PE 文件格式详解(上)

作者： MSDN
译者：李马 (http://home.nuc.edu.cn/~titilima)

摘要

Windows NT 3.1 引入了一种名为 PE 文件格式的新可执行文件格式。PE 文件格式的规范包含在了 MSDN 的 CD 中（Specs and Strategy, Specifications, Windows NT File Format Specifications），但是它非常之晦涩。

然而这一的文档并未提供足够的信息，所以开发者们无法很好地弄懂 PE 格式。本文旨在解决这一问题，它会对整个的 PE 文件格式作一个十分彻底的解释，另外，本文中还带有对所有必需结构的描述以及示范如何使用这些信息的源码示例。

为了获得 PE 文件中所包含的重要信息，我编写了一个名为 PEFILE.DLL 的动态链接库，本文中所有出现的源码示例亦均摘自于此。这个 DLL 和它的源代码都作为 PEFile 示例程序的一部分包含在了 CD 中（译注：示例程序请在 MSDN 中寻找，本站恕不提供），你可以在你自己的应用程序中使用这个 DLL；同样，你亦可以依你所愿地使用并构建它的源码。在本文末尾，你会找到 PEFILE.DLL 的函数导出列表和一个如何使用它们的说明。我觉得你会发现这些函数会让你从容应付 PE 文件格式的。

介绍

Windows 操作系统家族最近增加的 Windows NT 为开发环境和应用程序本身带来了很大的改变，这之中一个最为重大的当属 PE 文件格式了。新的 PE 文件格式主要来自于 UNIX 操作系统所通用的 COFF 规范，同时为了保证与旧版本 MS-DOS 及 Windows 操作系统的兼容，PE 文件格式也保留了 MS-DOS 中那熟悉的 MZ 头部。

在本文之中，PE 文件格式是以自顶而下的顺序解释的。在你从头开始研究文件内容的过程之中，本文会详细讨论 PE 文件的每一个组成部分。

许多单独的文件成分定义都来自于 Microsoft Win32 SDK 开发包中的 WINNT.H 文件，在这个文件中你会发现用来描述文件头部和数据目录等各种成分的结构类型定义。但是，在 WINNT.H 中缺少对 PE 文件结构足够的定义，在这种情况下，我定义了自己的结构来存取文件数据。你会在 PEFILE.DLL 工程的 PEFILE.H 中找到这些结构的定义，整套的 PEFILE.H 开发文件包含在 PEFile 示例程序之中。

本文配套的示例程序除了 PEFILE.DLL 示例代码之外，还有一个单独的 Win32 示例应用程序，名为 EXEVIEW.EXE。创建这一示例目的有二：首先，我需要测试 PEFILE.DLL 的函数，并且某些情况要求我同时查看多个文件；其次，很多解决 PE 文件格式的工作和直接观看数据有关。例如，要弄懂导入地址名称表是如何构成的，我就得同时查看 .idata 段头部、导入映像数据目录、可选头部以及当前的 .idata 段实体，而 EXEVIEW.EXE 就是

查看这些信息的最佳示例。

闲话少叙，让我们开始吧。

PE 文件结构

PE 文件格式被组织为一个线性的数据流，它由一个 MS-DOS 头部开始，接着是一个是模式的程序残余以及一个 PE 文件标志，这之后紧接着 PE 文件头和可选头部。这些之后是所有的段头部，段头部之后跟随着所有的段实体。文件的结束处是一些其它的区域，其中是一些混杂的信息，包括重分配信息、符号表信息、行号信息以及字串表数据。我将所有这些成分列于图 1。

图 1.PE 文件映像结构

从 MS-DOS 文件头结构开始，我将按照 PE 文件格式各成分的出现顺序依次对其进行讨论，并且讨论的大部分是以示例代码为基础来示范如何获得文件的信息的。所有的源码均摘自 PEFILE.DLL 模块的 PEFILE.C 文件。这些示例都利用了 Windows NT 最酷的特色之一——内存映射文件，这一特色允许用户使用一个简单的指针来存取文件中所包含的数据，因此所有的示例都使用了内存映射文件来存取 PE 文件中的数据。

注意：请查阅本文末尾关于如何使用 PEFILE.DLL 的那一段。

MS-DOS 头部/实模式头部

如上所述，PE 文件格式的第一个组成部分是 MS-DOS 头部。在 PE 文件格式中，它

并非一个新概念，因为它与 MS-DOS 2.0 以来就已有的 MS-DOS 头部是完全一样的。保留这个相同结构的最主要原因是，当你尝试在 Windows 3.1 以下或 MS-DOS 2.0 以上的系统下装载一个文件的时候，操作系统能够读取这个文件并明白它是和当前系统不相兼容的。换句话说，当你在 MS-DOS 6.0 下运行一个 Windows NT 可执行文件时，你会得到这样一条消息："This program cannot be run in DOS mode."如果 MS-DOS 头部不是作为 PE 文件格式的第一部分的话，操作系统装载文件的时候就会失败，并提供一些完全没用的信息，例如："The name specified is not recognized as an internal or external command, operable program or batch file."

MS-DOS 头部占据了 PE 文件的头 64 个字节，描述它内容的结构如下：

```
//WINNT.H

typedef struct _IMAGE_DOS_HEADER { // DOS 的.EXE 头部
  USHORT e_magic; // 魔术数字
  USHORT e_cblp; // 文件最后页的字节数
  USHORT e_cp; // 文件页数
  USHORT e_crlc; // 重定义元素个数
  USHORT e_cparhdr; // 头部尺寸，以段落为单位
  USHORT e_minalloc; // 所需的最小附加段
  USHORT e_maxalloc; // 所需的最大附加段
  USHORT e_ss; // 初始的 SS 值（相对偏移量）
  USHORT e_sp; // 初始的 SP 值
  USHORT e_csum; // 校验和
  USHORT e_ip; // 初始的 IP 值
  USHORT e_cs; // 初始的 CS 值（相对偏移量）
  USHORT e_lfarlc; // 重分配表文件地址
  USHORT e_ovno; // 覆盖号
  USHORT e_res[4]; // 保留字
  USHORT e_oemid; // OEM 标识符（相对 e_oeminfo）
  USHORT e_oeminfo; // OEM 信息
  USHORT e_res2[10]; // 保留字
  LONG e_lfanew; // 新 exe 头部的文件地址
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

第一个域 e_magic，被称为魔术数字，它被用于表示一个 MS-DOS 兼容的文件类型。所有 MS-DOS 兼容的可执行文件都将这个值设为 0x5A4D，表示 ASCII 字符 MZ。MS-DOS 头部之所以有的时候被称为 MZ 头部，就是这个缘故。还有许多其它的域对于 MS-DOS 操作系统来说都有用，但是对于 Windows NT 来说，这个结构中只有一个有用的域——最后一个域 e_lfnew，一个 4 字节的文件偏移量，PE 文件头部就是由它定位的。对于

Windows NT 的 PE 文件来说，PE 文件头部是紧跟在 MS-DOS 头部和实模式程序残余之后的。

实模式残余程序

实模式残余程序是一个在装载时能够被 MS-DOS 运行的实际程序。对于一个 MS-DOS 的可执行映像文件，应用程序就是从这里执行的。对于 Windows、OS/2、Windows NT 这些操作系统来说，MS-DOS 残余程序就代替了主程序的位置被放在这里。这种残余程序通常什么也不做，而只是输出一行文本，例如："This program requires Microsoft Windows v3.1 or greater."当然，用户可以在此放入任何的残余程序，这就意味着你可能经常看到像这样的东西："You can''t run a Windows NT application on OS/2, it''s simply not possible."

当为 Windows 3.1 构建一个应用程序的时候，链接器将向你的可执行文件中链接一个名为 WINSTUB.EXE 的默认残余程序。你可以用一个基于 MS-DOS 的有效程序取代 WINSTUB，并且用 STUB 模块定义语句指示链接器，这样就能够取代链接器的默认行为。为 Windows NT 开发的应用程序可以通过使用 -STUB: 链接器选项来实现。

PE 文件头部与标志

PE 文件头部是由 MS-DOS 头部的 e_lfanew 域定位的，这个域只是给出了文件的偏移量，所以要确定 PE 头部的实际内存映射地址，就需要添加文件的内存映射基地址。例如，以下的宏是包含在 PEFILE.H 源文件之中的：

```
//PEFILE.H

#define NTSIGNATURE(a) ((LPVOID)((BYTE *)a + \
                                 ((PIMAGE_DOS_HEADER)a)->e_lfanew))
```

在处理 PE 文件信息的时候，我发现文件之中有些位置需要经常查阅。既然这些位置仅仅是对文件的偏移量，那么用宏来实现这些定位就比较容易，因为它们较之函数有更好的表现。

请注意这个宏所获得的是 PE 文件标志，而并非 PE 文件头部的偏移量。那是由于自 Windows 与 OS/2 的可执行文件开始，.EXE 文件都被赋予了目标操作系统的标志。对于 Windows NT 的 PE 文件格式而言，这一标志在 PE 文件头部结构之前。在 Windows 和 OS/2 的某些版本中，这一标志是文件头的第一个字。同样，对于 PE 文件格式，Windows NT 使用了一个 DWORD 值。

以上的宏返回了文件标志的偏移量，而不管它是哪种类型的可执行文件。所以，文件头部是在 DWORD 标志之后，还是在 WORD 标志处，是由这个标志是否 Windows NT 文件标志所决定的。要解决这个问题，我编写了 ImageFileType 函数（如下），它返回了映像文件的类型：

```
//PEFILE.C

DWORD WINAPI ImageFileType (LPVOID lpFile)
{
  /* 首先出现的是 DOS 文件标志 */
  if (*(USHORT *)lpFile == IMAGE_DOS_SIGNATURE)
  {
    /* 由 DOS 头部决定 PE 文件头部的位置 */
    if (LOWORD (*(DWORD *)NTSIGNATURE (lpFile)) ==
         IMAGE_OS2_SIGNATURE ||
         LOWORD (*(DWORD *)NTSIGNATURE (lpFile)) ==
         IMAGE_OS2_SIGNATURE_LE)
      return (DWORD)LOWORD(*(DWORD *)NTSIGNATURE (lpFile));
    else if (*(DWORD *)NTSIGNATURE (lpFile) ==
      IMAGE_NT_SIGNATURE)
    return IMAGE_NT_SIGNATURE;
    else
      return IMAGE_DOS_SIGNATURE;
  }
  else
    /* 不明文件种类 */
    return 0;
}
```

以上列出的代码立即告诉了你 NTSIGNATURE 宏有多么有用。对于比较不同文件类型并且返回一个适当的文件种类来说，这个宏就会使这两件事变得非常简单。WINNT.H 之中定义的四种不同文件类型有：

```
//WINNT.H

#define IMAGE_DOS_SIGNATURE 0x5A4D // MZ
#define IMAGE_OS2_SIGNATURE 0x454E // NE
#define IMAGE_OS2_SIGNATURE_LE 0x454C // LE
#define IMAGE_NT_SIGNATURE 0x00004550 // PE00
```

首先，Windows 的可执行文件类型没有出现在这一列表中，这一点看起来很奇怪。但是，在稍微研究一下之后，就能得到原因了：除了操作系统版本规范的不同之外，Windows 的可执行文件和 OS/2 的可执行文件实在没有什么区别。这两个操作系统拥有相同的可执行文件结构。

现在把我们的注意力转向 Windows NT PE 文件格式，我们会发现只要我们得到了文件标志的位置，PE 文件之后就会有 4 个字节相跟随。下一个宏标识了 PE 文件的头部：

```
//PEFILE.C

#define PEFHDROFFSET(a) ((LPVOID)((BYTE *)a + \
                                  ((PIMAGE_DOS_HEADER)a)->e_lfanew + \
                                  SIZE_OF_NT_SIGNATURE))
```

这个宏与上一个宏的唯一不同是这个宏加入了一个常量 SIZE_OF_NT_SIGNATURE。不幸的是，这个常量并未定义在 WINNT.H 之中，于是我将它定义在了 PEFILE.H 中，它是一个 DWORD 的大小。

既然我们知道了 PE 文件头的位置，那么就可以检查头部的数据了。我们只需要把这个位置赋值给一个结构，如下：

```
PIMAGE_FILE_HEADER pfh;
pfh = (PIMAGE_FILE_HEADER)PEFHDROFFSET(lpFile);
```

在这个例子中，lpFile 表示一个指向可执行文件内存映像基地址的指针，这就显出了内存映射文件的好处：不需要执行文件的 I/O，只需使用指针 pfh 就能存取文件中的信息。PE 文件头结构被定义为：

```
//WINNT.H

typedef struct _IMAGE_FILE_HEADER {
  USHORT Machine;
  USHORT NumberOfSections;
  ULONG TimeDateStamp;
  ULONG PointerToSymbolTable;
  ULONG NumberOfSymbols;
  USHORT SizeOfOptionalHeader;
  USHORT Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;

#define IMAGE_SIZEOF_FILE_HEADER 20
```

请注意这个文件头部的大小已经定义在这个包含文件之中了，这样一来，想要得到这个结构的大小就很方便了。但是我觉得对结构本身使用 sizeof 运算符（译注：原文为“function”）更简单一些，因为这样的话我就不必记住这个常量的名字 IMAGE_SIZEOF_FILE_HEADER，而只需要记住结构 IMAGE_FILE_HEADER 的名字就可以了。另一方面，记住所有结构的名字已经够有挑战性的了，尤其在是这些结构只有 WINNT.H 中才有的情况下。

PE 文件中的信息基本上是一些高级信息，这些信息是被操作系统或者应用程序用来决定如何处理这个文件的。第一个域是用来表示这个可执行文件被构建的目标机器种类，例如 DEC(R) Alpha、MIPS R4000、Intel(R) x86 或一些其它处理器。系统使用这一信息来在读取这个文件的其它数据之前决定如何处理它。

Characteristics 域表示了文件的一些特征。比如对于一个可执行文件而言，分离调试文件是如何操作的。调试器通常使用的方法是将调试信息从 PE 文件中分离，并保存到一个调试文件（.DBG）中。要这么做的话，调试器需要了解是否要在一个单独的文件中寻找调试信息，以及这个文件是否已经将调试信息分离了。我们可以通过深入可执行文件并寻找调试信息的方法来完成这一工作。要使调试器不在文件中查找的话，就需要用到 IMAGE_FILE_DEBUG_STRIPPED 这个特征，它表示文件的调试信息是否已经被分离了。这样一来，调试器可以通过快速查看 PE 文件的头部的方法来决定文件中是否存在着调试信息。

WINNT.H 定义了若干其它表示文件头信息的标记，就和以上的例子差不多。我把研究这些标记的事情留给读者作为练习，由你们来看看它们是不是很有趣，这些标记位于 WINNT.H 中的 IMAGE_FILE_HEADER 结构之后。

PE 文件头结构中另一个有用的入口是 NumberOfSections 域，它表示如果你要方便地提取文件信息的话，就需要了解多少个段——更明确一点来说，有多少个段头部和多少个段实体。每一个段头部和段实体都在文件中连续地排列着，所以要决定段头部和段实体在哪里结束的话，段的数目是必需的。以下的函数从 PE 文件头中提取了段的数目：

PEFILE.C

```
int WINAPI NumOfSections(LPVOID lpFile)
{
  /* 文件头部中所表示出的段数目 */
  return (int)((PIMAGE_FILE_HEADER)
    PEFHDROFFSET (lpFile))->NumberOfSections);
}
```

如你所见，PEFHDROFFSET 以及其它宏用起来非常方便。

PE 可选头部

PE 可执行文件中接下来的 224 个字节组成了 PE 可选头部。虽然它的名字是"可选头部"，但是请确信：这个头部并非"可选"，而是"必需"的。OPTHDROFFSET 宏可以获得指向可选头部的指针：

//PEFILE.H

```
#define OPTHDROFFSET(a) ((LPVOID)((BYTE *)a + \
                        ((PIMAGE_DOS_HEADER)a)->e_lfanew +
\
```

```
                                SIZE_OF_NT_SIGNATURE + \
                                sizeof(IMAGE_FILE_HEADER)))
```

可选头部包含了很多关于可执行映像的重要信息，例如初始的堆栈大小、程序入口点的位置、首选基地址、操作系统版本、段对齐的信息等等。IMAGE_OPTIONAL_HEADER 结构如下：

```
//WINNT.H

typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // 标准域
    //
    USHORT Magic;
    UCHAR MajorLinkerVersion;
    UCHAR MinorLinkerVersion;
    ULONG SizeOfCode;
    ULONG SizeOfInitializedData;
    ULONG SizeOfUninitializedData;
    ULONG AddressOfEntryPoint;
    ULONG BaseOfCode;
    ULONG BaseOfData;
    //
    // NT 附加域
    //
    ULONG ImageBase;
    ULONG SectionAlignment;
    ULONG FileAlignment;
    USHORT MajorOperatingSystemVersion;
    USHORT MinorOperatingSystemVersion;
    USHORT MajorImageVersion;
    USHORT MinorImageVersion;
    USHORT MajorSubsystemVersion;
    USHORT MinorSubsystemVersion;
    ULONG Reserved1;
    ULONG SizeOfImage;
    ULONG SizeOfHeaders;
    ULONG CheckSum;
    USHORT Subsystem;
```

```
  USHORT DllCharacteristics;
  ULONG SizeOfStackReserve;
  ULONG SizeOfStackCommit;
  ULONG SizeOfHeapReserve;
  ULONG SizeOfHeapCommit;
  ULONG LoaderFlags;
  ULONG NumberOfRvaAndSizes;
  IMAGE_DATA_DIRECTORY
DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

如你所见，这个结构中所列出的域实在是冗长得过分。为了不让你对所有这些域感到厌烦，我会仅仅讨论有用的——就是说，对于探究 PE 文件格式而言有用的。

标准域

首先，请注意这个结构被划分为"标准域"和"NT 附加域"。所谓标准域，就是和 UNIX 可执行文件的 COFF 格式所公共的部分。虽然标准域保留了 COFF 中定义的名字，但是 Windows NT 仍然将它们用作了不同的目的——尽管换个名字更好一些。

·Magic。我不知道这个域是干什么的，对于示例程序 EXEVIEW.EXE 示例程序而言，这个值是 0x010B 或 267（译注：0x010B 为.EXE，0x0107 为 ROM 映像，这个信息我是从 eXeScope 上得来的）。

·MajorLinkerVersion、MinorLinkerVersion。表示链接此映像的链接器版本。随 Window NT build 438 配套的 Windows NT SDK 包含的链接器版本是 2.39（十六进制为 2.27）。

·SizeOfCode。可执行代码尺寸。

·SizeOfInitializedData。已初始化的数据尺寸。

·SizeOfUninitializedData。未初始化的数据尺寸。

·AddressOfEntryPoint。在标准域中，AddressOfEntryPoint 域是对 PE 文件格式来说最为有趣的了。这个域表示应用程序入口点的位置。并且，对于系统黑客来说，这个位置就是导入地址表（IAT）的末尾。以下的函数示范了如何从可选头部获得 Windows NT 可执行映像的入口点。

```
//PEFILE.C

LPVOID WINAPI GetModuleEntryPoint(LPVOID lpFile)
{
  PIMAGE_OPTIONAL_HEADER poh;
  poh = (PIMAGE_OPTIONAL_HEADER)OPTHDROFFSET(lpFile);
  if (poh != NULL)
```

```
    return (LPVOID)poh->AddressOfEntryPoint;
  else
    return NULL;
}
```

·BaseOfCode。已载入映像的代码（".text"段）的相对偏移量。

·BaseOfData。已载入映像的未初始化数据（".bss"段）的相对偏移量。

Windows NT 附加域

添加到 Windows NT PE 文件格式中的附加域为 Windows NT 特定的进程行为提供了装载器的支持，以下为这些域的概述。

·ImageBase。进程映像地址空间中的首选基地址。Windows NT 的 Microsoft Win32 SDK 链接器将这个值默认设为 0x00400000，但是你可以使用 -BASE:linker 开关改变这个值。

·SectionAlignment。从 ImageBase 开始，每个段都被相继的装入进程的地址空间中。SectionAlignment 则规定了装载时段能够占据的最小空间数量——就是说，段是关于 SectionAlignment 对齐的。

Windows NT 虚拟内存管理器规定，段对齐不能少于页尺寸（当前的 x86 平台是 4096 字节），并且必须是成倍的页尺寸。4096 字节是 x86 链接器的默认值，但是它可以通过 -ALIGN: linker 开关来设置。

·FileAlignment。映像文件首先装载的最小的信息块间隔。例如，链接器将一个段实体（段的原始数据）加零扩展为文件中最接近的 FileAlignment 边界。早先提及的 2.39 版链接器将映像文件以 0x200 字节的边界对齐，这个值可以被强制改为 512 到 65535 这么多。

·MajorOperatingSystemVersion。表示 Windows NT 操作系统的主版本号；通常对 Windows NT 1.0 而言，这个值被设为 1。

·MinorOperatingSystemVersion。表示 Windows NT 操作系统的次版本号；通常对 Windows NT 1.0 而言，这个值被设为 0。

·MajorImageVersion。用来表示应用程序的主版本号；对于 Microsoft Excel 4.0 而言，这个值是 4。

·MinorImageVersion。用来表示应用程序的次版本号；对于 Microsoft Excel 4.0 而言，这个值是 0。

·MajorSubsystemVersion。表示 Windows NT Win32 子系统的主版本号；通常对于 Windows NT 3.10 而言，这个值被设为 3。

·MinorSubsystemVersion。表示 Windows NT Win32 子系统的次版本号；通常对于 Windows NT 3.10 而言，这个值被设为 10。

·Reserved1。未知目的，通常不被系统使用，并被链接器设为 0。

·SizeOfImage。表示载入的可执行映像的地址空间中要保留的地址空间大小，这个

数字很大程度上受 SectionAlignment 的影响。例如，考虑一个拥有固定页尺寸 4096 字节的系统，如果你有一个 11 个段的可执行文件，它的每个段都少于 4096 字节，并且关于 65536 字节边界对齐，那么 SizeOfImage 域将会被设为 11 * 65536 = 720896（176 页）。而如果一个相同的文件关于 4096 字节对齐的话，那么 SizeOfImage 域的结果将是 11 * 4096 = 45056（11 页）。这只是个简单的例子，它说明每个段需要少于一个页面的内存。在现实中，链接器通过个别地计算每个段的方法来决定 SizeOfImage 确切的值。它首先决定每个段需要多少字节，并且最后将页面总数向上取整至最接近的 SectionAlignment 边界，然后总数就是每个段个别需求之和了。

·SizeOfHeaders。这个域表示文件中有多少空间用来保存所有的文件头部，包括 MS-DOS 头部、PE 文件头部、PE 可选头部以及 PE 段头部。文件中所有的段实体就开始于这个位置。

·CheckSum。校验和是用来在装载时验证可执行文件的，它是由链接器设置并检验的。由于创建这些校验和的算法是私有信息，所以在此不进行讨论。

·Subsystem。用于标识该可执行文件目标子系统的域。每个可能的子系统取值列于 WINNT.H 的 IMAGE_OPTIONAL_HEADER 结构之后。

·DllCharacteristics。用来表示一个 DLL 映像是否为进程和线程的初始化及终止包含入口点的标记。

·SizeOfStackReserve、SizeOfStackCommit、SizeOfHeapReserve、SizeOfHeapCommit。这些域控制要保留的地址空间数量，并且负责栈和默认堆的申请。在默认情况下，栈和堆都拥有 1 个页面的申请值以及 16 个页面的保留值。这些值可以使用链接器开关 -STACKSIZE: 与 -HEAPSIZE: 来设置。

·LoaderFlags。告知装载器是否在装载时中止和调试，或者默认地正常运行。

·NumberOfRvaAndSizes。这个域标识了接下来的 DataDirectory 数组。请注意它被用来标识这个数组，而不是数组中的各个入口数字，这一点非常重要。

·DataDirectory。数据目录表示文件中其它可执行信息重要组成部分的位置。它事实上就是一个 IMAGE_DATA_DIRECTORY 结构的数组，位于可选头部结构的末尾。当前的 PE 文件格式定义了 16 种可能的数据目录，这之中的 11 种现在在使用中。

数据目录

WINNT.H 之中所定义的数据目录为：

```
//WINNT.H

// 目录入口
// 导出目录
#define IMAGE_DIRECTORY_ENTRY_EXPORT 0
// 导入目录
#define IMAGE_DIRECTORY_ENTRY_IMPORT 1
```

```
// 资源目录
#define IMAGE_DIRECTORY_ENTRY_RESOURCE 2
// 异常目录
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION 3
// 安全目录
#define IMAGE_DIRECTORY_ENTRY_SECURITY 4
// 重定位基本表
#define IMAGE_DIRECTORY_ENTRY_BASERELOC 5
// 调试目录
#define IMAGE_DIRECTORY_ENTRY_DEBUG 6
// 描述字串
#define IMAGE_DIRECTORY_ENTRY_COPYRIGHT 7
// 机器值（MIPS GP）
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR 8
// TLS 目录
#define IMAGE_DIRECTORY_ENTRY_TLS 9
// 载入配置目录
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG 10
```

基本上，每个数据目录都是一个被定义为 IMAGE_DATA_DIRECTORY 的结构。虽然数据目录入口本身是相同的，但是每个特定的目录种类却是完全唯一的。每个数据目录的定义在本文的以后部分被描述为"预定义段"。

```
//WINNT.H

typedef struct _IMAGE_DATA_DIRECTORY {
  ULONG VirtualAddress;
  ULONG Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

每个数据目录入口指定了该目录的尺寸和相对虚拟地址。如果你要定义一个特定的目录的话，就需要从可选头部中的数据目录数组中决定相对的地址，然后使用虚拟地址来决定该目录位于哪个段中。一旦你决定了哪个段包含了该目录，该段的段头部就会被用于查找数据目录的精确文件偏移量位置。

所以要获得一个数据目录的话，那么首先你需要了解段的概念。我在下面会对其进行描述，这个讨论之后还有一个有关如何定位数据目录的示例。

PE 文件段

PE 文件规范由目前为止定义的那些头部以及一个名为"段"的一般对象组成。段包

含了文件的内容，包括代码、数据、资源以及其它可执行信息，每个段都有一个头部和一个实体（原始数据）。我将在下面描述段头部的有关信息，但是段实体则缺少一个严格的文件结构。因此，它们几乎可以被链接器按任何的方法组织，只要它的头部填充了足够能够解释数据的信息。

段头部

PE 文件格式中，所有的段头部位于可选头部之后。每个段头部为 40 个字节长，并且没有任何的填充信息。段头部被定义为以下的结构：

```
//WINNT.H

#define IMAGE_SIZEOF_SHORT_NAME 8
typedef struct _IMAGE_SECTION_HEADER {
UCHAR Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
    ULONG PhysicalAddress;
    ULONG VirtualSize;
  } Misc;
  ULONG VirtualAddress;
  ULONG SizeOfRawData;
  ULONG PointerToRawData;
  ULONG PointerToRelocations;
  ULONG PointerToLinenumbers;
  USHORT NumberOfRelocations;
  USHORT NumberOfLinenumbers;
  ULONG Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

你如何才能获得一个特定段的段头部信息？既然段头部是被连续的组织起来的，而且没有一个特定的顺序，那么段头部必须由名称来定位。以下的函数示范了如何从一个给定了段名称的 PE 映像文件中获得一个段头部：

```
//PEFILE.C

BOOL WINAPI GetSectionHdrByName(LPVOID lpFile,
IMAGE_SECTION_HEADER *sh, char *szSection)
{
  PIMAGE_SECTION_HEADER psh;
  int nSections = NumOfSections (lpFile);
```

```
  int i;
  if ((psh = (PIMAGE_SECTION_HEADER)SECHDROFFSET(lpFile))
      != NULL)
  {
    /* 由名称查找段 */
    for (i = 0; i < nSections; i++)
    {
      if (!strcmp(psh->Name, szSection))
      {
        /* 向头部复制数据 */
        CopyMemory((LPVOID)sh, (LPVOID)psh,
            sizeof(IMAGE_SECTION_HEADER));
        return TRUE;
      }
      else
        psh++;
    }
  }
  return FALSE;
}
```

这个函数通过 SECHDROFFSET 宏将第一个段头部定位，然后它开始在所有段中循环，并将要寻找的段名称和每个段的名称相比较，直到找到了正确的那一个为止。当找到了段的时候，函数将内存映像文件的数据复制到传入函数的结构中，然后 IMAGE_SECTION_HEADER 结构的各域就能够被直接存取了。

段头部的域

·Name。每个段都有一个 8 字符长的名称域，并且第一个字符必须是一个句点。

·PhysicalAddress 或 VirtualSize。第二个域是一个 union 域，现在已不使用了。

·VirtualAddress。这个域标识了进程地址空间中要装载这个段的虚拟地址。实际的地址由将这个域的值加上可选头部结构中的 ImageBase 虚拟地址得到。切记，如果这个映像文件是一个 DLL，那么这个 DLL 就不一定会装载到 ImageBase 要求的位置。所以一旦这个文件被装载进入了一个进程，实际的 ImageBase 值应该通过使用 GetModuleHandle 来检验。

·SizeOfRawData。这个域表示了相对 FileAlignment 的段实体尺寸。文件中实际的段实体尺寸将少于或等于 FileAlignment 的整倍数。一旦映像被装载进入了一个进程的地址空间，段实体的尺寸将会变得少于或等于 FileAlignment 的整倍数。

·PointerToRawData。这是一个文件中段实体位置的偏移量。

·PointerToRelocations、PointerToLinenumbers、NumberOfRelocations、NumberOfLinenumbers。这些域在 PE 格式中不使用。

·Characteristics。定义了段的特征。这些值可以在 WINNT.H 及本光盘（译注：MSDN 的光盘）的 PE 格式规范中找到。

值	定义
0x00000020	代码段
0x00000040	已初始化数据段
0x00000080	未初始化数据段
0x04000000	该段数据不能被缓存
0x08000000	该段不能被分页
0x10000000	共享段
0x20000000	可执行段
0x40000000	可读段
0x80000000	可写段

定位数据目录

数据目录存在于它们相应的数据段中。典型地来说，数据目录是段实体中的第一个结构，但不是必需的。由于这个缘故，如果你需要定位一个指定的数据目录的话，就需要从段头部和可选头部中获得信息。

为了让这个过程简单一点，我编写了以下的函数来定位任何一个在 WINNT.H 之中定义的数据目录。

```
// PEFILE.C

LPVOID WINAPI ImageDirectoryOffset(LPVOID lpFile,
    DWORD dwIMAGE_DIRECTORY)
{
  PIMAGE_OPTIONAL_HEADER poh;
  PIMAGE_SECTION_HEADER psh;
  int nSections = NumOfSections(lpFile);
  int i = 0;
  LPVOID VAImageDir;
  /* 必须为 0 到(NumberOfRvaAndSizes-1)之间 */
  if (dwIMAGE_DIRECTORY >= poh->NumberOfRvaAndSizes)
    return NULL;
  /* 获得可选头部和段头部的偏移量 */
  poh = (PIMAGE_OPTIONAL_HEADER)OPTHDROFFSET(lpFile);
```

```
  psh = (PIMAGE_SECTION_HEADER)SECHDROFFSET(lpFile);
  /* 定位映像目录的相对虚拟地址 */
  VAImageDir = (LPVOID)poh->DataDirectory
       [dwIMAGE_DIRECTORY].VirtualAddress;
  /* 定位包含映像目录的段 */
  while (i++ < nSections)
  {
    if (psh->VirtualAddress <= (DWORD)VAImageDir &&
         psh->VirtualAddress +
         psh->SizeOfRawData > (DWORD)VAImageDir)
      break;
    psh++;
  }
  if (i > nSections)
    return NULL;
  /* 返回映像导入目录的偏移量 */
  return (LPVOID)(((int)lpFile +
    (int)VAImageDir. psh->VirtualAddress) +
    (int)psh->PointerToRawData);
}
```

该函数首先确认被请求的数据目录入口数字，然后它分别获取指向可选头部和第一个段头部的两个指针。它从可选头部决定数据目录的虚拟地址，然后它使用这个值来决定数据目录定位在哪个段实体之中。如果适当的段实体已经被标识了，那么数据目录特定的位置就可以通过将它的相对虚拟地址转换为文件中地址的方法来找到。

PE 文件格式详解(下)

作者： MSDN

译者：李马 (http://home.nuc.edu.cn/~titilima)

预定义段

一个 Windows NT 的应用程序典型地拥有 9 个预定义段，它们是.text、.bss、.rdata、.data、.rsrc、.edata、.idata、.pdata 和.debug。一些应用程序不需要所有的这些段，同样还有一些应用程序为了自己特殊的需要而定义了更多的段。这种做法与 MS-DOS 和 Windows 3.1 中的代码段和数据段相似。事实上，应用程序定义一个独特的段的方法是使用标准编译器来指示对代码段和数据段的命名，或者使用名称段编译器选项-NT——就和 Windows 3.1 中应用程序定义独特的代码段和数据段一样。

以下是一个关于 Windows NT PE 文件之中一些有趣的公共段的讨论。

可执行代码段，.text

Windows 3.1 和 Windows NT 之间的一个区别就是 Windows NT 默认的做法是将所有的代码段（正如它们在 Windows 3.1 中所提到的那样）组成了一个单独的段，名为“.text”。既然 Windows NT 使用了基于页面的虚拟内存管理系统，那么将分开的代码放入不同的段之中的做法就不太明智了。因此，拥有一个大的代码段对于操作系统和应用程序开发者来说，都是十分方便的。

.text 段也包含了早先提到过的入口点。IAT 亦存在于.text 段之中的模块入口点之前。（IAT 在.text 段之中的存在非常有意义，因为这个表事实上是一系列的跳转指令，并且它们的跳转目标位置是已固定的地址。）当 Windows NT 的可执行映像装载入进程的地址空间时，IAT 就和每一个导入函数的物理地址一同确定了。要在.text 段之中查找 IAT，装载器只用将模块的入口点定位，而 IAT 恰恰出现于入口点之前。既然每个入口拥有相同的尺寸，那么向后退查找这个表的起始位置就很容易了。

数据段，.bss、.rdata、.data

.bss 段表示应用程序的未初始化数据，包括所有函数或源模块中声明为 static 的变量。

.rdata 段表示只读的数据，比如字符串文字量、常量和调试目录信息。

所有其它变量（除了出现在栈上的自动变量）存储在.data 段之中。基本上，这些是应用程序或模块的全局变量。

资源段，.rsrc

.rsrc 段包含了模块的资源信息。它起始于一个资源目录结构，这个结构就像其它大多数结构一样，但是它的数据被更进一步地组织在了一棵资源树之中。以下的 IMAGE_RESOURCE_DIRECTORY 结构形成了这棵树的根和各个结点。

```
//WINNT.H

typedef struct _IMAGE_RESOURCE_DIRECTORY {
  ULONG Characteristics;
  ULONG TimeDateStamp;
  USHORT MajorVersion;
  USHORT MinorVersion;
  USHORT NumberOfNamedEntries;
```

```
  USHORT NumberOfIdEntries;
} IMAGE_RESOURCE_DIRECTORY, *PIMAGE_RESOURCE_DIRECTORY;
```

请看这个目录结构，你将会发现其中竟然没有指向下一个结点的指针。但是，在这个结构中有两个域 NumberOfNamedEntries 和 NumberOfIdEntries 代替了指针，它们被用来表示这个目录附有多少入口。附带说一句，我的意思是目录入口就在段数据之中的目录后边。有名称的入口按字母升序出现，再往后是按数值升序排列的 ID 入口。

一个目录入口由两个域组成，正如下面 IMAGE_RESOURCE_DIRECTORY_ENTRY 结构所描述的那样：

```
// WINNT.H

typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
  ULONG Name;
  ULONG OffsetToData;
} IMAGE_RESOURCE_DIRECTORY_ENTRY,
*PIMAGE_RESOURCE_DIRECTORY_ENTRY;
```

根据树的层级不同，这两个域也就有着不同的用途。Name 域被用于标识一个资源种类，或者一种资源名称，或者一个资源的语言 ID。OffsetToData 与常常被用来在树之中指向兄弟结点——即一个目录结点或一个叶子结点。

叶子结点是资源树之中最底层的结点，它们定义了当前资源数据的尺寸和位置。IMAGE_RESOURCE_DATA_ENTRY 结构被用于描述每个叶子结点：

```
// WINNT.H

typedef struct _IMAGE_RESOURCE_DATA_ENTRY {
  ULONG OffsetToData;
  ULONG Size;
  ULONG CodePage;
  ULONG Reserved;
} IMAGE_RESOURCE_DATA_ENTRY, *PIMAGE_RESOURCE_DATA_ENTRY;
```

OffsetToData 和 Size 这两个域表示了当前资源数据的位置和尺寸。既然这一信息主要是在应用程序装载以后由函数使用的，那么将 OffsetToData 作为一个相对虚拟的地址会更有意义一些。——幸甚，恰好是这样没错。非常有趣的是，所有其它的偏移量，比如从目录入口到其它目录的指针，都是相对于根结点位置的偏移量。

要更清楚地了解这些内容，请参考图 2。

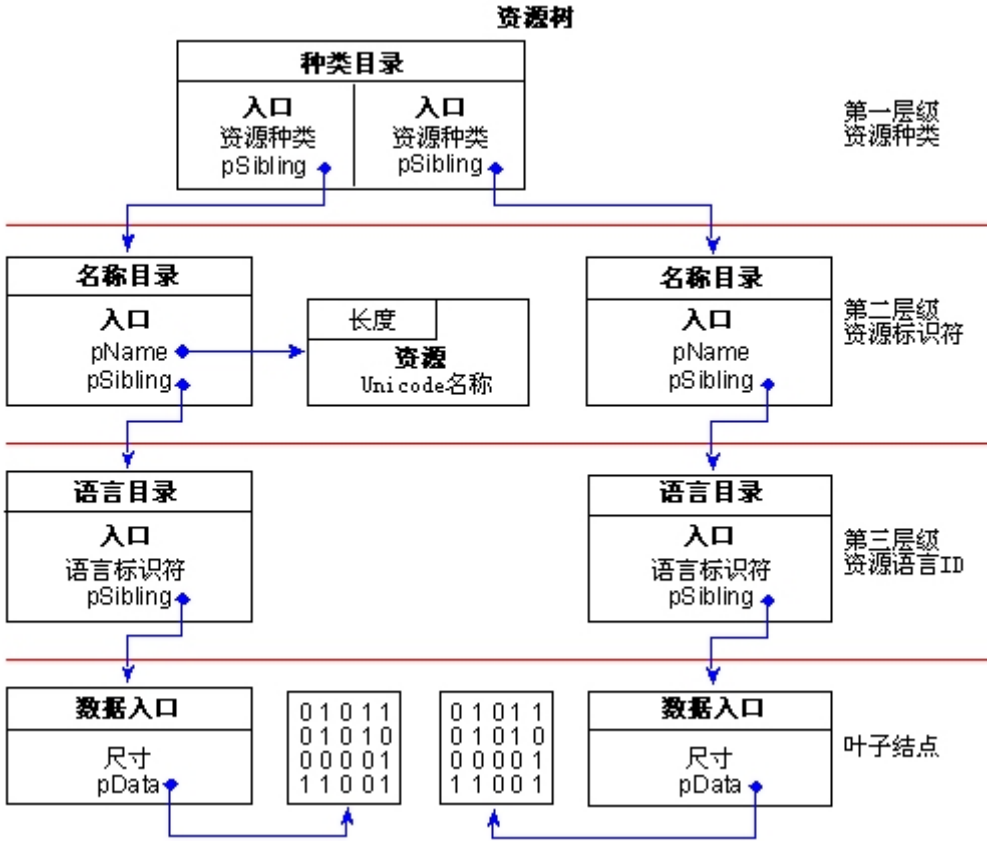

图 2.一个简单的资源树结构

图 2 描述了一个非常简单的资源树，它包含了仅仅两个资源对象：一个菜单和一个字串表。更深一层地来说，它们各自都有一个子项。然而，你仍然可以看到资源树有多么复杂——即使它像这个一样只有一点点资源。

在树的根部，第一个目录有一个文件中包含的所有资源种类的入口，而不管资源种类有多少。在图 2 中，有两个由树根标识的入口，一个是菜单的，另一个是字串表的。如果文件中拥有一个或多个对话框资源，那么根结点会再拥有一个入口，因此，就有了对话框资源的另一个分支。

WINUSER.H 中标识了基本的资源种类，我将它们列到了下面：

```
//WINUSER.H

/*
* 预定义的资源种类
*/
#define RT_CURSOR MAKEINTRESOURCE(1)
#define RT_BITMAP MAKEINTRESOURCE(2)
#define RT_ICON MAKEINTRESOURCE(3)
#define RT_MENU MAKEINTRESOURCE(4)
#define RT_DIALOG MAKEINTRESOURCE(5)
#define RT_STRING MAKEINTRESOURCE(6)
```

```
#define RT_FONTDIR MAKEINTRESOURCE(7)
#define RT_FONT MAKEINTRESOURCE(8)
#define RT_ACCELERATOR MAKEINTRESOURCE(9)
#define RT_RCDATA MAKEINTRESOURCE(10)
#define RT_MESSAGETABLE MAKEINTRESOURCE(11)
```

在树的第一层级，以上列出的 MAKEINTRESOURCE 值被放置在每个种类入口的 Name 处，它标识了不同的资源种类。

每个根目录的入口都指向了树中第二层级的一个兄弟结点，这些结点也是目录，并且每个都拥有它们自己的入口。在这一层级，目录被用来以给定的种类标识每一个资源种类。如果你的应用程序中有多个菜单，那么树中的第二层级会为每个菜单都准备一个入口。

你可能意识到了，资源可以由名称或整数标识。在这一层级，它们是通过目录结构的 Name 域来分辨的。如果如果 Name 域最重要的位被设置了，那么其它的 31 个位就会被用作一个到 IMAGE_RESOURCE_DIR_STRING_U 结构的偏移量。

```
// WINNT.H

typedef struct _IMAGE_RESOURCE_DIR_STRING_U {
  USHORT Length;
  WCHAR NameString[1];
} IMAGE_RESOURCE_DIR_STRING_U, *PIMAGE_RESOURCE_DIR_STRING_U;
```

这个结构仅仅是由一个 2 字节长的 Length 域和一个 UNICODE 字符 Length 组成的。

另一方面，如果 Name 域最重要的位被清空，那么它的低 31 位就被用于表示资源的整数 ID。图 2 示范的就是菜单资源作为一个命名的资源，以及字串表作为一个 ID 资源。

如果有两个菜单资源，一个由名称标识，另一个由资源标识，那么它们二者就会在菜单资源目录之后拥有两个入口。有名称的资源入口在第一位，之后是由整数标识的资源。目录域 NumberOfNamedEntries 和 NumberOfIdEntries 将各自包含值 1，表示当前的 1 个入口。

在第二层级的下面，资源树就不再更深一步地扩展分支了。第一层级分支至表示每个资源种类的目录中，第二层级分支至由标识符表示的每个资源的目录中，第三层级是被个别标识的资源与它们各自的语言 ID 之间一对一的映射。要表示一个资源的语言 ID，目录入口结构的 Name 域就被用来表示资源的主语言 ID 和子语言 ID 了。Windows NT 的 Win32 SDK 开发包中列出了默认的值资源，例如对于 0x0409 这个值来说，0x09 表示主语言 LANG_ENGLISH，0x04 则被定义为子语言的 SUBLANG_ENGLISH_CAN。所有的语言 ID 值都定义于 Windows NT Win32 SDK 开发包的文件 WINNT.H 中。

既然语言 ID 结点是树中最后的目录结点，那么入口结构的 OffsetToData 域就是到

一个叶子结点（即前面提到过的 IMAGE_RESOURCE_DATA_ENTRY 结构）的偏移量。

再回过头来参考图 2，你会发现每个语言目录入口都对应着一个数据入口。这个结点仅仅表示了资源数据的尺寸以及资源数据的相对虚拟地址。

在资源数据段（.rsrc）之中拥有这么多结构有一个好处，就是你可以不存取资源本身而直接可以从这个段收集很多信息。例如，你可以获得有多少种资源、哪些资源（如果有的话）使用了特别的语言 ID、特定的资源是否存在以及单独种类资源的尺寸。为了示范如何利用这一信息，以下的函数说明了如何决定一个文件中包含的不同种类的资源：

```
// PEFILE.C

int WINAPI GetListOfResourceTypes(LPVOID lpFile, HANDLE hHeap,
 char **pszResTypes)
{
  PIMAGE_RESOURCE_DIRECTORY prdRoot;
  PIMAGE_RESOURCE_DIRECTORY_ENTRY prde;
  char *pMem;
  int nCnt, i;
  /* 获得资源树的根目录 */
  if ((prdRoot =
(PIMAGE_RESOURCE_DIRECTORY)ImageDirectoryOffset
       (lpFile, IMAGE_DIRECTORY_ENTRY_RESOURCE)) == NULL)
    return 0;
  /* 在堆上分配足够的空间来包括所有类型 */
  nCnt = prdRoot->NumberOfIdEntries * (MAXRESOURCENAME + 1);
  *pszResTypes = (char *)HeapAlloc(hHeap, HEAP_ZERO_MEMORY,
       nCnt);
  if ((pMem = *pszResTypes) == NULL)
    return 0;
  /* 将指针指向第一个资源种类的入口 */
  prde = (PIMAGE_RESOURCE_DIRECTORY_ENTRY)((DWORD)prdRoot +
       sizeof (IMAGE_RESOURCE_DIRECTORY));
  /* 在所有的资源目录入口类型中循环 */
  for (i = 0; i < prdRoot->NumberOfIdEntries; i++)
  {
    if (LoadString(hDll, prde->Name, pMem, MAXRESOURCENAME))
      pMem += strlen(pMem) + 1;
    prde++;
  }
  return nCnt;
```

}

这个函数将一个资源种类名称的列表写入了由 pszResTypes 标识的变量中。请注意，在这个函数的核心部分，LoadString 是使用各自资源种类目录入口的 Name 域来作为字符串 ID 的。如果你查看 PEFILE.RC，你会发现我定义了一系列的资源种类的字符串，并且它们的 ID 与它们在目录入口中的定义完全相同。PEFILE.DLL 还有有一个函数，它返回了.rsrc 段中的资源对象总数。这样一来，从这个段中提取其它的信息，借助这些函数或另外编写函数就方便多了。

导出数据段，.edata

.edata 段包含了应用程序或 DLL 的导出数据。在这个段出现的时候，它会包含一个到达导出信息的导出目录。

```
// WINNT.H

typedef struct _IMAGE_EXPORT_DIRECTORY {
  ULONG Characteristics;
  ULONG TimeDateStamp;
  USHORT MajorVersion;
  USHORT MinorVersion;
  ULONG Name;
  ULONG Base;
  ULONG NumberOfFunctions;
  ULONG NumberOfNames;
  PULONG *AddressOfFunctions;
  PULONG *AddressOfNames;
  PUSHORT *AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

导出目录中的 Name 域标识了可执行模块的名称。NumberOfFunctions 域和 NumberOfNames 域表示模块中有多少导出的函数以及这些函数的名称。

AddressOfFunctions 域是一个到导出函数入口列表的偏移量。AddressOfNames 域是到一个导出函数名称列表起始处偏移量的地址，这个列表是由 null 分隔的。AddressOfNameOrdinals 是一个到相同导出函数顺序值（每个值 2 字节长）列表的偏移量。

三个 AddressOf...域是当模块装载时进程地址空间中的相对虚拟地址。一旦模块被装载，那么要获得进程地质空间中的确切地址的话，就应该在相对虚拟地址上加上模块的基地址。可是，在文件被装载前，仍然可以决定这一地址：只要从给定的域地址中减

去段头部的虚拟地址（VirtualAddress），再加上段实体的偏移量（PointerToRawData），这个结果就是映像文件中的偏移量了。以下的例子解说了这一技术：

```
// PEFILE.C

int WINAPI GetExportFunctionNames(LPVOID lpFile, HANDLE hHeap,
 char **pszFunctions)
{
  IMAGE_SECTION_HEADER sh;
  PIMAGE_EXPORT_DIRECTORY ped;
  char *pNames, *pCnt;
  int i, nCnt;
  /* 获得.edata 域中的段头部和指向数据目录的指针 */
  if ((ped = (PIMAGE_EXPORT_DIRECTORY)ImageDirectoryOffset
       (lpFile, IMAGE_DIRECTORY_ENTRY_EXPORT)) == NULL)
    return 0;
  GetSectionHdrByName (lpFile, &sh, ".edata");
  /* 决定导出函数名称的偏移量 */
  pNames = (char *)(*(int *)((int)ped->AddressOfNames -
    (int)sh.VirtualAddress + (int)sh.PointerToRawData +
    (int)lpFile) - (int)sh.VirtualAddress +
    (int)sh.PointerToRawData + (int)lpFile);
  /* 计算出要为所有的字符串分配多少内存 */
  pCnt = pNames;
  for (i = 0; i < (int)ped->NumberOfNames; i++)
    while (*pCnt++);
  nCnt = (int)(pCnt.pNames);
  /* 在堆上为函数名称分配内存 */
  *pszFunctions = HeapAlloc (hHeap, HEAP_ZERO_MEMORY, nCnt);
  /* 将所有字符串复制到缓冲区 */
  CopyMemory((LPVOID)*pszFunctions, (LPVOID)pNames, nCnt);
  return nCnt;
}
```

请注意，在这个函数之中，变量 pNames 是由决定偏移量地址和当前偏移量位置的方法来赋值的。偏移量的地址和偏移量本身都是相对虚拟地址，因此在使用之前必须进行转换——函数之中体现了这一点。虽然你可以编写一个类似的函数来决定顺序值或函数入口点，但是我为什么不为你做好呢？——GetNumberOfExportedFunctions、GetExportFunctionEntryPoints 和 GetExportFunctionOrdinals 已经存在于 PEFILE.DLL 之中了。

导入数据段，.idata

.idata 段是导入数据，包括导入库和导入地址名称表。虽然定义了 IMAGE_DIRECTORY_ENTRY_IMPORT，但是 WINNT.H 之中并无相应的导入目录结构。作为代替，其中有若干其它的结构，名为 IMAGE_IMPORT_BY_NAME、IMAGE_THUNK_DATA 与 IMAGE_IMPORT_DESCRIPTOR。在我个人看来，我实在不知道这些结构是如何和.idata 段发生关联的，所以我花了若干个小时来破译.idata 段实体并且得到了一个更简单的结构，我名之为 IMAGE_IMPORT_MODULE_DIRECTORY。

```
// PEFILE.H

typedef struct tagImportDirectory
{
  DWORD dwRVAFunctionNameList;
  DWORD dwUseless1;
  DWORD dwUseless2;
  DWORD dwRVAModuleName;
  DWORD dwRVAFunctionAddressList;
} IMAGE_IMPORT_MODULE_DIRECTORY,
*PIMAGE_IMPORT_MODULE_DIRECTORY;
```

和其它段的数据目录不同的是，这个是作为文件中的每个导入模块重复出现的。你可以将它看作模块数据目录列表中的一个入口，而不是一个整个数据段的数据目录。每个入口都是一个指向特定模块导入信息的目录。

IMAGE_IMPORT_MODULE_DIRECTORY 结构中的一个域 dwRVAModuleName 是一个相对虚拟地址，它指向模块的名称。结构中还有两个 dwUseless 参数，它们是为了保持段的对齐。PE 文件格式规范提到了一些东西，关于导入标记、时间/日期标志以及主/次版本，但是在我的实验中，这两个域自始而终都是空的，所以我仍然认为它们没有什么用处。

基于这个结构的定义，你便可以获得可执行文件中导入的所有模块和函数名称了。以下的函数示范了如何获得特定的 PE 文件中的所有导入函数名称：

```
//PEFILE.C

int WINAPI GetImportModuleNames(LPVOID lpFile, HANDLE hHeap,
char **pszModules)
{
  PIMAGE_IMPORT_MODULE_DIRECTORY pid;
  IMAGE_SECTION_HEADER idsh;
  BYTE *pData;
```

```
  int nCnt = 0, nSize = 0, i;
  char *pModule[1024];
  char *psz;
  pid = (PIMAGE_IMPORT_MODULE_DIRECTORY)ImageDirectoryOffset
      (lpFile, IMAGE_DIRECTORY_ENTRY_IMPORT);
  pData = (BYTE *)pid;
  /* 定位.idata 段头部 */
  if (!GetSectionHdrByName(lpFile, &idsh, ".idata"))
    return 0;
  /* 提取所有导入模块 */
  while (pid->dwRVAModuleName)
  {
    /* 为绝对字符串偏移量分配缓冲区 */
    pModule[nCnt] = (char *)(pData +
        (pid->dwRVAModuleName-idsh.VirtualAddress));
    nSize += strlen(pModule[nCnt]) + 1;
    /* 增至下一个导入目录入口 */
    pid++;
    nCnt++;
  }
  /* 将所有字符串赋值到一大块的堆内存中 */
  *pszModules = HeapAlloc(hHeap, HEAP_ZERO_MEMORY, nSize);
  psz = *pszModules;
  for (i = 0; i < nCnt; i++)
  {
    strcpy(psz, pModule[i]);
    psz += strlen (psz) + 1;
  }
  return nCnt;
}
```

这个函数非常好懂，然而有一点值得指出——注意 while 循环。这个循环当 pid->dwRVAModuleName 为 0 的时候终止，这就暗示了在 IMAGE_IMPORT_MODULE_DIRECTORY 结构列表的末尾有一个空的结构，这个结构拥有一个 0 值，至少 dwRVAModuleName 域为 0。这便是我在对文件的实验中以及之后在 PE 文件格式中研究的行为。

这个结构中的第一个域 dwRVAFunctionNameList 是一个相对虚拟地址，这个地址

指向一个相对虚拟地址的列表，这些地址是文件中的一些文件名。如下面的数据所示，所有导入模块的模块和函数名称都列于.idata 段数据中了：

```
E6A7 0000 F6A7 0000 08A8 0000 1AA8 0000 ................
28A8 0000 3CA8 0000 4CA8 0000 0000 0000 (...<...L.......
0000 4765 744F 7065 6E46 696C 654E 616D ..GetOpenFileNam
6541 0000 636F 6D64 6C67 3332 2E64 6C6C eA..comdlg32.dll
0000 2500 4372 6561 7465 466F 6E74 496E ..%.CreateFontIn
6469 7265 6374 4100 4744 4933 322E 646C directA.GDI32.dl
6C00 A000 4765 7444 6576 6963 6543 6170 l...GetDeviceCap
7300 C600 4765 7453 746F 636B 4F62 6A65 s...GetStockObje
6374 0000 D500 4765 7454 6578 744D 6574 ct....GetTextMet
7269 6373 4100 1001 5365 6C65 6374 4F62 ricsA...SelectOb
6A65 6374 0000 1601 5365 7442 6B43 6F6C ject....SetBkCol
6F72 0000 3501 5365 7454 6578 7443 6F6C or..5.SetTextCol
6F72 0000 4501 5465 7874 4F75 7441 0000 or..E.TextOutA..
```

以上的数据是 EXEVIEW.EXE 示例程序.idata 段的一部分。这个特别的段表示了导入模块列表和函数名称列表的起始处。如果你开始检查数据中的这个段，你应该认出一些熟悉的 Win32 API 函数以及模块名称。从上往下读的话，你可以找到 GetOpenFileNameA，紧接着是 COMDLG32.DLL。然后你能发现 CreateFontIndirectA，紧接着是模块 GDI32.DLL，以及之后的 GetDeviceCaps、GetStockObject、GetTextMetrics 等等。

这样的式样会在.idata 段中重复出现。第一个模块是 COMDLG32.DLL，第二个是 GDI32.DLL。请注意第一个模块只导出了一个函数，而第二个模块导出了很多函数。在这两种情况下，函数和模块的排列的方法是首先出现一个函数名，之后是模块名，然后是其它的函数名（如果有的话）。

以下的函数示范了如何获得指定模块的所有函数名。

```
// PEFILE.C

int WINAPI GetImportFunctionNamesByModule(LPVOID lpFile,
HANDLE hHeap,
    char *pszModule, char **pszFunctions)
{
  PIMAGE_IMPORT_MODULE_DIRECTORY pid;
  IMAGE_SECTION_HEADER idsh;
  DWORD dwBase;
  int nCnt = 0, nSize = 0;
  DWORD dwFunction;
  char *psz;
  /* 定位.idata 段的头部 */
```

```
  if (!GetSectionHdrByName(lpFile, &idsh, ".idata"))
    return 0;
  pid = (PIMAGE_IMPORT_MODULE_DIRECTORY)ImageDirectoryOffset
      (lpFile, IMAGE_DIRECTORY_ENTRY_IMPORT);
  dwBase = ((DWORD)pid. idsh.VirtualAddress);
  /* 查找模块的 pid */
  while (pid->dwRVAModuleName && strcmp (pszModule,
      (char *)(pid->dwRVAModuleName+dwBase)))
    pid++;
  /* 如果模块未找到，就退出 */
  if (!pid->dwRVAModuleName)
    return 0;
  /* 函数的总数和字符串长度 */
  dwFunction = pid->dwRVAFunctionNameList;
  while (dwFunction && *(DWORD *)(dwFunction + dwBase) &&
      *(char *)((*(DWORD *)(dwFunction + dwBase)) + dwBase+2))
  {
    nSize += strlen ((char *)((*(DWORD *)(dwFunction +
      dwBase)) + dwBase+2)) + 1;
    dwFunction += 4;
    nCnt++;
  }
  /* 在堆上分配函数名称的空间 */
  *pszFunctions = HeapAlloc (hHeap, HEAP_ZERO_MEMORY, nSize);
  psz = *pszFunctions;
  /* 向内存指针复制函数名称 */
  dwFunction = pid->dwRVAFunctionNameList;
  while (dwFunction && *(DWORD *)(dwFunction + dwBase) &&
    *((char *)((*(DWORD *)(dwFunction + dwBase)) + dwBase+2)))
  {
    strcpy (psz, (char *)((*(DWORD *)(dwFunction + dwBase)) +
        dwBase+2));
    psz += strlen((char *)((*(DWORD *)(dwFunction + dwBase))+
        dwBase+2)) + 1;
    dwFunction += 4;
  }
  return nCnt;
}
```

就像 GetImportModuleNames 函数一样，这一函数依靠每个信息列表的末端来获得一个置零的入口。这在种情况下，函数名称列表就是以零结尾的。

最后一个域 dwRVAFunctionAddressList 是一个相对虚拟地址，它指向一个虚拟地址表。在文件装载的时候，这个虚拟地址表会被装载器置于段数据之中。但是在文件装载前，这些虚拟地址会被一些严密符合函数名称列表的虚拟地址替换。所以在文件装载之前，有两个同样的虚拟地址列表，它们指向导入函数列表。

调试信息段，.debug

调试信息位于.debug 段之中，同时 PE 文件格式也支持单独的调试文件（通常由.DBG 扩展名标识）作为一种将调试信息集中的方法。调试段包含了调试信息，但是调试目录却位于早先提到的.rdata 段之中。这其中每个目录都涉及了.debug 段之中的调试信息。调试目录的结构 IMAGE_DEBUG_DIRECTORY 被定义为：

```
// WINNT.H

typedef struct _IMAGE_DEBUG_DIRECTORY {
  ULONG Characteristics;
  ULONG TimeDateStamp;
  USHORT MajorVersion;
  USHORT MinorVersion;
  ULONG Type;
  ULONG SizeOfData;
  ULONG AddressOfRawData;
  ULONG PointerToRawData;
} IMAGE_DEBUG_DIRECTORY, *PIMAGE_DEBUG_DIRECTORY;
```

这个段被分为单独的部分，每个部分为不同种类的调试信息数据。对于每个部分来说都是一个像上边一样的调试目录。不同的调试信息种类如下：

```
// WINNT.H

#define IMAGE_DEBUG_TYPE_UNKNOWN 0
#define IMAGE_DEBUG_TYPE_COFF 1
#define IMAGE_DEBUG_TYPE_CODEVIEW 2
#define IMAGE_DEBUG_TYPE_FPO 3
#define IMAGE_DEBUG_TYPE_MISC 4
```

每个目录之中的 Type 域表示该目录的调试信息种类。如你所见，在上边的表中，PE 文件格式支持很多不同的调试信息种类，以及一些其它的信息域。对于那些来说，

IMAGE_DEBUG_TYPE_MISC 信息是唯一的。这一信息被添加到描述可执行映像的混杂信息之中，这些混杂信息不能被添加到 PE 文件格式任何结构化的数据段之中。这就是映像文件中最合适的位置，映像名称则肯定会出现在这里。如果映像导出了信息，那么导出数据段也会包含这一映像名称。

每种调试信息都拥有自己的头部结构，该结构定义了它自己的数据。这些结构都列于 WINNT.H 之中。关于 IMAGE_DEBUG_DIRECTORY 一件有趣的事就是它包括了两个标识调试信息的域。第一个是 AddressOfRawData，为相对文件装载的数据虚拟地址；另一个是 PointerToRawData，为数据所在 PE 文件之中的实际偏移量。这就使得定位指定的调试信息相当容易了。

作为最后的例子，请你考虑以下的函数代码，它从 IMAGE_DEBUG_MISC 结构中提取了映像名称。

```
//PEFILE.C

int WINAPI RetrieveModuleName(LPVOID lpFile, HANDLE hHeap,
char **pszModule)
{
  PIMAGE_DEBUG_DIRECTORY pdd;
  PIMAGE_DEBUG_MISC pdm = NULL;
  int nCnt;
  if (!(pdd =
(PIMAGE_DEBUG_DIRECTORY)ImageDirectoryOffset(lpFile,
       IMAGE_DIRECTORY_ENTRY_DEBUG)))
  return 0;
  while (pdd->SizeOfData)
  {
    if (pdd->Type == IMAGE_DEBUG_TYPE_MISC)
    {
      pdm = (PIMAGE_DEBUG_MISC)((DWORD)pdd->PointerToRawData
+ (DWORD)lpFile);
      nCnt = lstrlen(pdm->Data) * (pdm->Unicode ? 2 : 1);
      *pszModule = (char *)HeapAlloc(hHeap, HEAP_ZERO_MEMORY,
nCnt+1);
      CopyMemory(*pszModule, pdm->Data, nCnt);
      break;
    }
    pdd ++;
  }
  if (pdm != NULL)
```

```
    return nCnt;
  else
    return 0;
}
```

你看到了，调试目录结构使得定位一个特定种类的调试信息变得相对容易了些。只要定位了 IMAGE_DEBUG_MISC 结构，提取映像名称就如同调用 CopyMemory 函数一样简单。

如上所述，调试信息可以被剥离到单独的 .DBG 文件中。Windows NT SDK 包含了一个名为 REBASE.EXE 的程序可以实现这一目的。例如，以下的语句可以将一个名为 TEST.EXE 的调试信息剥离：

```
rebase -b 40000 -x c:\samples\testdir test.exe
```

调试信息被置于一个新的文件中，这个文件名为 TEST.DBG，位于 c:\samples\testdir 之中。这个文件起始于一个单独的 IMAGE_SEPARATE_DEBUG_HEADER 结构，接着是存在于原可执行映像之中的段头部的一份拷贝。在段头部之后，是 .debug 段的数据。也就是说，在段头部之后，就是一系列的 IMAGE_DEBUG_DIRECTORY 结构及其相关的数据了。调试信息本身保留了如上所描述的常规映像文件调试信息。

PE 文件格式总结

Windows NT 的 PE 文件格式向熟悉 Windows 和 MS-DOS 环境的开发者引入了一种全新的结构。然而熟悉 UNIX 环境的开发者会发现 PE 文件格式与 COFF 规范很相像（如果它不是以 COFF 为基础的话）。

整个格式的组成：一个 MS-DOS 的 MZ 头部，之后是一个实模式的残余程序、PE 文件标志、PE 文件头部、PE 可选头部、所有的段头部，最后是所有的段实体。

可选头部的末尾是一个数据目录入口的数组，这些相对虚拟地址指向段实体之中的数据目录。每个数据目录都表示了一个特定的段实体数据是如何组织的。

PE 文件格式有 11 个预定义段，这是对 Windows NT 应用程序所通用的，但是每个应用程序可以为它自己的代码以及数据定义它自己独特的段。

.debug 预定义段也可以分离为一个单独的调试文件。如果这样的话，就会有一个特定的调试头部来用于解析这个调试文件，PE 文件中也会有一个标志来表示调试数据被分离了出去。

PEFILE.DLL 函数描述

PEFILE.DLL 主要由一些函数组成，这些函数或者被用来获得一个给定的 PE 文件中的偏移量，或者被用来把文件中的一些数据复制到一个特定的结构中去。每个函数都有一个需求——第一个参数是一个指针，这个指针指向 PE 文件的起始处。也就是说，这个文件必须首先被映射到你进程的地址空间中，然后映射文件的位置就可以作为每个函数

第一个参数的 lpFile 的值来传入了。

我意在使函数的名称使你能够一见而知其意，并且每个函数都随一个详细描述其目的的注释而列出。如果在读完函数列表之后，你仍然不明白某个函数的功能，那么请参考 EXEVIEW.EXE 示例来查明这个函数是如何使用的。以下的函数原型列表可以在 PEFILE.H 中找到：

```
// PEFILE.H

/* 获得指向 MS-DOS MZ 头部的指针 */
BOOL WINAPI GetDosHeader(LPVOID, PIMAGE_DOS_HEADER);

/* 决定.EXE 文件的类型 */
DWORD WINAPI ImageFileType(LPVOID);

/* 获得指向 PE 文件头部的指针 */
BOOL WINAPI GetPEFileHeader(LPVOID, PIMAGE_FILE_HEADER);

/* 获得指向 PE 可选头部的指针 */
BOOL WINAPI GetPEOptionalHeader(LPVOID,
PIMAGE_OPTIONAL_HEADER);

/* 返回模块入口点的地址 */
LPVOID WINAPI GetModuleEntryPoint(LPVOID);

/* 返回文件中段的总数 */
int WINAPI NumOfSections(LPVOID);

/* 返回当可执行文件被装载入进程地址空间时的首选基地址 */
LPVOID WINAPI GetImageBase(LPVOID);

/* 决定文件中一个特定的映像数据目录的位置 */
LPVOID WINAPI ImageDirectoryOffset(LPVOID, DWORD);

/* 获得文件中所有段的名称 */
int WINAPI GetSectionNames(LPVOID, HANDLE, char **);

/* 复制一个特定段的头部信息 */
BOOL WINAPI GetSectionHdrByName(LPVOID, PIMAGE_SECTION_HEADER,
 char *);
```

```
/* 获得由空字符分隔的导入模块名称列表 */
int WINAPI GetImportModuleNames(LPVOID, HANDLE, char **);

/* 获得一个模块由空字符分隔的导入函数列表 */
int WINAPI GetImportFunctionNamesByModule(LPVOID, HANDLE,
char *, char **);

/* 获得由空字符分隔的导出函数列表 */
int WINAPI GetExportFunctionNames(LPVOID, HANDLE, char **);

/* 获得导出函数总数 */
int WINAPI GetNumberOfExportedFunctions(LPVOID);

/* 获得导出函数的虚拟地址入口点列表 */
LPVOID WINAPI GetExportFunctionEntryPoints(LPVOID);

/* 获得导出函数顺序值列表 */
LPVOID WINAPI GetExportFunctionOrdinals(LPVOID);

/* 决定资源对象的种类 */
int WINAPI GetNumberOfResources (LPVOID);

/* 返回文件中所使用的所有资源对象的种类 */
int WINAPI GetListOfResourceTypes(LPVOID, HANDLE, char **);

/* 决定调试信息是否已从文件中分离 */
BOOL WINAPI IsDebugInfoStripped(LPVOID);

/* 获得映像文件名称 */
int WINAPI RetrieveModuleName(LPVOID, HANDLE, char **);

/* 决定文件是否是一个有效的调试文件 */
BOOL WINAPI IsDebugFile(LPVOID);

/* 从调试文件中返回调试头部 */
BOOL WINAPI GetSeparateDebugHeader(LPVOID,
PIMAGE_SEPARATE_DEBUG_HEADER);
```

除了以上所列的函数之外，本文中早先提到的宏也定义在了PEFILE.H 中，完整的列表如下：

```
/* PE 文件标志的偏移量 */
#define NTSIGNATURE(a) ((LPVOID)((BYTE *)a + \
                        ((PIMAGE_DOS_HEADER)a)->e_lfanew))

/* MS 操作系统头部标识了双字的 NT PE 文件标志；PE 文件头部就紧跟在这个双字之后 */
#define PEFHDROFFSET(a) ((LPVOID)((BYTE *)a + \
                        ((PIMAGE_DOS_HEADER)a)->e_lfanew + \
                        SIZE_OF_NT_SIGNATURE))

/* PE 可选头部紧跟在 PE 文件头部之后 */
#define OPTHDROFFSET(a) ((LPVOID)((BYTE *)a + \
                        ((PIMAGE_DOS_HEADER)a)->e_lfanew + \
                        SIZE_OF_NT_SIGNATURE + \
                        sizeof(IMAGE_FILE_HEADER)))

/* 段头部紧跟在 PE 可选头部之后 */
#define SECHDROFFSET(a) ((LPVOID)((BYTE *)a + \
                        ((PIMAGE_DOS_HEADER)a)->e_lfanew + \
                        SIZE_OF_NT_SIGNATURE + \
                        sizeof(IMAGE_FILE_HEADER) + \
                        sizeof(IMAGE_OPTIONAL_HEADER)))
```

要使用 PEFILE.DLL，你只用包含 PEFILE.H 文件并在应用程序中链接到这个 DLL 即可。所有的这些函数都是互斥性的函数，但是有些函数的功能可以相互支持以获得文件信息。例如，GetSectionNames 可以用于获得所有段的名称，这样一来，为了获得一个拥有独特段名称（在编译期由应用程序开发者定义的）的段头部，你就需要首先获得所有名称的列表，然后再对那个准确的段名称调用函数 GetSectionHeaderByName 了。现在，你可以享受我为你带来的这一切了！