在 WIN64 系统上运行的原生 64 位应用程序，其 PE 格式称为 PE32+，当然是和 32 位程序有所不同的。但是，相同的部分是占大多数的，**不同的地方**，基本只是在 PE 头部分而已（IMAGE_NT_HEADERS64）。所以，本文只讲述 PE32 和 PE32+不同的地方，相同的地方就略过不讲了。本文所有的资料均来源于微软官方的电子书籍《Microsoft 可移植可执行文件和通用目标文件格式文件规范》和 WDK7 自带的 ntimage.h，所以在数据的准确性方面应该是没有问题的。

首先对比一下 IMAGE_NT_HEADERS32 和 IMAGE_NT_HEADERS64：

```
typedef struct _IMAGE_NT_HEADERS {
    ULONG Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

```
typedef struct _IMAGE_NT_HEADERS64 {
    ULONG Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER64 OptionalHeader;
} IMAGE_NT_HEADERS64, *PIMAGE_NT_HEADERS64;
```

可见只是“可选头”部分不同。再对比 IMAGE_OPTIONAL_HEADER32 和 IMAGE_OPTIONAL_HEADER64 的异同：

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    USHORT  Magic;
    UCHAR   MajorLinkerVersion;
    UCHAR   MinorLinkerVersion;
    ULONG   SizeOfCode;
    ULONG   SizeOfInitializedData;
    ULONG   SizeOfUninitializedData;
    ULONG   AddressOfEntryPoint;
    ULONG   BaseOfCode;
    ULONG   BaseOfData;
    //以上是标准域，以下是特定域
    ULONG   ImageBase;
    ULONG   SectionAlignment;
    ULONG   FileAlignment;
    USHORT  MajorOperatingSystemVersion;
    USHORT  MinorOperatingSystemVersion;
    USHORT  MajorImageVersion;
    USHORT  MinorImageVersion;
    USHORT  MajorSubsystemVersion;
    USHORT  MinorSubsystemVersion;
    ULONG   Win32VersionValue;
    ULONG   SizeOfImage;
    ULONG   SizeOfHeaders;
    ULONG   CheckSum;
    USHORT  Subsystem;
    USHORT  DllCharacteristics;
    ULONG   SizeOfStackReserve;
    ULONG   SizeOfStackCommit;
```

```
typedef struct _IMAGE_OPTIONAL_HEADER64 {
    USHORT      Magic;
    UCHAR       MajorLinkerVersion;
    UCHAR       MinorLinkerVersion;
    ULONG       SizeOfCode;
    ULONG       SizeOfInitializedData;
    ULONG       SizeOfUninitializedData;
    ULONG       AddressOfEntryPoint;
    ULONG       BaseOfCode;
    //以上是标准域，以下是特定域
    ULONGLONG   ImageBase;
    ULONG       SectionAlignment;
    ULONG       FileAlignment;
    USHORT      MajorOperatingSystemVersion;
    USHORT      MinorOperatingSystemVersion;
    USHORT      MajorImageVersion;
    USHORT      MinorImageVersion;
    USHORT      MajorSubsystemVersion;
    USHORT      MinorSubsystemVersion;
    ULONG       Win32VersionValue;
    ULONG       SizeOfImage;
    ULONG       SizeOfHeaders;
    ULONG       CheckSum;
    USHORT      Subsystem;
    USHORT      DllCharacteristics;
    ULONGLONG   SizeOfStackReserve;
    ULONGLONG   SizeOfStackCommit;
    ULONGLONG   SizeOfHeapReserve;
```

```
    ULONG    SizeOfHeapReserve;
    ULONG    SizeOfHeapCommit;
    ULONG    LoaderFlags;
    ULONG    NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY
DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES
];
} IMAGE_OPTIONAL_HEADER32,
*PIMAGE_OPTIONAL_HEADER32;
```

```
    ULONGLONG    SizeOfHeapCommit;
    ULONG        LoaderFlags;
    ULONG        NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY
DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES
];
} IMAGE_OPTIONAL_HEADER64,
*PIMAGE_OPTIONAL_HEADER64;
```

可见两者基本上是相同的，具体各部分含义如下（以下内容来自《Microsoft 可移植可执行文件和通用目标文件格式文件规范》）。

首先是标准域部分：

偏移	大小	域	描述
0	2	Magic	这个无符号整数指出了映像文件的状态。最常用的数字是 0x10B，它表明这是一个正常的可执行文件。0x107 表明这是一个 ROM 映像，0x20B 表明这是一个 PE32+可执行文件。
2	1	MajorLinkerVersion	链接器的主版本号。
3	1	MinorLinkerVersion	链接器的次版本号。
4	4	SizeOfCode	代码节（.text）的大小。如果有多个代码节的话，它是所有代码节的和。
8	4	SizeOfInitializedData	已初始化数据节的大小。如果有多个这样的数据节的话，它是所有这些数据节的和。
12	4	SizeOfUninitializedData	未初始化数据节（.bss）的大小。如果有多个.bss 节的话，它是所有这些节的和。
16	4	AddressOfEntryPoint	当可执行文件被加载进内存时其入口点相对于映像基址的偏移地址。对于一般程序映像来说，它就是启动地址。对于设备驱动程序来说，它是初始化函数的地址。入口点对于 DLL 来说是可选的。如果不存在入口点的话，这个域必须为 0。
20	4	BaseOfCode	当映像被加载进内存时代码节的开头相对于映像基址的偏移地址。

PE32 中在 BaseOfCode 域后面是下面这个附加域，它并不存在于 PE32+中：

偏移	大小	域	描述
24	4	BaseOfData	当映像被加载进内存时数据节的开头相对于映像基址的偏移地址。

特定域部分：

偏移（PE32/PE32+）	大小（PE32/PE32+）	域	描述
28/24	4/8	ImageBase	当加载进内存时映像的第一个字节的首选地址。它必须是 64K 的倍数。DLL 默认是 0x10000000。Windows CE EXE 默认是 0x00010000。Windows NT、Windows 2000、Windows XP、Windows 95、Windows 98 和 Windows Me 默认是 0x00400000。
32/32	4	SectionAlignment	当加载进内存时节的对齐值（以字节计）。它必须大于或等于 FileAlignment。默认是相应系统的页面大小。
36/36	4	FileAlignment	用来对齐映像文件的节中的原始数据的对齐因子（以字节计）。它应该是界于 512 和 64K 之间的 2 的幂（包括这两个边界值）。默认是 512。如果 SectionAlignment 小于相应系统的页面大小，那么 FileAlignment 必须与 SectionAlignment 匹配。
40/40	2	MajorOperatingSystemVersion	所需操作系统的主版本号。
42/42	2	MinorOperatingSystemVersion	所需操作系统的次版本号。
44/44	2	MajorImageVersion	映像的主版本号。
46/46	2	MinorImageVersion	映像的次版本号。
48/48	2	MajorSubsystemVersion	子系统的主版本号。
50/50	2	MinorSubsystemVersion	子系统的次版本号。
52/52	4	Win32VersionValue	保留，必须为 0。
56/56	4	SizeOfImage	当映像被加载进内存时的大小（以字节计），包括所有的文件头。它必须是 SectionAlignment 的倍数。
60/60	4	SizeOfHeaders	MS-DOS 占位程序、PE 文件头和节头的总大小，向上舍入为 FileAlignment 的倍数。
64/64	4	CheckSum	映像文件的校验和。计算校验和的算法被合并到了 IMAGEHLP.DLL 中。以下程序在加载时被校验以确定其是否合法：所有的驱动程序、任何在引导时被加载的 DLL 以及加载进关键 Windows 进程中的 DLL。
68/68	2	Subsystem	运行此映像所需的子系统。
70/70	2	DllCharacteristics	DLL特征。

偏移（PE32/PE32+）	大小（PE32/PE32+）	域	描述
72/72	4/8	SizeOfStackReserve	保留的堆栈大小。只有 SizeOfStackCommit 指定的部分被提交；其余的每次可用一页，直到到达保留的大小为止。
76/80	4/8	SizeOfStackCommit	提交的堆栈大小。
80/88	4/8	SizeOfHeapReserve	保留的局部堆空间大小。只有 SizeOfHeapCommit 指定的部分被提交；其余的每次可用一页，直到到达保留的大小为止。
84/96	4/8	SizeOfHeapCommit	提交的局部堆空间大小。
88/104	4	LoaderFlags	保留，必须为 0。
92/108	4	NumberOfRvaAndSizes	可选文件头其余部分中数据目录项的个数。每个数据目录描述了一个表的位置和大小。

一些常数：

Windows 子系统

为可选文件头的 Subsystem 域定义了以下值以确定运行映像所需的 Windows 子系统（如果存在）：

常量	值	描述
IMAGE_SUBSYSTEM_UNKNOWN	0	未知子系统
IMAGE_SUBSYSTEM_NATIVE	1	设备驱动程序和 Native Windows 进程
IMAGE_SUBSYSTEM_WINDOWS_GUI	2	Windows 图形用户界面（GUI）子系统
IMAGE_SUBSYSTEM_WINDOWS_CUI	3	Windows 字符模式（CUI）子系统
IMAGE_SUBSYSTEM_POSIX_CUI	7	Posix 字符模式子系统
IMAGE_SUBSYSTEM_WINDOWS_CE_GUI	9	Windows CE
IMAGE_SUBSYSTEM_EFI_APPLICATION	10	可扩展固件接口（EFI）应用程序
IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER	11	带引导服务的 EFI 驱动程序
IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER	12	带运行时服务的 EFI 驱动程序
IMAGE_SUBSYSTEM_EFI_ROM	13	EFI ROM 映像
IMAGE_SUBSYSTEM_XBOX	14	XBOX

DLL 特征

为可选文件头的 DllCharacteristics 域定义了以下值：

常量	值	描述
	0x0001	保留，必须为 0。
	0x0002	保留，必须为 0。
	0x0004	保留，必须为 0。
	0x0008	保留，必须为 0。
IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE	0x0040	DLL 可以在加载时被重定位。
IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY	0x0080	强制进行代码完整性校验。
IMAGE_DLLCHARACTERISTICS_NX_COMPAT	0x0100	映像兼容于 NX。
IMAGE_DLLCHARACTERISTICS_NO_ISOLATION	0x0200	可以隔离，但并不隔离此映像。
IMAGE_DLLCHARACTERISTICS_NO_SEH	0x0400	不使用结构化异常（SE）处理。在此映像中不能调用 SE 处理程序。
IMAGE_DLLCHARACTERISTICS_NO_BIND	0x0800	不绑定映像。
	0x1000	保留，必须为 0。
IMAGE_DLLCHARACTERISTICS_WDM_DRIVER	0x2000	WDM 驱动程序。
IMAGE_DLLCHARACTERISTICS_ TERMINAL_SERVER_AWARE	0x8000	可以用于终端服务器。

在整个可选头结构体里，最引人关注的恐怕就是 DataDirectory（数据目录）了。DataDirectory 其实是一个结构体数组，它有 16 个元素：

```
typedef struct _IMAGE_DATA_DIRECTORY {
    ULONG   VirtualAddress;
    ULONG   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES    16
```

```
#define IMAGE_DIRECTORY_ENTRY_EXPORT          0   // Export Directory
#define IMAGE_DIRECTORY_ENTRY_IMPORT          1   // Import Directory
#define IMAGE_DIRECTORY_ENTRY_RESOURCE        2   // Resource Directory
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION       3   // Exception Directory
#define IMAGE_DIRECTORY_ENTRY_SECURITY        4   // Security Directory
#define IMAGE_DIRECTORY_ENTRY_BASERELOC       5   // Base Relocation Table
#define IMAGE_DIRECTORY_ENTRY_DEBUG           6   // Debug Directory
//      IMAGE_DIRECTORY_ENTRY_COPYRIGHT       7   // (X86 usage)
#define IMAGE_DIRECTORY_ENTRY_ARCHITECTURE    7   // Architecture Specific Data
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR       8   // RVA of GP
#define IMAGE_DIRECTORY_ENTRY_TLS             9   // TLS Directory
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG     10  // Load Configuration Directory
#define IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT    11  // Bound Import Directory in headers
#define IMAGE_DIRECTORY_ENTRY_IAT             12  // Import Address Table
#define IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT    13  // Delay Load Import Descriptors
#define IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR 14   // COM Runtime descriptor
```

有人可能觉得奇怪，说为什么定义了 16 个元素，可是只有 15 个宏呢？这是因为还有一个 IMAGE_DIRECTORY_ENTRY 尚未使用，处于 Reserve 状态。在这已有的 15 个元素里，每个元素的 VirtualAddress 都指向一个结构体，Size 都指出了这个结构体的大小。其中大家最为关注的输入表、导出表、重定位表、资源的结构体跟 PE32 一样，没有发生任何变化。但是也不是全部没有发生变化，比如 TLS 就发生了变化，不过这个似乎关注的人不多。

关于 PE32+相对于 PE32 的变化，该说的我基本都说完了，不过如果这样子就结束文章，估计有些朋友会不太高兴。所以，我“做了一个艰难的决定”，一是把我亲自修改而成的 PE32+文件结构超高清大图（3056*1910）奉献给大家，二是把我之前做的一个程序《Simple PE64 Viewer》开源，以飨读者。接下来详细讲述一下我的 PE 查看器源码。

类似于制作 ARK，制作 PE 信息查看器也是从“底层新人”晋升为“底层高手”的必经之路。目前，支持 PE32+文件的 PE 信息查看器不多，LordPE 算是一个。我就模仿 LordPE，把一个 PE 文件的基本信息，以及输入表和导出表的信息显示出来。在展示源码之前，先说思路，这个思路很简单，就是把 PE 文件载入内存，此时 ReadFile 返回的 **buffer** 指针就是 IMAGE_DOS_HEADER 结构体的首地址。当确认文件**是 PE32 文件且是 PE32+文件**时，根据 DOS 头结构体 e_lfanew 成员的值获得 NT 头的偏移，然后展示一系列文件头和可选头的信息之后，显示导出表的信息（如果导出表不存在的话，就跳到后面显示输入表的信息）。由于用语言描述获得导出表和输入表的信息十分麻烦，所以用列表的方式描述。

获得导出表信息（假设导出表存在）：

1.把 OptionalHeader.DataDirectory[0].VirtualAddress 的值（RVA，相对虚拟地址）转化为 VA（虚拟地址），获得 IMAGE_EXPORT_DIRECTORY 结构体相对于 buffer 的地址；
2.再使用三次 ImageRvaToVa，把 IMAGE_EXPORT_DIRECTORY 结构体里 AddressOfNames、AddressOfFunctions、AddressOfNameOrdinals 这三个成员的值（RVA）转化为 VA，获得导出函数的名字，RVA 和序号，然后使用 printf 把它们打印出来。

获得输入表的信息（假设输入表存在）：

1.把 OptionalHeader.DataDirectory[1].VirtualAddress 的值（RVA）转化为 VA，获得 IMAGE_EXPORT_DIRECTORY 结构体相对于 buffer 的地址；
2.获得被输入的 DLL 的名字；
3.获得第一个 Thunk 的值（假定名为 dwThunk），并把这个值（RVA）转化为 VA，得到 IMAGE_THUNK_DATA 相对于 buffer 的地址（假定名为_pThunk）；
4.把 pThunk->u1.AddressOfData->Name 的值（RVA）转化为 VA，获得函数名和 Hint，然后用 printf 把 Hint（序号）、dwThunk（RVA）以及函数名打印出来；
5.重复步骤 3，直到_pThunk->u1.AddressOfData 为 NULL；
6.重复步骤 2，直到“输入的 DLL 的名字”为 NULL。

详细代码如下：

```
#include <stdio.h>
#include <Windows.h>
#include <IMAGEHLP.H>
#pragma comment(lib,"ImageHlp.lib")

void MyCls(HANDLE hConsole)
{
    COORD   coordScreen={0,0};//设置清屏后光标返回的屏幕左上角坐标
    BOOL bSuccess;
    DWORD   cCharsWritten;
    CONSOLE_SCREEN_BUFFER_INFO   csbi;//保存缓冲区信息
    DWORD   dwConSize;//当前缓冲区可容纳的字符数
    bSuccess=GetConsoleScreenBufferInfo(hConsole,&csbi);//获得缓冲区信息
    dwConSize=csbi.dwSize.X*csbi.dwSize.Y;//缓冲区容纳字符数目
    bSuccess=FillConsoleOutputCharacter(hConsole,(TCHAR)'
',dwConSize,coordScreen,&cCharsWritten);
    bSuccess=GetConsoleScreenBufferInfo(hConsole,&csbi);//获得缓冲区信息

bSuccess=FillConsoleOutputAttribute(hConsole,csbi.wAttributes,dwConSize,coordScreen,&cCharsWritt
en);
    bSuccess=SetConsoleCursorPosition(hConsole,coordScreen);
    return;
}

void clrscr(void)
{
    HANDLE hStdOut=GetStdHandle(STD_OUTPUT_HANDLE);
    MyCls(hStdOut);
    return;
}

DWORD FileLen(char *filename)
{
     WIN32_FIND_DATAA fileInfo={0};
     DWORD fileSize=0;
     HANDLE hFind;
     hFind = FindFirstFileA(filename ,&fileInfo);
     if(hFind != INVALID_HANDLE_VALUE)
     {
          fileSize = fileInfo.nFileSizeLow;
          FindClose(hFind);
     }
     return fileSize;
```

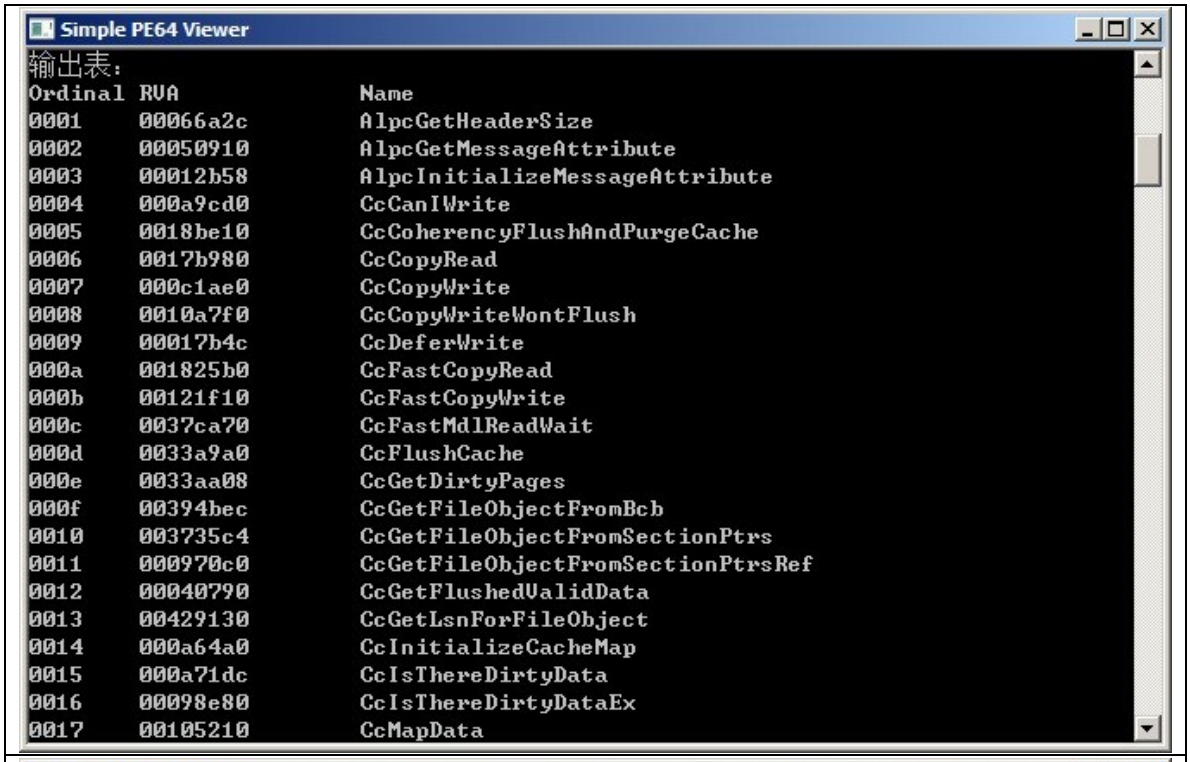
```
}

CHAR *LoadFile(char *filename)
{
    DWORD dwReadWrite, LenOfFile=FileLen(filename);
    HANDLE hFile = CreateFileA(filename, GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ |
FILE_SHARE_WRITE, 0, OPEN_EXISTING, 0, 0);
    if (hFile != INVALID_HANDLE_VALUE)
    {
        PCHAR buffer=(PCHAR)malloc(LenOfFile);
        SetFilePointer(hFile, 0, 0, FILE_BEGIN);
        ReadFile(hFile, buffer, LenOfFile, &dwReadWrite, 0);
        CloseHandle(hFile);
        return buffer;
    }
    return NULL;
}

VOID ShowPE64Info(char *filename)
{
    PIMAGE_NT_HEADERS64 pinths64;
    PIMAGE_DOS_HEADER pdih;
    char *filedata;
    filedata=LoadFile(filename);
    pdih=(PIMAGE_DOS_HEADER)filedata;
    pinths64=(PIMAGE_NT_HEADERS64)(filedata+pdih->e_lfanew);
    if(pinths64->Signature!=0x00004550)
    {
        printf("无效的 PE 文件！\n");
        return ;
    }
    if(pinths64->OptionalHeader.Magic!=0x20b)
    {
        printf("不是 PE32+格式的文件！\n");
        return ;
    }
    printf("\n");
    printf("入口点：          %llx\n",pinths64->OptionalHeader.AddressOfEntryPoint);
    printf("镜像基址：        %llx\n",pinths64->OptionalHeader.ImageBase);
    printf("镜像大小：        %llx\n",pinths64->OptionalHeader.SizeOfImage);
    printf("代码基址：        %llx\n",pinths64->OptionalHeader.BaseOfCode);
    printf("块对齐：          %llx\n",pinths64->OptionalHeader.SectionAlignment);
    printf("文件块对齐：      %llx\n",pinths64->OptionalHeader.FileAlignment);
    printf("子系统：          %llx\n",pinths64->OptionalHeader.Subsystem);
```

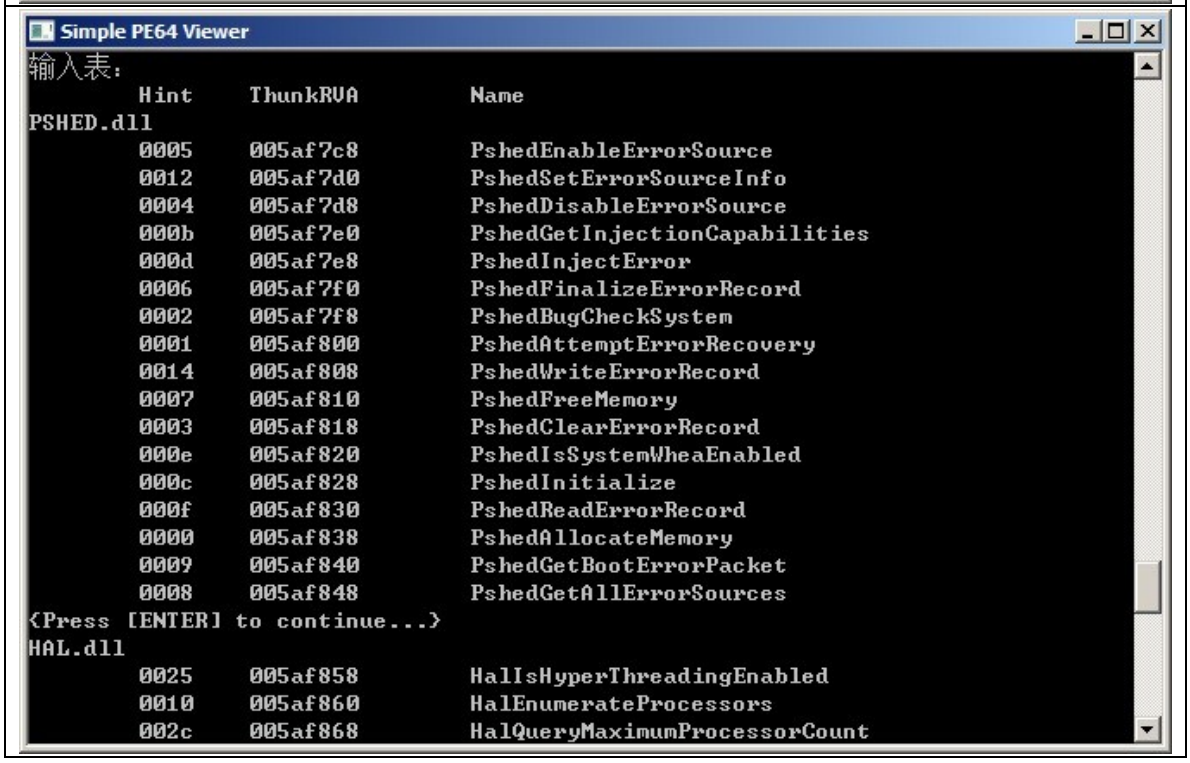
```
    printf("区段数目:          %llx\n",pinths64->FileHeader.NumberOfSections);
    printf("时间日期标志:      %llx\n",pinths64->FileHeader.TimeDateStamp);
    printf("首部大小:          %llx\n",pinths64->OptionalHeader.SizeOfHeaders);
    printf("特征值:            %llx\n",pinths64->FileHeader.Characteristics);
    printf("校验和:            %llx\n",pinths64->OptionalHeader.CheckSum);
    printf("可选头部大小:      %llx\n",pinths64->FileHeader.SizeOfOptionalHeader);
    printf("RVA 数及大小:      %llx\n",pinths64->OptionalHeader.NumberOfRvaAndSizes);
    getchar();
    printf("\n");
    printf("输出表: \n");
    printf("Ordinal\tRVA\t\tName\n");
    PIMAGE_EXPORT_DIRECTORY pied;
    if(pinths64,pdih,pinths64->OptionalHeader.DataDirectory[0].VirtualAddress==0) goto imp;
    pied=(PIMAGE_EXPORT_DIRECTORY)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64,pdih,pinths64->Optio
nalHeader.DataDirectory[0].VirtualAddress,NULL);
    DWORD i = 0;
    DWORD NumberOfNames = pied->NumberOfNames;
    ULONGLONG **ppdwNames = (ULONGLONG **)pied->AddressOfNames;
    ppdwNames =
(PULONGLONG*)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64,pdih,(ULONG)ppdwNames,NULL);
    ULONGLONG **ppdwAddr  = (ULONGLONG **)pied->AddressOfFunctions;
    ppdwAddr = (PULONGLONG*)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64,pdih,(DWORD)ppdwAddr,NULL);
    ULONGLONG
*ppdwOrdin=(ULONGLONG*)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64,pdih,(DWORD)pied->AddressOfName0
rdinals,NULL);
    char* szFun=(PSTR)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64,pdih,(ULONG)*ppdwNames,NULL);
    for(i=0; i<NumberOfNames; i++)
    {
        printf("%0.4x\t%0.8x\t%s\n",i+1,*ppdwAddr,szFun);
        szFun=szFun + strlen(szFun)+1;
        ppdwAddr++;
        if(i%200==0 && i/200>=1)
        {
            printf("{Press [ENTER] to continue...}");
            getchar();
        }
    }
imp:
    printf("\n\n输入表: \n");
    printf("\tHint\tThunkRVA\tName\n");
    PIMAGE_IMPORT_DESCRIPTOR piid;
    PIMAGE_THUNK_DATA _pThunk=NULL;
    DWORD dwThunk=NULL;
    USHORT Hint;
```

```
    if(pinths64->OptionalHeader.DataDirectory[1].VirtualAddress==0) return;
    piid=(PIMAGE_IMPORT_DESCRIPTOR)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64,pdih,pinths64->Opti
onalHeader.DataDirectory[1].VirtualAddress,NULL);
    for(;piid->Name!=NULL;)
    {
        char *szName=(PSTR)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64, pdih, (ULONG)piid->Name,
0);
        printf("%s\n",szName);
        if(piid->OriginalFirstThunk!=0)
        {
            dwThunk=piid->OriginalFirstThunk;

    _pThunk=(PIMAGE_THUNK_DATA)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64,pdih,(ULONG)piid->Origi
nalFirstThunk,NULL);
        }
        else
        {
            dwThunk=piid->FirstThunk;

    _pThunk=(PIMAGE_THUNK_DATA)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64,pdih,(ULONG)piid->First
Thunk,NULL);
        }
        for(;_pThunk->u1.AddressOfData!=NULL;)
        {
            char
*szFun=(PSTR)ImageRvaToVa((PIMAGE_NT_HEADERS)pinths64,pdih,(ULONG)(((PIMAGE_IMPORT_BY_NAME)_pThu
nk->u1.AddressOfData)->Name), 0);
            if(szFun!=NULL)
                memcpy(&Hint,szFun-2,2);
            else
                Hint=-1;
            printf("\t%0.4x\t%0.8x\t%s\n",Hint,dwThunk,szFun);
            dwThunk+=8;
            _pThunk++;
        }
        piid++;
        printf("{Press [ENTER] to continue...}");
        getchar();
    }
}

int main()
{
    char filename[MAX_PATH]={0};
```

```
    SetConsoleTitleA("Simple PE64 Viewer");
bgn:
    printf("Simple PE64 Viewer\n====================\nAuthor: Tesla.Angela\nVersion:
0.01\nSupport: PE32+ file\n\n\n");
    printf("输入文件名（支持文件拖拽，直接按回车则默认打开 ntoskrnl.exe，输入 exit 退出）：");
    gets(filename);
    if (FileLen(filename)==0)
    {
        if( stricmp(filename,"exit") )
        {
            CopyFileA("c:\\windows\\system32\\ntoskrnl.exe","c:\\ntoskrnl.exe",0);
            strcpy(filename,"c:\\ntoskrnl.exe");
            printf("c:\\ntoskrnl.exe\n");
        }
        else
            goto end;
    }
    ShowPE64Info(filename);
    clrscr();
    goto bgn;
end:
    DeleteFileA("c:\\ntoskrnl.exe");
    return 0;
}
```

运行效果：





本文到此结束。示例代码在附件里。