

目录

=====

Inside WINDOWS NT Object Manager

- 00."对象化的" Windows NT
- 01.Windows NT对象管理器
- 02.对象类型
- 03.Object Directory 与 SymbolicLink
- 04.对象的创建与对象的结构
- 05.Object Type 结构体
- 06.Дальнейшая жизнь объекта - простейший с
лучай
- 07.句柄数据库
- 08.对象的安全性
- 09.命名对象
- 0a.对象的消亡
- 0b.其它管理器函数
- 0c.结语

附录

- 13.从用户模式下获取对象信息
- 14.某些与对象管理相关的系统服务

00."对象化的" Windows NT

=====

如果您熟悉Windows NT体系结构,当然会知道在这个系统中资源都是以对象的形式存在的。这样集中了对资源的管理。实际上,所有的Windows NT子系统无论如何都要使用资源,也就是说都要经过对象管理器。对象管理器是Windows NT的子系统,它对对象提供支持。对对象的使用贯穿了整个操作系统,真正的"对象化"的NT对应用程序隐藏了Win32子系统,甚至于部分的隐藏了系统调用接口。

通常,当提到NT的体系结构时,都会说到各种管理器(对象管理器、内存管理器等等)。所有这些管理器都是Windows NT内核(ntoskrnl.exe)的一部分。NT的内核主要使用C语言编写的,每一个管理器本身都是一组函数,这些函数都被封装在相应的模块中。并不是所有的都定义了接口。管理器之间借助于函数调用相互协作(在内核内部)。一部分函数从内核中导出的并公开了使用方法以在内核模式下使用。

对象管理器是一组形如ObXXXX的函数。实际上并不是所有的函数都是从内核中导出的。一般说来,对象管理器主要在内核中实现。例如,许多系统调用(NtCreateProcess、NtCreateEvent...)的处理都与这样或是那样的对象相联系。但是服务隐藏了对象管理器,不让我们的程序知道。甚至于驱动程序的设计人员未必需要大量直接的与对象管理器打交道。尽管如此,了解内核是如何实际工作的无疑非常重要,更为重要的是对象管理器还与安全管理器密切相关。理解了对象管理器就揭开了NT中许多起初并不明显,但潜在具有的可能性.....

01.Windows NT对象管理器

=====

对象管理器是NT内核中许多形如ObXXX的函数。某些函数并未导出并只在内核内部由子系统使

用。对于在内核之外的使用，主要是通过未公开的函数。

ObAssignSecurity, ObCheckCreateObjectAccess, ObCheckObjectAccess,
ObCreateObject, ObDereferenceObject, ObFindHandleForObject,
ObGetObjectPointerCount, ObGetObjectSecurity, ObInsertObject,
ObMakeTemporaryObject, ObOpenObjectByName, ObOpenObjectByPointer,
ObQueryNameString, ObQueryObjectAuditingByHandle,
ObReferenceObjectByHandle, ObReferenceObjectByName,
ObReferenceObjectByPointer, ObReleaseObjectSecurity,
ObSetSecurityDescriptorInfo, ObfDereferenceObject, ObfReferenceObject

为了描述对象管理器，我们先来看一下什么是对象。

02.对象类型

=====

对象管理器本身就工作在某些对象之上。其中之一就是object type。object type用于描述所有对象的共同属性。object type由函数ObCreateObjectType(...)在内核初始化子系统的时候创建(也就是是在系统初始化的时候)。我们所感兴趣的并不是ObCreateObjectType(...)函数，因为其并未由内核导出。但是，在object type中的信息是非常重要的。关于这些我们少后再说。

大概的讲，任何对象都可以分为两个部分：一部分包含管理上必需的信息(我们称之为首部)，另一部分填充并确定创建该对象的子系统的必需信息(这是对象的body)。对象管理器对首部进行操作，而实际上并不对对象的内容感兴趣，还有一些对象是对象管理器本身使用的。object type正如其名字所示的那样，定义了对象的类型。每一个对象都有对其object type的引用，对象管理器也会非常频繁的使用对象的body。object type结构体的一个域是类型的名字。在NT下有以下各类型：

Type, Directory, SymbolicLink, Event, EventPair, Mutant, Semaphore,
Windows, Desktop, Timer, File, IoCompletion, Adapter, Controller, Device,
Driver, Key, Port, Section, Process, Thread, Token, Profile

Type类型的object type不足一惧。要知道扼要的讲object type是用于管理器的对象，其和所有其它的对象是一样的，也有自己的Type类型(它自己的Type类型对象)。如果好好的想一下的话，这样做似乎也并不奇怪。下面，在Type这个话题下，主要来讲几个与object type相关的名词。

03.Object Directory与SymbolicLink

=====

所有列举出的类型都是由不同的子系统创建的。我们现在讨论Directory和SymbolicLink类型，因为在对象管理器本身的工作中要用到这些类型的对象(这些类型是其在自己初始化时创建的)。

NT的对象可以有名字，还可以组织成树型的结构。Directory对象正是用来组织这种结构的。它的用途非常简单——保存对其它对象的引用，例如，另外一个Directory对象。Directory的body的结构非常简单：

```
typedef _DIR_ITEM{  
    PDIR_ITEM Next;  
    PVOID Object;
```

```

    }DIR_ITEM,*PDIR_ITEM;

typedef struct _DIRECTORY{
    PDIR_ITEM HashEntries[37];
    PDIR_ITEM LastHashAccess; //94h
    DWORD LastHashResult;    //98h
}DIRECTORY,*PDIRECTORY;

```

路径用于保存命名对象。路径本身是37个entry的哈希表。表中的元素除保存着指向后面元素的指针之外，还保存着指向属于该路径对象的body的指针。哈希函数形式如下：

```

DWORD Hash(const char* str);
{
    char *ptr=str;
    int str_len=strlen(str);
    char Sym;
    int hash=0;
    while(str_len--)
    {
        Sym=*ptr++;
        ToUpperCase(&Sym);
        Sym-=' ';
        hash=hash*3+(hash>>1)+Sym;// умножение знаковое
    }
    return hash%37;
}

```

当然，这里的代码只是给出程序的逻辑，并不是真正的实现。这个例子主要是基于对ObpLookupDirectoryEntry(...)函数的分析写出的。最后的两个域反映出对哈希表的最后访问。LastHashAccess是指向最后访问的entry的指针。LastHashResult包含成功查找到的单元。NT想优化对路径的查找，而且如果在查找的结果中找到的entries中的一个entry不在表的开头，则这个表的元素就要被移到表的开头(认为后面对该对象的访问的几率最大)。

Directory是对象，和其它的对象一样，它也可以有名字。如果再考虑到对象还有对自己所在目录的引用，则对可以画出树型的层级结构这个问题就能够理解了。

在Windows NT中包含着根目录ObpRootDirectoryObject，Windows NT的命名对象树就从这里开始。同时，还有ObpTypeDirectoryObject目录，它包含所有的object type(使用这个树是为了防止object type重名)。内核表ObpObjectTypes中保存着指向object type的指针。再有，对于每一个类型的创建，子系统都保存着单独的指针。

SymbolicLink这个object type用于保存字符串。在树中进行查找时它们被用作引用。例如，对象\\?\\C: (这表示，在ObpRootDirectoryObject中有对路径"\\?\\"的引用，在这个路径下包含着对象"C:")就是SymbolicLink对象并包含字符串"\\Device\\Harddisk0\\Partition1"。该对象格式如下：

```

typedef struct _SYMBOLIC_LINK{
    LARGE_INTEGER CreationTime; // время создания от 1601 год
    UNICODE_STRING Link;
}SYMBOLIC_LINK,*PSYMBOLIC_LINK;

```

现在有了对管理器用到的类型和对象的一般概念，我们可以来研究更为具体的信息了。

04.对象的创建与对象的结构

=====

对象由ObCreateObject函数创建。这个函数由ntoskrnl.exe导出，下面给出其原型：

```
NTSTATUS NTOSKRNL
ObCreateObject
(
    KPROCESSOR_MODE bMode,          // kernel / user
    POBJECT_TYPE Type,              // 对象类型
    POBJECT_ATTRIBUTES Attributes,  // 属性
    BOOLEAN bObjectMode,           // kernel/user对象类型
    DWORD Reserved,                 // 函数未使用
    DWORD BodySize,                 // 对象body的大小
    DWORD PagedPoolQuota OPTIONAL, // 如果为0
    DWORD NonPagedPoolQuota OPTIONAL, // 则回收
    PVOID* pObjectBody              // 指向body的指针
);
```

参数Reserved可以简单的忽略。bObjectMode定义了是否要从用户模式访问。PagedPollQuota和NonPagedPollQuota是与这些对象相关联的分页池和非分页池的限额。限额记录了进程每一次其打开句柄的情况。进程有一个限额的阈值，其不能超越。POBJECT_ATTRIBUTES结构体是公开的(ntdef.h中有)，但我这里还要将其给出，因为后面会经常引用其中的域。

```
typedef struct _OBJECT_ATTRIBUTES {
    ULONG Length;
    HANDLE RootDirectory;
    PUNICODE_STRING ObjectName;
    ULONG Attributes;
    PVOID SecurityDescriptor;
    PVOID SecurityQualityOfService;
} OBJECT_ATTRIBUTES;
typedef OBJECT_ATTRIBUTES *POBJECT_ATTRIBUTES;
```

现在来集中讨论重要的域RootDirectory和ObjectName。RootDirectory是什么，现在我想已经能猜到了——将要包含对象(如果需要)的目录。ObjectName——一见名知意。顺便说一下，可以不指明RootDirectory，但在ObjectName中要指定对象的完整路径，这时对象将在起始于ObpRootDirectoryObject的树中创建。

关于安全方面的话我们单独讲，所以Security域嘛.....暂时留下不提。剩下Attributes域。ntdef.h中描述了各位的意义。下面给出这些属性。

```
#define OBJ_INHERIT          0x00000002L
#define OBJ_PERMANENT        0x00000010L
#define OBJ_EXCLUSIVE        0x00000020L
#define OBJ_CASE_INSENSITIVE 0x00000040L
#define OBJ_OPENIF           0x00000080L
#define OBJ_OPENLINK         0x00000100L
```

结果，函数返回指向对象body的指针。

到了描述对象结构的时候啦。

```
+-----+<-----> -??h
|         |
```

```

| ..... | --长度可变的结构体 (到 30h)
|       |
+-----+<--对象首部-----> 00h
| Header | --标准的首部18h字节
|       |
+-----+<--对象body-----> 18h
|       |
| Body   | - тело объекта желаемого размера
|       |
+-----+<-----> 18h+BodySize

```

管理器经常要操作指向对象首部的指针。它返回给我们的是指向**body**的指针。首部之上还有一块区域, 这块区域长度可变并依赖于所建对象的参数和类型。这个“帽子”可以保存下面这个结构体:

```

typedef struct _POLLED_QUOTA // 如果限额不等于default则增加
{
    // 保存限额的消耗量
    DWORD PagedPoolQuota;
    DWORD NonPagedPoolQuota;
    DWORD QuotaInformationSize; // PagedPollQuota的总和
    PPROCESS_OBJECT pProcess; // 拥有此对象的进程
}POLLED_QUOTA;

```

```

typedef struct _HANDLE_DB // 关于打开句柄的信息
{
    union {
        PPROCESS_OBJECT pProcess; // 如果只有一个进程打开了
        // 指针
        PVOID HandlesDBInfo; //-----+-----
    }HanInfo; // |dd Num;2
    // +-----
    // |dd pProcess1
    // |dd Counter2
    // +-----
    // |dd pProcess2
    // |dd Counter2
    // +-----
    // |....
    DWORD Counter; // 句柄总数.
}HANDLE_DB;

```

```

typedef struct _OBJECT_NAME
{
    PDIRECTORY Directory; // 对象所属的路径
    UNICODE_STRING ObjectName; //对象名
    DWORD Reserved; //对齐
}OBJECT_NAME;

```

```

typedef struct _CREATOR_INFO
{
    LIST_ENTRY Creator; // 闭合的链表 (包含着类型)
    DWORD UniqueProcessId; // 对象父进程的ID
    DWORD Reserved; // 对齐
}

```

```
}CREATOR_INFO;
```

这个或是其它的结构体可以存在，也可以不存在，但顺序不能变。在注释中写明了这些结构体都是作什么用的了。它们中最后一个(哪怕只有一个)的后面就是标准的对象首部了。

```
typedef struct _OBJECT_HEADER
{
    DWORD RefCounter; // 对象的引用计数           00
    DWORD HandleCounter; // 句柄数目               04
    POBJECT_TYPE ObjectType; // 对象类型           08
    DWORD SubHeaderInfo; // 后面会讲           0c
    UNION // 10
    {
        POBJECT_INFO ObjectInfo;
        PQQUOTA_BLOCK pQuotaBlock;
    } a;
    PSECURITY_DESCRIPTOR SecurityDescriptor; // 14 Optional
} OBJECT_HEADER;
```

先不讲这个联合体，以后再说。其它域的作用都很显然。对象管理器将统计打开的对象句柄以及对对象的引用，以此来知道何时删除对象的名字和**body**（如果对象不是**permanent**的，也就是说没有设置**OBJ_PERMANENT**位）。还有对对象类型的引用和有关安全的信息。首部前面的结构体中的域在**SubHeaderInfo**域中也有体现。

SubHeaderInfo中各个位的含义如下:

The diagram illustrates the structure of an Object Header, which is 32 bits long. It is divided into four 8-bit sections:

- Quota offset** (bits 31-24)
- HandleDB offset** (bits 23-16)
- Name Offset** (bits 15-8)
- Unnamed section** (bits 7-0)

Fields and their bit positions:

- OBJECT_NAME** (距首部起始的正偏移(向上)) - bits 31-24
- HANDLE_DB** (距首部起始的正偏移(向上)) - bits 23-16
- POLLED_QUOTA** (距首部起始的正偏移(向上)) - bits 15-8
- QuotaBlock/ObjectInfo** (Quota charged) - bits 7-0
- User/Kernel mode** - bits 31-30
- CREATOR_INFO** - bits 29-28
- OBJ_EXCLUSIVE** - bit 27
- OBJ_PERMANENT** - bit 26
- SecurityDescriptor** - bits 25-24
- HandleInfo** (未初始化) - bits 23-22
- Unused/Reserved** - bits 21-0

*在SubHeaderInfo没有CREATOR_INFO的字节偏移, 但因为这个结构体(如果有)总是最后一个, 故指向它的指针可以获得, 只需从首部指针中减去sizeof(CREATOR_INFO)就可以了。

当我第一次开始反汇编对象管理器函数并看到ObCreateObject(...)的时候, 我断定这个函数做了使对象开始其“生活”的所有必需的工作。但是一个普遍的原则——一直到最后再下定论(很好的原则), 在这里也成立。实际上, 对于前面描述的各域, 这个函数只填充了结构体的最低限度的域。倒不如说这个函数在内存中把对象折腾了一番, 捎带脚儿地填充了几个域。

再详细些。SubHeaderInfo中的位: Q=1 总是, S=0 总是, H=1 如果存在HANDLE_DB。OBJECT_NAME中 Directory = NULL。CREATOR_INFO中的LIST_ENTRY指向自己。POLLED_QUOTA中 pProcess = NULL。HANDLE_DB清零。HandleCounter = 0; RefCounter = 1; 其余的域以相应的information entries和对象类型中的信息来填充。除此之外, 分配并填充OBJECT_INFO结构体, 首部的UNION {..}a中的相应指针(达, 当然所有的内存都是从NonPaged pool里分配的)。

```
typedef struct _OBJECT_INFO{
    DWORD Attributes;          //00h OBJECT_ATTRIBUTES.Attributes
    HANDLE RootDirectory;      //04h
    DWORD Reserved;            //08h - Unknown or Res.
    KPROCESSOR_MODE bMode;     //0ch
    BYTE Reserved[3];          //0dh - Alignment
    DWORD PagedPoolQuota;      //10h
    DWORD NonPagedPoolQuota;   //14h
    DWORD QuotaInformationSize; //18h - 组SID的大小
                               //+ DACL的大小(圆整后)
    PSECURITY_DESCRIPTOR SelfRelSecDescriptor;
                               //1ch - 指向Self Relativ的指针.
                               //Non Paed Pool里的安全描述符
    PSECURITY_QUALITY_OF_SERVICE pSecQual; //20h
    SECURITY_QUALITY_OF_SERVICE SecQuality; //24h
                               //30h
} OBJECT_INFO,*POBJECT_INFO;
```

顺便说一句, 系统为OBJECT_INFO结构体维护了一个叫ObpCreateInfoLookasideList的Lookaside list(见DDK)。实际上Reserved域有时也会用到(在对象方法中作参数), 但我尚未碰到过这个域的值不为0的情况。QuotaInformationSize用在从Paged和NonPaged pool中移除限额用的。

05.Object Type 结构体

=====

这里我只描述一下object type的body的结构。

```
typedef struct _OBJECT_TYPE
{
    ERESOURCE TypeAsResource; //0x0 可用作资源
                               //34h
    PLIST_ENTRY FirstCreatorInfo; //38h 我注意到了这个结构体
    PLIST_ENTRY LastCreatorInfo; //3ch 只是用于object type
    UNICODE_STRING TypeName; //40h 类型名
    DWORD Unknown2[2];        //48h
```

```

DWORD RefCount;          //50h 该类型对象的计数
DWORD HanCount;          //54h 该类型句柄的计数
DWORD PeakRef;           //58h 对象的峰值
DWORD PeakHandles;       //5ch 句柄的峰值
DWORD Unknown3;          //60h
DWORD AllowedAttributeMask; //64h 可能的属性 0 - 允许所有的
GENERIC_MAPPING GenericMapping; //68 отображение родовых пра
в на специальные
DWORD AllowedAccessMask; //78h (ACCESS_SYSTEM_SECURITY 总是设置的)
BOOLEAN bInNameSpace;    //7ch 这个类型的对象在对象路径中
                        // 可能我会弄错, 但也类似.
BOOLEAN bHandleDB;       //7dh 是否包含对象句柄的信息(HANDLE_DB)
BOOLEAN bCreatorInfo;     //7eh ----//---- CreatorInfo + 38h处的链表
BOOLEAN Unknown5;         //7fh
DWORD Unknown6;           //80h 如果 !=0 则在NpSuccess里创建
DWORD PagedPoolQuota;     //84h default
DWORD NonPagedPoolQuota; //88h 限额
PVOID DumpProcedure;      //8ch 原型未知 (?)
PVOID OpenProcedure;      //90h 原型已知
PVOID CloseProcedure;     //94h 原型已知
PVOID DeleteProcedure;    //98h 原型已知
PVOID ParseProcedure;     //9ch 原型已知
PVOID SecurityProcedure;  //a0h 原型已知
// 可以有4种调用情况:
//0-set sec_info, 1-query descriptor, 2-delete, 3-assign

PVOID QueryNameProcedure; //a4h 原型已知
PVOID Tag;                //a8h 通过高层次信息判断
// 这应该是方法OkayToCloseProcedure;
// 实际上, 对于所有的对象我都发现在这个地址上有四字符的类型Tag,例如Dire
(Directory)
} OBJECT_TYPE,*POBJECT_TYPE;

```

我就不给出已知方法的原型了, 因为这里不大用得到。于是, 类型种包含了类型名, 操作对象的函数, 还有属于该类型的用于对象的一般信息。结构体的域都已有注释, 不再细说。

我们将视线转向偏移0处的ERESOURCE结构体(可以在NTDDK.H中找到)。换句话说, 这个对象可以用作资源。对象管理器使用ExAcquireResourceExclusiveLite函数(见DDK)从object type中取得资源。PagedPoolQuota和NonPagedPoolQuota将限额指定为默认值。如果我们使用限额值为零来调用ObCreateObject函数(或与默认的限额值相同)而且如果没有设置OBJ_EXCLUSIVE且如果QuotaInformationSize小于0x800, 则在创建对象时将不会创建POLLED_QUOTA结构体。AllowedAttributeMask和AllowedAccessMask域指定了属性的掩码和允许的访问。最后有指向方法的指针, 这些方法在定义的时候调用。例如, 在增加句柄的时候, 在这个进程句柄的基址里调用OpenProcedure函数。这并不是说这些域就是我们想象的那样(以及我从Хелен Касер(译注:我想是Helen Custer)的关于Windows NT的书中读到的那样)是必需的。但是它们能够扩展对象的功能。下面我来讲调用某些方法的条件。

06. Дальнейшая жизнь объекта - простейший случай

=====

ObCreateObject创建对象, 但没有什么该对象本质上的东西。没有它的句柄, 不能用名字打开它(ObCreateObject函数并不向对象树中添加对象)。我们回来看一下对象管理器导出的

函数。下面的三个函数很令人好奇:ObOpenObjectByName, ObOpenObjectByPointer, ObInsertObject。第一个函数先不说。那是说第二个呢还是第三个? ObOpenObjectByPointer自然应该讲, 可是ObInsertObject平时更少被提到, 所以我们从它开始。

NTSTATUS NTOSKRNL

```
ObInsertObject(
    PVOID pObject,                //body
    PACCESS_STATE pAccessState OPTIONAL,
    ACCESS_MASK Access,
    DWORD RefCounterDelta OPTIONAL, //0- default (r.e. 1)
    PVOID OUT *ObjectExist OPTIONAL, //如果已经存在
    PHANDLE OUT Handle             //句柄
);
```

函数的逻辑十分明显。它从填充好的先前的OBJECT_INFO结构体得到主要的信息是其即将用到的从对象首部中获得的一个指针(指向对象首部的指针可以简单的用body指针减去18h)。函数的工作方式有两种:一, 对象没有名字且没有在类型中设置bInNameSpace标志;二, 设置了该标志或是有名字。我们来看这两种情况。

如果对象没有名字, 则调用ObpCreateUnnamedHandle(..)。后面, 我将引用一个内部函数而不给出其原型, 其原型是已知的。这个函数本身是以许多内部函数为基础的, 这些内部函数共同完成了该函数的工作。

当前进程记录了限额。在这种情况下, 填充a.pQuotaBlock域并清SubHeaderInfo中的Q位。现在对象“填充好了”限额。a.pQuotaBlock指向与每一个进程相联系的QUOTA_BLOCK结构体。我们现在先不研究对象—进程的结构, 我想下面这个结构体值得一提。

```
typedef _QUOTA_BLOCK{
    DWORD KSPIN_LOCK QuotaLock;
    DWORD RefCounter; // 用于计数该block的进程
    DWORD PeakNonPagedPoolUsage;
    DWORD PeakPagedPoolUsage;
    DWORD NonPagedpoolUsage;
    DWORD PagedPoolUsage;
    DWORD NonPagedPoolLimit;
    DWORD PagedPoolLimit;
    DWORD PeakPagefileUsage;
    DWORD PagefileUsage;
    DWORD PageFileLimit;
}QUOTA_BLOCK,*PQUOTA_BLOCK;
```

我想各域的名字已经很清楚的说明了其用途。行, 我们接着往下进行。增加句柄的数目并恢复HANDLE_DB中的信息, 如果有的话。HANDLE_DB本身是个结构体, 保存了对打开对象的各进程的引用。它为每一个进程都维护了一个计数器。在第一遍中将HanInfo与此结构体相联结显示了对“懒惰”原则的遵守。如果对象只是从一个进程中打开, 则联结本身就是指向这个进程的指针。不可命名的对象可以是exclusive的(属性中的OBJ_EXCLUSIVE位)。这样控制了只能从一个与POLLED_QUOTA中的域对应的进程中打开对象。我们记得, 这个结构体总是存在于exclusive的对象中。这样, 进程就独自占据了对象——如此就没有了控制访问的问题。合情合理的, 对象独占的句柄能够继承(OBJ_INHERIT), 并被管理器检查。如果CREATOR_INFO是必需的信息, 则要有CREATOR_INFO链表, object type有相应的域指向它。object type本身也在这个链中。实际上, 我发现bCreatorInfo位只在类型对象“Type”中设置。这表明链表只用于对象类型。对于类型已定的对象, 则不含此链表。好了, 只剩下描述ObpIncrementUnnamedHandleCount(..)函数的主要内容的逻辑了。在该函数执行时还会调用OpenProcedure(如果有)。

当上述操作完成时, 会使首部中的计数器RefCounter增加RefCounterDelta+1。最后, 使用ExCreateHandle创建句柄。在成功创建句柄后, 就离开相应的OBJECT_INFO结构体, 这个结构体役期已满, 不再需要.....

07.句柄数据库

进程(也是个对象, 这个以后再说)有个与对象相联系的句柄数据库。知道了对象的句柄就可以很容易的对对象进行访问, 而同时也出现了继承和复制的问题。我们来详细研究一下句柄数据库。

我们不在'thread'和'process'对象的具体格式上做停顿, 演示如何得到指向句柄数据库的指针(自然, 要在内核模式下)。

```
mov eax,large fs:124h ; Got current thread body pointer
mov eax,[eax+44h]    ; Got current process body pointer
mov ecx,[eax+104h]   ; Got handle_db header pointer !
```

数据库的结构可以分成两部分——header和body(实际上就是数据库本身)。header的形式如下:

```
typedef struct _HANDLE_DB_HEADER{
    WORD Unknown0[2]; //0x0 - эти поля всегда были 0
    DWORD Unknown1; //0x4 - -----..-----
    SPIN_LOCK SpinLock; //0x8
    PVOID Head; //0xc
    PVOID Tail; //0x10
    DWORD HandleCount; //0x14
    PPROCESS pProcess; //0x18
    DWORD ProcessUniqueId;//0x1c
    WORD GrowDelta; //0x20
    LIST_ENTRY HandleTableList; //0x24 系统有个句柄数据库链表. 链表的
entry - HandleTableListHead
    KEVENT EventObj; //0x2c
    KSEMAPHORE SemaphoreObj; //0x3c
}HANDLE_DB_HEADER,*PHANDLE_DB_HEADER;
```

数据库本身形式如下:

```

+-----+
+-----|+-----|+-----+
+-->NullEntry>+|Han0|Han1|..+>| LIST_ENTRY<|>LIST_ENTRY>|.. |>LIST_ENTRY<|+
+
| +-----+-----+ +-----+-----+ |
+-----+
[Head]    [ Handles ]      [ Free spaces for handles ] [Tail]
```

DB首部中的Head指向数据库的起始位置, Tail指向末尾。数据库初始化时, 数据库的全部内存都填充为循环链表(除链表外没别的东西, 也就是说每一个元素都是LIST_ENTRY)。并且链表是从左到右依次建立的。当向数据库中添加句柄时, 句柄被加在NullEntry之后。head有指向链表中空闲位置的指针, 句柄就在此处添加。当没有空闲位置时(链表的head指向了自己), 就将数据库扩展GrowDelta个位置。当然, 在句柄数据库工作时会产生相应的限额的“装填”。所有这些都很简单.....


```

return 0;
}

ExLockHandleTableEntry(Table,TableEntry)
// Table not used

{

ObjectPointer = TableEntry->0x00;
if (!ObjectPointer) return 0;

if (ObjectPointer > 0)
{
    NewObjectPointer = ObjectPointer | 0x80000000;
    //xmpxchg    NewObjectPointer,ObjectPointer in ObjectPointer
}
else {
    // wait logic
}
return 1;
}

```

为了完整, 我给出用于句柄数据库的主要内部函数的原型。

用于创建数据库的基本函数:

```

NTSTATUS ExCreateHandleTable(
    PPROCESS Process,
    WORD Size,    //Optional
    WORD GrownDelta //Optional
);

```

这个函数用于数据库的扩展:

```

NTSTATUS ExpAllocateHandleTableEntries(
    PHANDLE_DB_HEADER HandleDB,
    PLIST_ENTRY Head,
    DWORD OldSize, // 在元素中
    DWORD NewSize
);

```

更高层的函数:

```

NTSTATUS ObInitProcess(
    PPROCESS pFatherProcess, //如果!=0 - 数据库是复制的.
    //否则 - 是创建的
    //包含audit
    PPROCESS pProcess
)

```

对象句柄本身是怎样的结构呢? 看这里:

```

typedef struct _OBJECT_HANDLE{

```

```
PVOID ObjectHeaderPointer,
ACCESS_MASK GrantedAccess
}OBJECT_HANDLE,*POBJECT_HANDLE;
```

正如我们看到的——句柄只是简单的一个指向对象首部的指针，再加上一个访问掩码。是的，ObjectHeaderPointer还保存了某些信息。问题在于所有对象的首部的地址都是8字节对齐的。这样，对于对对象的访问只需使用ObjectHeaderPointer的位[31:3]。低三位是OBJECT_HANDLE_INFORMATION结构体中的HandleAttributes(见DDK中的ObReferenceObjectByHandle函数)。各个位的值:0x01 - protect from close, 0x02 - inherit, 0x04 - generate on close。因此，使用句柄能进行有效的寻址。返回给用户的句柄值，左移两位后等于OBJECT_HANDLE结构体在数据库中的序号。在ntdef.h中，OBJ_HANDLE_TAGBITS定义的附近可以看到以下几行：

```
//
// Low order two bits of a handle are ignored by the system and available
// for use by application code as tag bits. The remaining bits are opaque
// and used to store a serial number and table index.
//
```

怎么样，全明白了吧。

如果您还没忘的话，我们只是分析过ObInsertObject的最简单的情况，即非命名对象的情况。如果一定要将对象放入命名树中会怎样呢？简单讲就是对命名树进行操作，最终将对象放入目录中，这样就可以用名字打开对象了。如果这个目录是系统树的根目录，则这个对象就对所有进程可见。如果这个目录是某个进程的一——则只对该进程可见(通常所有这样的对象都在系统树中创建，因为大多数系统服务都使用系统树)。对此，为了展示进程的完整格局，我们先来分析清楚某些问题.....

08.对象的安全性

=====

之前，我一直都在回避这个问题，因为在前面不涉及这方面的东西是可以的。但是，要向下继续的话，就要知道一些了。幸运的是，这里要讲的大多数结构体都在这个或那个头文件中有描述，是公开的。但是，我觉得将这些必需的信息结合在一起是合乎情理的，因为下面我将经常引用这些结构体。有时，我会讲到一些未公开的东西。在创建时，帐户记录生成一个随机的SID，这样就保证了SID的随机性。

为了识别认证用户和组，安全系统使用了SID(Security Identifier)。

```
typedef struct _SID_IDENTIFIER_AUTHORITY {
    BYTE Value[6];
} SID_IDENTIFIER_AUTHORITY, *PSID_IDENTIFIER_AUTHORITY;
```

```
typedef struct _SID {
    BYTE Revision;
    BYTE SubAuthorityCount;
    SID_IDENTIFIER_AUTHORITY IdentifierAuthority;
    DWORD SubAuthority[ANYSIZE_ARRAY];
} SID, *PISID;
```

当前Revision的值为1。SubAuthorityCount的最大值为15。SubAuthority域最后一个值为RID(Relativ. Id.)。从文本的角度看，NT的SID的形式为S-1-A-SA1-SA2-...-RID。其中S-

1表示SID rev.1, A - IdentifierAuthority, SA? - SubAuthority, RID - SubAuthority的最后一项。在NT中有标识的预定义, 同时也有RID值的预定义。例如, RID等于500表示帐户administrator。详细信息可参考WINNT.H。

用户可被授予某种特权级SeXXXPrivilege, 每一个特权级在系统中都是8位数字的LUID(Local Unique Id.)。起初, 每一个特权级都是一个文本字符串(特权级的文本表示可以参见WINNT.H)。

这个信息不依赖于环境。在用户进入系统时, LSA(Local Security Administrator)创建访问令牌(TOKEN), 访问令牌包含着用于安全检查的重要信息。该结构体形式如下:

```
typedef struct _ACCESS_TOKEN{
    char SourceName[8]; //0 Source of token'a
    LUID SourceIdentifier; //8
    LUID TokenId; //10
    LUID AuthenticationId; //18
    LARGE_INTEGER ExpirationTime; //20
    LUID ModifiedId; //28 修改token'a时改变
    DWORD NumberOfUserAndGroupSids; //30
    DWORD NumberOfPrivileges //34
    DWORD Unknown; //38
    DWORD DynamicCharged; //3c
    DWORD DynamicAvailable; //40
    DWORD NumberOfOwnerSid //44
    PVOID SidDB; //48
    PSID PrimaryGroup; //4c
    PLUID_AND_ATTRIBUTES TokenPrivileges; //50
    DWORD Unknown1; //54
    PACL DefaultDacl; //58
    TOKEN_TYPE TokenType; //5c 原始的还是
    // IMPERSONATION
    SECURITY_IMPERSONATION_LEVEL ImpLevel; //60 IMPERSONATION
    DWORD UnknownFlag; //64
    DWORD Tail[ANYSIZE_ARRAY]; //???
}ACCESS_TOKEN,*PACCESS_TOKEN;
```

数据库SidDB大约是下面这个样子:

```

+-TokenUser-----+
|00 PSID      |
|04 Attributes |
+-TokenGroups--+
|....      |
|          |
|          |
+-----+
      <--NumberOfUserAndGroupSids

+-OwnerSid-----+
|          | <--NumberOfOwnerSid
+-----+
|....
```

我再给出与之相应的未公开的函数, 其用于取得'Token'对象。

```
PACCESS_TOKEN NTOSKRNL PsReferencePrimaryToken(KPROCESS* Process);
```

9

SecurityDescriptor描述了对对象的安全性——这是需要说的。

Control指定了不同的标志, 其中一个指示描述符**Self Relative**(就是说其中所有的指针都是相对于结构体首部的)。详细描述见**WINNT.H**。域**Owner**和**Group**的值的意义是显然的。**Dacl**——一枚举用户和组的链表, 通过其来允许或是禁止对对象的访问。**Sacl**与**audit**密切相关。与**ACL**有关的结构体在**WINNT.H**中都有很好的描述。所以, 描述用于各种用户的对象访问权限的信息与对象密切相联(在对象首部中有指向安全描述符的指针)。

在取得对对象的访问权时(如ObInsertObject)会被赋予ACCESS_MASK。通常,这时要使用相应的类型定义STANDARD_RIGHTS_READ等等。现在,重要的是来看一下这个信息,该信息的位置结构体是下面这个样子:

```

3 3 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-----+-----+-----+
|G|G|G|G|Res'd|A| StandardRights| SpecificRights |
|R|W|E|A| |S| | | |
+---+---+---+---+---+---+---+---+
||| | | | | |
||| | | | | SpecificRights;
||| | | | | StandardRights;
||| | | | | AccessSystemAcl
||| +GenericAll
|| +--GenericExecute

```

```
| +----GenericWrite
+-----GenericRead
```

Generic类的位是为了方便程序员(都是标准的而且很简单)。实际上, 在对象类型结构体中有**GenericMapping**域, 该域包含着关于在**SpecificRights**(特殊权限——此位的含义依赖于对象类型)中转换**Generic**位的信息。**StandardRights**, 从名字就可看出, 它对于所有的对象有相同的含义。对象句柄有个**GrantedAccess**域, 这样, 在使用句柄时不必总是解析安全描述符, 这里的信息就足够了。除此之外, 对象类型还包含着**AllowedAccessMask**, 这样就可以控制对于对象类型的上下文所进行的可能的访问。

在内部, NT使用了下面这样一些结构体, 如**ACCESS_STATE**。

```
typedef struct _ACCESS_STATE {
    LUID OperationID;
    BOOLEAN SecurityEvaluated;
    BOOLEAN GenerateAudit;
    BOOLEAN GenerateOnClose;
    BOOLEAN PrivilegesAllocated;
    ULONG Flags;
    ACCESS_MASK RemainingDesiredAccess;
    ACCESS_MASK PreviouslyGrantedAccess;
    ACCESS_MASK OriginalDesiredAccess;
    SECURITY_SUBJECT_CONTEXT SubjectSecurityContext;
    PSECURITY_DESCRIPTOR SecurityDescriptor;
    PVOID AuxData;
    union {
        INITIAL_PRIVILEGE_SET InitialPrivilegeSet;
        PRIVILEGE_SET PrivilegeSet;
    } Privileges;

    BOOLEAN AuditPrivileges;
    UNICODE_STRING ObjectName;
    UNICODE_STRING ObjectTypeNames;
} ACCESS_STATE, *PACCESS_STATE;

typedef struct _SECURITY_SUBJECT_CONTEXT {
    PACCESS_TOKEN ClientToken;
    SECURITY_IMPERSONATION_LEVEL ImpersonationLevel;
    PACCESS_TOKEN PrimaryToken;
    PVOID ProcessAuditId;
} SECURITY_SUBJECT_CONTEXT, *PSECURITY_SUBJECT_CONTEXT;
```

正像我们看到的, 这个结构体在其中结合了在访问对象时所有用于控制对象访问的必需的信息。我给出几个与之相关的未公开函数。

```
// 填充ACCESS_STATE结构体
// 留下了一个关于优先级和对象的信息未填充
```

```
NTSTATUS NTOSKRNL SeCreateAccessState(
    PACCESS_STATE AccessState,
    PVOID AuxData,
    ACCESS_MASK Access,
    PGENERIC_MAPPING GenericMapping
);
```

```
NTSTATUS NTOSKRNL SeDeleteAccessState(PACCESS_STATE AccessState);
```

```
// 填充SUBJECT_CONTEXT, 用在SeCreateAccessState
```

```
void NTOSKRNL SeCaptureSubjectContext(  
    PSUBJECT_CONTEXT SubjectContext  
);
```

在DDK中同时描述了ObGetObjectSecurity和ObReleaseObjectSecurity函数。其中第一个函数使用代号1调用了方法SecurityProcedure(参见对象类型结构体的格式)。如果有安全描述符,则这个方法负责收回。否则就通过对象首部来取得描述符。

09.命名对象

=====

我们来研究ObInsertObject的第二个分支,该分支用于处理命名对象。如果对象是未命名的或是exclusive的,则其不需要安全描述符。否则可以通过名字访问对象。

在开始部分,函数创建ACCESS_STATE结构体并在以OBJECT_INFO.RootDirectory(这个结构体当时是存在的)为根在树中查找这个名字。查找是通过ObpLookupObjectName函数进行的。其工作逻辑十分显然——解析相应对象的Directory和SymbolicLink。一般ObpLookupObjectName不只用于查找,还用于在树中创建对象。在选择路径时同时要用安全检查点(ObpCheckTraverseAccess函数)在树中验证对象的访问权。到目前,暂时还没有结束对路径的解析或是在路径中没有遇到对象管理器不知道的对象(这叫做另一类的名字空间)。这时调用方法ParseProcedure(如果有的话),其中的一个参数是指向未被选中的路径的指针。下面的分析要用到这个方法。使用Parse方法可以查找整个名字空间,但对象可能只在其“亲本”的树中。ParseProcedure方法只用于查找并在在树中创建对象时被忽略(如果它没有指向默认的函数ObpParseSymbolicLink)。但符号链接对于对象描述符来说是自有的,而且总是可以被处理的。如果发生了名字冲突(对象已存在),则ObInsertObject的进一步的动作就依赖于OBJ_OPENIF标志。这个标志可被译为open if exist。其它的词将打开现有的对象。一般情况下正常的控制流在对象创建的最后会调用未公开函数ObAssignSecurity(其“逆”函数ObGetObjectSecurity是公开的)。这个函数被封装在了公开函数SeAssignSecurity(...)中。

```
BOOLEAN NTOSKRNL ObAssignSecurity(  
    PACCESS_STATE AccessState,  
    PSECURITY_DESCRIPTOR OPTIONAL OldSecurityDescriptor,  
    PVOID Object,  
    POBJECT_TYPE Type);
```

参数SecurityDescriptor只用作传递给SecurityProcedure方法的参数,函数用代号3来调用此方法。所有的必需的安全管理信息都在ACCESS_STATE结构体中。在该函数完成后,对象首部中的SecurityDescriptor域就被填充好(对于非命名对象则无此项工作)。然后调用内部函数ObpCreateHandle。尽管该函数未被导出,但是为了比较函数ObpCreateHandle和ObpCreateUnnamedHandle,我在这里给出其原型。

```
NTSTATUS ObpCreateHandle(  
    DWORD SecurityMode, // 通常==1, 访问在这里被检查  
    PVOID Object,  
    POBJECT_TYPE Type,  
    PACCESS_STATE AccessState,  
    DWORD RefCounterDelta,  
    DWORD Attributes,  
    DWORD OPTIONAL UnknErrMode,
```

```
KPROCESSOR_MODE bMode,
PVOID Object,
PHANDLE Handle
);
```

```
NTSTATUS ObpCreateUnnamedHandle(
    PVOID Object,
    ACCESS_MASK Access,
    DWORD RefCounterDelta,
    DWORD Attributes,
    KPROCESSOR_MODE bMode,
    PVOID Object,
    PHANDLE Handle
);
```

两个函数做的工作实际上是相同的，最终都要调用**ExCreateHandle(..)**。但是，从原型中可以直观的看出，第一个函数处理的是安全描述符，创建句柄的过程我已经描述过，我们都已经知道了。最后，**OBJECT_INFO**结构体被释放。我已经说过这些结构体都是分配在**LookasideList**中的，现在明白是为什么了吧。在系统中经常会对象被创建和删除，对**OBJECT_INFO**的创建工作会大量进行(这些结构体从对象被创建起就存在于每一个对象中)。**LookasideList**专门用于优化这类经常用到的内存分配/回收操作(见**DDK**)。

0a.对象的消亡

=====

如果对象管理器不再需要某对象，或对象不是**permanent**(**OBJ_PERMANENT**位)的，则管理器就将对象删除。这里对象的消亡分为两步。在首部中有两个域，**RefCounter**和**HandleCounter**。在句柄关闭时(**ZwClose**)会依次调用**ExDeleteHandle**和**ObpDecrementHandleCount**。第一个函数只是从句柄数据库中删除句柄(释放的空间被返回给链表的空闲位置)。**ObpDecrementHandleCount**包含着关于句柄打开数量的信息，同时保存着句柄对象数据库的信息(**HANDLE_DB**)。除此之外，在这些函数中还要调用方法**CloseProcedure**。如果关闭的是最后一个句柄且对象不是**permanent**的，则调用**ObpDeleteNameCheck**函数，该函数从对象名树中将有关该对象的信息删除。这时，以代号**delete**调用方法**SecurityProcedure**。但是对象继续存在——它只是不再可见。

除此之外，在首部中还有对对象的引用计数域，这个域通过**ObDereferenceObject**函数来递减。(在递增/递减句柄的引用计数时该域也同时递增/递减，但可以修改对象的引用计数而不改变句柄的数量)。如果引用计数值达到了零，则一般会在删除队列中添加这个对象(这要依情况而定，对象可以被立即删除)。这时，**RefCount**和**HandleCount**就扮演了**LIST_ENTRY**的角色。对于这些域，我没有在对象首部的格式中将它们结合起来，以不使内容过快的复杂化。在直接删除对象后就要用代号2调用方法**SecurityProcedure**和**DeleteProcedure**。现在所提到的所有的内核函数，除了**ObDereferenceObject**都是内部函数，它们的原型我就不给出了，都是已知的。

0b.其它管理器函数

=====

实际上，对**ObInsertObject**函数的研究打开了对象管理器逻辑的主要部分。我们来简要的研究一下剩下的管理器导出函数。

DDK文档中有以下函数：

ObReferenceObject, ObDereferenceObject, ObGetObjectSecurity,
ObReleaseObjectSecurity, ObReferenceObjectByHandle,
ObReferenceObjectByPointer.

从已知的信息来看, 在这些函数里的管理器的工作逻辑都是明显的。其工作就是处理对象首部、句柄数据库和同步内部对象。剩下的一些函数我们不再讨论:

ObCheckCreateObjectAccess , ObCheckObjectAccess,
ObFindHandleForObject, ObGetObjectPointerCount,
ObMakeTemporaryObject, ObOpenObjectByName, ObOpenObjectByPointer,
ObQueryNameString, ObQueryObjectAuditingByHandle,
ObReferenceObjectByName, ObSetSecurityDescriptorInfo

关于所有这些函数的信息都是已知的。

```
// 将完整的对象名返回到参数中
// 该参数就是用于保存UNICODE_STRING和名字的缓冲区
// 通常开始先不用参数Result调用, 以获得
// 缓冲区的长度。
// 如果定义了方法QueryName - 则调用它
```

```
NTSTATUS NTOSKRNL ObQueryNameString(
    PVOID Object,
    PUNICODE_STRING Result OPTIONAL,
    DWORD Len,
    PDWORD RetLen OPTIONAL
);
```

```
// 调用ObpLookupObjectName
```

```
NTSTATUS NTOSKRNL ObReferenceObjectByName(
    PUNICODE_NAME Name,
    DWORD Attributes, // 经常使用OBJ_CASE...
    PACCESS_STATE AccessState OPTIONAL,
    ACCESS_MASK Access,
    POBJECT_TYPE Type,
    KPROCESSOR_MODE bMode,
    DWORD Unknown OPTIONAL, // 与其说是保留,
    // 不如说是只用作SecurityProcedure的参数
    // 总是为0
    PVOID BodyPointer
);
```

```
// ObpLookupObjectName/ ObpCreateHandle的主要逻辑
```

```
NTSTATUS ObOpenObjectByName(
    POBJECT_ATTRIBUTES Attributes,
    POBJECT_TYPE Type,
    KPROCESSOR_MODE bMode,
    PACCESS_STATE AccessState OPTIONAL,
    ACCESS_MASK Access,
    DWORD Unknown OPTIONAL, // 见ObReferenceObjectByName中的unknown
    PHANDLE Handle OUT
);
```

// ObReferenceObjectByPointer / ObpCreateHandle的主要逻辑

```
NTSTATUS NTOSKRNL ObOpenObjectByPointer(  
    PVOID Object,  
    DWORD Attributes,  
    PACCESS_STATE AccessState OPTIONAL,  
    ACCESS_MASK Access,  
    POBJECT_TYPE Type,  
    KPROCESSOR_MODE bMode,  
    PHANDLE Handle OUT  
);
```

```
// 函数代码 :  
// mov eax,[esp+pObject]  
// mov eax,[eax-18h]  
// ret 4  
// 完了!
```

DWORD NTOSKRNL ObGetObjectPointerCount(PVOID Object);

```
// 有趣的函数. 一般不用在内核中同时也未公开.  
// 对于域Object, Type和HandleAttributes其值可以为0,  
// 此时将查找任意符合的值  
// 此函数可以查找关于句柄的信息
```

```
BOOLEAN NTOSKRNL ObFindHandleForObject(  
    PPROCESS Process,  
    PVOID Object,  
    POBJECT_TYPE Type,  
    DWORD HandleAttributes, //低3位. 见 '句柄表'          PHANDLE Handle O  
    UT // 返回句柄  
);
```

```
// 此函数清除对象首部中的相应标志  
// 如果不再引用该对象, 则允许将其删除, 并调用内部函数ObpDeleteNameCheck
```

```
NTSTATUS NTOSKRNL ObMakeTemporaryObject(  
    PVOID Object  
);
```

// ObGetObjectSecurity/SeAccessCheck的主要逻辑

```
BOOLEAN NTOSKRNL ObCheckObjectAccess(  
    PVOID Object,  
    PACCESS_STATE AccessState,  
    BOOLEAN bTypeLocked,  
    KPROCESSOR_MODE Mode,  
    OUT PNTSTATUS AccessStatus  
);
```

// 当在树中创建对象时, 在ObpLookupObjectName中被调用

```
BOOLEAN NTOSKRNL ObCheckCreateObjectAccess(  
    PDIRECTORY Directory,
```

```

        ACCESS_MASK Access,
        PACCESS_STATE AccessState,
        PUNICODE NameFromDir,
        DWORD UnknLockedFlags,
        KPROCESSOR_MODE Mode,
        OUT PNTSTATUS AccessStatus
    );

// последовательность ExMapHandleToPointer/取出位0x4 (见'句柄数
// 据库')

BOOLEAN NTOSKRNL ObQueryObjectAuditingByHandle(
    HANDLE Handle,
    PDWORD ReturnVal);

// 此函数在方法SpDefaultObjectMetod中调用, 用于代号0 - set security descriptor.
// 该函数的工作是基于SeSetSecurityDescriptorInfo函数的。现在我们不详细研究安全描述
// 符, 除此之外, 某些参数还不明了,
// 故此函数的原型我就不给出了
// ObSetSecurityDescriptorInfo(...);

```

Не останавливаться на бу
 дущем -
 Я понял, что его не бу
 дет
 И ты знаешь, когда я уйду
 -
 Ты услышишь мой плач в ветре...
 (c) by Anathema

0с.结语
 =====

实际上, 对象管理器的大多数导出函数都不是完全functional的, 这点现在是完全显然的了。
 一般来说, 如果提出关于可行性的问题, 可以表达这样的见解。Kernel驱动并不使用句柄。
 一般说来, 在普通情况下并不知道位于驱动代码在哪个进程的上下文中执行。实际使用的
 函数允许用名字或指针来引用对象。

对于剩下的函数, 如果我们没写过系统服务的话, 是很难想到其真实的用法的(但是完全可
 能的)。对于实现Windows NT的潜力, 管理器运作的知识总是很有益的。我们怎样确信处理
 对象的系统是十分灵活的——要知道NT的执行系统要使用它来模拟不同的操作系统。当然,
 本文可能还有不准确的地方。任何评论意见请发至peter_k@vivos.ru (Peter Kosyh)。

Best regards, Gloomy

附录

13.从用户模式下获取对象信息
 =====

在Windows NT中有许多系统调用以Query或QueryInformation命名。这些函数都能获得有趣的

信息, 而且大多是未公开的。下面给出NtQueryObject函数的原型, 使用该函数能轻松获取关于对象格式的信息。

```
typedef enum _OBJECTINFOCLASS {
    BaseObjectInfo,
    NameObjectInfo,
    TypeObjectInfo,
    AllTypesObjectInfo,
    HandleObjectInfo
} OBJECTINFOCLASS;

NTSYSAPI NTSTATUS NTAPI NtQueryObject(
    HANDLE ObjHandle,
    OBJECTINFOCLASS ObjectInfoClass,
    OUT PVOID ObjectInfo, // 信息缓冲区
    DWORD ObjectInfoLen, // 缓冲区长度
    OUT PDWORD RetInfoLen // 返回信息的长度
);

typedef struct _BASE_OBJECT_INFO{
    DWORD HandleAttributes,
    ACCESS_MASK GrantedAccess,
    DWORD HandleCount,
    DWORD RefCount,
    DWORD PagedPollQuota,
    DWORD NonPagedPollQuota,
    DWORD ReservedAndAlwaysNull[3],
    DWORD NameObjectInfoLength,
    DWORD TypeObjectInfoLength,
    DWORD SecurityDescriptorLengh,
    LARGE_INTEGER SymLinkCreationTime //自1601年起的时间
};

typedef struct _NAME_OBJECT_INFO{
    UNICODE_STRING Name;
    // 这里应当是名字的位置. 参数ObjectInfo可能为0
    // 以获取缓冲区的大小
}NAME_OBJECT_INFO;

typedef struct _TYPE_OBJECT_INFO{
    UNICODE_STRING Name;
    DWORD InstanceCount;
    DWORD HandleCount;
    DWORD PeakObjects;
    DWORD PeakHandles;
    DWORD AllowedAttributesMask;
    GENERIC_MAPPING GenericMapping;
    DWORD AllowedAccessMask;
    BOOLEAN bInNameSpace;
    BOOLEAN bHandleDBInfo;
    BOOLEAN Align[2];
    DWORD Unknown6; // 见对象类型的unknown6域
    DWORD DefaultPagedPollQuota;
    DWORD DefaultNonPagedPollQuota;
}TYPE_OBJECT_INFO;
```

```
typedef struct _ALL_TYPES_OBJECT_INFO{
    DWORD NumOfTypes;
    TYPE_OBJECT_INFO [ANY_SIZE_ARRAY];
}ALL_TYPES_OBJECT_INFO;
```

```
typedef struct _HANDLE_OBJECT_INFO{
    BOOLEAN Inherit;
    BOOLEAN ProtectFromClose;
}HANDLE_OBJECT_INFO;
```

14.某些与对象管理相关的系统服务

=====

在Недокументированные возможности Windows NT一书中

A.B. Коберниченко给出了某些系统服务接口的原型。作者的目的是描述未公开系统调用的原型，于是从kernel32.dll中使用的系统调用出发。大概对于这个目的，这是个捷径(恰巧，这本书也试图描述NtQueryObject函数，但是其描述在这里是不适用的，而且书中的函数的描述很不完整)。对于某些函数的反汇编，我检查了它的输出并确信定义的正确性。下面我给出某些调用，因为作者对它们进行了描述。

//用于任何对象的函数

```
NTSYSAPI NTSTATUS NTAPI
    NtClose(IN HANDLE Handle);
```

```
NTSYSAPI NTSTATUS NTAPI NtMakeTemporaryObject(
    IN HANDLE Handle
);
```

```
#define DUPLICATE_CLOSE_SOURCE    0x00000001
#define DUPLICATE_SAME_ACCESS     0x00000002
```

```
NTSYSAPI NTSTATUS NTAPI
NtDuplicateObject(
    IN HANDLE SourceProcessHandle,
    IN HANDLE SourceHandle,
    IN HANDLE TargetProcessHandle,
    OUT PHANDLE TargetHandle OPTIONAL,
    IN ACCESS_MASK DesiredAccess,
    IN ULONG Attributes,//OBJ_***
    IN ULONG Options
);
```

//对象目录

```
#define DIRECTORY_QUERY            (0x0001)
#define DIRECTORY_TRAVERSE        (0x0002)
#define DIRECTORY_CREATE_OBJECT   (0x0004)
#define DIRECTORY_CREATE_SUBDIRECTORY (0x0008)
```

```
#define DIRECTORY_ALL_ACCESS (STANDARD_RIGHTS_REQUIRED | 0xF)
```

```

NTSYSAPI NTSTATUS NTAPI
NtCreateDirectoryObject(
    OUT PHANDLE DirectoryHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes
);

```

```

NTSYSAPI NTSTATUS NTAPI
NtOpenDirectoryObject(
    OUT PHANDLE DirectoryHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes
);

```

```

typedef struct _OBJECT_NAME_TYPE_INFO {
    UNICODE_STRING ObjectName;
    UNICODE_STRING ObjectType;
} OBJECT_NAME_TYPE_INFO, *POBJECT_NAME_TYPE_INFO;

```

```

typedef enum _DIRECTORY_INFORMATION_CLASS {
    ObjectArray,
    ObjectByOne
} DIRECTORY_INFORMATION_CLASS, *PDIRECTORY_INFORMATION_CLASS;

```

```

NTSYSAPI NTSTATUS NTAPI
NtQueryDirectoryObject(
    IN HANDLE DirectoryObjectHandle,
    OUT PVOID ObjectInfoBuffer,
    IN ULONG ObjectInfoBufferLength,
    IN DIRECTORY_INFORMATION_CLASS DirectoryInformationClass,
    IN BOOLEAN First,
    IN OUT PULONG ObjectIndex,
    OUT PULONG LengthReturned
);

```

//对象符号链接

```

#define SYMBOLIC_LINK_QUERY (0x0001)

```

```

#define SYMBOLIC_LINK_ALL_ACCESS (STANDARD_RIGHTS_REQUIRED | 0x1)

```

```

NTSYSAPI NTSTATUS NTAPI
NtCreateSymbolicLinkObject(
    OUT PHANDLE ObjectHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    IN PUNICODE_STRING SubstituteString
);

```

```

NTSYSAPI NTSTATUS NTAPI
NtOpenSymbolicLinkObject(
    OUT PHANDLE ObjectHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes
);

```

```

);

typedef struct _OBJECT_NAME_INFORMATION {
    UNICODE_STRING Name;
} OBJECT_NAME_INFORMATION, *POBJECT_NAME_INFORMATION;

NTSYSAPI NTSTATUS NTAPI
NtQuerySymbolicLinkObject(
    IN HANDLE ObjectHandle,
    OUT POBJECT_NAME_INFORMATION SubstituteString,
    OUT PULONG SubstituteStringLength //字节
);

```

我想, 现在领悟描述过的大部分管理器系统调用行为应该不难了.....

在网上可以找到 "**Недокументированные возможности Windows NT**" 一书的例子——作者做了一件大好事。在源代码中描述了大部分未公开的系统调用以及给出的例子的用法。