柄 my_key，下面如此来读取值：
status = ZwQueryValueKey(
    my_key,
    &my_key_name,
    KeyValuePartialInformation,
    &key_infor,
    sizeof(KEY_VALUE_PARTIAL_INFORMATION),
    &ac_length);
if(!NT_SUCCESS(status) &&
    status != STATUS_BUFFER_OVERFLOW &&
    status != STATUS_BUFFER_TOO_SMALL)
{
    // 错误处理
    …
}
// 如果没失败，那么分配足够的空间，再次读取
ac_key_infor = (PKEY_VALUE_PARTIAL_INFORMATION)
    ExAllocatePoolWithTag(NonpagedPool,ac_length ,MEM_TAG);
if(ac_key_infor == NULL)
{
    stauts = STATUS_INSUFFICIENT_RESOURCES;
    // 错误处理
    …
}
status = ZwQueryValueKey(
    my_key,
    &my_key_name,
    KeyValuePartialInformation,
    ac_key_infor,
    ac_length,
    &ac_length);
// 到此为止，如果 status 为 STATUS_SUCCESS,则要读取的数据已经
// 在 ac_key_infor->Data 中。请利用前面学到的知识，转换为
// UNICODE_STRING
```

```
……
```

4.3 注册值的写

实际上注册表的写入比读取要简单。因为这省略了一个尝试数据的大小的过程。直接将数据写入即可。写入值一般使用函数 ZwSetValueKey 。这个函数的原型如下：

```
NTSTATUS ZwSetValueKey(
    IN HANDLE  KeyHandle,
    IN PUNICODE_STRING  ValueName,
    IN ULONG  TitleIndex  OPTIONAL,
    IN ULONG  Type,
    IN PVOID  Data,
    IN ULONG  DataSize
);
```

其中的 TileIndex 参数请始终填入 0。

KeyHandle、ValueName、Type 这三个参数和 ZwQueryValueKey 中对应的参数相同。不同的是 Data 和 DataSize。Data 是要写入的数据的开始地址，而 DataSize 是要写入的数据的长度。由于 Data 是一个空指针，因此，Data 可以指向任何数据。也就是说，不管 Type 是什么，都可以在 Data 中填写相应的数据写入。

ZwSetValueKey 的时候，并不需要该 Value 已经存在。如果该 Value 已经存在，那么其值会被这次写入覆盖。如果不存在，则会新建一个。下面的例子写入一个名字为“Test”，而且值为“My Test Value”的字符串值。假设 my_key 是一个已经打开的子键的句柄。

```
UNICODE_STRING name = RTL_CONSTANT_STRING(L”Test”);
PWCHAR value = { L”My Test Value” };
…
// 写入数据。数据长度之所以要将字符串长度加上 1，是为了把最后一个空结束符
// 写入。我不确定如果不写入空结束符会不会有错，有兴趣的读者请自己测试一下。
status = ZwSetValueKey(my_key,
        &name,0,REG_SZ,value,(wcslen(value)+1)*sizeof(WCHAR));
if(!NT_SUCCESS(status))
{
    // 错误处理
    ……
}
```

关于注册表的操作就介绍到这里了。如果有进一步的需求，建议读者阅读 WDK 相关的文档。

第五章 时间与定时器

5.1 获得当前滴答数

在编程中，获得当前的系统日期和时间，或者是获得一个从启动开始的毫秒数，是很常见的需求。获得系统日期和时间往往是为了写日志。获得启动毫秒数很适合用来做一个随机数的种子。有时也使用时间相关的函数来寻找程序的性能瓶颈。

熟悉 Win32 应用程序开发的读者会知道有一个函数 GetTickCount(),这个函数返回系统自启动之后经历的毫秒数。在驱动开发中有一个对应的函数 KeQueryTickCount(),这个函数的原型如下：

```
VOID
    KeQueryTickCount(
        OUT PLARGE_INTEGER  TickCount
    );
```

遗憾的是，被返回到 TickCount 中的并不是一个简单的毫秒数。这是一个“滴答”数。但是一个“滴答”到底为多长的时间，在不同的硬件环境下可能有所不同。为此，必须结合另一个函数使用。下面这个函数获得一个“滴答”的具体的 100 纳秒数。

```
ULONG
    KeQueryTimeIncrement(
        );
```

得知以上的关系之后，下面的代码可以求得实际的毫秒数：

```
void MyGetTickCount (PULONG msec)
{
    LARGE_INTEGER tick_count;
    ULONG myinc = KeQueryTimeIncrement();
    KeQueryTickCount(&tick_count);
    tick_count.QuadPart *= myinc;
    tick_count.QuadPart /=  10000;
    *msec = tick_count.LowPart;
}
```

这不是一个简单的过程。不过所幸的是，现在有代码可以拷贝了。

5.2 获得当前系统时间

接下来的一个需求是得到当前的可以供人类理解的时间。包括年、月、日、时、分、秒这些要素。在驱动中不能使用诸如 CTime 之类的 MFC 类。不过与之对应的有 TIME_FIELDS，这个结构中含有对应的时间要素。

KeQuerySystemTime()得到当前时间。但是得到的并不是当地时间，而是一个格林威治时间。之后请使用 ExSystemTimeToLocalTime()转换可以当地时间。这两个函数的原型如下：

```
VOID
KeQuerySystemTime(
    OUT PLARGE_INTEGER  CurrentTime
    );
VOID
ExSystemTimeToLocalTime(
    IN PLARGE_INTEGER  SystemTime,
    OUT PLARGE_INTEGER  LocalTime
    );
```

这两个函数使用的“时间”都是长长整型数据结构。这不是人类可以阅读的。必须通过函数 RtlTimeToTimeFields 转换为 TIME_FIELDS。这个函数原型如下：

```
VOID
RtlTimeToTimeFields(
    IN PLARGE_INTEGER  Time,
    IN PTIME_FIELDS  TimeFields
);
```

读者需要实际应用一下来加深印象。下面写出一个函数：这个函数返回一个字符串。这个字符串写出当前的年、月、日、时、分、秒，这些数字之间用“-”号隔开。这是一个很有用的函数。而且同时用到上面三个函数，此外，请读者回忆前面关于字符串的打印的相关章节。

```
PWCHAR MyCurTimeStr()
{
    LARGE_INTEGER snow,now;
    TIME_FIELDS now_fields;
    static WCHAR time_str[32] = { 0 };
    // 获得标准时间
    KeQuerySystemTime(&snow);
    // 转换为当地时间
    ExSystemTimeToLocalTime(&snow,&now);
    // 转换为人类可以理解的时间要素
    RtlTimeToTimeFields(&now,&now_fields);
```

```
    // 打印到字符串中
    RtlStringCchPrintfW(
        time_str,
        32*2,
        L"%4d-%2d-%2d %2d-%2d-%2d",
        now_fields.Year,now_fields.Month,now_fields.Day,
        now_fields.Hour,now_fields.Minute,now_fields.Second);
    return time_str;
}
```

请注意 time_str 是静态变量。这使得这个函数不具备多线程安全性。请读者考虑一下，如何保证多个线程同时调用这个函数的时候，不出现冲突？

5.3 使用定时器

使用过 Windows 应用程序编程的读者的读者一定对 SetTimer()映像尤深。当需要定时执行任务的时候，SetTimer()变得非常重要。这个功能在驱动开发中可以通过一些不同的替代方法来实现。比较经典的对应是 KeSetTimer()，这个函数的原型如下：

```
BOOLEAN
      KeSetTimer(
           IN PKTIMER  Timer,                  // 定时器
            IN LARGE_INTEGER  DueTime,     // 延后执行的时间
            IN PKDPC  Dpc  OPTIONAL         // 要执行的回调函数结构
        );
```

其中的定时器 Timer 和要执行的回调函数结构 Dpc 都必须先初始化。其中 Timer 的初始化比较简单。下面的代码可以初始化一个 Timer：

```
KTIMER my_timer;
KeInitializeTimer(&my_timer);
```

Dpc 的初始化比较麻烦。这是因为需要提供一个回调函数。初始化 Dpc 的函数原型如下：

```
VOID
      KeInitializeDpc(
           IN PRKDPC  Dpc,
            IN PKDEFERRED_ROUTINE  DeferredRoutine,
            IN PVOID  DeferredContext
        );
```

PKDEFERRED_ROUTINE 这个函数指针类型所对应的函数的类型实际上是这样的：

```
VOID
```

```
CustomDpc(
        IN struct _KDPC  *Dpc,
        IN PVOID  DeferredContext,
        IN PVOID  SystemArgument1,
        IN PVOID  SystemArgument2
    );
```

读者需要关心的只是 DeferredContext。这个参数是 KeInitializeDpc 调用时传入的参数。用来提供给 CustomDpc 被调用的时候，让用户传入一些参数。
至于后面的 SystemArgument1 和 SystemArgument2 则请不要理会。Dpc 是回调这个函数的 KDPC 结构。

请注意这是一个“延时执行”的过程。而不是一个定时执行的过程。因此每次执行了之后，下次就不会再被调用了。如果想要定时反复执行，就必须在每次 CustomDpc 函数被调用的时候，再次调用 KeSetTimer，来保证下次还可以执行。

值得注意的是，CustomDpc 将运行在 APC 中断级。因此并不是所有的事情都可以做（在调用任何内核系统函数的时候，请注意 WDK 说明文档中标明的中断级要求。）
这些事情非常的烦恼，因此要完全实现定时器的功能，需要自己封装一些东西。下面的结构封装了全部需要的信息：

```
// 内部时钟结构
typedef struct MY_TIMER_
{
    KDPC dpc;
    KTIMER timer;
    PKDEFERRED_ROUTINE func;
    PVOID private_context;
} MY_TIMER,*PMY_TIMER;

// 初始化这个结构：
void MyTimerInit(PMY_TIMER timer, PKDEFERRED_ROUTINE func)
{
    // 请注意，我把回调函数的上下文参数设置为 timer,为什么要
    // 这样做呢？
    KeInitializeDpc(&timer->dpc,sf_my_dpc_routine,timer);
    timer->func = func;
    KeInitializeTimer(&timer->timer);
    return (wd_timer_h)timer;
}

// 让这个结构中的回调函数在 n 毫秒之后开始运行：
BOOLEAN MyTimerSet(PMY_TIMER timer,ULONG msec,PVOID context)
{
    LARGE_INTEGER due;
    // 注意时间单位的转换。这里 msec 是毫秒。
```

```
    due.QuadPart = -10000*msec;
    // 用户私有上下文。
    timer->private_context = context;
    return KeSetTimer(&timer->timer,due,&mytimer->dpc);
};

// 停止执行
VOID MyTimerDestroy(PMY_TIMER timer)
{
    KeCancelTimer(&mytimer->timer);
};
```

使用结构 PMY_TIMER 已经比结合使用 KDPC 和 KTIMER 简便许多。但是还是有一些要注意的地方。真正的 OnTimer 回调函数中，要获得上下文，必须要从 timer->private_context 中获得。此外，OnTimer 中还有必要再次调用 MyTimerSet()，来保证下次依然得到执行。

```
VOID
MyOnTimer (
        IN struct _KDPC  *Dpc,
        IN PVOID  DeferredContext,
        IN PVOID  SystemArgument1,
        IN PVOID  SystemArgument2
    )
{
    // 这里传入的上下文是 timer 结构，用来下次再启动延时调用
    PMY_TIMER timer = (PMY_TIMER)DeferredContext;
    // 获得用户上下文
    PVOID my_context = timer->private_context;

    // 在这里做 OnTimer 中要做的事情
    ……

    // 再次调用。这里假设每 1 秒执行一次
    MyTimerSet(timer,1000,my_context);
};
```

关于定时器就介绍到这里了。

第六章 内核线程

6.1 使用线程

有时候需要使用线程来完成一个或者一组任务。这些任务可能耗时过长，而开发者又不想让当前系统停止下来等待。在驱动中停止等待很容易使整个系统陷入“停顿”，最后可能只能重启电脑。但一个单独的线程长期等待，还不至于对系统造成致命的影响。另一些任务是希望长期、不断的执行，比如不断写入日志。为此启动一个特殊的线程来执行它们是最好的方法。

在驱动中生成的线程一般是系统线程。系统线程所在的进程名为“System”。用到的内核 API 函数原型如下：

```
NTSTATUS
    PsCreateSystemThread(
        OUT PHANDLE  ThreadHandle,
        IN ULONG  DesiredAccess,
        IN POBJECT_ATTRIBUTES  ObjectAttributes  OPTIONAL,
        IN HANDLE  ProcessHandle  OPTIONAL,
        OUT PCLIENT_ID  ClientId  OPTIONAL,
        IN PKSTART_ROUTINE  StartRoutine,
        IN PVOID  StartContext);
```

这个函数的参数也很多。不过作者本人的使用经验如下：ThreadHandle 用来返回句柄。放入一个句柄指针即可。DesiredAccess 总是填写 0。后面三个参数都填写 NULL。最后的两个参数一个用于改线程启动的时候执行的函数。一个用于传入该函数的参数。

下面要关心的就是那个启动函数的原型。这个原型比起定时器回调函数倒是异常的简单，没有任何多余的东西：

```
VOID CustomThreadProc(IN PVOID context)
```

可以传入一个参数，就是那个 context。context 就是 PsCreateSystemThread 中的 StartContext 。值得注意的是，线程的结束应该在线程中自己调用 PsTerminateSystemThread 来完成。此外得到的句柄也必须要用 ZwClose 来关闭。但是请注意：关闭句柄并不结束线程。

下面举一个例子。这个例子传递一个字符串指针到一个线程中打印一下。然后结束该线程。当然打印字符串这种事情没有必要单独开一个线程来做。这里只是一个简单的示例。请注意，这个代码中有一个隐藏的错误，请读者指出这个错误是什么：

```
// 我的线程函数。传入一个参数，这个参数是一个字符串。
```

```
VOID MyThreadProc(PVOID context)
{
    PUNICODE_STRING str = (PUNICODE_STRING)context;
    // 打印字符串
    KdPrint((“PrintInMyThread:%wZ\r\n”,str));
    // 结束自己。
    PsTerminateSystemThread(STATUS_SUCCESS);
}

VOID MyFunction()
{
    UNICODE_STRING str = RTL_CONSTANT_STRING(L“Hello!”);
    HANDLE thread = NULL;
    NTSTATUS status;

    status = PsCreateSystemThread(
        &thread,0L,NULL,NULL,NULL,MyThreadProc,(PVOID)&str);
    if(!NT_SUCCESS(status))
    {
        // 错误处理。
        …
    }
    // 如果成功了，可以继续做自己的事。之后得到的句柄要关闭
    ZwClose(thread);
}
```

以上错误之处在于：MyThreadProc 执行的时候，MyFunction 可能已经执行完毕了。执行完毕之后，堆栈中的 str 已经无效。此时再执行 KdPrint 去打印 str 一定会蓝屏。这也是一个非常隐蔽，但是非常容易犯下的错误。

合理的方法是是在堆中分配 str 的空间。或者 str 必须在全局空间中。请读者自己写出正确的方法。

但是读者会发现，以上的写法在正确的代码中也是常见的。原因是这样做的时候，在 PsCreateSystemThread 结束之后，开发者会在后面加上一个等待线程结束的语句。

这样就没有任何问题了，因为在这个线程结束之前，这个函数都不会执行完毕，所以栈内存空间不会失效。

这样做的目的一般不是为了让任务并发。而是为了利用线程上下文环境而做的特殊处理。比如防止重入等等。在后面的章节读者会学到这方面的技巧。

如何等待线程结束在后面 1.6.3“使用事件通知”中进一步的讲述。

6.2 在线程中睡眠

许多读者一定使用过 Sleep 函数。这能使程序停下一段时间。许多需要连续、长期执行，但是又不希望占太多 CPU 使用率的任务，可以在中间加入睡眠。这样能使 CPU 使用率大大降

低。即使睡眠的时间非常短（几十个毫秒）。

在驱动中也可以睡眠。使用到的内核函数的原型如下：

```
NTSTATUS
KeDelayExecutionThread(
    IN KPROCESSOR_MODE  WaitMode,
    IN BOOLEAN  Alertable,
    IN PLARGE_INTEGER  Interval);
```

这个函数的参数简单明了。WaitMode 请总是填写 KernelMode，因为现在是在内核编程中使用。Alertable 表示是否允许线程报警（用于重新唤醒）。但是目前没有必要用到这么高级的功能，请总是填写 FALSE。剩下的就是 Interval 了，表明要睡眠多久。

但是这个看似简单的参数说明起来却异常的复杂。为此作者建议读者使用下面简单的睡眠函数，这个函数可以指定睡眠多少毫秒，而没有必要自己去换算时间（这个函数中有睡眠时间的转换）：

```
#define DELAY_ONE_MICROSECOND   (-10)
#define DELAY_ONE_MILLISECOND   (DELAY_ONE_MICROSECOND*1000)
VOID MySleep(LONG msec)
{
    LARGE_INTEGER my_interval;
    my_interval.QuadPart = DELAY_ONE_MILLISECOND;
    my_interval.QuadPart *= msec;
    KeDelayExecutionThread(KernelMode,0,&my_interval);
}
```

当然要睡眠几秒也是可以的，1 毫秒为千分之一秒。所以乘以 1000 就可以表示秒数。

在一个线程中用循环进行睡眠，也可以实现一个自己的定时器。考虑前面说的定时器的缺点：中断级较高，有一些事情不能做。在线程中用循环睡眠，每次睡眠结束之后调用自己的回调函数，也可以起到类似的效果。而且系统线程执行中是 Passive 中断级。睡眠之后依然是这个中断级，所以不像前面提到的定时器那样有限制。

请读者自己写出用线程+睡眠来实现定时器的例子。

6.3 使用事件通知

一些读者可能熟悉“事件驱动”编程技术。但是这里的“事件”与之不同。内核中的事件是一个数据结构。这个结构的指针可以当作一个参数传入一个等待函数中。如果这个事件不被“设置”，则这个等待函数不会返回，这个线程被阻塞。如果这个事件被“设置”，则等待结束，可以继续下去。

这常常用于多个线程之间的同步。如果一个线程需要等待另一个线程完成某事后才能做某事，则可以使用事件等待。另一个线程完成后设置事件即可。

这个数据结构是 KEVENT。读者没有必要去了解其内部结构。这个结构总是用 KeInitlizeEvent 初始化。这个函数原型如下：

```
VOID
    KeInitializeEvent(
        IN PRKEVENT  Event,
        IN EVENT_TYPE  Type,
        IN BOOLEAN  State
    );
```

第一个参数是要初始化的事件。第二个参数是事件类型，这个详见于后面的解释。第三个参数是初始化状态。一般的说设置为 FALSE。也就是未设状态。这样等待者需要等待设置之后才能通过。

事件不需要销毁。

设置事件使用函数 KeSetEvent。这个函数原型如下：

```
LONG
    KeSetEvent(
        IN PRKEVENT  Event,
        IN KPRIORITY  Increment,
        IN BOOLEAN  Wait
    );
```

Event 是要设置的事件。Increment 用于提升优先权。目前设置为 0 即可。Wait 表示是否后面马上紧接着一个 KeWaitSingleObject 来等待这个事件。一般设置为 TRUE。（事件初始化之后，一般就要开始等待了。）

使用事件的简单代码如下：

```
// 定义一个事件
KEVENT event;
// 事件初始化
KeInitializeEvent(&event,SynchronizationEvent,TRUE);
……

// 事件初始化之后就可以使用了。在一个函数中，你可以等待某
// 个事件。如果这个事件没有被人设置，那就会阻塞在这里继续
// 等待。
KeWaitForSingleObject(&event,Executive,KernelMode,0,0);
……

// 这是另一个地方，有人设置这个事件。只要一设置这个事件，
// 前面等待的地方，将继续执行。
KeSetEvent(&event);
```

由于在 KeInitializeEvent 中使用了 SynchronizationEvent，导致这个事件成为所谓的“自动重设”事件。一个事件如果被设置，那么所有 KeWaitForSingleObject 等待这个事件的地方都会通过。如果要能继续重复使用这个时间，必须重设（Reset）这个事件。当

KeInitializeEvent 中第二个参数被设置为 NotificationEvent 的时候，这个事件必须要手动重设才能使用。手动重设使用函数 KeResetEvent。

```
LONG
    KeResetEvent(
        IN PRKEVENT  Event
    );
```

如果这个事件初始化的时候是 SynchronizationEvent 事件，那么只有一个线程的 KeWaitForSingleObject 可以通过。通过之后被自动重设。那么其他的线程就只能继续等待了。这可以起到一个同步作用。所以叫做同步事件。不能起到同步作用的是通知事件（NotificationEvent）。请注意不能用手工设置通知事件的方法来取代同步事件。请读者思考一下这是为什么。

回忆前面的 1.6.1 “使用线程”的最后的例子。在那里曾经有一个需求：就是等待线程中的函数 KdPrint 结束之后，外面生成线程的函数再返回。 这可以通过一个事件来实现：线程中打印结束之后，设置事件。外面的函数再返回。为了编码简单我使用了一个静态变量做事件。这种方法在线程同步中用得极多，请务必熟练掌握：

```
static KEVENT s_event;

// 我的线程函数。传入一个参数，这个参数是一个字符串。
VOID MyThreadProc(PVOID context)
{
    PUNICODE_STRING str = (PUNICODE_STRING)context;
    KdPrint((“PrintInMyThread:%wZ\r\n”,str));
    KeSetEvent(&s_event);       // 在这里设置事件。
    PsTerminateSystemThread(STATUS_SUCCESS);
}

// 生成线程的函数:
VOID MyFunction()
{
    UNICODE_STRING str = RTL_CONSTANT_STRING(L“Hello!”);
    HANDLE thread = NULL;
    NTSTATUS status;

    KeInitializeEvent(&event,SynchronizationEvent,TRUE);    // 初始化事件
    status = PsCreateSystemThread(
        &thread,0L,NULL,NULL,NULL,MyThreadProc,(PVOID)&str);
    if(!NT_SUCCESS(status))
    {
        // 错误处理。
        …
    }
```

```
        ZwClose(thread);
        // 等待事件结束再返回：
        KeWaitForSingleObject(&s_event,Executive,KernelMode,0,0);
    }
```

实际上等待线程结束并不一定要用事件。线程本身也可以当作一个事件来等待。但是这里为了演示事件的用法而使用了事件。以上的方法调用线程则不必担心 str 的内存空间会无效了。因为这个函数在线程执行完 KdPrint 之后才返回。缺点是这个函数不能起到并发执行的作用。

第七章 驱动与设备

7.1 驱动入口与驱动对象

驱动开发程序员所编写的驱动程序对应有一个结构。这个结构名为 DRIVER_OBJECT。对应一个“驱动程序”。下面的代码展示的是一个最简单的驱动程序。

```
#include <ntddk.h>
NTSTATUS
    DriverEntry (
        IN PDRIVER_OBJECT   DriverObject,
        IN PUNICODE_STRING  RegistryPath
    )
{
    NTSTATUS status = STATUS_UNSUCCESSFUL;
    return status;
}
```

函数 DriverEntry 是每个驱动程序中必须的。如同 Win32 应用程序里的 WinMain。DriverEntry 的第一个参数就是一个 DRIVER_OBJECT 的指针。这个 DRIVER_OBJECT 结构就对应当前编写的驱动程序。其内存是 Windows 系统已经分配的。

第二个参数 RegistryPath 是一个字符串。代表一个注册表子键。这个子键是专门分配给这个驱动程序使用的。用于保存驱动配置信息到注册表中。至于读写注册表的方法，请参照前面章节中的内容。

DriverEntry 的返回值决定这个驱动的加载是否成功。如果返回为 STATUS_SUCCESS，则驱动将成功加载。否则，驱动加载失败。

7.2 分发函数与卸载函数

DRIVER_OBJECT 中含有分发函数指针。这些函数用来处理发到这个驱动的各种请求。Windows 总是自己调用 DRIVER_OBJECT 下的分发函数来处理这些请求。所以编写一个驱动程序，本质就是自己编写这些处理请求的分发函数。

DRIVER_OBJECT 下的分发函数指针的个数为 IRP_MJ_MAXIMUM_FUNCTION。保存在一个数组中。下面的代码设置所有分发函数的地址为同一个函数：

```
NTSTATUS
    DriverEntry (
        IN PDRIVER_OBJECT   DriverObject,
```

```
        IN PUNICODE_STRING  RegistryPath
    )
{
    ULONG i;
    for(i=0;i<IRP_MJ_MAXIMUM_FUNCTION;++i)
    {
        DriverObject->MajorFunctions[i] = MyDispatchFunction;
    }
    …
}
```

这个设置固然不难。难的工作都在编写 MyDispatchFunction 这个函数上。因为所有的分发函数指针都指向这一个函数，那么这个函数当然要完成本驱动所有的功能。下面是这个函数的原型。这个原型是 Windows 驱动编程的规范，不能更改：

```
NTSTATUS MyDispatchFunction(PDEVICE_OBJECT device,PIRP irp)
{
    ……
}
```

这里出现了 DEVICE_OBJECT 和 IRP 这两大结构。前一个表示一个由本驱动生成的设备对象。后一个表示一个系统请求。也就是说，现在要处理的是：发给设备 device 的请求 irp。请完成这个处理吧。这两个结构在后面再进一步描述。

还有一个不放在分发函数数组中的函数，称为卸载函数也非常重要。如果存在这个函数，则该驱动程序可以动态卸载。在卸载时，该函数会被执行。该函数原型如下：

```
VOID MyDriverUnload(PDRIVER_OBJECT driver)
{
    ……
}
```

这个函数的地址设置到 DriverObject->DriverUnload 即可。

由于没有返回值，所以实际上在 DriverUnload 中，已经无法决定这个驱动能否卸载。只能做善后处理。

7.3 设备与符号链接

驱动程序和系统其他组件之间的交互是通过给设备发送或者接受发给设备的请求来交互的。换句话说，一个没有任何设备的驱动是不能按规范方式和系统交互的。当然也不会收到任何 IRP，分发函数也失去了意义。

但并不意味着这样的驱动程序不存在。如果一个驱动程序只是想写写日志文件、Hook 某些内核函数或者是做一些其他的小动作，也可以不生成任何设备，也不需要关心分发函数的设置。

如果驱动程序要和应用程序之间通信，则应该生成设备。此外还必须为设备生成应用程

序可以访问的符号链接。下面的驱动程序生成了一个设备，并设置了分发函数：

```
#include <ntifs.h>  // 之所以用 ntifs.h 而不是 ntddk.h 是因为我习惯开发文件
                    // 系统驱动，实际上目前对读者来说这两个头文件没区别。
NTSTATUS DriverEntry(
    PDRIVER_OBJECT driver,
    PUNICODE_STRING reg_path)
{
    NTSTATUS status;
    PDEVICE_OBJECT device;
    // 设备名
    UNICODE_STRING device_name =
        RTL_CONSTANT_STRING("\\Device\\MyCDO");
    // 符号链接名
    UNICODE_STRING symb_link =
        RTL_CONSTANT_STRING("\\DosDevices\\MyCDOSL");

    // 生成设备对象
    status = IoCreateDevice(
            driver,
            0,
            device_name,
            FILE_DEVICE_UNKNOWN,
            0,
            FALSE,
            &device);

    // 如果不成功，就返回。
    if(!NT_SUCCESS(status))
        return status;

    // 生成符号链接
    status = IoCreateSymbolicLink(
        &symb_link,
        &device_name);
    if(!NT_SUCCESS(status))
    {
        IoDeleteDevice(device);
        return status;
    }
    // 设备生成之后，打开初始化完成标记
    device->Flags &= ~DO_DEVICE_INITIALIZING;
    return status;
}
```

这个驱动成功加载之后，生成一个名叫“\Device\MyCDO”的设备。然后在给这个设备生成了一个符号链接名字叫做“\DosDevices\MyCDOSL”。应用层可以通过打开这个符号链接来打开设备。应用层可以调用 CreateFile 就像打开文件一样打开。只是路径应该是“"\\.\MyCDOSL”。前面的“\\.\”意味后面是一个符号链接名，而不是一个普通的文件。请注意，由于 C 语言中斜杠要双写，所以正确的写法应该是“\\\\.\\”。与应用层交互的例子在下一节“IRP 和 IO_STACK_LOCATION”中。

7.4 设备的生成安全性限制

上一节的例子只是简单的例子。很多情况下那些代码会不起作用。为了避免读者在实际编程中遇到哪些特殊情况的困绕，下面详细说明生成设备和符号链接需要注意的地方。生成设备的函数原型如下：

```
NTSTATUS
IoCreateDevice(
    IN PDRIVER_OBJECT  DriverObject,
    IN ULONG  DeviceExtensionSize,
    IN PUNICODE_STRING  DeviceName  OPTIONAL,
    IN DEVICE_TYPE  DeviceType,
    IN ULONG  DeviceCharacteristics,
    IN BOOLEAN  Exclusive,
    OUT PDEVICE_OBJECT  *DeviceObject
);
```

这个函数的参数也非常复杂。但是实际上需要注意的并不多。

第一个参数是生成这个设备的驱动对象。

第二个参数 DeviceExtensionSize 非常重要。由于分发函数中得到的总是设备的指针。当用户需要在每个设备上记录一些额外的信息（比如用于判断这个设备是哪个设备的信息、以及不同的实际设备所需要记录的实际信息，比如网卡上数据包的流量、过滤器所绑定真实设备指针等等），需要指定的设备扩展区内存的大小。如果 DeviceExtensionSize 设置为非 0，IoCreateDevice 会分配这个大小的内存在 DeviceObject->DeviceExtension 中。以后用户就可以从根据 DeviceObject-> DeviceExtension 来获得这些预先保存的信息。

DeviceName 如前例，是设备的名字。目前生成设备，请总是生成在\Device\目录下。所以前面写的名字是“\Device\MyCDO”。其他路径也是可以的，但是这在本书描述范围之外。

DeviceType 表示设备类型。目前的范例无所谓设备类型，所以填写 FILE_DEVICE_UNKNOWN 即可。

DeviceCharacteristics 目前请简单的填写 0 即可。

Exclusive 这个参数必须设置 FALSE。文档没有做任何解释。

最后生成的设备对象指针返回到 DeviceObject 中。

这种设备生成之后，必须有系统权限的用户才能打开（比如管理员）。所以如果编程者写了一个普通的用户态的应用程序去打开这个设备进行交互，那么很多情况下可以（用管理员登录的时候）。但是偶尔又不行（用普通用户登录的时候）。结果困绕很久。其实是权限问题。

为了保证交互的成功与安全性，应该用服务程序与之交互。

但是依然有时候必须用普通用户打开设备。为了这个目的，设备必须是对所有的用户开放的。此时不能用 IoCreateDevice。必须用 IoCreateDeviceSecure。这个函数的原型如下：

```
NTSTATUS
IoCreateDeviceSecure(
    IN PDRIVER_OBJECT  DriverObject,
    IN ULONG  DeviceExtensionSize,
    IN PUNICODE_STRING  DeviceName  OPTIONAL,
    IN DEVICE_TYPE  DeviceType,
    IN ULONG  DeviceCharacteristics,
    IN BOOLEAN  Exclusive,
    IN PCUNICODE_STRING  DefaultSDDLString,
    IN LPCGUID  DeviceClassGuid,
    OUT PDEVICE_OBJECT  *DeviceObject
)
```

这个函数增加了两个参数（其他的没变）。一个是 DefaultSDDLString。这个一个用于描述权限的字符串。描述这个字符串的格式需要大量的篇幅。但是没有这个必要。字符串“D:P(A;;GA;;;WD)”将满足“人人皆可以打开”的需求。

另一个参数是一个设备的 GUID。请随机手写一个 GUID。不要和其他设备的 GUID 冲突（不要复制粘贴即可）。

下面是例子：

```
// 随机手写一个 GUID
const GUID DECLSPEC_SELECTANY MYGUID_CLASS_MYCDO =
{0x26e0d1e0L, 0x8189, 0x12e0, {0x99,0x14, 0x08, 0x00, 0x22, 0x30, 0x19, 0x03}};
// 全用户可读写权限
UNICODE_STRING sddl =
    RLT_CONSTANT_STRING(L"D:P(A;;GA;;;WD)");
// 生成设备
status = IoCreateDeviceSecure( DriverObject,
                0,
                &device_name,
                FILE_DEVICE_UNKNOWN,
                FILE_DEVICE_SECURE_OPEN,
                FALSE,
                &sddl,
                (LPCGUID)&SFGUID_CLASS_MYCDO,
                &device);
```

使用这个函数的时候，必须连接库 wdmsec.lib。

7.5 符号链接的用户相关性

从前面的例子看，符号链接的命名貌似很简单。简单的符号链接（之所以称为简单，是因为还有一种使用 GUID 的符号链接，这在本书讨论范围之外）总是命名在\DosDevices\之下。但是实际上这会有一些问题。

比较高级的 Windows 系统（哪个版本的操作系统很难讲，可能必须判定补丁号），符号链接也带有用户相关性。换句话说，如果一个普通用户创建了符号链接“\DosDevices\MyCDOSL”，那么，其实其他的用户是看不见这个符号链接的。

但是读者又会发现，如果在 DriverEntry 中生成符号链接，则所有用户都可以看见。原因是 DriverEntry 总是在进程“System”中执行。系统用户生成的符号链接是大家都可以看见的。

当前用户总是取决于当前启动当前进程的用户。实际编程中并不一定要在 DriverEntry 中生成符号链接。一旦在一个不明用户环境下生成符号链接，就可能出现注销然后换用户登录之后，符号链接“不见了”的严重错误。这也是常常让初学者抓狂几周都不知道如何解决的一个问题。

其实解决的方案很简单，任何用户都可以生成全局符号链接，让所有其他用户都能看见。路径“\DosDevices\MyCDOSL”改为“\DosDevices\Global\MyCDOSL”即可。

但是在不支持符号链接用户相关性的系统上，生成“\DosDevices\Global\MyCDOSL”这样的符号链接是一种错误。为此必须先判断一下。幸运的是，这个判断并不难。下面是一个例子，这个例子生成的符号链接总是随时可以使用，不用担心用户注销：

```
    UNICODE_STRING device_name;
    UNICODE_STRING symbl_name;
    if (IoIsWdmVersionAvailable(1, 0x10))
    {
        // 如果是支持符号链接用户相关性的版本的系统，用\DosDevices\Global.
        RtlInitUnicodeString(&symbl_name,
L"\\DosDevices\\Global\\SymbolicLinkName");
    }
    else
    {
        // 如果是不支持的，则用\DosDevices
        RtlInitUnicodeString(&symbl, L"\\DosDevices\\SymbolicLinkName");
    }
    // 生成符号链接
    IoCreateSymbolicLink(&symbl_name, &device_name);
```

第八章 处理请求

8.1 IRP 与 IO_STACK_LOCATION

开发一个驱动要有可能要处理各种 IRP。但是本书范围内，只处理为了应用程序和驱动交互而产生的 IRP。IRP 的结构非常复杂，但是目前的需求下没有必要去深究它。应用程序为了和驱动通信，首先必须打开设备。然后发送或者接收信息。最后关闭它。这至少需要三个 IRP：第一个是打开请求。第二个发送或者接收信息。第三个是关闭请求。

IRP 的种类取决于主功能号。主功能号就是前面的说的 DRIVER_OBJECT 中的分发函数指针数组中的索引。打开请求的主功能号是 IRP_MJ_CREATE，而关闭请求的主功能号是 IRP_MJ_CLOSE。

如果写有独立的处理 IRP_MJ_CREATE 和 IRP_MJ_CLOSE 的分发函数，就没有必要自然判断 IRP 的主功能号。如果像前面的例子一样，使用一个函数处理所有的 IRP，那么首先就要得到 IRP 的主功能号。IRP 的主功能号在 IRP 的当前栈空间中。

IRP 总是发送给一个设备栈。到每个设备上的时候拥有一个“当前栈空间”来保存在这个设备上的请求信息。读者请暂时忽略这些细节。下面的代码在 MyDispatch 中获得主功能号，同时展示了几个常见的主功能号：

```
NTSTATUS MyDispatchFunction(PDEVICE_OBJECT device,PIRP irp)
{
    // 获得当前 irp 调用栈空间
    PIO_STACK_LOCATION irpsp = IoGetCurrentIrpStackLocation(irp);
    NTSTATUS status = STATUS_UNSUCCESSFUL;
    swtich(irpsp->MajorFunction)
    {
    // 处理打开请求
    case IRP_MJ_CREATE:
        ……
        break;
    // 处理关闭请求
    case IRP_MJ_CLOSE:
        ……
        break;
    // 处理设备控制信息
    case IRP_MJ_DEVICE_CONTROL:
        ……
        break;
    // 处理读请求
    case IRP_MJ_READ:
        ……
        break;
```

```
        // 处理写请求
        case IRP_MJ_WRITE:
            ……
            break;
        default:
            …
            break;
        }
        return status;
    }
```

用于与应用程序通信时，上面这些请求都由应用层 API 引发。对应的关系大致如下：

应用层调用的 API	驱动层收到的 IRP 主功能号
CreateFile	IRP_MJ_CREATE
CloseHandle	IRP_MJ_CLOSE
DeviceIoControl	IRP_MJ_DEVICE_CONTROL
ReadFile	IRP_MJ_READ
WriteFile	IRP_MJ_WRITE

了解以上信息的情况下，完成相关 IRP 的处理，就可以实现应用层和驱动层的通信了。具体的编程在紧接后面的两小节里完成。

8.2 打开与关闭的处理

如果打开不能成功，则通信无法实现。要打开成功，只需要简单的返回成功就可以了。在一些有同步限制的驱动中（比如每次只允许一个进程打开设备）编程要更加复杂一点。但是现在忽略这些问题。暂时认为我们生成的设备任何进程都可以随时打开，不需要担心和其他进程冲突的问题。

简单的返回一个 IRP 成功（或者直接失败）是三部曲,如下：

1. 设置 irp->IoStatus.Information 为 0。关于 Information 的描述，请联系前面关于 IO_STATUS_BLOCK 结构的解释。
2. 设置 irp->IoStatus.Status 的状态。如果成功则设置 STATUS_SUCCESS,否则设置错误码。
3. 调用 IoCompleteRequest (irp,IO_NO_INCREMENT)。这个函数完成 IRP。

以上三步完成后，直接返回 irp->IoStatus.Status 即可。示例代码如下。这个函数能完成打开和关闭请求。

```
NTSTATUS
MyCreateClose(
    IN PDEVICE_OBJECT device,
    IN PIRP           irp)
{
```

```
    irp->IoStatus.Information = 0;
    irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest (irp,IO_NO_INCREMENT);
    return irp->IoStatus.Status;
}
```

当然，在前面设置分发函数的时候，应该加上：

```
DriverObject->MajorFunctions[IRP_MJ_CREATE] = MyCreateClose;
DriverObject->MajorFunctions[IRP_MJ_CLOSE] = MyCreateClose;
```

在应用层，打开和关闭这个设备的代码如下：

```
HANDLE device=CreateFile("\\\\.\\MyCDOSL",
    GENERIC_READ|GENERIC_WRITE,0,0,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_SYSTEM,0);
if (device == INVALID_HANDLE_VALUE)
{
    // …. 打开失败，说明驱动没加载，报错即可
}

// 关闭
CloseHandle(device);
```

8.3 应用层信息传入

应用层传入信息的时候，可以使用 WriteFile，也可以使用 DeviceIoControl。DeviceIoControl 是双向的，在读取设备的信息也可以使用。因此本书以 DeviceIoControl 为例子进行说明。DeviceIoControl 称为设备控制接口。其特点是可以发送一个带有特定控制码的 IRP。同时提供输入和输出缓冲区。应用程序可以定义一个控制码，然后把相应的参数填写在输入缓冲区中。同时可以从输出缓冲区得到返回的更多信息。

当驱动得到一个 DeviceIoControl 产生的 IRP 的时候，需要了解的有当前的控制码、输入缓冲区的位置和长度，以及输出缓冲区的位置和长度。其中控制码必须预先用一个宏定义。定义的示例如下：

```
#define MY_DVC_IN_CODE \
    (ULONG)CTL_CODE(FILE_DEVICE_UNKNOWN, \
        0xa01, \
        METHOD_BUFFERED, \
        FILE_READ_DATA|FILE_WRITE_DATA)
```

其中 0xa01 这个数字是用户可以自定义的。其他的参数请照抄。

下面是获得这三个要素的例子：

```
NTSTATUS MyDeviceIoControl(
    PDEVICE_OBJECT  dev,
    PIRP    irp)
{
    // 得到 irpsp 的目的是为了得到功能号、输入输出缓冲
    // 长度等信息。
    PIO_STACK_LOCATION irpsp =
        IoGetCurrentIrpStackLocation(irp);
    // 首先要得到功能号
    ULONG code = irpsp->Parameters.DeviceIoControl.IoControlCode;
    // 得到输入输出缓冲长度
    ULONG in_len =
        irpsp->Parameters.DeviceIoControl.InputBufferLength;
    ULONG out_len =
        irpsp->Parameters.DeviceIoControl.OutputBufferLength;
    // 请注意输入输出缓冲是公用内存空间的
    PVOID buffer = irp->AssociatedIrp.SystemBuffer;

    // 如果是符合定义的控制码，处理完后返回成功
    if(code == MY_DVC_IN_CODE)
    {
        … 在这里进行需要的处理动作

        // 因为不返回信息给应用，所以直接返回成功即可。
        // 没有用到输出缓冲
        irp->IoStatus.Information = 0;
        irp->IoStatus.Status = STATUS_SUCCESS;
    }
    else
    {
        // 其他的请求不接受。直接返回错误。请注意这里返
        // 回错误和前面返回成功的区别。
        irp->IoStatus.Information = 0;
        irp->IoStatus.Status = STATUS_INVALID_PARAMETER;
    }
    IoCompleteRequest (irp,IO_NO_INCREMENT);
    return irp->IoStatus.Status;
}
```

在前面设置分发函数的时候，要加上：

```
DriverObject->MajorFunctions[IRP_MJ_DEVICE_CONTROL] = MyCreateClose;
```

应用程序方面，进行 DeviceIoControl 的代码如下：

```
HANDLE device=CreateFile("\\\\.\\MyCDOSL",
    GENERIC_READ|GENERIC_WRITE,0,0,
    OPEN_EXISTING,
    FILE_ATTRIBUTE_SYSTEM,0);
BOOL ret;
DWORD length = 0;            // 返回的长度

if (device == INVALID_HANDLE_VALUE)
{
    // … 打开失败，说明驱动没加载，报错即可
}

BOOL ret = DeviceIoControl(device,
    MY_DVC_IN_CODE,      // 功能号
    in_buffer,           // 输入缓冲，要传递的信息，预先填好
    in_buffer_len,       // 输入缓冲长度
    NULL,                // 没有输出缓冲
    0,                   // 输出缓冲的长度为 0
    &length,             // 返回的长度
    NULL);

if(!ret)
{
    // … DeviceIoControl 失败。报错。
}

// 关闭
CloseHandle(device);
```

8.4 驱动层信息传出

驱动主动通知应用和应用通知驱动的通道是同一个。只是方向反过来。应用程序需要开启一个线程调用 DeviceIoControl，（调用 ReadFile 亦可）。而驱动在没有消息的时候，则阻塞这个 IRP 的处理。等待有信息的时候返回。

有的读者可能听说过在应用层生成一个事件，然后把事件传递给驱动。驱动有消息要通知应用的时候，则设置这个事件。但是实际上这种方法和上述方法本质相同：应用都必须开启一个线程去等待（等待事件）。而且这样使应用和驱动之间交互变得复杂（需要传递事件句柄）。这毫无必要。

让应用程序简单的调用 DeviceIoControl 就可以了。当没有消息的时候，这个调用不返回。应用程序自动等待（相当于等待事件）。有消息的时候这个函数返回。并从缓冲区中读到消息。

实际上，驱动内部要实现这个功能，还是要用事件的。只是不用在应用和驱动之间传递事件了。

驱动内部需要制作一个链表。当有消息要通知应用的时候，则把消息放入链表中（请参考前面的“使用 LIST_ENTRY”），并设置事件（请参考前面的“使用事件”）。在 DeviceIoControl 的处理中等待事件。下面是一个例子：这个例子展示的是驱动中处理 DeviceIoControl 的控制码为 MY_DVC_OUT_CODE 的部分。实际上驱动如果有消息要通知应用，必须把消息放入队列尾并设置事件 g_my_notify_event。MyGetPendingHead 获得第一条消息。请读者用以前的知识自己完成其他的部分。

```
NTSTATUS MyDeviceIoCtrlOut(PIRP irp,ULONG out_len)
{
    MY_NODE *node;
    ULONG pack_len;
    // 获得输出缓冲区。
    PVOID buffer = irp->AssociatedIrp.SystemBuffer;

    // 从队列中取得第一个。如果为空，则等待直到不为空。
    while((node = MyGetPendingHead()) == NULL)
    {
        KeWaitForSingleObject(
            &g_my_notify_event,// 一个用来通知有请求的事件
            Executive,KernelMode,FALSE,0);
    }

    // 有请求了。此时请求是 node。获得 PACK 要多长。
    pack_len = MyGetPackLen(node);
    if(out_len < pack_len)
    {
        irp->IoStatus.Information = pack_len;   // 这里写需要的长度
        irp->IoStatus.Status = STATUS_INVALID_BUFFER_SIZE;
        IoCompleteRequest (irp,IO_NO_INCREMENT);
        return irp->IoStatus.Status;
    }

    // 长度足够，填写输出缓冲区。
    MyWritePackContent(node,buffer);
    // 头节点被发送出去了，可以删除了
    MyPendingHeadRemove ();
    // 返回成功
    irp->IoStatus.Information = pack_len;   // 这里写填写的长度
    irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest (irp,IO_NO_INCREMENT);
    return irp->IoStatus.Status;
}
```

这个函数的处理要追加到 MyDeviceIoControl 中。如下：

```
NTSTATUS MyDeviceIoControl(
    PDEVICE_OBJECT  dev,
    PIRP    irp)
{
    …
    if(code == MY_DVC_OUT_CODE)
        return MyDeviceIoCtrlOut(dev,irp);
    …
}
```

在这种情况下，应用可以循环调用 DeviceIoControl,来取得驱动驱动通知它的信息。

后记：我的闲言碎语

写这本小册子的时候，我正在 NED-LS 办离职手续。

想当初在东京的时候，NED 的田上每夜好酒好肉的招待。北京几个同事跳槽，搞得项目特别尴尬。田上特意说道：“拜托你们不要转职.....”。

半年不到，我就抛出一纸辞职信，真是负心人啊.....

一眨眼间，就在 NED-LS 混了三年了。不是我非要走人，一方面汇率节节攀升，外包越来越困难。欧美企业不退反进，纷纷把更高档的玩意搬来国内来发。许多日本公司却还把外包当主业。另一方面我日语暴烂，又不肯学。继续摆烂显然不是办法。

相对于 Intel，我其实一直是看好 AMD 的。上次帮小 D 选笔记本，还特意选了 AMD 的 CPU。都说 AMD 的东西便宜量又足，最适合国内的劳苦大众，这可不是吹的。只可惜貌似每次都被 Intel 揍得鼻青脸肿。所以我便去面试了。跟我去后面那天去的还有那个写“碟中谍虚拟光驱”的说话有点像唐僧的万春。

上海的 AMD 在浦东的荒郊。先把 2 号线坐到终点站，然后打的到一处无人知道的野外。只看见长长的公路和茂盛的野草。两边有无数片工地，横七竖八的堆着许多建筑材料，却没有一个人。好处是内急的时候不用找公厕（也不可能找得到）。直接在路边就可以解决。

工地的旁边有一部分成品。AMD 的绿色标记就坐落其中。有两栋楼，都不大，袖珍型的。他们的面试没有笔试（说本来是有，但我去的时候卷子还没准备好，就免了）。是四个人轮番上阵，前三个是工程师，最后一个是经理。

我面的是存储芯片驱动开发。问到驱动开发相关或者纯 C 语言的问题，我自然是对答如流，这许多年的苦工不是白干的。偏偏他们对效率很有兴趣，总是不时抛出几个位运算的妙用之类的几个优化题。我只好明说了平时并不怎么关心效率。所以这个不擅长。待加盟了你们项目组之后，一定好好学习天天向上云云。

最终当然是没过了。他们 Pending 了好长一段时间。最终结论是我不适合做硬件驱动开发。因为我以前的经验比较上层。做虚拟 SCSI 设备的万春按理比我好点，但是他死得更惨：结论是只在小公司干过，组织性、纪律性会比较差（其实这也没说错==!）。杂牌军被 BS 了。

万春没多久就去广州了。真是“浮云游子意，落日故人情”啊。不过话撂这儿了，莫怪劳资以后不支持 AMD...

我的零八年的春夏真是惬意。做的几个程序都还没出大问题。有空还写写书，《天书夜读》扔给出版社了。只可惜一审再编的没完没了。到现在也还没有头。我又开始写新书。但不知道怎么说，因为题目还没有拟好。然后又见到小 D，一个不小心就掉到蜜罐子里了。

后来又面试了几个。Mavell 的面试官实在太强了，无所不知，成功的鄙视了我。MS 的面试官是老外。虽然我不懂他说什么，但是我说的他也未见得明白。

不过 MS 的职位真的是很棒啊。做 Windows Kernel，而且还在上海。一通过马上先送去美利坚培训。很美啊。

后来又去面 Intel（屡败屡面==!）。Intel 的环境真是不错。我到的时候正是早上。天气又好，一个人也没有，对面是一大片几乎望不到边的浅水，长着人深的草。几只长脚的苍鹭站在水里。还有一些棕色的像小鸭子的玩意在水里抢食吃。一些小鸟在空中掠过，发出铜铃一样的声音。Intel 就在那边沼泽地的对面。比 AMD 大。几栋大楼。有上千人在那里工作。吃饭都在食堂。方圆 2 公里内没有饭店。

Intel 的面试和 AMD 很像。没有笔试。四人轮番上阵。不过他们四个人稍微有些分工。每个人问的方向都不大一样。另一个情况是他们喜欢给你水笔，然后请直接在白板上写代码。

还不能写简意，非要一行一行写出来才行。写白板手抖得厉害，没点心理素质还真不行。

第一个人问内核编程。问到我的得意之处了。不过这帮人还真的有两把刷子——他们能看 Windows 的代码，我还得自己反汇编。世界真不公平啊。

然后来一个人问了很多设计模式和代码管理之类的问题。这方面我当然是更滔滔不绝了。最有挑战性的是第三人，发型很像爱因斯坦那个，进来坐定之后，也不说什么，就给出一张白纸，让我写一个矩形相交判断以及一个形状覆盖的算法。时间又大概只有二十分钟，大脑一片空白，汗就出来了。空白了大概十分钟，还一个字没写。面试官都急了，就说：“你如果有什么中间结果，就先拿出来。”意思就是你多少写点，别交白卷啊。

不过好歹我以前是做过 3D 引擎的（虽然做得很烂的说）。慢慢冷静下来，和他说了计算的步骤。不算优秀也不算高效（十几分钟哪里有空考虑那么多啊）。但是面试官说也算是逻辑完整。ok，又出了一道图论算法题。这时候我已经缓过劲来，五分钟内轻松搞定。爱因斯坦满意的走了。

最后一个是部门经理。相与言欢。然后他请客在他们食堂吃饭。但是真的很难吃的说。下午回 NEC-AS 继续上班，一面构思辞职信的措辞。

告别 luO，告别 wowocock。

本书献给小 D。她是我今夏遇到的，生命里最美的一缕阳光。

谭文 于 2008 年端午

（全书完）