程序的本质，就是内存里的一串串的数字（它们被 CPU 当作指令解析，才能够有意义，否则就是一坨垃圾）；因此在正式讲解驱动编程之前，首先讲解内存使用。

内存使用，无非就是申请、复制、设置、释放。在 C 语言里，它们对应的函数是：malloc、memcpy、memset、free；在内核编程里，他们分别对应 ExAllocatePool、RtlMoveMemory、RtlFillMemory、ExFreePool。它们的原型分别是：

PVOID ExAllocatePool(POOL_TYPE PoolType, SIZE_T NumberOfBytes);
VOID RtlMoveMemory(PVOID Destination, PVOID Source, SIZE_T Length);
VOID RtlFillMemory(PVOID Destination, SIZE_T Length, UCHAR Fill);
VOID ExFreePool(PVOID P);

需要注意的是，RtlFillMemory 和 memset 的原型不同、ExAllocatePool 和 malloc 的原型也不同。前者只是参数前后调换了一下位置，但是后者则多了一个参数：PoolType。这个 PoolType 也是必须了解的。**PoolType 在 MSDN 的介绍上有 N 种，其实常用的只有 2 种：PagedPool 和 NonPagedPool。PagedPool 是分页内存，简单来说就是物理内存不够时，会把这片内存移动到硬盘上，而 NonPagedPool 是无论物理内存如何紧缺，都绝对不把这片内存的内容移动到硬盘上。**在往下讲之前，先补充一个知识，**就是我们操作的内存，都是虚拟内存，和物理内存是两码事。但虚拟内存的数据是放在物理内存上的，两者存在映射关系，一般来说，一片物理内存可以映射为多片虚拟内存，但一片虚拟内存必定只对应一片物理内存。**假设虚拟内存是 0Xfffff80001234567 在物理内存的地址是 0x123456，当物理内存不够用时，物理内存 0x123456 的原始内容就挪到硬盘上，然后把另外一片急需要用的内容移到物理内存里。此时，当你再读取 0Xfffff80001234567 的内容时，就会引发缺页异常，系统就会把在硬盘上的内容再次放到物理内存中（如果这个过程失败，一般就死机了）。以上说了这么多废话，总结两句：**1.NonPagedPool 的总量是有限的（具体大小和你物理内存的大小相关），而 PagedPool 的总量较多。申请了内存忘记释放都会造成内存泄漏，但是很明显忘记释放 NonPagedPool 的后果要严重得多；2.一般来说，PagedPool 用来放数据（比如你用 ZwQuerySystemInformation 枚举内核模块，可以申请一大片 PagedPool 存放返回的数据），而 NonPagedPool 用来放代码（你写内核 shellcode 并需要执行时，必须使用 NonPagedPool 存放 shellcode）。以我的经验来说，访问到切换出去的内存没事，但是执行到切换出去的内存必然蓝屏（这只是我的经验，正确性待定）。3.在用户态，内存是有属性的，有的内存片只能读不能写不能执行（PAGE_READ），有的内存片可以读可以写也可以执行（PAGE_READ_WRITE_EXECUTE）。在内核里，PagedPool 和 NonPagedPool 都是可读可写可执行的，而且没有类似 VirtualProtect 之类的函数。**示例代码：

```
void test()
{
	PVOID ptr1 = ExAllocatePool(PagedPool,0x100);
	PVOID ptr2 = ExAllocatePool(NonPagedPool,0x200);
	RtlFillMemory(ptr2,0x200,0x90);
	RtlMoveMemory(ptr1,ptr2,0x50);
	ExFreePool(ptr1);
	ExFreePool(ptr2);
}
```

在内核里想要写入“别人的”内存（一般指 NTOS 等系统模块的内存空间），还有另外的

规矩，这里又涉及到另外两个概念：IRQL 和内存保护。IRQL 成为中断请求级别，从 0～31 共 32 个级别；内存保护可以打开和关闭，如果在内存处于保护状态时写入，会导致蓝屏。**一般来说，要写入"别人的"内核内存，必须关闭内存写保护，并把 IRQL 提升到 2 才行（绝大多数时候 IRQL 都为 0，当 IRQL=2 时，会阻断大部分线程执行，防止执行出错）**。内存是否处于写保护的状态记录在 CR0 寄存器上，因此直接修改 CR0 寄存器的值即可；而提升或降低 IRQL 则使用 KeRaiseIrqlToDpcLevel 和 KeLowerIrql 实现（WIN64 的 IRQL 值记录在 CR8 寄存器上，而 WIN32 的 IRQL 值记录在 KPCR 上）。代码如下：

```
KIRQL WPOFFx64()
{
	KIRQL irql=KeRaiseIrqlToDpcLevel();
	UINT64 cr0=__readcr0();
	cr0 &= 0xfffffffffffeffff;
	__writecr0(cr0);
	_disable();
	return irql;
}

void WPONx64(KIRQL irql)
{
	UINT64 cr0=__readcr0();
	cr0 |= 0x10000;
	_enable();
	__writecr0(cr0);
	KeLowerIrql(irql);
}

void test()
{
	KIRQL irql=WPOFF();
	RtlMoveMemory(NtOpenProcess,HookCode,15);
	WPON(irql);
}
```

至于写入"别人的"内存，还有一种微软推荐的安全方式，就是 MDL 映射内存的方式。这个比较麻烦，大概方法是**申请一个 MDL（类似句柄的玩意），然后尝试锁定页面，如果成功，则让系统分配一个"安全"的虚拟地址再行写入，写入完毕后解锁页面并释放掉 MDL**。以下是某人写的 SafeCopyMemory：

```
BOOLEAN SafeCopyMemory( PVOID pDestination, PVOID pSourceAddress, SIZE_T SizeOfCopy )
{
	PMDL pMdl = NULL;
	PVOID pSafeAddress = NULL;
	if( !MmIsAddressValid(pDestination) || !MmIsAddressValid(pSourceAddress) )
		return FALSE;
	pMdl = IoAllocateMdl(pDestination, (ULONG)SizeOfCopy, FALSE, FALSE, NULL );
```

```
    if( !pMdl )
        return FALSE;
    __try
    {
        MmProbeAndLockPages( pMdl, KernelMode, IoReadAccess );
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        IoFreeMdl( pMdl );
        return FALSE;
    }
    pSafeAddress = MmGetSystemAddressForMdlSafe( pMdl, NormalPagePriority );
    if( !pSafeAddress )
        return FALSE;
    __try
    {
        RtlMoveMemory(pSafeAddress, pSourceAddress, SizeOfCopy );
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        ;
    }
    MmUnlockPages( pMdl );
    IoFreeMdl( pMdl );
    return TRUE;
}
```

内存部分的基础知识讲解到此完毕，但内存管理涉及的方方面面太多了，以后想到什么，我还会随时补充新内容。最后，**对那些把指针写成 ULONG 的人，表示深深的鄙视。这种只考虑眼前方便而不考虑后期移植的人，终究会自尝苦果**。