

2021/02/05_PE_第5课_导入表、地址导入表

笔记本： PE

创建时间： 2021/2/5 星期五 10:39

作者： ileemi

- [表](#)
 - [导入表](#)
 - [IMAGE_IMPORT_DESCRIPTOR](#)
 - [IMAGE_IMPORT_BY_NAME](#)
 - [IMAGE_THUNK_DATA32](#)
 - [操作系统加载API的流程](#)

静态反汇编工具会解析可执行文件中代码区的反汇编代码。还原可执行文件的OEP其实是有很大价值的。

防止可执行文件被内存dump，可以程序运行起来破坏其内存结构。

使用调试器对可执行文件进行内存dump时，破坏可执行文件在内存中的PE头后，再进行内存dump时，一般做法就是调试器从文件中获取目标进程的PE头（但是存在一定的风险（可执行文件的OEP可能不正确））。

防止内存dump，还可以将目标进程的二进制文件进行破坏。随机基址不会影响RVA，可执行文件的入口地址为：模块基址+偏移（RVA）。

表

- **导入表** (IMAGE_DIRECTORY_ENTRY_IMPORT)：记录程序中所使用的所有API，操作系统加载DII后，操作系统 将API的地址加载到程序中。
- **导出表** (IMAGE_DIRECTORY_ENTRY_EXPORT)
- **导入地址表** (IMAGE_DIRECTORY_ENTRY_IAT)
- **资源表** (IMAGE_DIRECTORY_ENTRY_RESOURCE)
- **重定位表** (IMAGE_DIRECTORY_ENTRY_BASERELOC)
- **线程局部存储** (IMAGE_DIRECTORY_ENTRY_TLS)
- **绑定导入表** (IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT)：将程序中使用的API地址固定写入到导入表中，现在基本不使用，老版本的软件使用。
- **延时加载表** (IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT)：程序中首次使用API函数时，就加载模块并将API在模块中的地址加载到程序中，解决软件启动速度的问题，现在基本不使用。
- **.net表** (IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR)
- **异常表** (IMAGE_DIRECTORY_ENTRY_EXCEPTION)：高版本编译器使用

导入表

导入表 (IMAGE_DIRECTORY_ENTRY_IMPORT)：记录程序中所使用的所有API，操作系统加载Dll后，操作系统会将API的地址加载到程序中。

导入地址表 (IMAGE_DIRECTORY_ENTRY_IAT)

程序中调用API，程序编译的时候，编译器并不知道所使用的API的地址，但是会将所用的API相关信息填写到PE文件中（API名字，所属模块（程序需要加载对应的模块））。在程序运行的时候，操作系统会根据PE文件中所使用的API的信息将对应的地址填写到对应的位置上。

API名称和模块名称属于多对多的数据关系（不使用数据库的情况下可以使用指针（结构体的首地址）来当作外键）。导入表由编译器编写到可执行文件中，由操作系统去读取。

数据目录在选项头中，一个可执行程序中一定存在导入表，没有导入表，可执行文件就不能调用系统的任意一个API。

WinHex - [Hello.exe]

文件(F) 编辑(E) 搜索(S) 导航(N) 查看(V) 工具(T) 专业工具(I) 选项(O) 窗口(W) 帮助(H)

19.9 x86

Hello.exe

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000000A0	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA
000000B0	50	45	00	00	4C	01	03	00	AA	AA	AA	AA	AA	AA	AA	AA
000000C0	AA	AA	AA	AA	E0	00	0F	01	0B	01	AA	AA	AA	AA	AA	AA
000000D0	AA	AA	AA	AA	AA	AA	AA	AA	23	10	00	00	AA	AA	AA	AA
000000E0	AA	AA	AA	AA	00	00	40	00	00	10	00	00	00	02	00	00
000000F0	AA	AA	AA	AA	AA	AA	AA	AA	04	00	AA	AA	AA	AA	AA	AA
00000100	AA	3A	00	00	00	04	00	00	AA	AA	AA	AA	02	00	00	00
00000110	AA	AA	AA	AA	00	AA	1A	00	AA	AA	AA	00	AA	1A	00	00
00000120	AA	AA	AA	AA	10	00	00	00	00	00	00	00	00	00	00	00
00000130	4C	10	00	00	3C	00	00	00	00	00	00	00	00	00	00	00
00000140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000150	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000170	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000180	00	00	00	00	00	00	00	00	00	10	00	00	10	00	00	00
00000190	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001A0	00	00	00	00	00	00	00	00	2E	35	31	61	73	6D	00	00
000001B0	00	10	00	00	00	10	00	00	AA	01	00	00	00	04	00	00
000001C0	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA
000001D0	2E	6E	65	77	00	00	00	00	00	10	00	00	00	20	00	00
000001E0	00	02	00	00	00	06	00	00	AA	AA	AA	AA	AA	AA	AA	AA
000001F0	AA	AA	AA	AA	AA	AA	AA	FA	2E	35	32	61	73	6D	00	00
00000200	00	10	00	00	00	30	00	00	00	02	00	00	00	08	00	00
00000210	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	AA	FA
00000220	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000230	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000240	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

ANSI ASCII

PE L à # @ : L <

.51asm

.new

ú. 52asm

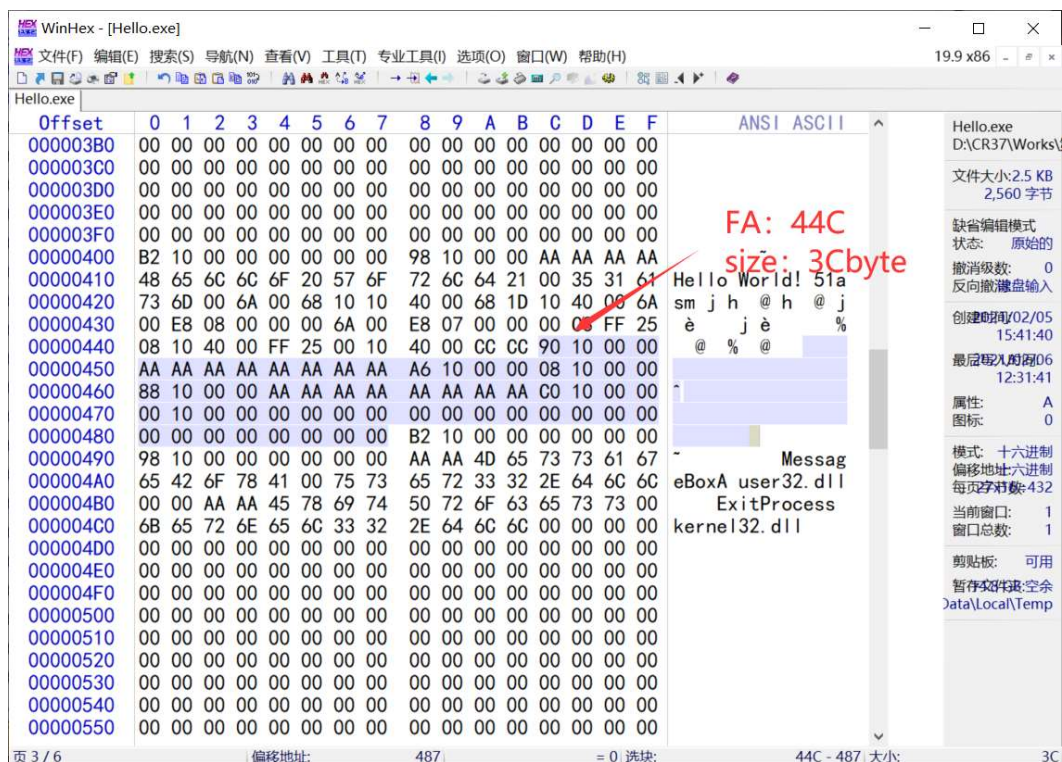
0 ú

1A8 - 1CF 大小: 28

页 1 / 6 偏移地址: 1CF = 106 选块:

Hello.exe
D:\CR37\Works\
文件大小: 2.5 KB
2,560 字节
缺省编辑模式
状态: 原始的
撤消级数: 0
反向撤消输入
创建时间: 02/05
15:41:40
最后写入时间: 06
12:31:41
属性: A
图标: 0
模式: 十六进制
偏移地址: 十六进制
每页字节数: 432
当前窗口: 1
窗口总数: 1
剪贴板: 可用
暂存文件: 空余
Data\Local\Temp

表的大小
在第一个节表中
其FA = 4C + 400 = 44C



表的大小由表的格式决定。数据目录结构体: "IMAGE_DATA_DIRECTORY".

导入表（可使用结构体首地址当作外键）会涉及到三个表（三个结构体），分别如下：

- IMAGE_IMPORT_DESCRIPTOR（主表）
- IMAGE_IMPORT_BY_NAME
- IMAGE_THUNK_DATA32

IMAGE_IMPORT_DESCRIPTOR

导入表结构（表的大不固定，受所使用API的数量决定）如下（一项共20byte，一项表示一个库）：

```
///@[comment("MVI_tracked")]
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD Characteristics; // 0 for terminating null import

        // 指向函数名称表 (INT)
        DWORD OriginalFirstThunk; // RVA to original unbound IAT
    } DUMMYUNIONNAME;
    // 0 if not bound,
    // -1 if bound, and real date\time stamp
    // in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new BIND)
    // 0.W. date/time stamp of DLL bound to (Old BIND)
    DWORD TimeDateStamp; // 时间戳描述信息
    DWORD ForwarderChain; // -1 if no forwarders
}
```



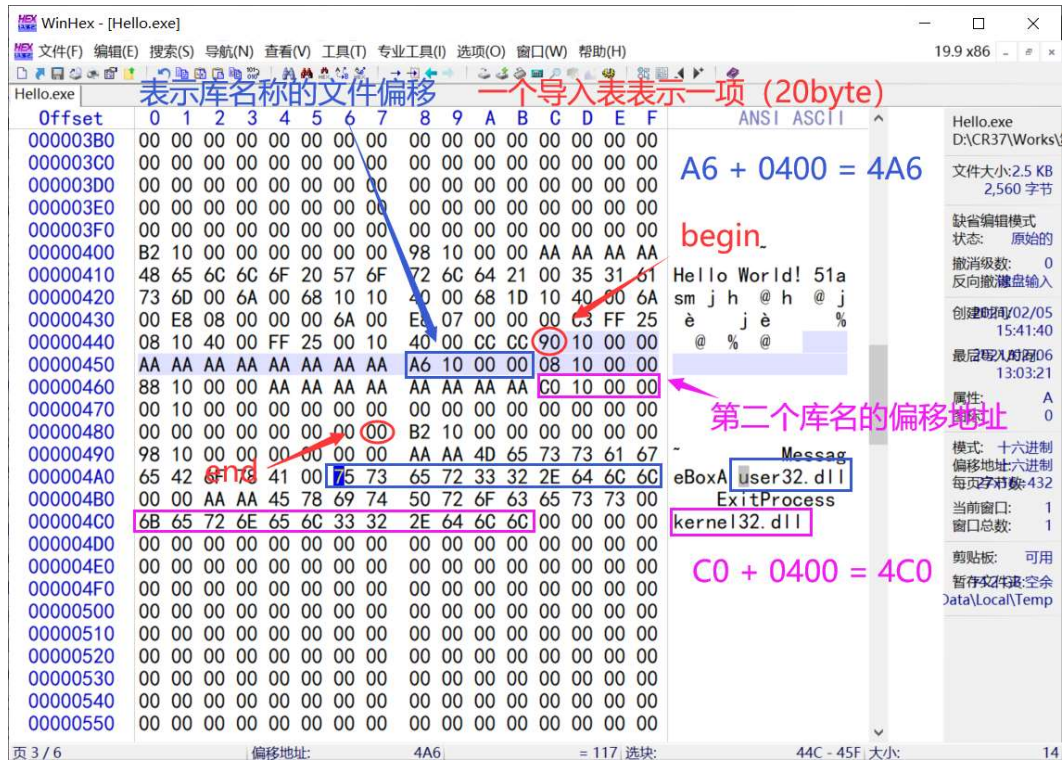
```

DWORD Name; // 库名
// 指向 IMAGE_THUNK_DATA32 (导入地址表 "IAT")
// 将API在模块中的地址进行保存
DWORD FirstThunk; // RVA to IAT (if bound this IAT has actual
// addresses)

} IMAGE_IMPORT_DESCRIPTOR;

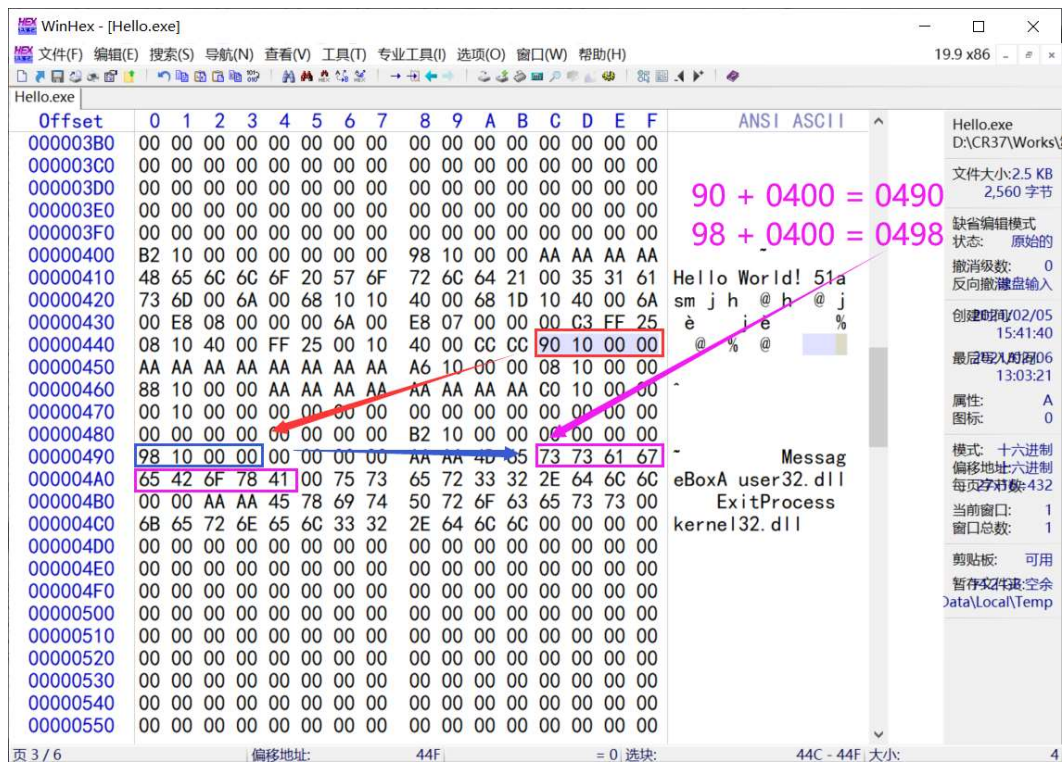
typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED *PIMAGE_IMPORT_DESCRIPTOR;

```



编译器生成可执行文件，导入表一般在代码断后面存储，由上图可知，该可执行文件加载了两个库。

可执行文件所需加载的API名称通过导入表的第一个成员进行表示（指向IMAGE_IMPORT_BY_NAME数据在文件中偏移），解析格式如下：

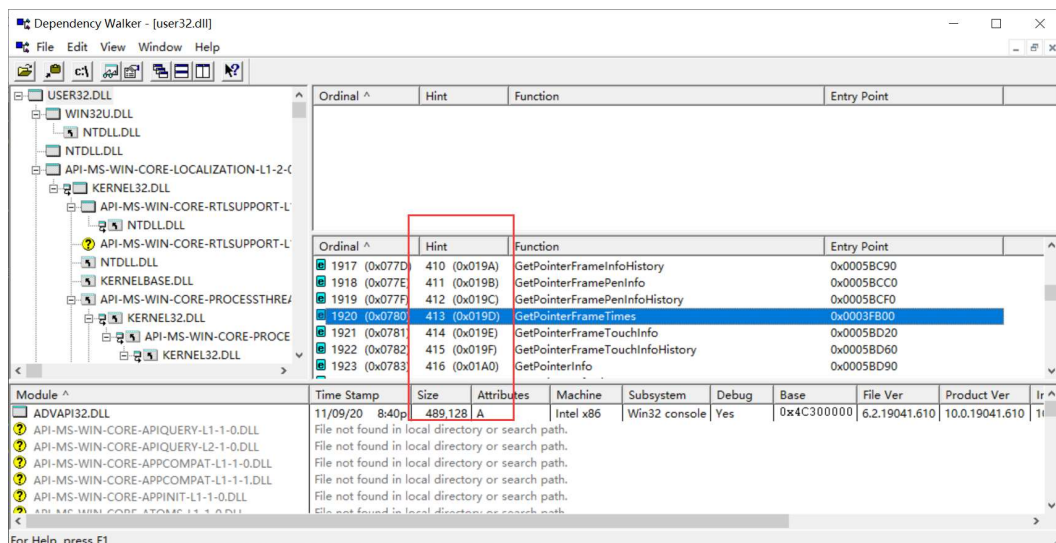


IMAGE_IMPORT_BY_NAME

结构如下:

```
//API名称 模块名称(多对多)
typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD Hint; //函数序号
    CHAR Name[1]; //函数名称(柔性数组, 零结尾)
} IMAGE_IMPORT_BY_NAME, * PIMAGE_IMPORT_BY_NAME;
```

Hint: 模块中导出函数的序号 (该序号受模块版本影响, 不一定准确), 方便快速定位模块中目标API函数 (目前已经不准确了)。

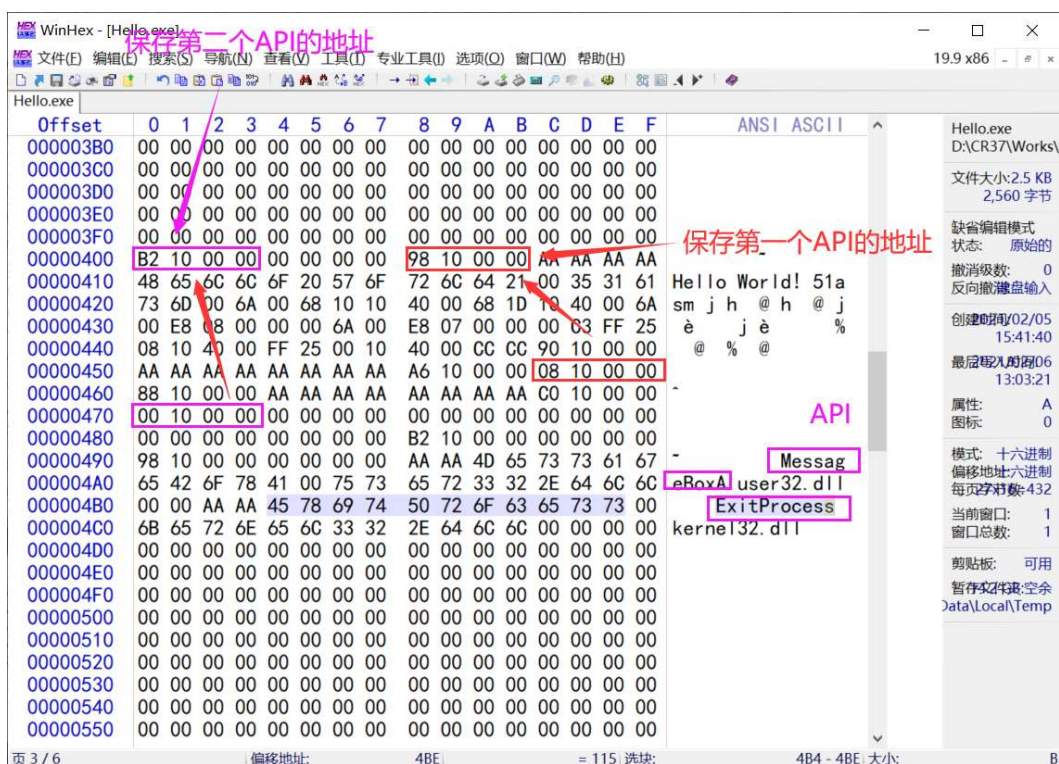


IMAGE_THUNK_DATA32

导入地址表 (IAT) 结构如下:

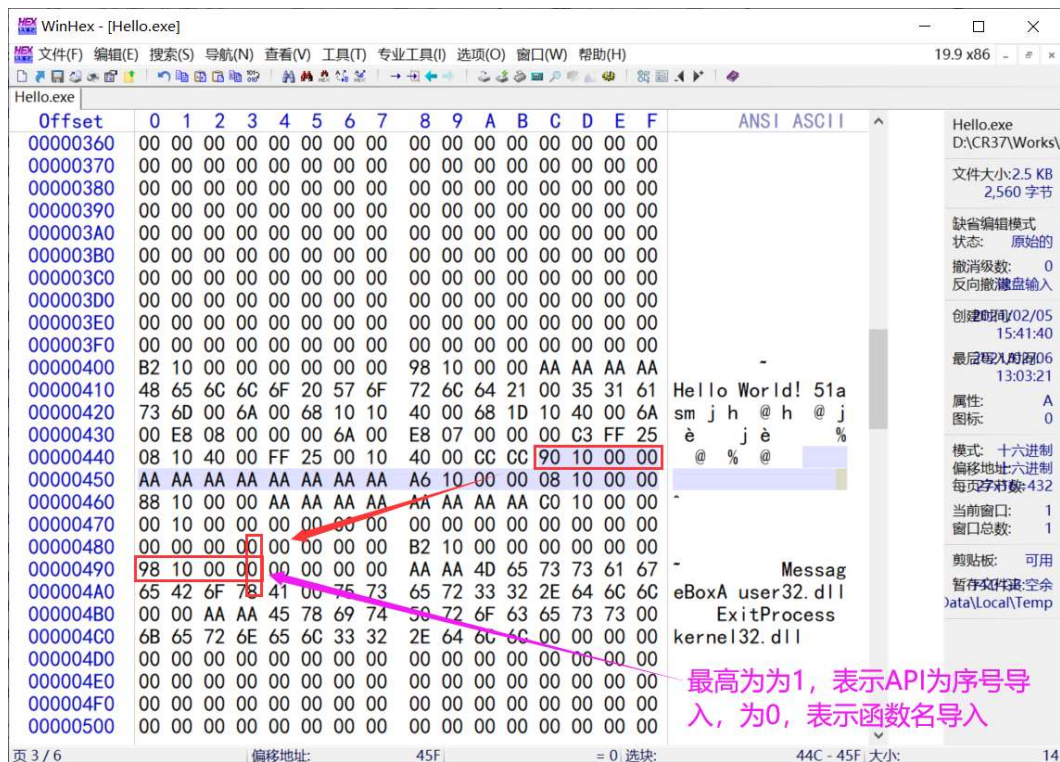
```
typedef struct _IMAGE_THUNK_DATA32 {  
    union {  
        DWORD ForwarderString; // PBYTE  
        DWORD Function; // PDWORD  
        DWORD Ordinal;  
        DWORD AddressOfData; // PIMAGE_IMPORT_BY_NAME  
    } u1;  
} IMAGE_THUNK_DATA32;
```

最高位1表示
序号导入



导入的每个模块都会有一个IAT表, 模块对应的IAT表中的保存的API地址在内存中是连续存放的。编译器生成的可执行文件中, 每个模块对应的IAT表在内存中也是连续的 (可以对其进行修改)。

模块中导出函数可以通过函数名导出以及序号导出, 使用序号导出, 所使用API的名称也就不需要保存到可执行文件中, 操作系统需要通过结构体 "IMAGE_THUNK_DATA32" 的成员 "u1" 的最高位进行区分 (最高位为1表示序号导入, 低2个字节为导出序号)。MFC程序多使用序号进行导入 (需要动态使用, 就是 "在共享 DLL 中使用 MFC")。



动态库静态使用会使用 "LoadLibrary", "#pragma comment(lib, "text.lib" -- 会生成导入表, "text.lib" 库中存放了模块名以及API名 (用于生成导入表), 并没有函数的实现代码, 告诉操作系统去加载对应的模块以及填充API的地址)。

操作系统加载API的流程

- LoadLibrary(Name), 失败退出
- GetProcAddress(OriginalFirstThunk[i]), 获取API地址
- 填充到IAT[i], 将API地址填到导入地址表中

