Microsoft Windows

Microsoft 可移植可执行文件和通用目标文件格式文件规范

修订版 8.0 — 2006 年 5 月 16 日

摘要

本规范描述了 Microsoft® Windows®操作系统家族下的可执行文件（映像）和目标文件的结构。这些文件分别被称为可移植可执行（PE）文件和通用目标文件格式（COFF）文件。

注意

提供本文档是为了辅助开发用于 Microsoft Windows 操作系统上的工具和应用程序，但并不保证它在各个方面都是完整的规范。Microsoft 保留更改本文档而不通知的权利。

Microsoft 可移植可执行文件和通用目标文件格式文件规范的此次修订版取代了本规范的 6.0 修订版。

本规范的最新版在万维网的以下地址被维护：
http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

SmartTech 译

电子信箱：zhzhtst@163.com

法律声明

Microsoft 可移植可执行文件和通用目标文件格式文件规范
Microsoft Corporation
修订版 8.0

注意：*提供本规范是为了辅助开发某些用于 Microsoft Windows 操作系统平台上的开发工具。但是 Microsoft 并不保证它在各个方面都是完整的规范，也无法保证这里的所有信息在发布之后一直都是准确的。Microsoft 保留更改本规范而不通知的权利。*

在合理的和非歧视性条款和条件下，Microsoft 将针对任何 Microsoft 认为仅在面向 Microsoft Windows 的被称为编译器、链接器以及汇编程序的软件开发工具中实现和遵守本规范中所需部分这种有限用途下所需要的 Microsoft 权利要求书（如果存在）授予您免版税许可。

遵守所有适用的版权法是用户的责任。在不限制版权许可权利的前提下，未经 Microsoft 明确书面许可，不得出于任何目的或以任何形式或任何手段（电子、机械、影印、记录或其他方法）复制本规范的任何部分，或者将其存储或引入检索系统，或者修改或用于其衍生作品，或者将其进行传播。

对于本规范中的主题，Microsoft 可能拥有知识产权。除非在书面许可中明确规定，否则提供本规范并不意味着 Microsoft 通过暗示、默许或其它方式针对这些知识产权和其它权利授予您任何许可。

本规范依“原样”提供。Microsoft 不针对下列情形作任何明示的、暗示的或者法定的表示或担保：(1)本规范中的信息，包括对适销性、针对特定目的的适用性、不侵权或所有权的担保；(2)本规范中的内容适用于任何用途；(3)对这些内容的实现不会侵犯任何第三方的专利、版权、商标或其它权利。

Microsoft 不就由本规范及其相关的使用或分发所带来的任何直接的、间接的、特别的、偶然的或者后果性的损害负任何责任。

Microsoft、MS-DOS、MSDN、Visual Studio、Visual C++、Win32、 Windows、Windows NT、Windows Server 和 Windows Vista 是 Microsoft Corporation 在美国和/或其它国家的注册商标或商标。这里提到的其它产品和公司名称可能是其各自所有者的商标。

对于上述名称和商标，在没有事先得到其各自所有者明确书面许可的情况下，不得以任何方式使用，包括广告或宣传本规范及其内容。

目录

1 基本概念

本文档详细说明了 Microsoft® Windows®操作系统家族下的可执行文件（映像）和目标文件的结构。这些文件分别被称为可移植可执行（PE）文件和通用目标文件格式（COFF）文件。“可移植可执行”这个名称道出了这种格式与平台体系结构无关的事实。

下表描述了贯穿于本规范中的一些概念：

名称	描述
属性证书	用来将可校验的声明与映像关联起来的证书。有许多不同的可校验声明可以与文件关联，最常用的一种就是软件制造商用来指明映像的消息摘要是什么的声明。消息摘要与校验和类似，但想要伪造它却极其困难。因此对一个文件进行修改并保持它的消息摘要与原始文件一致是非常困难的。正如制造商所做的那样，可以使用公钥或私钥加密机制来校验声明。本文档除了描述允许将它们插入到映像文件中外还描述了有关属性证书的详细信息。
日期/时间戳	由于不同目的而用于 PE 或 COFF 文件中好几个地方的戳。戳的格式与 C 运行时库时间函数所使用的戳的格式相同。
文件指针	某项内容在文件被链接器处理前（如果是目标文件文件）或者被加载器处理前（如果是映像文件）在文件自身中的位置。换句话说，这是一个位于存储在磁盘上的文件中的位置。
链接器	指的是随 Microsoft Visual Studio®提供的链接器。
映像文件	可执行文件：或者是.EXE 文件，或者是 DLL。映像文件可以被认为是“内存映像”。术语“映像文件”经常用来代替“可执行文件”，因为后者有时仅用来指代.EXE 文件。
目标文件	作为链接器的输入的文件。链接器生成一个映像文件，而这个映像文件又作为加载器的输入。
保留，必须为 0	对一个域的这种描述表明，对于生成这个域的程序来说必须将这个域设置为 0，对于使用这个域的程序来说必须忽略它的值。
RVA	相对虚拟地址。对于映像文件来说，它是某项内容被加载进内存后的地址减去映像文件的基地址。某项内容的 RVA 几乎总是与它在磁盘上的文件中的位置（文件指针）不同。 对于目标文件来说，RVA 并没有什么意义，因为内存位置尚未分配。在这种情况下，RVA 是一个节（后面将要描述）中的地址，这个地址在以后链接时要被重定位。为了简单起见，编译器应该将每个节的首个 RVA 设置为 0。
节	节是 PE 或 COFF 文件中代码或数据的基本单元。例如一个目标文件中所有代码可以被组合成单个节，或者（依赖于编译器的行为）每个函数独占一个节。增加节的数目会增加文件开销，但是链接器在链接代码时有更大的选择余地。节与 Intel 8086 体系结构中的段非常相似。一个节中的所有原始数据必须被加载到连续的内存中。另外，映像文件可能包含一些具有特殊用途的节，例如.tls 节或.reloc 节。
VA	虚拟地址。除了不减去映像文件的基地址外，与 RVA 相同。这个地址之所以被叫做“虚拟地址”是因为 Windows 为每个进程创建一个私有的虚拟地址空间，它独立于物理内存。无论出于何种目的，VA 都只应该被认为是一个地址。VA 并不能像 RVA 那样能够预先得到，因为加载器可能不把映像加载到它的首选位置上。

2 概览

图 1 解释了 Microsoft PE 可执行文件格式：

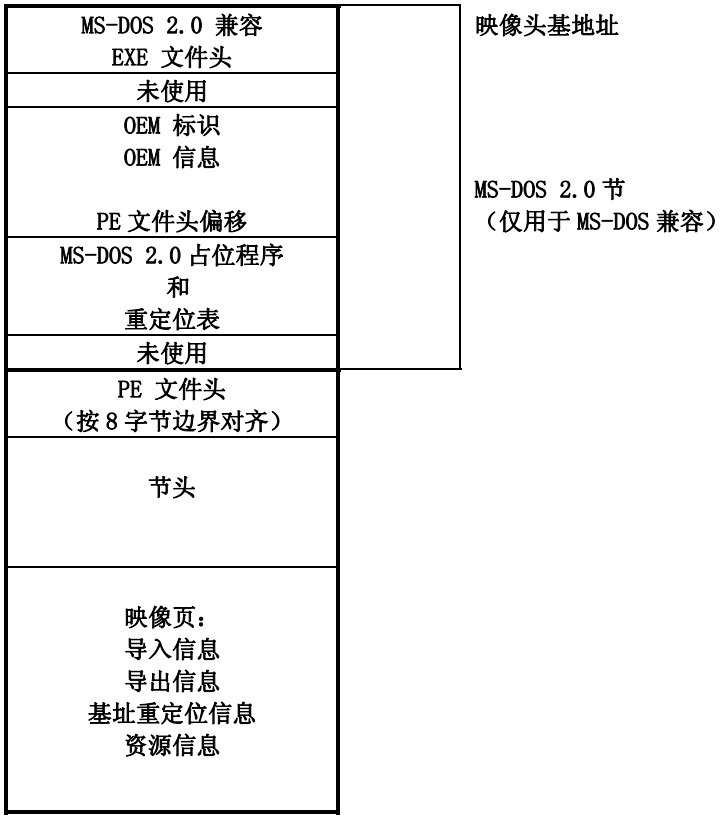

图 1.典型的可移植 EXE 文件布局

图 2 解释了 Microsoft COFF 目标模块格式：

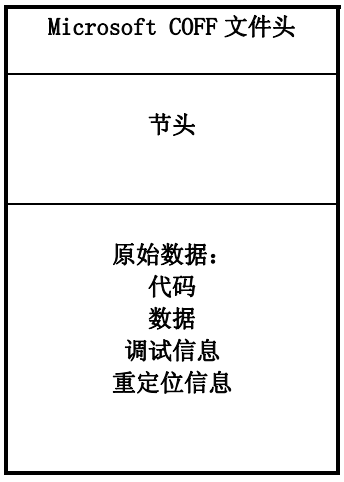

图 2. 典型的 COFF 目标模块布局

3 文件头

PE 文件头由 Microsoft MS-DOS®占位程序、PE 文件签名、COFF 文件头以及可选文件头组成，COFF 目标文件头由 COFF 文件头和可选文件头组成。在这两种情况下，文件头后面紧跟着的都是节头。

3.1 MS-DOS 占位程序（仅适用于映像文件）

MS-DOS 占位程序是一个运行于 MS-DOS 下的合法应用程序,它被放在 EXE 映像的最前面。链接器在这里放一个默认的占位程序，当映像运行于 MS-DOS 下时，这个占位程序显示“This program cannot be run in DOS mode（此程序不能在 DOS 模式下运行）”这条消息。用户可以使用/STUB 链接器选项来指定一个不同的占位程序。

在位置 0x3C 处，这个占位程序包含 PE 文件签名的偏移地址。此信息的存在使得即使映像文件中有一个 MS-DOS 占位程序，Windows 仍然能够正常执行它。这个文件偏移是在链接时被放在 0x3C 处的。

3.2 签名（仅适用于映像文件）

在 MS-DOS 占位程序后面、在偏移 0x3C 指定的文件偏移处，是一个 4 字节的签名，它用来标识文件为一个 PE 格式的映像文件。这个签名是“PE\0\0”（字母“P”和“E”后跟着两个空字节）。

3.3 COFF 文件头（适用于目标文件和映像文件）

在目标文件的开头，或者紧跟着映像文件签名之后，是一个如下格式的标准 COFF 文件头。注意 Windows 加载器限制节的最大数目为 96。

偏移	大小	域	描述
0	2	Machine	标识目标机器类型的数字。要获取更多信息，请参考 3.3.1 节“机器类型”。
2	2	NumberOfSections	节的数目。它给出了节表的大小，而节表紧跟着文件头。
4	4	TimeDateStamp	从 UTC 时间 1970 年 1 月 1 日 00:00 起的总秒数（一个 C 运行时 time_t 类型的值）的低 32 位，它指出文件何时被创建。
8	4	PointerToSymbolTable	COFF 符号表的文件偏移。如果不存在 COFF 符号表，此值为 0。对于映像文件来说，此值应该为 0，因为已经不赞成使用 COFF 调试信息了。
12	4	NumberOfSymbols	符号表中的元素数目。由于字符串表紧跟符号表，所以可以利用这个值来定位字符串表。对于映像文件来说，此值应该为 0，因为已经不赞成使用 COFF 调试信息了。
16	2	SizeOfOptionalHeader	可选文件头的大小。可执行文件需要可选文件头而目标文件并不需要。对于目标文件来说，此值应该为 0。要获取可选文件头格式的详细描述，请参考 3.4 节“可选文件头（仅适用于映像文件）”。
18	2	Characteristics	指示文件属性的标志。对于特定的标志，请参考 3.3.2 节“特征”。

3.3.1 机器类型

Machine 域可以取以下各值中的一个来指定 CPU 类型。 映像文件仅能运行于指定处理器或者能够模拟指定处理器的系统上。

常量	值	描述
IMAGE_FILE_MACHINE_UNKNOWN	0x0	适用于任何类型处理器
IMAGE_FILE_MACHINE_AM33	0x1d3	Matsushita AM33 处理器
IMAGE_FILE_MACHINE_AMD64	0x8664	x64 处理器
IMAGE_FILE_MACHINE_ARM	0x1c0	ARM 小尾处理器
IMAGE_FILE_MACHINE_EBC	0xebc	EFI 字节码处理器
IMAGE_FILE_MACHINE_I386	0x14c	Intel 386 或后继处理器及其兼容处理器
IMAGE_FILE_MACHINE_IA64	0x200	Intel Itanium 处理器家族
IMAGE_FILE_MACHINE_M32R	0x9041	Mitsubishi M32R 小尾处理器
IMAGE_FILE_MACHINE_MIPS16	0x266	MIPS16 处理器
IMAGE_FILE_MACHINE_MIPSFPU	0x366	带 FPU 的 MIPS 处理器
IMAGE_FILE_MACHINE_MIPSFPU16	0x466	带 FPU 的 MIPS16 处理器
IMAGE_FILE_MACHINE_POWERPC	0x1f0	PowerPC 小尾处理器
IMAGE_FILE_MACHINE_POWERPCFP	0x1f1	带符点运算支持的 PowerPC 处理器
IMAGE_FILE_MACHINE_R4000	0x166	MIPS 小尾处理器
IMAGE_FILE_MACHINE_SH3	0x1a2	Hitachi SH3 处理器
IMAGE_FILE_MACHINE_SH3DSP	0x1a3	Hitachi SH3 DSP 处理器
IMAGE_FILE_MACHINE_SH4	0x1a6	Hitachi SH4 处理器
IMAGE_FILE_MACHINE_SH5	0x1a8	Hitachi SH5 处理器
IMAGE_FILE_MACHINE_THUMB	0x1c2	Thumb 处理器
IMAGE_FILE_MACHINE_WCEMIPSV2	0x169	MIPS 小尾 WCE v2 处理器

3.3.2 特征

Characteristics 域包含指示目标文件或映像文件属性的标志。当前定义了以下值：

标志	值	描述
IMAGE_FILE_RELOCS_STRIPPED	0x0001	仅适用于映像文件，适用于 Windows CE、Microsoft Windows NT® 及其后继操作系统。它表明此文件不包含基址重定位信息，因此必须被加载到其首选基地址上。如果基地址不可用，加载器会报错。链接器默认会移除可执行（EXE）文件中的重定位信息。
IMAGE_FILE_EXECUTABLE_IMAGE	0x0002	仅适用于映像文件。它表明此映像文件是合法的，可以被运行。如果未设置此标志，表明出现了链接器错误。
IMAGE_FILE_LINE_NUMS_STRIPPED	0x0004	COFF 行号信息已经被移除。不赞成使用此标志，它应该为 0。
IMAGE_FILE_LOCAL_SYMS_STRIPPED	0x0008	COFF 符号表中有关局部符号的项已经被移除。不赞成使用此标志，它应该为 0。
IMAGE_FILE_AGGRESSIVE_WS_TRIM	0x0010	此标志已经被舍弃。它用于调整工作集。Windows 2000 及其后继操作系统不赞成使用此标志，它应该为 0。
IMAGE_FILE_LARGE_ADDRESS_AWARE	0x0020	应用程序可以处理大于 2GB 的地址。
	0x0040	此标志保留供将来使用。
IMAGE_FILE_BYTES_REVERSED_LO	0x0080	小尾：在内存中，最不重要位（LSB）在最重要位（MSB）前面。不赞成使用此标志，它应该为 0。
IMAGE_FILE_32BIT_MACHINE	0x0100	机器类型基于 32 位字体系结构。
IMAGE_FILE_DEBUG_STRIPPED	0x0200	调试信息已经从此映像文件中移除。
IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP	0x0400	如果此映像文件在可移动介质上，完全加载它并把它复制到交换文件中。
IMAGE_FILE_NET_RUN_FROM_SWAP	0x0800	如果此映像文件在网络介质上，完全加载它并把它复制到交换文件中。
IMAGE_FILE_SYSTEM	0x1000	此映像文件是系统文件，而不是用户程序。
IMAGE_FILE_DLL	0x2000	此映像文件是动态链接库（DLL）。这样的文件总被认为是可执行文件，尽管它们并不能直接被运行。
IMAGE_FILE_UP_SYSTEM_ONLY	0x4000	此文件只能运行于单处理器机器上。
IMAGE_FILE_BYTES_REVERSED_HI	0x8000	大尾：在内存中，MSB 在 LSB 前面。不赞成使用此标志，它应该为 0。

3.4 可选文件头（仅适用于映像文件）

每个映像文件都有一个可选文件头，用于为加载器提供信息。这个文件头之所以说是可选的是从一些文件（明确地说是目标文件）并不包含它这层意义上来说的。对于映像文件来说，这个文件头是必须的。目标文件可以包含一个可选文件头，但是通常它除了增加文件的大小外并无其它作用。

注意可选文件头的大小并不固定。COFF 文件头中的 SizeOfOptionalHeader 域用来验证某个特别的数据目录不超过 SizeOfOptionalHeader。要获取更多信息，请参考 3.3 节“COFF 文件头（适用于目标文件和映像文件）”

可选文件头中的 NumberOfRvaAndSizes 域也应该被用来确保某个特别的数据目录不超出可选文件头。另外出于格式兼容目的而去校验可选文件头中的幻数也很重要。

可选文件头中的幻数决定一个映像是 PE32 还是 PE32+格式的可执行文件：

幻数	PE 格式
0x10b	PE32
0x20b	PE32+

PE32+映像在限制映像大小最大为 2G 时可以访问 64 位地址空间。PE32+的其它更改在它们各自的节中描述。

可选文件头自身由三个主要部分组成：

偏移（PE32/PE32+）	大小（PE32/PE32+）	文件头部分	描述
0	28/24	标准域	这些域被所有 COFF 实现所定义，其中包括 UNIX。
28/24	68/88	Windows 特定域	支持 Windows 特性（例如子系统）的附加域。
96/112	Variable	数据目录	映像文件中的特殊表（例如导入表和导出表）的地址/大小组合，它们供操作系统使用。

3.4.1 可选文件头中的标准域（仅适用于映像文件）

可选文件头的前 8 个域为标准域，它们被所有 COFF 实现所定义。这些域包含对加载和运行可执行文件有用的常规信息。它们在 PE32+中并未改变。

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。

3.4.2 可选文件头中的 Windows 特定域（仅适用于映像文件）

接下来的 21 个域是 COFF 可选文件头格式的扩展。它们包含 Windows 中的链接器和加载器所必需的附加信息。

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。要获取更多信息，请参考后面的“Windows 子系统”部分。
70/70	2	DllCharacteristics	要获取更多信息，请参考后面的“DLL 特征”部分。

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。

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。

常量	值	描述
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。

3.4.3 可选文件头中的数据目录（仅适用于映像文件）

每个数据目录给出了 Windows 使用的表或字符串的地址和大小。这些数据目录项全部被被加载进内存以备系统运行时使用。数据目录是按照如下格式定义的一个 8 字节结构：

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

第一个域——VirtualAddress，实际上是表的 RVA。这个 RVA 是当表被加载进内存时相对于映像基地址的偏移地址。第二个域给出了表的大小（以字节计）。 数据目录组成了可选文件头的最后一部分，它被列于下表中。

注意目录的数目是不固定的。在查看一个特定的目录之前，先检查可选文件头中的 NumberOfRvaAndSizes 域。

同样，不要想当然地认为在这个表中的 RVA 指向节的开头，或者认为包含特定表的节有特定的名字。

偏移（PE/PE32+）	大小	域	描述
96/112	8	Export Table	导出表的地址和大小。要获取更多信息，请参考 6.3 节“.edata 节（仅适用于映像文件）”。
104/120	8	Import Table	导入表的地址和大小。要获取更多信息，请参考 6.4 节“.idata 节”。
112/128	8	Resource Table	资源表的地址和大小。要获取更多信息，请参考 6.9 节“.rsrc 节”。
120/136	8	Exception Table	异常表的地址和大小。要获取更多信息，请参考 6.5 节“.pdata 节”。
128/144	8	Certificate Table	属性证书表的地址和大小。要获取更多信息，请参考 5.7 节“属性证书表（仅适用于映像文件）”
136/152	8	Base Relocation Table	基址重定位表的地址和大小。要获取更多信息，请参考 6.6 节“.reloc 节（仅适用于映像文件）”。
144/160	8	Debug	调试数据起始地址和大小。要获取更多信息，请参考 6.1 节“.debug 节”。
152/168	8	Architecture	保留，必须为 0。
160/176	8	Global Ptr	将被存储在全局指针寄存器中的一个值的 RVA。这个结构的 Size 域必须为 0。
168/184	8	TLS Table	线程局部存储（TLS）表的地址和大小。要获取更多信息，请参考 6.7 节“.tls 节”。

偏移（PE/PE32+）	大小	域	描述
176/192	8	Load Config Table	加载配置表的地址和大小。要获取更多信息，请参考 6.8 节“加载配置结构（仅适用于映像文件）”。
184/200	8	Bound Import	绑定导入表的地址和大小。
192/208	8	IAT	导入地址表的地址和大小。要获取更多信息，请参考 6.4.4 节“导入地址表”。
200/216	8	Delay Import Descriptor	延迟导入描述符的地址和大小。要获取更多信息，请参考 5.8 节“延迟加载导入表（仅适用于映像文件）”。
208/224	8	CLR Runtime Header	CLR 运行时头部的地址和大小。要获取更多信息，请参考 6.10 节“.cormeta 节（仅适用于目标文件）”
216/232	8	保留，必须为 0。	

Certificate Table 域指向属性证书表。这些证书并不作为映像的一部分被加载进内存。此时它的第一个域是一个文件指针，而不是通常的 RVA。

4 节表（节头）

节表的每一行等效于一个节头。这个表紧跟可选文件头（如果存在的话）。之所以必须在这个位置是因为文件头中并没有一个直接指向节表的指针，节表的位置是通过计算文件头后的第一个字节的位置来得到的。一定要确保使用文件头中指定的可选文件头的大小（来进行计算）。

节表中的元素数目由文件头中的 NumberOfSections 域给出。而元素的编号是从 1 开始的。表示代码节和数据节的元素的顺序由链接器决定。

在映像文件中，每个节的 VA 值必须由链接器决定。这样能够保证这些节位置相邻且按升序排列，并且这些 VA 值必须是可选文件头中 SectionAlignment 域的倍数。

每个节头（节表项）格式如下，共 40 个字节：

偏移	大小	域	描述
0	8	Name	这是一个 8 字节的 UTF-8 编码的字符串，不足 8 字节时用 NULL 填充。如果它正好是 8 字节，那就没有最后的 NULL 字符。如果名称更长的话，这个域中是一个斜杠（/）后跟一个用 ASCII 码表示的十进制数，这个十进制数表示字符串表中的偏移。可执行映像不使用字符串表也不支持长度超过 8 字节的节名。如果目标文件中有长节名的节最后要出现在可执行文件中，那么相应的长节名会被截断。
8	4	VirtualSize	当加载进内存时这个节的总大小。如果此值比 SizeOfRawData 大，那么多出的部分用 0 填充。这个域仅对可执行映像是合法的，对于目标文件来说，它应该为 0。
12	4	VirtualAddress	对于可执行映像来说，这个域的值是这个节被加载进内存之后它的第一个字节相对于映像基址的偏移地址。对于目标文件来说，这个域的值是没有重定位之前其第一个字节的地址；为了简单起见，编译器应该把此值设置为 0。否则这个值是个任意值，但是在重定位时应该从偏移地址中减去这个值。

偏移	大小	域	描述
16	4	SizeOfRawData	（对于目标文件来说）节的大小或者（对于映像文件来说）磁盘文件中已初始化数据的大小。对于可执行映像来说，它必须是可选文件头中 FileAlignment 域的倍数。如果它小于 VirtualSize 域的值，余下的部分用 0 填充。由于 SizeOfRawData 域要向上舍入，但是 VirtualSize 域并不舍入，因此可能出现 SizeOfRawData 域大于 VirtualSize 域的情况。当节中仅包含未初始化的数据时，这个域应该为 0。
20	4	PointerToRawData	指向 COFF 文件中节的第一个页面的文件指针。对于可执行映像来说，它必须是可选文件头中 FileAlignment 域的倍数。对于目标文件来说，要获得最好的性能，此值应该按 4 字节边界对齐。当节中仅包含未初始化的数据时，这个域应该为 0。
24	4	PointerToRelocations	指向节中重定位项开头的文件指针。对于可执行文件或者没有重定位项的文件来说，此值应该为 0。
28	4	PointerToLinenumbers	指向节中行号项开头的文件指针。如果没有 COFF 行号信息的话，此值应该为 0。对于映像来说，此值应该为 0，因为已经不赞成使用 COFF 调试信息了。
32	2	NumberOfRelocations	节中重定位项的个数。对于可执行映像来说，此值应该为 0。
34	2	NumberOfLinenumbers	节中行号项的个数。对于映像来说，此值应该为 0，因为已经不赞成使用 COFF 调试信息了。
36	4	Characteristics	描述节特征的标志。要获取更多信息，请参考 4.1 节“节标志”。

4.1 节标志

节头中的 Characteristics 域中的节标志指出了节的属性。

标志	值	描述
	0x00000000	保留供将来使用。
	0x00000001	保留供将来使用。
	0x00000002	保留供将来使用。
	0x00000004	保留供将来使用。
IMAGE_SCN_TYPE_NO_PAD	0x00000008	从这个节结尾到下一个边界之间不能填充。此标志被舍弃，它已经被 IMAGE_SCN_ALIGN_1BYTES 标志取代。此标志仅对目标文件合法。
	0x00000010	保留供将来使用。
IMAGE_SCN_CNT_CODE	0x00000020	此节包含可执行代码。
IMAGE_SCN_CNT_INITIALIZED_DATA	0x00000040	此节包含已初始化的数据。
IMAGE_SCN_CNT_UNINITIALIZED_DATA	0x00000080	此节包含未初始化的数据。
IMAGE_SCN_LNK_OTHER	0x00000100	保留供将来使用。
IMAGE_SCN_LNK_INFO	0x00000200	此节包含注释或者其它信息。**.drectve** 节具有这种属性。此标志仅对目标文件合法。
	0x00000400	保留供将来使用。

标志	值	描述
IMAGE_SCN_LNK_REMOVE	0x00000800	此节不会成为最终形成的映像文件的一部分。此标志仅对目标文件合法。
IMAGE_SCN_LNK_COMDAT	0x00001000	此节包含 COMDAT 数据。要获取更多信息，请参考 5.5.6 节“COMDAT 节（仅适用于目标文件）”。此标志仅对目标文件合法。
IMAGE_SCN_GPREL	0x00008000	此节包含通过全局指针（GP）来引用的数据。
IMAGE_SCN_MEM_PURGEABLE	0x00020000	保留供将来使用。
IMAGE_SCN_MEM_16BIT	0x00020000	保留供将来使用。
IMAGE_SCN_MEM_LOCKED	0x00040000	保留供将来使用。
IMAGE_SCN_MEM_PRELOAD	0x00080000	保留供将来使用。
IMAGE_SCN_ALIGN_1BYTES	0x00100000	按 1 字节边界对齐数据。此标志仅对目标文件合法。
IMAGE_SCN_ALIGN_2BYTES	0x00200000	按 2 字节边界对齐数据。此标志仅对目标文件合法。
IMAGE_SCN_ALIGN_4BYTES	0x00300000	按 4 字节边界对齐数据。此标志仅对目标文件合法。
IMAGE_SCN_ALIGN_8BYTES	0x00400000	按 8 字节边界对齐数据。此标志仅对目标文件合法。
IMAGE_SCN_ALIGN_16BYTES	0x00500000	按 16 字节边界对齐数据。此标志仅对目标文件合法。
IMAGE_SCN_ALIGN_32BYTES	0x00600000	按 32 字节边界对齐数据。此标志仅对目标文件合法。
IMAGE_SCN_ALIGN_64BYTES	0x00700000	按 64 字节边界对齐数据。此标志仅对目标文件合法。
IMAGE_SCN_ALIGN_128BYTES	0x00800000	按 128 字节边界对齐数据。此标志仅对目标文件合法。
IMAGE_SCN_ALIGN_256BYTES	0x00900000	按 256 字节边界对齐数据。此标志仅对目标文件合法。
IMAGE_SCN_ALIGN_512BYTES	0x00A00000	按 512 字节边界对齐数据。此标志仅对目标文件合法。
IMAGE_SCN_ALIGN_1024BYTES	0x00B00000	按 1024 字节边界对齐数据。此标志仅对目标文件合法。
IMAGE_SCN_ALIGN_2048BYTES	0x00C00000	按 2048 字节边界对齐数据。此标志仅对目标文件合法。
IMAGE_SCN_ALIGN_4096BYTES	0x00D00000	按 4096 字节边界对齐数据。此标志仅对目标文件合法。
IMAGE_SCN_ALIGN_8192BYTES	0x00E00000	按 8192 字节边界对齐数据。此标志仅对目标文件合法。
IMAGE_SCN_LNK_NRELOC_OVFL	0x01000000	此节包含扩展的重定位信息。
IMAGE_SCN_MEM_DISCARDABLE	0x02000000	此节可以在需要时被丢弃。
IMAGE_SCN_MEM_NOT_CACHED	0x04000000	此节不能被缓存。
IMAGE_SCN_MEM_NOT_PAGED	0x08000000	此节不能被交换到页面文件中。
IMAGE_SCN_MEM_SHARED	0x10000000	此节可以在内存中共享。
IMAGE_SCN_MEM_EXECUTE	0x20000000	此节可以作为代码执行。
IMAGE_SCN_MEM_READ	0x40000000	此节可读。

标志	值	描述
IMAGE_SCN_MEM_WRITE	0x80000000	此节可写。

IMAGE_SCN_LNK_NRELOC_OVFL 标志表明节中重定位项的个数超出了节头中为每个节保留的 16 位所能表示的范围。如果设置了此标志并且节头中的 NumberOfRelocations 域的值是 0xffff，那么实际的重定位项个数被保存在第一个重定位项的 VirtualAddress 域（32 位）中。如果设置了 IMAGE_SCN_LNK_NRELOC_OVFL 标志但节中的重定位项的个数少于 0xffff，则表示出现了错误。

4.2 成组的节（仅适用于目标文件）

目标文件节名中的“$”字符（美元符号）有特殊含义。

当由目标文件节中的内容生成映像文件的节时，链接器丢弃节名中“$”以及它后面的所有字符。因此目标文件中一个名为.text$X 的节实际上成了映像文件中.text 节的一部分。

但是“$”后面的字符决定了这些节在映像文件中的顺序。目标文件中节名（指的是节名中“$”字符以前的部分）相同的节被放在映像文件中连续的位置上，并且它们按节名的字母顺序排列。因此目标文件中节名为.text$X 的节中的内容被放在一起，并且它们在名为.text$W 节的内容之后，而在名为.text$Y 节的内容之前。

映像文件节名中永远不会包含“$”字符。

5 文件中的其它内容

到目前为止我们所描述的结构，直到可选文件头（含），都位于从文件开头算起的固定偏移处（如果是包含 MS-DOS 占位程序的映像文件的话则是从 PE 文件头开始算起）。

COFF 目标文件或映像文件中其余部分包含了那些并不需要处于某一特定文件偏移处的数据块。它们的位置由可选文件头或节头中相应的指针给出。

对于 SectionAlignment 值小于平台页面大小（Intel x86 和 MIPS 是 4K，Itanium 是 8K）的映像文件来说有一个例外。要获取关于 SectionAlignment 的详细描述，请参考 3.4 节“可选文件头（仅适用于映像文件）”。此时节中数据的文件偏移有一些限制。详细信息请参考 5.1 节“节中的数据”。另外一个例外情况是，属性证书和调试信息必须被放在映像文件的最后，并且属性证书表在调试信息节之前，这是因为加载器并不将这些信息映射进内存。但是对于属性证书和调试信息的这些限制并不适用于目标文件。

5.1 节中的数据

节中已初始化的数据就是简单的字节块。但是对于那些仅包含 0 的节来说，节中的数据就没有必要包含到文件中。

每个节中的数据都位于节头的 PointerToRawData 域指定的文件偏移处。数据的大小由 SizeOfRawData 域给出。如果 SizeOfRawData 小于 VirtualSize，那么余下的部分用 0 填充。

在映像文件中，节中的数据必须按可选文件头的 FileAlignment 域指定的边界对齐。节中的数据必须按相应节的 RVA 值的大小顺序出现在文件中（节表中单个的节头也是如此）。

如果可选文件头的 SectionAlignment 域的值小于相应平台的页面大小，那么映像文件有一些附加的限制。对于这种文件，当映像被加载到内存中时，节中数据在文件中的位置必须与它在内存中的位置匹配，因此节中数据的物理偏移与 RVA 相同。

5.2 COFF 重定位信息（仅适用于目标文件）

目标文件中包含 COFF 重定位信息，它用来指出当节中的数据被放进映像文件以及将来被加载进内存时应如何修改这些数据。

映像文件中并不包含 COFF 重定位信息，这是因为所有被引用的符号都已经被赋予了一个在平坦内存空间中的地址。映像文件中包含的重定位信息是位于.reloc 节中的基址重定位信息（除非映像具有 IMAGE_FILE_RELOCS_STRIPPED 属性）。要获取更多信息，请参考 6.6 节“.reloc 节（仅适用于映像文件）”。

对于目标文件中的每个节，都有一个由长度固定的记录组成的数组来保存此节的 COFF 重定位信息。此数组的位置和长度在节头中指定。数组的每个元素格式如下：

偏移	大小	域	描述
0	4	VirtualAddress	需要进行重定位的代码或数据的地址。这是从节开头算起的偏移，加上节的 RVA/Offset 域的值。请参考 4 节“节表（节头）”。例如如果节的第一个字节的地址是 0x10，那么第三个字节的地址就是 0x12。
4	4	SymbolTableIndex	符号表的索引（从 0 开始）。这个符号给出了用于重定位的地址。如果这个指定符号的存储类别为节，那么它的地址就是第一个与它同名的节的地址。
8	2	Type	重定位类型。合法的重定位类型依赖于机器类型。请参考 5.2.1 节“类型指示符”

如果 SymbolTableIndex 域指定的符号存储类别为 IMAGE_SYM_CLASS_SECTION，那么这个符号的地址就是节的地址。这个节通常就在同一个文件中，除非这个目标文件是档案（库）文件的一部分。如果是这种情况的话，那么这个节可能位于档案文件中与当前目标文件的档案文件成员名称相同的任何其它目标文件中。（与档案文件成员名称的联系用于链接生成导入表，即.idata 节。）

5.2.1 类型指示符

重定位记录的 Type 域指出了重定位类型。对于每种类型的机器都定义了不同的重定位类型。

x64 处理器

为 x64 及其兼容处理器定义了以下重定位类型指示符：

常量	值	描述
IMAGE_REL_AMD64_ABSOLUTE	0x0000	重定位被忽略。
IMAGE_REL_AMD64_ADDR64	0x0001	重定位目标的 64 位 VA。
IMAGE_REL_AMD64_ADDR32	0x0002	重定位目标的 32 位 VA。
IMAGE_REL_AMD64_ADDR32NB	0x0003	不包含映像基址的 32 位地址（RVA）。
IMAGE_REL_AMD64_REL32	0x0004	相对于重定位目标的 32 位地址。
IMAGE_REL_AMD64_REL32_1	0x0005	相对于距重定位目标 1 字节处的 32 位地址。
IMAGE_REL_AMD64_REL32_2	0x0006	相对于距重定位目标 2 字节处的 32 位地址。
IMAGE_REL_AMD64_REL32_3	0x0007	相对于距重定位目标 3 字节处的 32 位地址。
IMAGE_REL_AMD64_REL32_4	0x0008	相对于距重定位目标 4 字节处的 32 位地址。
IMAGE_REL_AMD64_REL32_5	0x0009	相对于距重定位目标 5 字节处的 32 位地址。

常量	值	描述
IMAGE_REL_AMD64_SECTION	0x000A	包含重定位目标的节的 16 位索引。用于支持调试信息。
IMAGE_REL_AMD64_SECREL	0x000B	重定位目标相对于它所在节开头的 32 位偏移。用于支持调试信息和静态线程局部存储。
IMAGE_REL_AMD64_SECREL7	0x000C	相对于重定位目标所在节基地址的 7 位偏移（无符号数）。
IMAGE_REL_AMD64_TOKEN	0x000D	CLR 记号。
IMAGE_REL_AMD64_SREL32	0x000E	放入目标文件中的 32 位跨度依赖值（符号数）。
IMAGE_REL_AMD64_PAIR	0x000F	与跨度依赖值成对出现，它必须紧跟每一个跨度依赖值。
IMAGE_REL_AMD64_SSPAN32	0x0010	链接时应用的 32 位跨度依赖值（符号数）。

ARM 处理器

为 ARM 处理器定义了以下重定位类型指示符：

常量	值	描述
IMAGE_REL_ARM_ABSOLUTE	0x0000	重定位被忽略。
IMAGE_REL_ARM_ADDR32	0x0001	重定位目标的 32 位 VA。
IMAGE_REL_ARM_ADDR32NB	0x0002	重定位目标的 32 位 RVA。
IMAGE_REL_ARM_BRANCH24	0x0003	重定位目标的 24 位相对偏移。
IMAGE_REL_ARM_BRANCH11	0x0004	对子程序调用的引用。这个引用由两个 16 位指令与 11 位偏移组成。
IMAGE_REL_ARM_SECTION	0x000E	包含重定位目标的节的 16 位索引。用于支持调试信息。
IMAGE_REL_ARM_SECREL	0x000F	重定位目标相对于它所在节开头的 32 位偏移。用于支持调试信息和静态线程局部存储。

Hitachi SuperH 处理器

为 SH3 和 SH4 处理器定义了以下重定位类型指示符。专用于 SH5 处理器的重定位类型都标有“SHM”字样，SHM 代表 SHMedia。

常量	值	描述
IMAGE_REL_SH3_ABSOLUTE	0x0000	重定位被忽略。
IMAGE_REL_SH3_DIRECT16	0x0001	对包含重定位目标符号 VA 的 16 位单元的引用。
IMAGE_REL_SH3_DIRECT32	0x0002	重定位目标符号的 32 位 VA。
IMAGE_REL_SH3_DIRECT8	0x0003	对包含重定位目标符号 VA 的 8 位单元的引用。
IMAGE_REL_SH3_DIRECT8_WORD	0x0004	对包含重定位目标符号 16 位有效 VA 的 8 位指令的引用。
IMAGE_REL_SH3_DIRECT8_LONG	0x0005	对包含重定位目标符号 32 位有效 VA 的 8 位指令的引用。
IMAGE_REL_SH3_DIRECT4	0x0006	对其低 4 位包含重定位目标符号 VA 的 8 位单元的引用。
IMAGE_REL_SH3_DIRECT4_WORD	0x0007	对其低 4 位包含重定位目标符号 16 位有效 VA 的 8 位指令的引用。
IMAGE_REL_SH3_DIRECT4_LONG	0x0008	对其低 4 位包含重定位目标符号 32 位有效 VA 的 8 位指令的引用。
IMAGE_REL_SH3_PCREL8_WORD	0x0009	对包含重定位目标符号 16 位有效相对偏移的 8 位指令的引用。

常量	值	描述
IMAGE_REL_SH3_PCREL8_LONG	0x000A	对包含重定位目标符号 32 位有效相对偏移的 8 位指令的引用。
IMAGE_REL_SH3_PCREL12_WORD	0x000B	对其低 12 位包含重定位目标符号 16 位有效相对偏移的 16 位指令的引用。
IMAGE_REL_SH3_STARTOF_SECTION	0x000C	对包含重定位目标符号所在节 VA 的 32 位单元的引用。
IMAGE_REL_SH3_SIZEOF_SECTION	0x000D	对包含重定位目标符号所在节大小的 32 位单元的引用。
IMAGE_REL_SH3_SECTION	0x000E	包含重定位目标的节的 16 位索引。用于支持调试信息。
IMAGE_REL_SH3_SECREL	0x000F	重定位目标相对于它所在节开头的 32 位偏移。用于支持调试信息和静态线程局部存储。
IMAGE_REL_SH3_DIRECT32_NB	0x0010	重定位目标符号的 32 位 RVA。
IMAGE_REL_SH3_GPREL4_LONG	0x0011	与 GP 相关。
IMAGE_REL_SH3_TOKEN	0x0012	CLR 记号。
IMAGE_REL_SHM_PCRELPT	0x0013	距当前指令的偏移（长字）。如果没有设置 IMAGE_REL_SHM_NOMODE 标志，那么将低位取反插入到第 32 位以选择 PTA 指令或 PTB 指令。
IMAGE_REL_SHM_REFLO	0x0014	32 位地址的低 16 位。
IMAGE_REL_SHM_REFHALF	0x0015	32 位地址的高 16 位。
IMAGE_REL_SHM_RELLO	0x0016	相对地址的低 16 位。
IMAGE_REL_SHM_RELHALF	0x0017	相对地址的高 16 位。
IMAGE_REL_SHM_PAIR	0x0018	只有紧跟 IMAGE_REL_SHM_REFHALF、IMAGE_REL_SHM_RELLO 或 IMAGE_REL_SHM_RELHALF 类型的重定位项时这种重定位类型才是合法的。此重定位项的 SymbolTableIndex 域包含的是偏移而不是符号表的索引。
IMAGE_REL_SHM_NOMODE	0x8000	重定位忽略节模式。

IBM PowerPC 处理器

为 PowerPC 处理器定义了以下重定位类型指示符：

常量	值	描述
IMAGE_REL_PPC_ABSOLUTE	0x0000	重定位被忽略。
IMAGE_REL_PPC_ADDR64	0x0001	重定位目标的 64 位 VA。
IMAGE_REL_PPC_ADDR32	0x0002	重定位目标的 32 位 VA。
IMAGE_REL_PPC_ADDR24	0x0003	重定位目标 VA 的低 24 位。只有当重定位目标符号是绝对符号且可以按符号扩展到它的原始值时才是合法的。
IMAGE_REL_PPC_ADDR16	0x0004	重定位目标 VA 的低 16 位。
IMAGE_REL_PPC_ADDR14	0x0005	重定位目标 VA 的低 14 位。只有当重定位目标符号是绝对符号且可以按符号扩展到它的原始值时才是合法的。
IMAGE_REL_PPC_REL24	0x0006	符号位置相对于 PC 的 24 位偏移。
IMAGE_REL_PPC_REL14	0x0007	符号位置相对于 PC 的 14 位偏移。
IMAGE_REL_PPC_ADDR32NB	0x000A	重定位目标的 32 位 RVA。
IMAGE_REL_PPC_SECREL	0x000B	重定位目标相对于它所在节开头的 32 位偏移。用于支持调试信息和静态线程局部存储。

常量	值	描述
IMAGE_REL_PPC_SECTION	0x000C	包含重定位目标的节的 16 位索引。用于支持调试信息。
IMAGE_REL_PPC_SECREL16	0x000F	重定位目标相对于它所在节开头的 16 位偏移。用于支持调试信息和静态线程局部存储。
IMAGE_REL_PPC_REFHI	0x0010	重定位目标 32 位 VA 的高 16 位。它用于加载一个完整地址所需的两指令序列中的第一条指令。这种重定位类型后面必须紧跟 IMAGE_REL_PPC_PAIR 类型的重定位项，而后者的 SymbolTableIndex 域包含的是一个 16 位偏移（符号数），这个偏移要被加到重定位目标位置的高 16 位。
IMAGE_REL_PPC_REFLO	0x0011	重定位目标 VA 的低 16 位。
IMAGE_REL_PPC_PAIR	0x0012	只有紧跟 IMAGE_REL_PPC_REFHI 或 IMAGE_REL_PPC_SECRELHI 类型的重定位时这种重定位类型才是合法的。此重定位项的 SymbolTableIndex 域包含的是偏移而不是符号表的索引。
IMAGE_REL_PPC_SECRELLO	0x0013	重定位目标相对于它所在节开头的 32 位偏移的低 16 位。
IMAGE_REL_PPC_SECRELHI	0x0014	重定位目标相对于它所在节开头的 32 位偏移的高 16 位。
IMAGE_REL_PPC_GPREL	0x0015	重定位目标相对于 GP 寄存器的 16 位偏移（带符号数）。
IMAGE_REL_PPC_TOKEN	0x0016	CLR 记号。

Intel 386 处理器

为 Intel 386 及其兼容处理器定义了以下重定位类型指示符：

常量	值	描述
IMAGE_REL_I386_ABSOLUTE	0x0000	重定位被忽略。
IMAGE_REL_I386_DIR16	0x0001	不支持。
IMAGE_REL_I386_REL16	0x0002	不支持。
IMAGE_REL_I386_DIR32	0x0006	重定位目标的 32 位 VA。
IMAGE_REL_I386_DIR32NB	0x0007	重定位目标的 32 位 RVA。
IMAGE_REL_I386_SEG12	0x0009	不支持。
IMAGE_REL_I386_SECTION	0x000A	包含重定位目标的节的 16 位索引。用于支持调试信息。
IMAGE_REL_I386_SECREL	0x000B	重定位目标相对于它所在节开头的 32 位偏移。用于支持调试信息和静态线程局部存储。
IMAGE_REL_I386_TOKEN	0x000C	CLR 记号。
IMAGE_REL_I386_SECREL7	0x000D	相对于重定位目标所在节基地址的 7 位偏移。
IMAGE_REL_I386_REL32	0x0014	重定位目标的 32 位相对偏移。用于支持 x86 的相对分支和 CALL 指令。

Intel Itanium 处理器家族（IPF）

为 Intel Itanium 处理器家族及其兼容处理器定义了以下重定位类型指示符。注意指令重定位使用它所在的指令包的偏移以及相应的指令槽编号作为它的偏移：

常量	值	描述
IMAGE_REL_IA64_ABSOLUTE	0x0000	重定位被忽略。

常量	值	描述
IMAGE_REL_IA64_IMM14	0x0001	这种指令重定位后面可以跟着 IMAGE_REL_IA64_ADDEND 类型的重定位项，而后者的值在被插入到 IMM14 指令包的指定的指令槽中之前被加到目标地址上。这种重定位目标必须是绝对符号，否则这个映像必须被修正。
IMAGE_REL_IA64_IMM22	0x0002	这种指令重定位后面可以跟着 IMAGE_REL_IA64_ADDEND 类型的重定位项，而后者的值在被插入到 IMM22 指令包的指定的指令槽中之前被加到目标地址上。这种重定位目标必须是绝对符号，否则这个映像必须被修正。
IMAGE_REL_IA64_IMM64	0x0003	这种重定位项的指令槽编号必须为 1。这种重定位后面可以跟着 IMAGE_REL_IA64_ADDEND 类型的重定位项，而后者的值在被存储到 IMM64 指令包的三个指令槽中之前被加到目标地址上。
IMAGE_REL_IA64_DIR32	0x0004	重定位目标的 32 位 VA。仅支持使用 /LARGEADDRESSAWARE:NO 链接器选项生成的映像。
IMAGE_REL_IA64_DIR64	0x0005	重定位目标的 64 位 VA。
IMAGE_REL_IA64_PCREL21B	0x0006	使用按 16 位边界对齐的重定位目标的 25 位相对偏移来修正指令。这个偏移的低 4 位全为 0，因此并没有被存储。
IMAGE_REL_IA64_PCREL21M	0x0007	使用按 16 位边界对齐的重定位目标的 25 位相对偏移来修正指令。这个偏移的低 4 位全为 0，因此并没有被存储。
IMAGE_REL_IA64_PCREL21F	0x0008	这种重定位目标偏移的 LSB 部分包含的是指令槽编号，其余部分包含的是指令包的地址。使用按 16 位边界对齐的重定位目标的 25 位相对偏移来修正指令。这个偏移的低 4 位全为 0，因此并没有被存储。
IMAGE_REL_IA64_GPREL22	0x0009	这种指令重定位后面可以跟着 IMAGE_REL_IA64_ADDEND 类型的重定位项，后者的值被加到目标地址上，而后计算 GPREL22 指令包相对于 GP 的偏移并应用。
IMAGE_REL_IA64_LTOFF22	0x000A	使用重定位目标符号的常量表项相对于 GP 的 22 位偏移来修正指令。链接器根据这个重定位项以及可能跟着它的 IMAGE_REL_IA64_ADDEND 类型的重定位项来创建这个常量表项。
IMAGE_REL_IA64_SECTION	0x000B	包含重定位目标的节的 16 位索引。用于支持调试信息。
IMAGE_REL_IA64_SECREL22	0x000C	使用重定位目标相对于它所在节开头的 22 位偏移来修正指令。这种类型的重定位项后面可以紧跟着 IMAGE_REL_IA64_ADDEND 类型的重定位项，后者的 Value 域包含重定位目标相对于它所在节开头的 32 位偏移（无符号数）。
IMAGE_REL_IA64_SECREL64I	0x000D	这种重定位项的指令槽编号必须为 1。使用重定位目标相对于它所在节开头的 64 位偏移来修正指令。这种类型的重定位项后面可以紧跟着 IMAGE_REL_IA64_ADDEND 类型的重定位项，后者的 Value 域包含重定位目标相对于它所在节开头的 32 位偏移（无符号数）。
IMAGE_REL_IA64_SECREL32	0x000E	使用重定位目标相对于它所在节开头的 32 位偏移来修正的数据的地址。

常量	值	描述
IMAGE_REL_IA64_LTOFF64	0x000F	
IMAGE_REL_IA64_DIR32NB	0x0010	目标的 32 位 RVA。
IMAGE_REL_IA64_SREL14	0x0011	用于包含两个重定位目标之差的 14 位立即数（符号数）。对于链接器来说这是一个说明域，表明编译器已经生成了这个值。
IMAGE_REL_IA64_SREL22	0x0012	用于包含两个重定位目标之差的 22 位立即数（符号数）。对于链接器来说这是一个说明域，表明编译器已经生成了这个值。
IMAGE_REL_IA64_SREL32	0x0013	用于包含两个重定位目标之差的 32 位立即数（符号数）。对于链接器来说这是一个说明域，表明编译器已经生成了这个值。
IMAGE_REL_IA64_UREL32	0x0014	用于包含两个重定位目标之差的 32 位立即数（无符号数）。对于链接器来说这是一个说明域，表明编译器已经生成了这个值。
IMAGE_REL_IA64_PCREL60X	0x0015	相对于 PC 的 60 位修正，用于 MLX 指令包的 BRL 指令。
IMAGE_REL_IA64_PCREL60B	0x0016	相对于 PC 的 60 位修正。如果重定位目标偏移不超过一个 25 位域所能表示的范围（符号数），那么就在 1 号指令槽中使用 NOP.B 指令、2 号指令槽中使用 25 位（最低 4 位全为 0，舍弃）的 BR 指令将整个指令包转换成 MBB 指令包。
IMAGE_REL_IA64_PCREL60F	0x0017	相对于 PC 的 60 位修正。如果重定位目标偏移不超过一个 25 位域所能表示的范围（符号数），那么就在 1 号指令槽中使用 NOP.F 指令、2 号指令槽中使用 25 位（最低 4 位全为 0，舍弃）的 BR 指令将整个指令包转换成 MFB 指令包。
IMAGE_REL_IA64_PCREL60I	0x0018	相对于 PC 的 60 位修正。如果重定位目标偏移不超过一个 25 位域所能表示的范围（符号数），那么就在 1 号指令槽中使用 NOP.I 指令、2 号指令槽中使用 25 位（最低 4 位全为 0，舍弃）的 BR 指令将整个指令包转换成 MIB 指令包。
IMAGE_REL_IA64_PCREL60M	0x0019	相对于 PC 的 60 位修正。如果重定位目标偏移不超过一个 25 位域所能表示的范围（符号数），那么就在 1 号指令槽中使用 NOP.M 指令、2 号指令槽中使用 25 位（最低 4 位全为 0，舍弃）的 BR 指令将整个指令包转换成 MMB 指令包。
IMAGE_REL_IA64_IMMGPREL64	0x001A	相对于 GP 的 64 位修正。
IMAGE_REL_IA64_TOKEN	0x001B	CLR 记号。
IMAGE_REL_IA64_GPREL32	0x001C	相对于 GP 的 32 位修正。
IMAGE_REL_IA64_ADDEND	0x001F	只有紧跟下列类型的重定位时这种重定位类型才是合法的：IMAGE_REL_IA64_IMM14、IMAGE_REL_IA64_IMM22、IMAGE_REL_IA64_IMM64、IMAGE_REL_IA64_GPREL22、IMAGE_REL_IA64_LTOFF22、IMAGE_REL_IA64_LTOFF64、IMAGE_REL_IA64_SECREL22、IMAGE_REL_IA64_SECREL64I 或 IMAGE_REL_IA64_SECREL32。它的值是应用到指令包中的指令上的加数，而不是用于数据。

MIPS 处理器

为 MIPS 处理器定义了以下重定位类型指示符：

常量	值	描述
IMAGE_REL_MIPS_ABSOLUTE	0x0000	重定位被忽略。
IMAGE_REL_MIPS_REFHALF	0x0001	重定位目标 32 位 VA 的高 16 位。
IMAGE_REL_MIPS_REFWORD	0x0002	重定位目标的 32 位 VA。
IMAGE_REL_MIPS_JMPADDR	0x0003	重定位目标 VA 的低 26 位。用于支持 MIPS 平台的 J 和 JAL 指令。
IMAGE_REL_MIPS_REFHI	0x0004	重定位目标 32 位 VA 的高 16 位。它用于加载一个完整地址所需的两指令序列中的第一条指令。这种重定位类型后面必须紧跟 IMAGE_REL_MIPS_PAIR 类型的重定位项，而后者的 SymbolTableIndex 域包含的是一个 16 位偏移（符号数），这个偏移要被加到重定位目标位置的高 16 位。
IMAGE_REL_MIPS_REFLO	0x0005	重定位目标 VA 的低 16 位。
IMAGE_REL_MIPS_GPREL	0x0006	重定位目标相对于 GP 寄存器的 16 位偏移（符号数）。
IMAGE_REL_MIPS_LITERAL	0x0007	与 IMAGE_REL_MIPS_GPREL 相同。
IMAGE_REL_MIPS_SECTION	0x000A	包含重定位目标的节的 16 位索引。用于支持调试信息。
IMAGE_REL_MIPS_SECREL	0x000B	重定位目标相对于它所在节开头的 32 位偏移。用于支持调试信息和静态线程局部存储。
IMAGE_REL_MIPS_SECRELLO	0x000C	重定位目标相对于它所在节开头的 32 位偏移的低 16 位。
IMAGE_REL_MIPS_SECRELHI	0x000D	重定位目标相对于它所在节开头的 32 位 VA 的高 16 位。这种重定位类型后面必须紧跟 IMAGE_REL_MIPS_PAIR 类型的重定位项，而后者的 SymbolTableIndex 域包含的是一个 16 位偏移（符号数），这个偏移要被加到重定位符号位置的高 16 位。
IMAGE_REL_MIPS_JMPADDR16	0x0010	重定位目标 VA 的低 26 位。用于支持 MIPS16 的 JAL 指令。
IMAGE_REL_MIPS_REFWORDNB	0x0022	重定位目标的 32 位 RVA。
IMAGE_REL_MIPS_PAIR	0x0025	只有紧跟 IMAGE_REL_MIPS_REFHI 或 IMAGE_REL_MIPS_SECRELHI 类型的重定位时这种重定位类型才是合法的。此重定位项的 SymbolTableIndex 域包含的是偏移而不是符号表索引。

Mitsubishi M32R 处理器

为 Mitsubishi M32R 处理器定义了以下重定位类型指示符：

常量	值	描述
IMAGE_REL_M32R_ABSOLUTE	0x0000	重定位被忽略。
IMAGE_REL_M32R_ADDR32	0x0001	重定位目标的 32 位 VA。
IMAGE_REL_M32R_ADDR32NB	0x0002	重定位目标的 32 位 RVA。
IMAGE_REL_M32R_ADDR24	0x0003	重定位目标的 24 位 VA。
IMAGE_REL_M32R_GPREL16	0x0004	重定位目标相对于 GP 寄存器的 16 位偏移。
IMAGE_REL_M32R_PCREL24	0x0005	重定位目标相对于程序计数器（PC）的 24 位偏移，已经左移 2 位并按符号扩展。

常量	值	描述
IMAGE_REL_M32R_PCREL16	0x0006	重定位目标相对于 PC 的 16 位偏移，已经左移 2 位并按符号扩展。
IMAGE_REL_M32R_PCREL8	0x0007	重定位目标相对于 PC 的 8 位偏移，已经左移 2 位并按符号扩展。
IMAGE_REL_M32R_REFHALF	0x0008	重定位目标 VA 的 16 位 MSB。
IMAGE_REL_M32R_REFHI	0x0009	重定位目标 VA 的 16 位 MSB，已经按 LSB 符号扩展调整。它用于加载一个完整的 32 位地址所需的两指令序列中的第一条指令。这种重定位类型后面必须紧跟 IMAGE_REL_M32R_PAIR 类型的重定位项，而后者的 SymbolTableIndex 域包含的是一个 16 位偏移（符号数），这个偏移要被加到重定位符号位置的高 16 位。
IMAGE_REL_M32R_REFLO	0x000A	重定位目标 VA 的 16 位 LSB。
IMAGE_REL_M32R_PAIR	0x000B	这种类型的重定位必须紧跟类型为 IMAGE_REL_M32R_REFHI 的重定位项。此重定位项的 SymbolTableIndex 域包含的是偏移而不是符号表索引。
IMAGE_REL_M32R_SECTION	0x000C	包含重定位目标的节的 16 位索引。用于支持调试信息。
IMAGE_REL_M32R_SECREL	0x000D	重定位目标相对于它所在节开头的 32 位偏移。用于支持调试信息和静态线程局部存储。
IMAGE_REL_M32R_TOKEN	0x000E	CLR 记号。

5.3 COFF 行号信息（不赞成使用）

Microsoft 的工具已经不再生成 COFF 行号信息，并且以后也不会再使用它了。

COFF 行号信息给出了源文件中代码与行号之间的关系。Microsoft 的 COFF 行号信息格式与标准 COFF 行号信息格式类似，但已经被扩展，以便用单个节就可以与多个源文件中的行号关联。

COFF 行号信息由长度固定的记录组成的数组构成。数组的位置（文件偏移）与大小由节头给出。每个行号记录格式如下：

偏移	大小	域	描述
0	4	Type (*)	这是由 SymbolTableIndex 和 VirtualAddress 两个域组成的共用体。使用 SymbolTableIndex 还是 RVA 依赖于 Linenumber 域的值。
4	2	Linenumber	如果此值非零，那么它表示行号（从 1 开始）。如果它为零，那么 Type 域表示一个函数在符号表中的索引。

Type 域是由 SymbolTableIndex 和 VirtualAddress 这两个 4 字节长的域组成的共用体：

偏移	大小	域	描述
0	4	SymbolTableIndex	当 Linenumber 为零时使用此域，它表示一个函数在符号表中的索引。此格式用来指明一组行号记录与哪个函数相关。
0	4	VirtualAddress	当 Linenumber 非零时使用此域，它表示与源代码中由 Linenumber 指定的那行代码对应的可执行代码的 RVA。在目标文件中，它表示节中的 VA。

一个行号记录，或者是将其 Linenumber 域设置为零，同时（另一个域）指向符号表中的一个函数；或者是作为一个标准的行号记录：给出一个正整数（行号）以及在目标代码中相应的地址。

一组行号记录总是以前一种格式——符号表中函数的索引开头。如果这个行号记录是节中的第一个行号记录，并且节的 COMDAT 标志设置的话，那么它同时也是函数的 COMDAT 符号名。详细信息请参考 5.5.6 节“COMDAT 节（仅适用于目标文件）”。此函数在符号表中的辅助记录有一个 PointerToLinenumber 域，它指向这个行号记录。

标识函数的记录后面跟着一些行号记录，它们给出了实际的行号信息（也就是说，这些记录的 Linenumber 域都大于 0）。这些项（中的 Linenumber 域）从 1 开始（相对于函数的开头），每一个表示函数源代码中的一行（除了第一行）。

例如在下面的例子中，第一个行号记录会列出 ReverseSign 函数（ReverseSign 函数的 SymbolTableIndex 和被设置为 0 的 Linenumber 域）。它后面的记录的 Linenumber 域的值依次为 1、2 和 3，它们分别对应着源代码中相应的行，如下所示：

```
// ReverseSign 函数前面的代码
    int ReverseSign(int i)
1:  {
2:      return -1 * i;
3:  }
```

5.4 COFF 符号表

本节中提到的符号表继承自传统的 COFF 格式。它与 Microsoft Visual C++®调试信息截然不同。一个文件可以同时包含 COFF 符号表和 Visual C++调试信息，二者是分开的。一些 Microsoft 的工具出于有限但重要的目的使用符号表，例如用于向链接器传递 COMDAT 信息。符号表中列出了节名、文件名以及代码与数据符号。

符号表的位置由 COFF 文件头给出。

符号表是一个由记录组成的数组，每个记录长 18 字节。它们或者是标准符号表记录，或者是辅助符号表记录。标准符号表记录定义了一个符号或名称，格式如下：

偏移	大小	域	描述
0	8	Name (*)	符号名称，这是一个由三个成员组成的共用体。如果名称的长度不超过 8 个字节，那么它就是一个 8 字节长的数组。要获取更多信息，请参考 5.4.1 节“符号名称表示”。
8	4	Value	与符号相关的值。其意义依赖于 SectionNumber 和 StorageClass 这两个域。它通常表示可重定位的地址。
12	2	SectionNumber	这个带符号整数是节表的索引（从 1 开始），用以标识定义此符号的节。一些值有特殊的含义，详细信息请参考 5.4.2 节“SectionNumber 域的值”。
14	2	Type	一个表示类型的数字。Microsoft 的工具将它设置为 0x20（如果是函数）或者 0x0（如果不是函数）。要获取更多信息，请参考 5.4.3 节“类型表示”。
16	1	StorageClass	这是一个表示存储类别的枚举类型值。要获取更多信息，请参考 5.4.4 节“存储类别”。
17	1	NumberOfAuxSymbols	跟在本记录后面的辅助符号表项的个数。

每个标准符号表记录后面紧跟着零个或多个辅助符号表记录，但通常情况下并不会超过一个（除了带长文件名的.file 记录）。每个辅助符号表记录的大小与标准符号表记录相同（都是 18 字节），但它并不是又定义了一个新的符号，而是给出了与上一个定义的符号相关的附加信息。辅助符号表记录有好几种格式，使用哪一种取决于 StorageClass 域的值。当前已经为辅助符号表记录定义的格式将在 5.5 节“辅助符号表记录”中列出。

处理 COFF 符号表的工具必须忽略意义不明的辅助符号表记录。这样允许日后对符号表的格式进行扩展以便添加新的辅助符号表记录而不会使现存的工具失效。

5.4.1 符号名称表示

如果符号名称长度不超过 8 个字节，那么符号表的 ShortName 域就是包含符号名本身的一个 8 字节长的数组；否则的话，它给出了字符串表中的一个偏移地址。要确定它到底是名称本身还是偏移地址，测试一下它的前 4 个字节是否等于 0。

通常名称都被认为是以 NULL 结尾的 UTF-8 格式的字符串。

偏移	大小	域	描述
0	8	ShortName	8 字节长的数组。如果名称长度小于 8 字节，在它的右边用 NULL 填充。
0	4	Zeroes	如果名称长度大于 8 字节，那么这个域被设置为全 0。
4	4	Offset	字符串表中的一个偏移地址。

5.4.2 SectionNumber 域的值

通常符号表项中的 SectionNumber 域是节表的索引（从 1 开始）。但是这个域是带符号整数，因此它可以为负值。下面这些小于 1 的值有特殊含义：

常量	值	描述
IMAGE_SYM_UNDEFINED	0	尚未为此符号记录分配一个节。这个零值表明引用了一个定义在其它地方的外部符号；而非零值则表明是一个普通符号，其大小由 Value 域给出。
IMAGE_SYM_ABSOLUTE	-1	此符号是个绝对符号（不可重定位），并且不是地址。
IMAGE_SYM_DEBUG	-2	此符号提供普通类型信息或者调试信息，但它并不对应于某一个节。Microsoft 的工具将.file 记录（存储类别为 FILE）设置为这个值。

5.4.3 类型表示

符号表项的 Type 域占 2 个字节，其中的每一个字节都表示类型信息。低位字节（LSB）表示简单（基本）数据类型，高位字节（MSB）表示复杂类型（如果存在）：

MSB	LSB
复杂类型：无、指针、函数、数组。	基本类型：整数、浮点数等。

尽管 Microsoft 的工具通常不使用这个域并将 LSB 设置为 0，但还是为基本类型定义了以下这些值。Microsoft 的工具使用 Visual C++调试信息来指明类型。但是出于完整性考虑，将可能出现的值列于下表：

常量	值	描述
IMAGE_SYM_TYPE_NULL	0	类型信息不存在，或者是未知的基本类型。Microsoft 的工具使用这个值。
IMAGE_SYM_TYPE_VOID	1	不是合法类型；用于 void 指针和函数。
IMAGE_SYM_TYPE_CHAR	2	字符（带符号的 1 个字节）。
IMAGE_SYM_TYPE_SHORT	3	长度为 2 个字节的带符号整数。

常量	值	描述
IMAGE_SYM_TYPE_INT	4	自然的整数类型（在 Windows 中通常为 4 个字节）。
IMAGE_SYM_TYPE_LONG	5	长度为 4 个字节的带符号整数。
IMAGE_SYM_TYPE_FLOAT	6	长度为 4 个字节的浮点数。
IMAGE_SYM_TYPE_DOUBLE	7	长度为 8 个字节的浮点数。
IMAGE_SYM_TYPE_STRUCT	8	结构体。
IMAGE_SYM_TYPE_UNION	9	共用体。
IMAGE_SYM_TYPE_ENUM	10	枚举类型。
IMAGE_SYM_TYPE_MOE	11	枚举类型成员（具体值）。
IMAGE_SYM_TYPE_BYTE	12	字节；长度为 1 个字节的无符号整数。
IMAGE_SYM_TYPE_WORD	13	字；长度两个字节的无符号整数。
IMAGE_SYM_TYPE_UINT	14	长度为自然尺寸的无符号整数（通常为 4 个字节）。
IMAGE_SYM_TYPE_DWORD	15	长度为 4 个字节的无符号整数。

高位字节指出符号是指向由 LSB 指定的基本类型的指针、返回由 LSB 指定的基本类型的函数还是由 LSB 指定的基本类型组成的数组。Microsoft 的工具仅使用这个域来指出这个符号是不是函数，因此最后的 Type 域只有 0x0 和 0x20 这两个值。但是其它工具可以使用这个域来传递更多信息。

正确地指定函数属性非常重要。增量链接需要这个信息才能正确地工作。对于某些平台而言，可能会出于其它目的而使用这个信息。

常量	值	描述
IMAGE_SYM_DTYPE_NULL	0	非导出类型；此符号是简单的标量变量。
IMAGE_SYM_DTYPE_POINTER	1	此符号是指向基本类型的指针。
IMAGE_SYM_DTYPE_FUNCTION	2	此符号是返回基本类型的函数。
IMAGE_SYM_DTYPE_ARRAY	3	此符号是由基本类型组成的数组。

5.4.4 存储类别

符号表中的 StorageClass 域指出符号具体的存储类别。下表列出了所有可能的取值。注意 StorageClass 域是长度为 1 个字节的无符号整数。因此如果这个域的值为-1 的话，实际上应该被看作是与它相等的无符号数，也就是 0xFF。

尽管传统的 COFF 格式使用许多存储类别，但是 Microsoft 的工具使用 Visual C++调试信息来表示大部分符号信息，它通常仅使用四种存储类别：EXTERNAL（2）、STATIC（3）、FUNCTION（101）和 FILE（103）。下表中出现的“Value”都表示符号表记录中的 Value 域（它的意义依赖于存储类别的值）。

常量	值	描述以及对 Value 域的解释
IMAGE_SYM_CLASS_END_OF_FUNCTION	-1 (0xFF)	表示函数结尾的特殊符号，用于调试。
IMAGE_SYM_CLASS_NULL	0	未被赋予存储类别。
IMAGE_SYM_CLASS_AUTOMATIC	1	自动（堆栈）变量。Value 域指出此变量在栈帧中的偏移。
IMAGE_SYM_CLASS_EXTERNAL	2	Microsoft 的工具使用此值来表示外部符号。如果 SectionNumber 域为 0（IMAGE_SYM_UNDEFINED），那么 Value 域给出大小；如果 SectionNumber 域不为 0，那么 Value 域给出节中的偏移。
IMAGE_SYM_CLASS_STATIC	3	符号在节中的偏移。如果 Value 域为 0，那么此符号表示节名。

常量	值	描述以及对 Value 域的解释
IMAGE_SYM_CLASS_REGISTER	4	寄存器变量。Value 域给出寄存器编号。
IMAGE_SYM_CLASS_EXTERNAL_DEF	5	在外部定义的符号。
IMAGE_SYM_CLASS_LABEL	6	模块中定义的代码标号。Value 域给出此符号在节中的偏移。
IMAGE_SYM_CLASS_UNDEFINED_LABEL	7	引用的未定义的代码标号。
IMAGE_SYM_CLASS_MEMBER_OF_STRUCT	8	结构体成员。Value 域指出是第几个成员。
IMAGE_SYM_CLASS_ARGUMENT	9	函数的形式参数（形参）。Value 域指出是第几个参数。
IMAGE_SYM_CLASS_STRUCT_TAG	10	结构体名。
IMAGE_SYM_CLASS_MEMBER_OF_UNION	11	共用体成员。Value 域指出是第几个成员。
IMAGE_SYM_CLASS_UNION_TAG	12	共用体名。
IMAGE_SYM_CLASS_TYPE_DEFINITION	13	Typedef 项。
IMAGE_SYM_CLASS_UNDEFINED_STATIC	14	静态数据声明。
IMAGE_SYM_CLASS_ENUM_TAG	15	枚举类型名。
IMAGE_SYM_CLASS_MEMBER_OF_ENUM	16	枚举类型成员。Value 域指出是第几个成员。
IMAGE_SYM_CLASS_REGISTER_PARAM	17	寄存器参数。
IMAGE_SYM_CLASS_BIT_FIELD	18	位域。Value 域指出是位域中的第几个位。
IMAGE_SYM_CLASS_BLOCK	100	**.bb**（beginning of block，块开头）或**.eb 记录**（end of block，块结尾）。Value 域是代码位置，它是一个可重定位的地址。
IMAGE_SYM_CLASS_FUNCTION	101	Microsoft 的工具用此值来表示定义函数范围的符号记录，这些符号记录分别是：**.bf**（begin function，函数开头）、**.ef**（end function，函数结尾）以及**.lf**（lines in function，函数中的行）。对于**.lf** 记录来说，Value 域给出了源代码中此函数所占的行数。对于**.ef** 记录来说，Value 域给出了函数代码的大小。
IMAGE_SYM_CLASS_END_OF_STRUCT	102	结构体末尾。
IMAGE_SYM_CLASS_FILE	103	Microsoft 的工具以及传统 COFF 格式都使用此值来表示源文件符号记录。这种符号表记录后面跟着给出文件名的辅助符号表记录。
IMAGE_SYM_CLASS_SECTION	104	节的定义（Microsoft 的工具使用 STATIC 存储类别代替）。
IMAGE_SYM_CLASS_WEAK_EXTERNAL	105	弱外部符号。要获取更多信息，请参考 5.5.3 节“辅助符号表记录格式之三：弱外部符号”。
IMAGE_SYM_CLASS_CLR_TOKEN	107	表示 CLR 记号的符号。它的名称是这个记号的十六进制值的 ASCII 码表示。要获取更多信息，请参考 5.5.7 节“CLR 记号定义”。

5.5 辅助符号表记录

辅助符号表记录总是跟在一些标准符号表记录之后并应用于这些符号表记录。它的格式并不固定，只要处理它的工具能够识别就可以了，但必须保证其长度是 18 个字节，以便整个符号表可以被当作一个数组来处理。当前 Microsoft 的工具可以识别

以下类型的辅助符号表记录：函数定义、函数开头和结尾符号（.bf 和.ef）、弱外部符号、文件名以及节的定义。

传统的 COFF 格式中还包括用于数组和结构体的辅助符号表记录。但是 Microsft 的工具并不使用它们，而是将符号信息按照 Visual C++调试信息格式存储在调试节中。

5.5.1 辅助符号表记录格式之一：函数定义

如果一个符号表记录拥有下列属性：存储类别为 EXTERNAL（2）、Type 域的值表明它是一个函数（0x20）以及 SectionNumber 域的值大于 0，它就标志着函数的开头。注意如果一个符号表记录 SectionNumber 域的值为 IMAGE_SYM_UNDEFINED（0），那么它并不定义一个函数，也没有相应的辅助符号表记录。能够定义函数的符号表记录后面跟着如下格式的辅助符号表记录：

偏移	大小	域	描述
0	4	TagIndex	相应的.bf（函数开头）记录在符号表中的索引。
4	4	TotalSize	函数经编译后生成的可执行代码的大小。如果此函数单独成节，那么根据对齐值的不同，节头中的 SizeOfRawData 域可能大于或等于这个域。
8	4	PointerToLinenumber	如果此函数存在行号记录，那么这个值表示它的第一个 COFF 行号记录的文件偏移；如果不存在，那么这个值为 0。要获取更多信息，请参考 5.3 节“COFF 行号信息（不赞成使用）”。
12	4	PointerToNextFunction	对应于下一个函数的符号表记录在符号表中的索引。如果此函数是符号表中的最后一个函数，那么这个域的值为 0。
16	2	未用	

5.5.2 辅助符号表记录格式之二：.bf 和.ef 符号

对于符号表中的每个函数定义，有三个相应的符号表记录分别用来描述函数开头、函数结尾以及此函数在源文件中所占的行数。这三个符号表记录的存储类别都是 FUNCTION（101）：

- 一个名为.bf 的符号表记录。此记录的 Value 域未用。
- 一个名为.lf 的符号表记录。此记录的 Value 域给出了这个函数所占的行数。
- 一个名为.ef 的符号表记录。此记录的 Value 域的值与定义函数的符号表记录的 TotalSize 域的值相同。

.bf 和.ef 符号表记录（不包括.lf 记录）后面跟着如下格式的辅助符号表记录：

偏移	大小	域	描述
0	4	未用	
4	2	Linenumber	源文件中实际的行号，按顺序排列（1、2、3 等等）。.bf 和.ef 记录都使用这个域。
6	6	未用	
12	4	PointerToNextFunction（仅用于.bf 符号表记录）	下一个.bf 符号表记录在符号表中的索引。如果此函数是符号表中的最后一个函数，那么这个域的值为 0。.ef 记录并不使用这个域。
16	2	未用	

5.5.3 辅助符号表记录格式之三：弱外部符号

“弱外部符号”是为了在链接时能获得更大的灵活性而应用于目标文件的一种机制。模块中可以包含一个无法解析的外部符号（假设为 sym1），但它也可以同时包含一个相应的辅助符号表记录来指明当链接时找不到 sym1 时可以用另外一个外部符号（假设为 sym2）来代替。

如果链接时能找到 sym1 的定义，那么外部对它的引用可以正常解析。否则所有对 sym1 这个弱外部符号的引用都用 sym2 来代替。而 sym2 这个外部符号在链接时必须能够找到；通常它就被定义在对 sym1 进行弱外部引用的模块中。

表示弱外部符号的符号表记录，其存储类别是 EXTERNAL，SectionNumber 域的值为 IMAGE_SYM_UNDEFINED（0），Value 域的值为 0。弱外部符号记录后面跟着如下格式的辅助符号表记录：

偏移	大小	域	描述
0	4	TagIndex	sym2 在符号表中的索引。如果链接时找不到 sym1 就用它代替。
4	4	Characteristics	如果这个值为 IMAGE_WEAK_EXTERN_SEARCH_NOLIBRARY，表明链接时不在库中查找 sym1。 如果这个值为 IMAGE_WEAK_EXTERN_SEARCH_LIBRARY，表明链接时在库中查找 sym1。 如果这个值为 IMAGE_WEAK_EXTERN_SEARCH_ALIAS，表明 sym1 是 sym2 的别名。
8	10	未用	

注意在 WINNT.H 文件中并没有定义这个 Characteristics 域，而是用 TotalSize 域来代替。

5.5.4 辅助符号表记录格式之四：文件

此格式的辅助符号表记录跟在存储类别为 FILE（103）的符号表记录之后。这个符号表记录的符号名本身应该是.file，而跟着它的辅助记录给出了源文件名。

偏移	大小	域	描述
0	18	File Name	表示源文件名的 ANSI 字符串。如果源文件名的长度小于最大长度，用 NULL 填充。

5.5.5 辅助符号表记录格式之五：节定义

此格式的辅助符号表记录跟在定义节的符号表记录之后。对于定义节的符号表记录来说，其符号名就是相应的节名（例如.text 或.drectve）且存储类别为 STATIC（3）。而相应的辅助记录提供了关于这个节的信息。因此这样的辅助符号表记录中的一些信息与节头中的信息是重复的。

偏移	大小	域	描述
0	4	Length	节中数据的大小。与节头的 SizeOfRawData 域一样。
4	2	NumberOfRelocations	此节中重定位项的数目。
6	2	NumberOfLinenumbers	此节中行号信息项的数目。
8	4	CheckSum	公共数据的校验和。只有节头中设置了 IMAGE_SCN_LNK_COMDAT 标志时才使用此域。要获取更多信息，请参考 5.5.6 节“COMDAT 节（仅适用于目标文件）”。

偏移	大小	域	描述
12	2	Number	与此节相关的节在节表中的索引（从 1 开始）。当 COMDAT 的 Selection 域为 5 时才使用这个域。
14	1	Selection	表示 COMDAT 选择方式的数字。这个域只用于 COMDAT 节。
15	3	未用	

5.5.6 COMDAT 节（仅适用于目标文件）

如果一个节为 COMDAT 节，那么定义这个节的符号表记录后跟着的辅助记录中的 Selection 域是可用的。COMDAT 节是一种可以由多个目标文件定义的节。（节头中的 Characteristics 域设置了 IMAGE_SCN_LNK_COMDAT 标志）。Selection 域的值决定了链接器在解析多个 COMDAT 节定义时所采用的方式。

第一个其 SectionNumber 域指向 COMDAT 节的符号必须是定义节的符号。这个符号的名称就是节名，其 Value 域的值为 0，SectionNumber 域指向我们正在讨论的 COMDAT 节，Type 域的值为 IMAGE_SYM_TYPE_NULL，StorageClass 域的值为 IMAGE_SYM_CLASS_STATIC，并且有一个辅助记录。第二个符号被称为“COMDAT 符号”，链接器将它与 Selection 域配合起来使用。

Selection 域的值如下所示：

常量	值	描述
IMAGE_COMDAT_SELECT_NODUPLICATES	1	如果此符号已经被定义过，链接器将生成一个“multiply defined symbol（符号多重定义）”错误。
IMAGE_COMDAT_SELECT_ANY	2	链接器从这些定义同一个 COMDAT 符号的节中任选一个，其余（未被选中）的节都被移除。
IMAGE_COMDAT_SELECT_SAME_SIZE	3	链接器从定义这个符号的多个节中任选一个。如果所有这些定义大小不等，链接器将生成一个“符号多重定义”错误。
IMAGE_COMDAT_SELECT_EXACT_MATCH	4	链接器从定义这个符号的多个节中任选一个。如果所有这些定义不严格一致，链接器将生成一个“符号多重定义”错误。
IMAGE_COMDAT_SELECT_ASSOCIATIVE	5	如果“其它某个”COMDAT 节被链接的话，此节也要被链接。这里的“其它某个”节由与定义此节的符号表记录相关的辅助符号表记录的 Number 域给出。这个设置对于那些在多个节中都有其相关部分（例如代码在一个节中而数据在另一个节中）但必须作为一个整体进行链接或丢弃的定义非常有用。与此节关联的这个“其它某个”节必须也是 COMDAT 节并且它不能再与其它 COMDAT 节关联（也就是说，这个“其它某个”节不能将 Selection 域设置为 IMAGE_COMDAT_SELECT_ASSOCIATIVE）。
IMAGE_COMDAT_SELECT_LARGEST	6	链接器从这个符号的所有定义中选取长度最大的进行链接。如果长度最大的不止一个，那么就在这几个最大的中任选一个。

5.5.7 CLR 记号定义（仅适用于目标文件）

这种辅助符号表记录通常跟在存储类别为 IMAGE_SYM_CLASS_CLR_TOKEN 的符号表记录之后。它用来联系 CLR 记号与 COFF 符号表的名称空间。

偏移	大小	域	描述
0	1	bAuxType	必须为 IMAGE_AUX_SYMBOL_TYPE_TOKEN_DEF（1）。
1	1	bReserved	保留，必须为 0。
2	4	SymbolTableIndex	此 CLR 记号定义涉及的 COFF 符号在符号表中的索引。
6	12	保留，必须为 0。	

5.6 COFF 字符串表

COFF 符号表后紧跟着的是 COFF 字符串表。它的位置可以通过将 COFF 文件头中符号表的地址加上符号总数乘以每个符号的大小得到。

COFF 字符串表开头的 4 个字节存储的是字符串表的总大小（以字节计）。这个大小包括这个域本身。因此如果字符串表中不包含任何字符串时，这个值应该为 4。

大小后面是一些以 NULL 结尾的字符串，COFF 符号表中的符号指向这些字符串。

5.7 属性证书表（仅适用于映像文件）

可以给映像文件添加属性证书表使它与属性证书相关联。有多种不同类型的属性证书，最常用的是 Authenticode®签名。

属性证书表包含一个或多个长度固定的表项，可以通过可选文件头中的数据目录中的 Certificate Table（证书表）域（偏移 128 处）找到它们。这个表的每个表项给出了相应证书的开始位置和长度。存储在这个节中的每个证书都有一个相应的证书表项。证书表项的数目可以通过将证书表的大小（可以在偏移 132 处找到）除以证书表中每一项的大小（8）得到。注意证书表的大小仅包括它的表项，并不包括这些表项实际指向的证书。

每个表项格式如下：

偏移	大小	域	描述
0	4	Certificate Data	指向证书实际数据的文件指针。它指向的地址总是按 8 字节倍数边界（即最低 3 个位都是 0）对齐。
0	4	Size of Certificate	这是一个无符号整数，它指出证书的大小（以字节计）。

注意证书总是从八进制字（从任意字节边界开始的 16 个连续字节）边界开始。如果一个证书的长度不是八进制字长度的偶数倍，那么就一直用 0 填充到下一个八进制字边界。但是证书长度并不包括这些填充的 0。因此任何处理证书的软件必须向上舍入到下一个八进制字才能找到另一个证书。

5.7.1 证书数据

这是表示属性证书的二进制数据。证书的起始位置和长度由证书表中相应的表项给出。每个证书都有惟一一个与它对应的表项。

5.8 延迟加载导入表（仅适用于映像文件）

将这些表添加到映像文件中是为了给“应用程序直到首次调用某个 DLL 中的函数或数据时才加载这个 DLL（即延迟加载）”这种行为提供一种通一的机制。这些表的

格式与将要在 6.4 节“.idata 节”中描述的传统导入表的格式一致。这里仅讨论少许细节。

5.8.1 延迟加载目录表

延迟加载目录表是导入目录表的副本，可以从可选文件头中的数据目录中的 Delay Import Descriptor（延迟导入描述符）域（偏移 200 处）获取它。其格式如下：

偏移	大小	域	描述
0	4	Attributes	必须为 0。
4	4	Name	被加载 DLL 名称的 RVA。这个名称位于映像文件的只读数据节中。
8	4	Module Handle	被延迟加载 DLL 模块句柄（在映像的数据节中）的 RVA。它供管理延迟加载的例程存储之用。
12	4	Delay Import Address Table	延迟加载导入地址表的 RVA。要获取更多信息，请参考 5.8.5 节“延迟导入地址表（IAT）”。
16	4	Delay Import Name Table	延迟加载名称表的 RVA。这个表包含了需要被加载的导入符号的名称。它与导入名称表的格式一致。要获取更多信息，请参考 6.4.3 节“提示/名称表”。
20	4	Bound Delay Import Table	绑定延迟加载地址表（如果存在）的 RVA。
24	4	Unload Delay Import Table	卸载延迟加载地址表（如果存在）的 RVA。这个表是延迟导入地址表的精确副本。如果调用者卸载了这个 DLL，应该将这个表复制回去以覆盖延迟导入地址表，以便后续对这个 DLL 的调用可以继续正确地使用形实转换机制。
28	4	Time Stamp	此映像绑定到的 DLL 的时间戳。

对于这个结构中使用到的表来说，它们的组织形式与存储格式与传统的导入表一样，就像是它们的副本。要获取更多信息，请参考 4.6 节“.idata 节”。

5.8.2 属性

至今尚未定义属性标志。链接器在生成映像文件时将此域设置为 0。我们可以在将来扩展这个结构时用它来指明添加了新域，或者用它来指明延迟加载或卸载辅助函数的行为。

5.8.3 名称

需要被延迟加载的 DLL 的名称被存储在映像文件的只读数据节中。可以通过 szName 域引用它。

5.8.4 模块句柄

要被延迟加载的 DLL 的句柄被存储在映像文件的数据节中。phmod 域指向这个句柄。延迟加载辅助函数使用这个位置存储要被延迟加载的 DLL 的模块句柄。

5.8.5 延迟导入地址表

可以通过延迟导入描述符的 pIAT 域引用延迟导入地址表（IAT）。延迟加载辅助函数用导入符号的实际地址来更新这些指针，以便起转换作用的这部分代码不会陷入循环调用之中。可以通过表达式 pINT->u1.Function 来访问这些函数指针。

5.8.6 延迟导入名称表

延迟导入名称表（INT）包含了可能需要被加载的导入符号的名称。它们的排列方式与 IAT 中的函数指针一样。它们的结构与标准的 INT 一样，可以通过表达式 `pINT->u1.AddressOfData->Name[0]`进行访问。

5.8.7 延迟绑定导入地址表和时间戳

延迟绑定导入地址表（BIAT）是由 IMAGE_THUNK_DATA 结构组成的数组，它是可选的。它与延迟加载目录表中的 Time Stamp 域一起被用于后处理绑定阶段。

5.8.8 延迟卸载导入地址表

延迟卸载导入地址表（UIAT）是由 IMAGE_THUNK_DATA 结构组成的数组，它是可选的。卸载代码用它来处理明确的卸载请求。它由只读节中已初始化的数据组成，这些数据是原始 IAT 的精确副本。在处理卸载请求时，可以释放这个 DLL，同时将 `*phmod` 清零，并用 UIAT 覆盖 IAT，以便将一切都还原到预加载时的状态。

6 特殊的节

典型的 COFF 节包含的是普通代码或数据，链接器与 Microsoft Win32®加载器并不需要知道其中的内容就可以处理它们。这些内容只与将要链接的或执行的应用程序有关。

但是目标文件与映像文件中的某些 COFF 节却有特殊含义。由于在节头中为这些节设置了特殊的标志，或者可选文件头中的某些域指向这些节，或者节名本身指出了节的特殊作用，所以工具和加载器可以识别它们。（尽管节名本身并不表明节的特殊作用，但节名都是约定俗成的，因此本规范的作者在任何情况下都可以使用节名。）

下表描述了保留的节以及它们的属性，后面是对出现在可执行文件中的节以及包含元数据用于扩展目的的节的详细描述。

节名	内容	特征
.bss	未初始化的数据（自由格式）	IMAGE_SCN_CNT_UNINITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE
.cormeta	CLR 元数据，它表明目标文件中包含托管代码	IMAGE_SCN_LNK_INFO
.data	已初始化的数据（自由格式）	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE
.debug$F	生成的 FPO 调试信息（仅适用于目标文件，仅用于 x86 平台，现已被舍弃）	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_DISCARDABLE
.debug$P	预编译的调试类型信息（仅适用于目标文件）	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_DISCARDABLE
.debug$S	调试符号信息（仅适用于目标文件）	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_DISCARDABLE
.debug$T	调试类型信息（仅适用于目标文件）	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_DISCARDABLE
.drectve	链接器选项	IMAGE_SCN_LNK_INFO

节名	内容	特征
.edata	导出表	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ
.idata	导入表	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE
.idlsym	包含已注册的 SEH（仅适用于映像文件），它们用以支持 IDL 属性。要获取更多信息，请参考本规范最后“参考信息”中的“IDL 属性”。	IMAGE_SCN_LNK_INFO
.pdata	异常信息	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ
.rdata	只读的已初始化数据	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ
.reloc	映像文件的重定位信息	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_DISCARDABLE
.rsrc	资源目录	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ
.sbss	与 GP 相关的未初始化数据（自由格式）	IMAGE_SCN_CNT_UNINITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE | IMAGE _SCN_GPREL 其中 IMAGE_SCN_GPREL 标志仅用于 IA64 平台，不能用于其它平台。此标志只能用于目标文件。当映像文件中出现这种类型的节时，一定不能设置这个标志。
.sdata	与 GP 相关的已初始化数据（自由格式）	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE | IMAGE _SCN_GPREL 其中 IMAGE_SCN_GPREL 标志仅用于 IA64 平台，不能用于其它平台。此标志只能用于目标文件。当映像文件中出现这种类型的节时，一定不能设置这个标志。
.srdata	与 GP 相关的只读数据（自由格式）	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE _SCN_GPREL 其中 IMAGE_SCN_GPREL 标志仅用于 IA64 平台，不能用于其它平台。此标志只能用于目标文件。当映像文件中出现这种类型的节时，一定不能设置这个标志。
.sxdata	已注册的异常处理程序数据（自由格式，仅适用于目标文件，仅用于 x86 平台）	IMAGE_SCN_LNK_INFO 这个节中包含目标文件中的代码所涉及到的所有异常处理程序在符号表中的索引。这些符号可以是 IMAGE_SYM_UNDEFINED 类型的符号，也可以是定义在那个模块中的符号。
.text	可执行代码（自由格式）	IMAGE_SCN_CNT_CODE | IMAGE_SCN_MEM_EXECUTE | IIMAGE_SCN_MEM_READ

节名	内容	特征
.tls	线程局部存储（仅适用于目标文件）	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE
.tls$	线程局部存储（仅适用于目标文件）	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE
.vsdata	与 GP 相关的已初始化数据（自由格式，仅适用于 ARM、SH4 和 Thumb 平台）	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE
.xdata	异常信息（自由格式）	IMAGE_SCN_CNT_INITIALIZED_DATA | IMAGE_SCN_MEM_READ

上表中列出的一些节被标记为“仅适用于目标文件”或“仅适用于映像文件”，它们表示这些节的特殊含义只是分别对于目标文件或者映像文件来说的。标记为“仅适用于映像文件”的节仍然可以首先出现在目标文件中，最后再成为映像文件的一部分，但这个节对于链接器来说并没有特殊含义，它仅对于映像文件加载器才有特殊含义。

6.1 .debug 节

在目标文件中，.debug 节包含编译器生成的调试信息；在映像文件中，它包含生成的全部调试信息。本节描述了目标文件和映像文件中调试信息的封装。

下一节描述调试目录的格式，它可以被放在映像文件中的任何地方。后面几节描述目标文件中包含调试信息的成组的节。

默认情况下调试信息并不映射到映像的地址空间中。只有当调试信息被映射到地址空间中时.debug 节才会存在。

6.1.1 调试目录（仅适用于映像文件）

映像文件中包含一个可选的调试目录，它给出了调试信息的格式及位置。这个目录是一个由调试目录项组成的数组，它的位置和大小由可选文件头给出。

调试目录可以位于一个可丢弃的.debug 节（如果存在）中，或者位于映像文件的其它节中，或者不在任何节中。

每个调试目录项给出了一块调试信息的位置和大小。如果调试信息没有被节头包含（也就是说，它位于映像文件中但并没有被映射到运行时地址空间中）的话，那么这个指定的 RVA 可以为 0。如果被映射的话，这个 RVA 就是它的实际地址。

调试目录项格式如下：

偏移	大小	域	描述
0	4	Characteristics	保留，必须为 0。
4	4	TimeDateStamp	调试数据被创建的日期和时间。
8	2	MajorVersion	调试数据格式的主版本号。
10	2	MinorVersion	调试数据格式的次版本号。
12	4	Type	调试信息的格式。这个域的存在使得可以支持多个调试器。要获取更多信息，请参考 6.1.2 节“调试类型”。
16	4	SizeOfData	调试数据（不包括调试目录本身）的大小。
20	4	AddressOfRawData	当被加载时调试数据相对于映像基址的偏移地址。

偏移	大小	域	描述
24	4	PointerToRawData	指向调试数据的文件指针。

6.1.2 调试类型

为调试目录项的 Type 域定义了以下值：

常量	值	描述
IMAGE_DEBUG_TYPE_UNKNOWN	0	未知值，所有工具均忽略此值。
IMAGE_DEBUG_TYPE_COFF	1	COFF 调试信息（行号信息、符号表和字符串表）。文件头中也有相关域指向这种类型的调试信息。
IMAGE_DEBUG_TYPE_CODEVIEW	2	Visual C++调试信息。
IMAGE_DEBUG_TYPE_FPO	3	帧指针省略（FPO）信息。这种信息告诉调试器如何解释非标准栈帧，这种帧将 EBP 寄存器用于其它目的而不是作为帧指针。
IMAGE_DEBUG_TYPE_MISC	4	DBG 文件的位置。
IMAGE_DEBUG_TYPE_EXCEPTION	5	.pdata 节的副本。
IMAGE_DEBUG_TYPE_FIXUP	6	保留。
IMAGE_DEBUG_TYPE_OMAP_TO_SRC	7	从经过代码重排后的映像中的 RVA 到原映像中的 RVA 的映射。
IMAGE_DEBUG_TYPE_OMAP_FROM_SRC	8	从原映像中的 RVA 到经过代码重排后的映像中的 RVA 的映射。
IMAGE_DEBUG_TYPE_BORLAND	9	保留，供 Borland 公司使用。
IMAGE_DEBUG_TYPE_RESERVED10	10	保留。
IMAGE_DEBUG_TYPE_CLSID	11	保留。

如果 Type 域被设置为 IMAGE_DEBUG_TYPE_FPO，那么原始的调试数据是一个数组，它的每个元素都描述了一个函数的栈帧。即使调试类型为 FPO，也不是映像文件中的每个函数都必须有相应的 FPO 信息。没有相应 FPO 信息的那些函数的栈帧被当作正常的栈帧对待。FPO 信息的格式如下：

```
#define FRAME_FPO   0
#define FRAME_TRAP  1
#define FRAME_TSS   2

typedef struct _FPO_DATA {
    DWORD       ulOffStart;             // 函数代码第一个字节的偏移
    DWORD       cbProcSize;             // 函数代码所占的字节数
    DWORD       cdwLocals;              // 局部变量所占字节数除以 4
    WORD        cdwParams;              // 参数所占字节数除以 4

    WORD        cbProlog : 8;           // 函数 prolog 代码所占字节数
    WORD        cbRegs   : 3;           // 保存的寄存器数
    WORD        fHasSEH  : 1;           // 如果函数中有 SEH，此值为 TRUE
    WORD        fUseBP   : 1;           // 如果 EBP 寄存器已经被分配，此值为 TRUE
    WORD        reserved : 1;           // 保留供将来使用
    WORD        cbFrame  : 2;           // 帧类型
} FPO_DATA;
```

6.1.3 .debug$F 节（仅适用于目标文件）

在 Visual C++ 7.0 及其后续版本中，这个节中的数据已经被保存在.debug$S 节中一组更详尽的数据取代。

目标文件可以包含内容为一个或多个 FPO_DATA 记录（帧指针省略信息）的.debug$F 节。要获取更多信息，请参考 6.1.2 节“调试类型”中的“IMAGE_DEBUG_TYPE_FPO”。

链接器能够识别这些.debug$F 记录。如果指定要生成调试信息的话，链接器根据函数的 RVA 将相应的 FPO_DATA 记录排序并分别为它们生成一个调试目录项。

编译器不应该为具有标准帧格式的过程生成 FPO 记录。

6.1.4 .debug$S 节（仅适用于目标文件）

这个节中包含的是 Visual C++调试信息（符号信息）。

6.1.5 .debug$P 节（仅适用于目标文件）

这个节中包含的是 Visual C++调试信息（预编译信息）。它们是在所有使用由这个目标文件生成的预编译头编译生成的目标文件之间共享的类型信息。

6.1.6 .debug$T 节（仅适用于目标文件）

这个节中包含的是 Visual C++调试信息（类型信息）。

6.1.7 链接器对 Microsoft 调试信息的支持

为了支持调试信息，链接器需要：

- 收集.debug$F、debug$S、.debug$P 和.debug$T 这些节中所有相关的调试数据。
- 将这些数据连同链接器生成的调试信息综合处理，生成一个 PDB 文件，同时创建一个指向这个文件的调试目录项。

6.2 .drectve 节（仅适用于目标文件）

如果在节头中将一个节设置了 IMAGE_SCN_LNK_INFO 标志，并且这个节被命名为.drectve，那么它就是一个指令节。链接器在处理完其中的信息后就会移除这个节，因此它并不会出现在链接生成的映像文件中。

.drectve 节由 ANSI 格式或 UTF-8 格式的文本字符串组成。如果 UTF-8 字节顺序标记（BOM，由 0xEF、0xBB 和 0xBF 组成的三字节前缀）不存在，那么指令字符串就被当作 ANSI 字符串。指令字符串是由空格分隔的一串链接器选项。每个选项包括一个连字符、选项名称以及相应的属性。如果某个选项包含空格，必须用双引号将整个选项括起来。.drectve 节一定不能有重定位或行号信息。

6.3 .edata 节（仅适用于映像文件）

导出数据节被命名为.edata，它包含的是与其它映像通过动态链接可以访问的符号相关的信息。导出符号通常出现在 DLL 中，但 DLL 也可以导入符号。

下表描述了导出节的一般结构。其中提到的各种表通常都是按下表中所示的顺序在文件中连续分布的（尽管这并不是必须的）。只需要导出目录表和导出地址表这两个表就可以将序数作为导出符号导出。（序数是一个导出符号，可以把它作为导出地址表中的索引直接引用。）导出名称指针表、导出序数表和导出名称表的存在都是为了支持使用导出名称。

表名	描述
导出目录表	一个只有一行的表（与调试目录不同）。它给出了其它各种导出表的位置和大小。

表名	描述
导出地址表	一个由导出符号的 RVA 组成的数组。它们是导出的函数和数据在可执行代码节和数据节内的实际地址。其它映像文件可以通过使用这个表的索引（序数）来导入符号，或者如果定义了一个与序数对应的公用名称的话，也可以用这个公用名称来导入符号。
导出名称指针表	一个由指向导出符号名称的指针组成的数组，按升序排列。
导出序数表	一个由对应于导出名称指针表中各个成员的序数组成的数组。它们的对应是通过位置来体现的，因此导出名称指针表与导出序数表成员数目必须相同。每个序数都是导出地址表的一个索引。
导出名称表	一系列以 NULL 结尾的 ASCII 码字符串。导出名称指针表中的成员都指向这个区域。它们都是公用名称，符号导入与导出就是通过它们。这些公用名称并不需要与映像文件内部使用的私有名称相同。

当其它映像文件通过名称导入符号时，Win32 加载器通过导出名称指针表来搜索匹配的字符串。如果找到，它就查找导出序数表中相应的成员（也就是说，将找到的导出名称指针表的索引作为导出序数表的索引来使用）来获取与导入符号相关联的序数。获取的这个序数是导出地址表的索引，这个索引对应的元素给出了所需符号的实际位置。每个导出符号都可以通过序数进行访问。

当其它映像文件通过序数导入符号时，就不再需要通过导出名称指针表来搜索匹配的字符串了。因此直接使用序数效率会更高。但是导出名称容易记忆，它不需要用户记住各个符号在表中的索引。

6.3.1 导出目录表

导出目录表是导出符号信息的开始部分，它描述了导出符号信息中其余部分的内容。导出目录表包含了有关将导入符号解析为相应入口点地址的信息。

偏移	大小	域	描述
0	4	Export Flags	保留，必须为 0。
4	4	Time/Date Stamp	导出数据被创建的日期和时间。
8	2	Major Version	主版本号。用户可以自行设置主版本号和次版本号。
10	2	Minor Version	次版本号。
12	4	Name RVA	包含这个 DLL 名称的 ASCII 码字符串相对于映像基址的偏移地址。
16	4	Ordinal Base	映像中导出符号的起始序数值。这个域指定了导出地址表的起始序数值。它通常被设置为 1。
20	4	Address Table Entries	导出地址表中元素的数目。
24	4	Number of Name Pointers	导出名称指针表中元素的数目。它同时也是导出序数表中元素的数目。
28	4	Export Address Table RVA	导出地址表相对于映像基址的偏移地址。
32	4	Name Pointer RVA	导出名称指针表相对于映像基址的偏移地址。它的大小由 Number of Name Pointers 域给出。
36	4	Ordinal Table RVA	导出序数表相对于映像基址的偏移地址。

6.3.2 导出地址表

导出地址表包含导出函数、导出数据以及绝对符号的地址。序数值是作为导出地址表的索引来使用的。

导出地址表的每一个元素都是一个域，其格式为下表所述的两种格式之一。如果指定的地址*不是*位于导出节（其地址和长度由可选文件头给出）中，那么这个域就是一个 Export RVA，这个地址就是导出符号在代码或数据中的实际地址；否则这个域是一个 Forwarder RVA，它给出了一个位于其它 DLL 中的符号的名称。

偏移	大小	域	描述
0	4	Export RVA	当加载进内存时，导出符号相对于映像基址的偏移地址。例如导出函数的地址。
0	4	Forwarder RVA	这是指向导出节中一个以 NULL 结尾的 ASCII 码字符串的指针。这个字符串必须位于 Export Table（导出表）数据目录项给出的范围之内。要获取更多信息，请参考 3.4.3 节“可选文件头中的数据目录（仅适用于映像文件）”。这个字符串给出了导出符号所在 DLL 的名称以及导出符号的名称（例如“MYDLL.expfunc”），或者 DLL 的名称以及导出符号的序数值（例如“MYDLL.#27”）。

Forwarder RVA 导出了其它映像中定义的符号，使它看起来好像是当前映像导出的一样。因此对于当前映像来说，这个符号同时既是导入符号又是导出符号。

例如对于 Windows XP 系统中的 Kernel32.dll 文件来说，它导出的“HeapAlloc”被转发到“NTDLL.RtlAllocateHeap”。这样就允许应用程序使用 Windows XP 系统中的 Ntdll.dll 模块而不需要实际包含任何相关的导入信息。应用程序的导入表只与 Kernel32.dll 有关。这样应用程序就不是特定于 Windows XP，它可以运行于任何 Win32 系统之上。

6.3.3 导出名称指针表

导出名称指针表是由导出名称表中的字符串的地址（RVA）组成的数组。每个地址都是相对于映像基址的 32 位指针。它们按词法顺序排列以便进行二进制搜索。

只有当导出名称指针表中包含指向某个导出名称的指针时，这个导出名称才算被定义。

6.3.4 导出序数表

导出序数表是由导出地址表的索引组成的一个数组，每个序数长 16 位。但是这个序数是移码形式，相应的偏移量由导出目录表中的 Ordinal Base 域给出。换句话说，必须从序数值中减去 Ordinal Base 域的值得到的才是导出地址表真正的索引。

导出名称指针表和导出序数表是两个并列的数组，将它们分开是为了使它们可以分别按照各自的自然边界（前者是 4 个字节，后者是 2 个字节）对齐。在进行操作时，这两个表等效于一个表（在这个等效的新表中，这两个表分别相当于新表中的两列），由导出名称指针这一列给出公共（导出）符号的名称，而由导出序数这一列给出这个导出符号对应的序数。导出名称指针表的成员和导出序数表的成员通过同一个索引相关联。

因此，当在导出名称指针表中搜索并找到匹配字符串的索引为 *i* 时，查找符号对应地址的算法如下：

```
i = Search_ExportNamePointerTable (ExportName);
ordinal = ExportOrdinalTable [i];
SymbolRVA = ExportAddressTable [ordinal - OrdinalBase];
```

6.3.5 导出名称表

导出名称表包含的是导出名称指针表实际指向的字符串。这个表中的字符串都是公用名称，其它映像可以通过它们导入这些符号。这些公用导出名称并不需要与这些符号所在的映像文件和源代码中的私有符号名称相同，尽管它们可以相同。

每个导出符号都有一个相应的序数值，这个值是导出地址表的索引（加上 Ordinal Base 域的值）。但是使用导出名称（来导出符号）是可选的。可以全部导出符号都有导出名称，也可以部分导出符号有导出名称，也可以全部导出符号都没有导出名称。对于那些有导出名称的导出符号来说，导出名称指针表和导出序数表中相应的项配合起来使每个名称与一个序数相关联。

导出名称表的结构就是长度可变的一系列以 NULL 结尾的 ASCII 码字符串。

6.4. idata 节

所有导入符号的映像文件，实际上几乎包括所有的可执行（EXE）文件，都有.idata 节。文件中导入信息的典型布局如下：

图 3. 典型的导入节布局

6.4.1 导入目录表

导入目录表是导入信息的开始部分，它描述了导入信息中其余部分的内容。导入目录表包含地址信息，这些地址信息用来修正对 DLL 映像中的相应函数的引用。导入目录表是由导入目录项组成的数组，每个导入目录项对应着一个导入的 DLL。最后一个导入目录项是空的（全部域的值都为 NULL），用来指明目录表的结尾。

每个导入目录项的格式如下：

偏移	大小	域	描述
0	4	Import Lookup Table RVA (Characteristics)	导入查找表的 RVA。这个表包含了每一个导入符号的名称或序数。（这个域在 Winnt.h 文件中的名称是“Characteristics”，但已经名不副实了。）

偏移	大小	域	描述
4	4	Time/Date Stamp	这个域一直被设置为 0，直到映像被绑定。当映像被绑定之后，这个域被设置为这个 DLL 的日期/时间戳。
8	4	Forwarder Chain	第一个转发项的索引。
12	4	Name RVA	包含 DLL 名称的 ASCII 码字符串相对于映像基址的偏移地址。
16	4	Import Address Table RVA (Thunk Table)	导入地址表的 RVA。这个表的内容与导入查找表的内容完全一样，直到映像被绑定。

6.4.2 导入查找表

导入查找表是由长度为 32 位（PE32）或 64 位（PE32+）的数字组成的数组。其中的每一个元素都是位域，其格式如下表所示。在这种格式中，位 31（PE32）或位 63（PE32+）是最高位。这些项描述了从给定的 DLL 导入的所有符号。最后一个项被设置为 0（NULL），用来指明表的结尾。

位	大小	位域	描述
31/63	1	Ordinal/Name Flag	如果这个位为 1，说明是通过序数导入的。否则是通过名称导入的。测试这个位的掩码为 0x80000000（PE32）或 0x8000000000000000（PE32+）。
15-0	16	Ordinal Number	序数值（16 位长）。只有当 Ordinal/Name Flag 域为 1（即通过序数导入）时才使用这个域。位 30-15（PE32）或 62-15（PE32+）必须为 0。
30-0	31	Hint/Name Table RVA	提示/名称表项的 RVA（31 位长）。只有当 Ordinal/Name Flag 域为 0（即通过名称导入）时才使用这个域。对于 PE32+来说，位 62-31 必须为 0。

6.4.3 提示/名称表

一个提示/名称表就能满足整个导入节的需要。提示/名称表中的每一个元素结构如下：

偏移	大小	域	描述
0	2	Hint	导出名称指针表的索引。当搜索匹配字符串时首选使用这个值。如果匹配失败，再在 DLL 的导出名称指针表中进行二进制搜索。
2	可变	Name	包含导入符号名称的 ASCII 码字符串。这个字符串必须与 DLL 导出的公用名称匹配。同时这个字符串区分大小写并且以 NULL 结尾。
*	0 或 1	Pad	为了让提示/名称表的下一个元素出现在偶数地址，这里可能需要填充 0 个或 1 个 NULL 字节。

6.4.4 导入地址表

导入地址表的结构和内容与导入查找表完全一样，直到文件被绑定。在绑定过程中，用导入符号的 32 位（PE32）或 64 位（PE32+）地址覆盖导入地址表中的相应项。这些地址是导入符号的实际内存地址，尽管技术上仍把它们称为“虚拟地址”。加载器通常会处理绑定。

6.5 .pdata 节

.pdata 节是由用于异常处理的函数表项组成的数组。可选文件头中的 Exception Table（异常表）域指向它。在将它们放进最终的映像文件之前，这些项必须按函数

地址（下列每个结构的第一个域）排序。下面描述了函数表项的三种格式，使用哪一种取决于目标平台。

对于 32 位的 MIPS 映像来说，其函数表项格式如下：

偏移	大小	域	描述
0	4	Begin Address	相应函数的 VA。
4	4	End Address	函数结尾的 VA。
8	4	Exception Handler	指向要执行的异常处理程序的指针。
12	4	Handler Data	指向要传递给异常处理程序的附加数据的指针。
16	4	Prolog End Address	函数 prolog 代码结尾的 VA。

对于 ARM、PowerPC、SH3 和 SH4 Windows CE 平台来说，其函数表项格式如下：

偏移	大小	域	描述
0	4	Begin Address	相应函数的 VA。
4	8 位	Prolog Length	函数 prolog 代码包含的指令数。
4	22 位	Function Length	函数代码包含的指令数。
4	1 位	32-bit Flag	如果此位为 1，表明函数由 32 位指令组成。否则，函数由 16 位指令组成。
4	1 位	Exception Flag	如果此位为 1，表明存在用于此函数的异常处理程序；否则，不存在异常处理程序。

对于 x64 和 Itanium 平台来说，其函数表项格式如下：

偏移	大小	域	描述
0	4	Begin Address	相应函数的 RVA。
4	4	End Address	函数结尾的 RVA。
8	4	Unwind Information	用于异常处理的展开（Unwind）信息的 RVA。

6.6 .reloc 节（仅适用于映像文件）

基址重定位表包含了映像中所有需要重定位的内容。可选文件头中的数据目录中的 Base Relocation Table（基址重定位表）域给出了基址重定位表所占的字节数。要获取更多信息，请参考 3.4.3 节“可选文件头中的数据目录（仅适用于映像文件）”。基址重定位表被划分成许多块，每一块表示一个 4K 页面范围内的基址重定位信息，它必须从 32 位边界开始。

加载器不需要处理由链接器解析的基址重定位信息，除非映像不能被加载到 PE 文件头中指定的映像基地址处。

6.6.1 基址重定位块

每个基址重定位块的开头都是如下结构：

偏移	大小	域	描述
0	4	Page RVA	将映像基址与这个域（页面 RVA）的和加到每个偏移地址处最终形成一个 VA，这个 VA 就是要进行基址重定位的地方。
4	4	Block Size	基址重定位块所占的总字节数，其中包括 Page RVA 域和 Block Size 域以及跟在它们后面的 Type/Offset 域。

Block Size 域后面跟着数目不定的 Type/Offset 位域。它们中的每一个都是一个 WORD（2 字节），其结构如下：

偏移	大小	域	描述
0	4 位	Type	它占这个 WORD 的最高 4 位，这个值指出需要应用的基址重定位类型。要获取更多信息，请参考 6.6.2 节“基址重定位类型”。
0	12 位	Offset	它占这个 WORD 的其余 12 位，这个值是从基址重定位块的 Page RVA 域指定的地址处开始的偏移。这个偏移指出需要进行基址重定位的位置。

为了进行基址重定位，需要计算映像的首选基地址与实际被加载到的基地址之差。如果映像本身就被加载到了其首选基地址，那么这个差为零，因此也就不需要进行基址重定位了。

6.6.2 基址重定位类型

常量	值	描述
IMAGE_REL_BASED_ABSOLUTE	0	基址重定位被忽略。这种类型可以用来对其它块进行填充。
IMAGE_REL_BASED_HIGH	1	进行基址重定位时将差值的高 16 位加到指定偏移处的一个 16 位域上。这个 16 位域是一个 32 位字的高半部分。
IMAGE_REL_BASED_LOW	2	进行基址重定位时将差值的低 16 位加到指定偏移处的一个 16 位域上。这个 16 位域是一个 32 位字的低半部分。
IMAGE_REL_BASED_HIGHLOW	3	进行基址重定位时将所有的 32 位差值加到指定偏移处的一个 32 位域上。
IMAGE_REL_BASED_HIGHADJ	4	进行基址重定位时将差值的高 16 位加到指定偏移处的一个 16 位域上。这个 16 位域是一个 32 位字的高半部分，而这个 32 位字的低半部分被存储在紧跟在这个 Type/Offset 位域后面的一个 16 位字中。也就是说，这一个基址重定位项占了两个 Type/Offset 位域的位置。
IMAGE_REL_BASED_MIPS_JMPADDR	5	对 MIPS 平台的跳转指令进行基址重定位。
	6	保留，必须为 0。
	7	保留，必须为 0。
IMAGE_REL_BASED_MIPS_JMPADDR16	9	对 MIPS16 平台的跳转指令进行基址重定位。
IMAGE_REL_BASED_DIR64	10	进行基址重定位时将差值加到指定偏移处的一个 64 位域上。

6.7 .tls 节

.tls 节为 PE 和 COFF 文件的静态线程局部存储（TLS）提供了直接支持。TLS 是 Windows 支持的一种特殊存储类别，它里面的数据对象不是自动（堆栈）变量，而是局限于运行相应代码的单个线程，因此每个线程都可以为使用 TLS 定义的变量维护一个不同的值。

通过调用 TlsAlloc、TlsFree、TlsSetValue 和 TlsGetValue 这些 API 可以支持任意数量的 TLS 数据。PE 和 COFF 对此的实现是变相地调用这些 API，其优点就是从高级语言程序员的观点来看这样更简单。这种实现方式使得 TLS 数据的定义与初始化就像程序中普通的静态变量那样。例如在 Visual C++中，对静态 TLS 变量的定义方式如下，不需要调用 Windows API 函数：

```
__declspec (thread) int tlsFlag = 1;
```

为了支持这种编程模式，PE 和 COFF 文件的.tls 节包含了以下信息：初始化数据、用于每个线程初始化和终止的回调函数以及下面将要讨论的 TLS 索引。

注意

静态定义的 TLS 数据对象只能用于静态加载的映像文件。这使得在 DLL 中使用静态 TLS 数据并不可靠，除非你能确定这个 DLL 以及静态链接到这个 DLL 的其它 DLL 永远不会被通过调用 LoadLibrary 这个 API 函数的方式动态加载。

可执行代码访问静态 TLS 数据需要经过以下步骤：

1. 在链接时，链接器设置 TLS 目录（见下文）中的 Address of Index 域。这个域指向一个位置，在这个位置保存程序用到的 TLS 索引。

   Microsoft 运行时库为了处理方便就定义了一个 TLS 目录的内存映像并给它取名为“__tls_used”（Intel x86 平台）或“_tls_used”（其它平台）。链接器查找这个内存映像并直接使用其中的数据创建 TLS 目录。其它支持 TLS 并与 Microsoft 的链接器配合使用的编译器必须使用这种技术。

2. 当创建线程时，加载器通过将线程环境块（TEB）的地址放入 FS 寄存器来传递线程的 TLS 数组地址。距 TEB 开头 0x2C 的位置处有一个指针指向 TLS 数组。这是特定于 Intel x86 平台的。

3. 加载器将 TLS 索引值保存到 Address of Index 域指向的位置处。

4. 可执行代码获取 TLS 索引以及 TLS 数组的位置。

5. 可执行代码使用 TLS 索引和 TLS 数组的位置（将索引乘以 4 并作为这个数组内的偏移地址来使用）来获取给定程序和模块的 TLS 数据区的地址。每个线程拥有它自己的 TLS 数据区，但这对于程序是透明的，它并不需要知道是如何为单个线程分配数据的。

6. 单个的 TLS 数据对象都位于 TLS 数据区的某个固定偏移处，因此可以用这种方式访问。

TLS 数组是系统为每个线程维护的一个地址数组。这个数组中的每个地址指出了程序中给定模块（EXE 或 DLL）的 TLS 数据区的位置。TLS 索引指出了是这个数组的哪个元素。这个索引是一个用来标识具体模块的数字（仅对系统有意义）。

6.7.1 TLS 目录

TLS 目录结构如下：

偏移（PE32/PE32+）	大小（PE32/PE32+）	域	描述
0	4/8	Raw Data Start VA	TLS 模板的起始地址。这个模板是一块数据，用于对 TLS 数据进行初始化。每当创建线程时，系统都要复制所有这些数据，因此这些数据一定不能出错。注意这个地址并不是一个 RVA，.reloc 节中应当有一个基址重定位信息是用于这个地址的。
4/8	4/8	Raw Data End VA	TLS 的最后一个字节的地址，不包括用于填充的 0。与 Raw Data Start VA 域一样，它是一个 VA 而不是 RVA。

偏移（PE32/PE32+）	大小（PE32/PE32+）	域	描述
8/16	4/8	Address of Index	用于保存 TLS 索引的位置，索引的具体值由加载器确定。这个位置在普通的数据节中，因此可以给它取一个容易理解的符号名，便于在程序中访问。
12/24	4/8	Address of Callbacks	这是一个指针，它指向由 TLS 回调函数组成的数组。这个数组是以 NULL 结尾的，因此如果没有回调函数的话，这个域指向的位置处应该是 4 个字节的 0。要获取有关这些函数原型方面的信息，请参考 6.7.2 节“TLS 回调函数”。
16/32	4	Size of Zero Fill	TLS 模板中除了由 Raw Data Start VA 和 Raw Data End VA 域组成的已初始化数据界限之外的大小（以字节计）。TLS 模板的总大小应该与映像文件中 TLS 数据的总大小一致。用 0 填充的数据就是已初始化的非零数据后面的那些数据。
20/36	4	Characteristics	保留，可能将来用作 TLS 标志。

6.7.2 TLS 回调函数

程序可以提供一个或多个 TLS 回调函数，用以支持对 TLS 数据进行附加的初始化和终止操作。调用对象的构造函数和析构函数应该算是对这种回调函数的典型应用。

尽管回调函数通常不超过一个，但还是将其作为一个数组来实现，以便在需要时可以另外添加回调函数。如果回调函数超过一个，将会按照它们的地址在数组中出现的顺序调用每个函数。一个空指针表示这个数组的结尾。让这个列表空着（不提供回调函数）也是完全合法的，这时这个数组就只有一个元素——一个空指针。

回调函数（由类型为 PIMAGE_TLS_CALLBACK 的函数指针指向）的原型与 DLL 入口点函数参数相同：

```
typedef VOID
(NTAPI *PIMAGE_TLS_CALLBACK) (
    PVOID DllHandle,
    DWORD Reason,
    PVOID Reserved
    );
```

Reserved 参数应该被设置为 0。Reason 参数可以取以下值：

设置	值	描述
DLL_PROCESS_ATTACH	1	启动了一个新进程，包括第一个线程。
DLL_THREAD_ATTACH	2	创建了一个新线程。创建所有线程时都会发送这个通知，除了第一个线程。
DLL_THREAD_DETACH	3	线程将要被终止。终止所有线程时都会发送这个通知，除了第一个线程。
DLL_PROCESS_DETACH	0	进程将要被终止，包括第一个线程。

6.8 加载配置结构（仅适用于映像文件）

加载配置结构（IMAGE_LOAD_CONFIG_DIRECTORY）最初用于 Windows NT 操作系统自身几种非常有限的场合——在映像文件头或可选文件头中描述各种特性太困难或这些信息尺寸太大。当前版本的 Microsoft 链接器和 Windows XP 以及后续版本的 Windows 使用的是这个结构的新版本，将之用于包含保留的 SEH 技术的基于 x86 的 32 位系统上。它提供了一个安全的结构化异常处理程序列表，操作系统在进行异常

派送时要用到这些异常处理程序。如果异常处理程序的地址在映像的 VA 范围之内，并且映像被标记为支持保留的 SEH（也就是说可选文件头中的 DllCharacteristics 域没有设置 IMAGE_DLLCHARACTERISTICS_NO_SEH 标志，前面已经提到过），那么这个异常处理程序必须在映像的已知安全异常处理程序列表中，否则操作系统将终止这个应用程序。这是为了防止利用“x86 异常处理程序劫持”来控制操作系统，它在以前已经被利用过。

Microsoft 的链接器自动提供一个默认的加载配置结构来包含保留的 SEH 数据。如果用户的代码已经提供了一个加载配置结构，那么它必须包含新添加的保留的 SEH 域。否则，链接器将不能包含保留的 SEH 数据，这样映像文件就不能被标记为包含保留的 SEH。

6.8.1 加载配置目录

对应于预保留的 SEH 加载配置结构的数据目录项必须为加载配置结构指定一个特别的大小，因为操作系统加载器总是希望它为这样一个特定值。事实上，这个大小只是用于检查这个结构的版本。为了与 Windows XP 以及以前版本的 Windows 兼容，x86 映像文件中这个结构的大小必须为 64。

6.8.2 加载配置结构布局

用于 32 位和 64 位 PE 文件的加载配置结构布局如下：

偏移	大小	域	描述
0	4	Characteristics	指示文件属性的标志，当前未用。
4	4	TimeDateStamp	日期/时间戳。这个值表示从 UTC 时间 1970 年 1 月 1 日午夜（00:00:00）以来经过的总秒数，它是根据系统时钟算出的。可以用 C 运行时函数 **time** 来获取这个时间戳。
8	2	MajorVersion	主版本号。
10	2	MinorVersion	次版本号。
12	4	GlobalFlagsClear	当加载器启动进程时，需要被清除的全局加载器标志。
16	4	GlobalFlagsSet	当加载器启动进程时，需要被设置的全局加载器标志。
20	4	CriticalSectionDefaultTimeout	用于这个进程处于无约束状态的临界区的默认超时值。
24	8	DeCommitFreeBlockThreshold	返回到系统之前必须释放的内存数量（以字节计）。
32	8	DeCommitTotalFreeThreshold	空闲内存总量（以字节计）。
40	8	LockPrefixTable	[仅适用于 x86 平台] 这是一个地址列表的 VA。这个地址列表中保存的是使用 LOCK 前缀的指令的地址，这样便于在单处理器机器上将这些 LOCK 前缀替换为 NOP 指令。
48	8	MaximumAllocationSize	最大的分配粒度（以字节计）。
56	8	VirtualMemoryThreshold	最大的虚拟内存大小（以字节计）。
64	8	ProcessAffinityMask	将这个域设置为非零值等效于在进程启动时将这个设定的值作为参数去调用 **SetProcessAffinityMask** 函数（仅适用于.exe 文件）。

偏移	大小	域	描述
72	4	ProcessHeapFlags	进程堆的标志，相当于 **HeapCreate** 函数的第一个参数。这些标志用于在进程启动过程中创建的堆。
76	2	CSDVersion	Service Pack 版本标识。
78	2	Reserved	必须为 0。
80	8	EditList	保留，供系统使用。
60/88	4/8	SecurityCookie	指向 cookie 的指针。cookie 由 Visual C++编译器的 GS 实现所使用。
64/96	4/8	SEHandlerTable	［仅适用于 x86 平台］这是一个地址列表的 VA。这个地址列表中保存的是映像中每个合法的、独一无二的 SE 处理程序的 RVA，并且它们已经按 RVA 排序。
68/104	4/8	SEHandlerCount	［仅适用于 x86 平台］表中独一无二的 SE 处理程序的数目。

6.9 .rsrc 节

一个多层的二叉排序树指向各种资源，树的深度可达 2^{31} 层。但是 Windows 通常使用如下 3 层：

类型
名称
语言

一系列资源目录表按如下方式与各层相联系：每个目录表后面跟着一系列目录项，它们给出那个层（类型、名称或语言）的名称或标识（ID）及其数据描述或另一个目录表的地址。如果这个地址指向一个数据描述，那么那个数据就是这棵树的叶子。如果这个地址指向另一个目录表，那么那个目录表列出了下一层的目录项。

一个叶子的类型、名称和语言 ID 由从目录表到这个叶子的路径决定。第一个表决定类型 ID，第二个表（由第一个表中的目录项指向）决定名称 ID，第三个表决定语言 ID。

.rsrc 节的一般结构如下：

数据	描述
资源目录表（以及资源目录项）	这是一系列表，其中每一个表都对应着树中的一组结点。所有的顶层（类型）结点都被列于第一个表中。这个表中的项指向第二层表。每个第二层树的类型 ID 相同但是名称 ID 不同。第三层树的类型 ID 和名称 ID 都相同但语言 ID 不同。 每个单个的表后面紧跟着目录项，每一项都有一个名称或数字标识和一个指向数据描述或下一层表的指针。
资源目录字符串	按 2 字节边界对齐的 Unicode 字符串，它是作为由目录项指向的字符串数据来使用的。
资源数据描述	一个由记录组成的数组，由表指向它，描述了资源数据的实际大小和位置。这些记录是资源描述树中的叶子。
资源数据	资源节的原始数据。资源数据描述域中的大小和位置信息将资源数据分成单个的区域。

6.9.1 资源目录表

每个资源目录表格式如下。此结构应该被看作是表头，因为这个表实际上是由多个目录项（在 6.9.2 节“资源目录项”中详细描述）和这个结构组成：

偏移	大小	域	描述
0	4	Characteristics	资源标志。保留供将来使用。当前它被设置为 0。
4	4	Time/Date Stamp	资源数据被资源编译器创建的时间。
8	2	Major Version	主版本号，由用户设定。
10	2	Minor Version	次版本号，由用户设定。
12	2	Number of Name Entries	紧跟着这个表的目录项的个数，这些目录项使用名称字符串来标识类型、名称或语言项（依赖于表的层次）。
14	2	Number of ID Entries	紧跟着名称项的目录项的个数，这些目录项使用数字 ID 来标识类型、名称或语言项。

6.9.2 资源目录项

这些目录项组成了表中的各行。每个资源目录项格式如下。这个项是名称项还是 ID 项取决于资源目录表，这个表指出跟着它的名称项和 ID 项各有多少个（表中所有的名称项在所有的 ID 项前面）。表中的所有项按升序排列：名称项是按不区分大小写的字符串，而 ID 项则是按数值。

偏移	大小	域	描述
0	4	Name RVA	表示类型、名称或语言 ID 项（依赖于表的层次）的名称字符串的地址。
0	4	Integer ID	标识类型、名称或语言 ID 项的 32 位整数。
4	4	Data Entry RVA	最高位为 0。资源数据项（叶子）的地址。
4	4	Subdirectory RVA	最高位为 1。低 31 位是另一个资源目录表（下一层）的地址。

6.9.3 资源目录字符串

资源目录字符串区由按字边界对齐的 Unicode 字符串组成。这些字符串被存储在最后一个资源目录项之后、第一个资源数据项之前。这样能够使这些长度可变的字符串对长度固定的目录项的对齐情况影响最小。每个资源目录字符串格式如下：

偏移	大小	域	描述
0	2	Length	字符串的长度，不包括 Length 域本身。
2	可变	Unicode String	按字边界对齐的可变长度的 Unicode 字符串。

6.9.4 资源数据项

每个资源数据项描述了资源数据区中一个实际单元的原始数据。资源数据项格式如下：

偏移	大小	域	描述
0	4	Data RVA	资源数据区中一个单元的资源数据的地址。
4	4	Size	由 Data RVA 域指向的资源数据的大小（以字节计）。
8	4	Codepage	用于解码资源数据中的代码点值的代码页。通常这个代码页应该是 Unicode 代码页。
12	4	保留，必须为 0。	

6.10 .cormeta 节（仅适用于目标文件）

CLR 元数据被存储在这个节中。它用来指明目标文件中包含托管代码。虽然元数据的格式并未公开，但是可以使用处理元数据的 CLR 接口来处理它们。

6.11 .sxdata 节

目标文件中合法的异常处理程序被列于这个目标文件的.sxdata 节中。这个节被标记为 IMAGE_SCN_LNK_INFO。它包含了每个合法的异常处理程序在 COFF 符号表中的索引，每个索引占 4 个字节。

另外，编译器通过生成一个名为“@feat.00”且其 Value 域的 LSB 为 1 的绝对符号来标识这个 COFF 目标文件中有已注册的 SEH。没有已注册的 SEH 的 COFF 目标文件有“@feat.00”符号，但没有.sxdata 节。

7 档案（库）文件格式

COFF 档案文件格式提供了一种存储目标文件集合的标准机制。这些目标文件集合在编程文档中通常被称为库（Library）。

档案文件的前 8 个字节是文件签名。其余部分是一系列档案文件成员，如下所示：

- 第一个和第二个成员是“链接器成员”。它们的格式如 8.3 节“导入名称类型”所述。通常链接器会将信息放在这些档案文件成员中。链接器成员包含了档案文件目录。
- 第三个成员是“长名称”成员。它由一系列以 NULL 结尾的 ASCII 码字符串组成，其中的每一个字符串都是其它档案文件成员的名称。
- 其余的都是标准成员（目标文件）。它们中的每一个都包含一个完整的目标文件。

每个档案文件成员前面都有一个头部。下图显示的就是一个普通档案文件的结构：

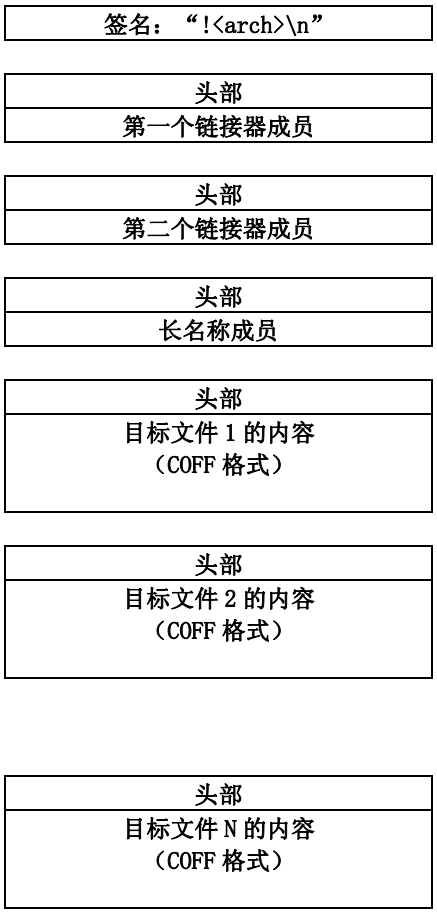

图 4. 档案文件结构

7.1 档案文件签名

档案文件签名标识了文件类型。任何处理档案文件的工具（例如链接器）都可以通过读取这个签名来检查文件类型。这个签名由下面的 ASCII 码字符组成，其中除了换行符（\n）为转义字符外其余的字符都是普通字符：

```
!<arch>\n
```

7.2 档案文件成员头部

每个成员（链接器成员、长名称成员或者目标文件成员）前面都是头部。档案文件成员头部格式如下表所示。其中的每个域都是一个左对齐的 ASCII 码字符串，长度不足的用空格填充。所有这些域都没有结尾的 NULL 字符。

每个档案文件成员头部都是从前一个档案文件成员结束后的首个偶数地址开始。

偏移	大小	域	描述
0	16	Name	档案文件成员的名称，最后添加一个斜杠（/）作为结束。如果它的第一个字符就是斜杠，那么这个名称具有特殊含义，将在后面的表中详细描述。
16	12	Date	此档案文件成员创建的日期与时间：这是一个用 ASCII 码表示的十进制数，它代表从 UTC 时间 1970 年 1 月 1 日午夜（00:00:00）开始的总秒数。

偏移	大小	域	描述
28	6	User ID	这是一个用 ASCII 码表示的十进制数，它代表用户 ID。在 Windows 平台上这个域并不包含任何有意义的值，因为 Microsoft 的工具将这个域全部用空格填充。
34	6	Group ID	这是一个用 ASCII 码表示的十进制数，它代表组 ID。在 Windows 平台上这个域并不包含任何有意义的值，因为 Microsoft 的工具将这个域全部用空格填充。
40	8	Mode	这是一个用 ASCII 码表示的八进制数，它代表档案文件成员的文件模式。它是 C 运行时函数_wstat 返回的_stat 结构中的 st_mode 域的值。
48	10	Size	这是一个用 ASCII 码表示的十进制数，它表示档案文件成员的大小，不包括头部。
58	2	End of Header	两字节的 C 语言格式字符串“`\n”（0x60 0x0A）。

Name 域的几种格式如下表所示。 如前文所述，每个字符串都向左对齐并且不足 16 字节的部分用空格填充：

Name 域的内容	描述
名称/	这个档案文件成员的名称。
/	这个档案文件成员是两个链接器成员之一。两个链接器成员都是这个名称。
//	这个档案文件成员是“长名称”成员，它由一系列以 NULL 结尾的 ASCII 码字符串组成。“长名称”成员是第三个档案文件成员，即使内容为空它也必须存在。
/n	这个档案文件成员的名称在“长名称”成员内的偏移 n 处。数字 n 是用十进制数表示的偏移。例如“/26”表明该档案文件成员的名称位于距“长名称”成员内容开头 26 字节的位置处。

7.3 第一个链接器成员

第一个链接器成员的名称是“/”。包含它是为了向后兼容。当前的链接器并不使用它，但是它的格式必须正确。这个链接器成员与第二个链接器成员一样，都提供了一个符号名称目录。对于其中的每个符号，相应的信息指出到哪里能够找到包含这个符号的档案文件成员。

第一个链接器成员格式如下，它位于档案文件成员头部之后：

偏移	大小	域	描述
0	4	Number of Symbols	这是一个无符号长整型数，表示被索引的符号数目。它是按大尾格式存储的。通常每个目标文件成员都会定义一个或多个外部符号。
4	4 * n	Offsets	这是一个由档案文件成员头部在文件中的偏移地址组成的数组，它的大小（4 * n）中的 n 指的是 Number of Symbols 域的值。这个数组中的每个元素都是一个按大尾格式存储的无符号长整型数。对于 String Table 域中命名的每个符号，这个数组中相应的元素给出了包含这个符号的档案文件成员的位置。
*	*	String Table	一系列以 NULL 结尾的字符串，它们给出了目录中所有符号的名称。每个字符串都紧跟前一个字符串最后的 NULL 字符。字符串的数目必须与 Number of Symbols 域的值相等。

偏移数组（上面的第二个域）中的元素必须按升序排列。这意味着字符串表（上面的第三个域）中的符号也必须根据档案文件成员的顺序进行排列。例如第一个目标文件成员中的符号都排在第二个目标文件成员中的符号之前。

7.4 第二个链接器成员

第二个链接器成员的名称与第一个链接器成员的名称一样，也是“/”。尽管这两个链接器成员都提供了符号目录以及包含这些符号的档案文件成员目录，但是第二个链接器成员是当前所有链接器的首选。它包含的符号名称是按词法顺序排列的，这样按照名称进行搜索会更快。

第二个链接器成员格式如下，它位于档案文件成员头部之后：

偏移	大小	域	描述
0	4	Number of Members	这是一个表示档案文件成员数目的无符号长整型数。
4	4 * m	Offsets	这是一个由档案文件成员头部在文件中的偏移地址组成的数组，这些偏移地址是按升序排列的，其中的每一个偏移地址都是一个无符号长整型数。它的大小（4 * m）中的 m 指的是 Number of Members 域的值。
*	4	Number of Symbols	这是一个无符号长整型数，表示被索引的符号数目。通常每个目标文件成员都会定义一个或多个外部符号。
*	2 * n	Indices	这是一个将符号名称映射到档案文件成员偏移的数组，它的索引（无符号短整型数）从 1 开始。它的大小（2 * n）中的 n 指的是 Number of Symbols 域的值。对于 String Table 域中命名的每一个符号，这个数组中相应的元素给出了偏移数组（第二个域）中的一个索引，而偏移数组中这个索引对应的元素最终给出了包含这个符号的档案文件成员的位置。
*	*	String Table	一系列以 NULL 结尾的字符串，它们给出了目录中所有符号的名称。每个字符串都紧跟前一个字符串最后的 NULL 字符。字符串的数目必须与 Number of Symbols 域的值相等。这个表按词法顺序且升序列出了所有符号的名称。

7.5 长名称成员

长名称成员的名称是“//”。它由一系列档案文件成员名称字符串组成。只有当 Name 域（长度为 16 字节）没有足够的空间来保存名称字符串时它才会出现在这里。尽管长名称成员的头部必须存在，但它本身却可以为空。

所有这些字符串都是以 NULL 结尾。每个字符串都紧跟前一个字符串最后的 NULL 字符。

8 导入库格式

传统的导入库，也就是描述由一个映像导出供其它映像使用的符号的库，其格式通常如 7 节“档案（库）文件格式”所示。主要区别是导入库成员包含的是伪造的目标文件而不是真正的目标文件，这些伪造的目标文件中仅包含用于创建 6.4 节“.idata 节”中描述的导入表所需的信息。链接器在创建需要导出符号的应用程序时会生成这种档案文件。

节中用于一个导入符号的部分可以从一小组信息中推断出来。在创建导入库时链接器可以为每一个成员生成完整且详细的信息，也可以仅写入一些标准信息而让以后使用这个库的应用程序很快生成所需的信息。

在长格式的导入库中，单个成员包含以下信息：

档案成员头
文件头

节头
对应于每个节头的数据
COFF 符号表
字符串

而短格式的导入库格式如下：

档案成员头
导入头
以 NULL 结尾的导入名称字符串
以 NULL 结尾的 DLL 名称字符串

有了这些信息就足以在使用这个成员时精确地重构它的整个内容。

8.1 导入头

导入头格式如下：

偏移	大小	域	描述
0	2	Sig1	必须为 IMAGE_FILE_MACHINE_UNKNOWN。要获取更多信息，请参考 3.3.1 节“机器类型”。
2	2	Sig2	必须为 0xFFFF。
4	2	Version	此结构的版本号。
6	2	Machine	标识目标机器类型的数字。要获取更多信息，请参考 3.3.1 节“机器类型”。
8	4	Time/Date Stamp	此文件被创建的时间和日期。
12	4	Size Of Data	此头后面跟着的字符串的长度。
16	2	Ordinal/Hint	或者是导入符号的序数，或者是导入符号的提示值，具体是哪一个取决于 Name Type 域的值。
18	2 位	Type	导入类型。要获取具体值以及相应的描述，请参考 8.2 节“导入类型”。
	3 位	Name Type	导入名称类型。要获取具体值以及相应的描述，请参考 8.3 节“导入名称类型”。
	11 位	Reserved	保留，必须为 0。

此结构后面跟着两个以 NULL 结尾的字符串，它们分别是导入符号的名称和这个导入符号所在的 DLL 的名称。

8.2 导入类型

为导入头中的 Type 域定义了以下值：

常量	值	描述
IMPORT_CODE	0	可执行代码。
IMPORT_DATA	1	数据。
IMPORT_CONST	2	在.def 文件中指定为 CONST。

这些值用来确定如果使用这个库的工具要访问那种数据时必须生成什么信息。

8.3 导入名称类型

以 NULL 结尾的导入符号名称字符串紧跟着与它相关的导入头。为导入头中的 Name Type 域定义了以下值。它们用来指明如何利用名称来生成表示导入符号的正确符号。

常量	值	描述
IMPORT_ORDINAL	0	通过序数导入。这表明导入头中 Ordinal/Hint 域的值是导入符号的序数。如果未指定这个值，那么 Ordinal/Hin 域的值就是导入符号的提示值。
IMPORT_NAME	1	导入符号的名称与公用符号名称相同。
IMPORT_NAME_NOPREFIX	2	导入符号名称是公用符号名称，但是没有前导的?、@或可选的_。
IMPORT_NAME_UNDECORATE	3	导入符号名称是公用符号名称，但是没有前导的?、@或可选的_，并且在第一个@处截断。

附录 A：计算 Authenticode® PE 映像散列

有好几种属性证书都可以用来校验映像的完整性，但最常用的是 Authenticode® 签名。它可以用来校验 PE 映像文件中相关的节没有从其原始形式以任何手段被更改过。要完成这个任务，Authenticode 签名包含了一些被称为 PE 映像散列的内容。

A.1 什么是 Authenticode PE 映像散列？

Authenticode PE 映像散列，简称为文件散列，与文件校验和类似，都是生成一个与文件完整性有关的比较小的值。但是校验和是用简单的算法生成的，它主要用来检测内存错误。也就是说，它用来检测保存在磁盘上的一块内存是否已经被破坏以及存储在那里的值是否已经被改变。文件散列与校验和的相同之处在于它也检测文件是否被破坏。但与大多数校验和算法不同的是，要对一个文件进行修改并同时保持它的文件散列与原始（未修改的）形式一致是极其困难的。也就是说，校验和主要是用来检测能够导致出错的简单内存错误，而文件散列可以用来检测诸如一些病毒、黑客或者木马程序对文件进行的有意的、甚至是非常细微的修改。

在 Authenticode 签名中，文件散列已经被这个文件的签名者用其私钥进行了数字签名。软件用户可以通过计算文件的散列值并与 Authenticode 数字签名中包含的已经签名的散列值进行比较来校验文件的完整性。如果文件散列不匹配，那就表明文件中被 PE 映像散列保护的部分已经被修改了。

A.2 Authenticode PE 映像散列保护了哪些内容？

计算 PE 映像散列时不需要、也不可能包含映像文件中的所有数据。有时候是不需要，例如公开发行的文件中包含的没有移除的调试信息；有时候则是不可能。例如，将映像文件中的所有信息都包含到 Authenticode 签名中并将这个包含 PE 映像散列的 Authenticode 签名插入到 PE 映像文件中，而后再次包含映像文件中的所有数据来计算 PE 映像散列，那么这两个 PE 映像散列不可能一致，因为文件中现在已经包含了 Authenticode 签名而之前并不包含它。

本附录解释了 PE 映像散列是如何计算的以及修改 PE 映像中的哪些部分不会使 Authenticode 签名失效。

值得注意的是，PE 映像散列可以被包含在一个单独的编录文件（.cat）中而不需要包含被散列的文件中的属性证书。这是相对的，因为通过修改并不实际包含 Authenticode 签名的 PE 映像使经过 Authenticode 签名的编录文件中的 PE 映像散列失效是可能的。

生成 Authenticode PE 映像散列的过程

PE 映像节表中指定的节中的所有数据要被整体散列，除了以下被排除的内容：

- **可选文件头中的 Windows 特定域中的 CheckSum 域**。这个校验和包含了整个文件（包括文件中的所有属性证书）。将 Authenticode 签名插入到文件中后，新的校验和十有八九与原来的值不同。
- **与属性证书有关的信息**。在计算 PE 映像散列时 PE 映像中与 Authenticode 签名有关的内容并不包含在内，因为给映像添加或移除 Authenticode 签名并不影响整个映像的完整性。这并不是问题，因为用户有时可能会重新对它签名或添加时间戳。Authenticode 在计算散列时排除以下信息：

  可选文件头中的数据目录中的 Certificate Table（证书表）域。

  Certificate Table（证书表）域指向的证书表以及相应的证书。

计算 PE 映像散列时，Authenticode 将节表中指定的节按地址范围排序，然后将产生的内容当成字节流进行散列，跳过被排除的内容。

- **最后一个节之后的信息**。最后一个节（通过最高偏移定义）之后的内容不散列。它们通常包含的是调试信息。调试信息一般是为了给调试器提供信息，它并不影响可执行程序实际的完整性。在产品交货之后将调试信息从映像中移除的可能性太大了，它并不影响程序的功能。实际上，有时这是节约磁盘空间的一种方法。值得注意的是，PE 映像的特定节中包含的调试信息不能从映像中移除，否则 Authenticode 签名就会失效。

你可以使用 Windows Platform SDK 中提供的 MakeCert 和 SignTool 工具进行创建和校验 Authenticode 签名方面的试验。要获取更多信息，请参考本规范最后的“参考信息”。

参考信息

IDL属性
http://msdn2.microsoft.com/en-US/library/8tesw2eh.aspx

创建、查看和管理证书
http://msdn2.microsoft.com/en-us/library/aa379872.aspx

Makecert
http://msdn2.microsoft.com/en-us/library/aa386968.aspx

SignTool
http://msdn2.microsoft.com/en-us/library/aa387764.aspx

Windows Server R2 2003 Platform SDK
http://www.microsoft.com/downloads/details.aspx?FamilyId=0BAF2B35-C656-4969-ACE8-E4C0C0716ADB&displaylang=en