链接器算法

作者：Matt Pietrek

在这个专栏中，我经常讨论一些新技术，至少是还未被广泛使用的技术。然而，随着越来越多的开发者加入到 Win32® 程序员队伍中来，有些对于老手来说是老生常谈的问题，对于新手来说却是神秘莫测的。链接器方面的主题就属于这个范畴。Visual Basic® 5.0 就使用了一个链接器。事实上，它使用的链接器与 Visual C++® 5.0 的一样。但是 Visual Basic 5.0 很好地隐藏了这个事实。如果你仔细观察就会发现，它产生出 OBJ 文件，然后把它们送往 Microsoft 链接器。

什么是链接器？它是如何工作的呢？本月我就讨论一些这方面的内容。作为这个专栏研究的一部分，我试图去找一些以前的资料。有趣的是，看起来，这里我要讲的内容不是停止印刷了，就是不再包含于 MSDN 光盘中了，即使链接器技术几乎影响到每一个 Windows 程序员。

为这个专栏考虑，我把 Microsoft 的 LINK.EXE 作为标准的链接器。（其它的链接器，例如 Borland 的 TLINK32，可能与我这里描述的有少许不同。）在以后的栏目中，我会讲一些关于 Microsoft 链接器方面更深入、更有趣、更有用的内容。首先，我需要给链接器一个极其概括的定义，然后再细化。链接器的工作是把一个或多个目标模块（典型地，就是 OBJ 文件）组合成一个可执行文件（也就是 EXE 或 DLL）。但是，这又引发一个问题：什么是目标模块呢？

目标模块是由一个程序产生的，这个程序把人类可读的文本转换成 CPU 可以理解的机器代码和数据。对于 C++来说，C++编译器读取 C++源文件。对于汇编语言来说，汇编程序（例如，MASM）读取汇编语言（ASM）文件，这种文件包含与 CPU 使用的代码和数据等价的指令。在 Visual Basic 5.0 中，输入文件就是工程中的 FRM，BAS，和 CLS 文件。这个概念对于其它大部分语言都是正确的，例如 Fortran。

目标模块中的主要部分是机器代码和数据。组成代码和数据的原始的字节被存储在连续的块中，这种块叫做节（section）。例如，Microsoft 编译器把他们的机器代码放进一个叫做.text 的节中，把数据放进一个叫做.data 的节中。这些名字除了提示节的用途之外并没有什么特别的意义。其它编译器能够（并且也是这么做的）对他们的节使用不同的名字。如果你曾经为 MS-DOS® 或 16 位 Windows®编写过程序，你把我前面讲的内容中的“节”都换成“段（segment）”，那么我讲的大部分也还是正确的。如果你系统中安装的有 Visual C++，你可以使用 DUMPBIN 程序来看一看 OBJ 文件中的节。执行以下的命令行：

```
DUMPBIN 目标文件名
```

目标文件名处是一个 OBJ 文件的名字。图 1 给出了一个常见节的片段。你可以用 DUMPBIN 对编译过的 C++程序中的 OBJ 文件试一下，例如 Visual C++\LIB 目录中的 CHKSTK.OBJ 文件：

```
Dump of file CHKSTK.OBJ

File Type: COFF OBJECT
```

Summary

0 .data

2F .text

图 1 常见节

.text　机器代码指令。

.data　已初始化的数据。

.rdata　只读数据。OLE 的 GUID 就保存在这里，还有其它东西。

.rsrc　资源。由资源编译器产生，被放进 RES 文件中。链接器把它复制到可执行文件中。

.reloc　基址重定位。由链接器产生。OBJ 文件中并没有。

.edata　导出函数表。由链接器创建，被放进 EXP 文件中。链接器把它复制到可执行中。

.idata　可执行文件中的导入函数表。

.idata$XXX　导入函数表的一部分。产生库的程序在导入库中创建这些节。在可执行文件中，链接器把它们组合成最终的.idata 节。

.CRT　可执行文件中的初始化表和关闭指针，它们供 Visual C++运行时库使用。

.CRT$XXX　链接器把它们组合进可执行文件之前，OBJ 中的初始化和关闭指针。

.bss　未初始化数据。

.drectve　OBJ 文件中包含链接器指令的节。它们并不被复制到可执行文件中。

.debug$XXX　OBJ 文件中的 COFF 符号表信息。

编译器或汇编器的输出有一个奇怪的名字叫做编译单元。然而，我们大部分人都认为它们只是 OBJ 文件。链接器最重要的工作就是收集所有编译单元并且从不同的编译单元中组合所有的节。当然，如果事情真的这么简单，那么链接器顶多不过是一个连接数据的奇怪程序。事实上，链接器工作中的复杂部分是处理修正问题。后面将详细叙述。

你可能想知道链接器是如何决定在最终的可执行文件中排列各个 OBJ 文件中的代码和数据节的。很明显，链接器有一整套详细规则要遵守。事实上，链接器的任务太复杂，因此，它不得不对输入文件进行两遍处理。链接器在第一遍时通篇查看要做的工作。在第二遍中，它应用所有规则产生可执行文件。

虽然对链接器规则的描述不能面面俱到，但我仍然会覆盖它的主要部分。链接器的主要规则就是从各个 OBJ 文件中提取代码和数据，并把它们放进最后的可执行文件中。如果你给链接器三个 OBJ 文件，这三个文件中的代码与数据最后按某种方式被组合进可执行文件中。然而，链接器并不是简单地把各个文件中的所有原始节一个挨一个地放在一起。相反，链接器把所有名字相同的节组合（也就是连接）在一起。例如，如果三个 OBJ 文件中每个都有.text 节，最后的可执行文件中只有单个.text 节，链接器按它们出现的顺序把它们组合在一起。

链接器遵守的另一个规则就是，可执行文件中节的顺序是由链接器处理节时遇到它们的顺序决定的。链接器严格按照命令行上指定的 OBJ 文件的顺序进行处理。但是，组合具有相同名字的节这条规则优先。

图 2 显示了三个 OBJ 文件，A.OBJ，B.OBJ，C.OBJ。每个文件都有三个节，其中.text 节和.data 节是三者共有的，但是在不同的文件中位置不同。它们都有一个与它们的源文件（也就是，a.asm，b.asm，c.asm）有关的节。调用 LINK，使用下面的内容作为参数：

A.OBJ B.OBJ C.OBJ

段的顺序（以及名字相同的节是如何被组合的）如图 2 所示。你可以从 http://www.microsoft.com/msj下载源文件和OBJ文件。提供OBJ文件主要是因为，如果你没有MASM或与其兼容的汇编程序，你可以使用OBJ文件，用“Link B.OBJ A.OBJ C.OBJ”这样的命令行来测试一下。

（文章中给出的原图是错误的，那是 Under the Hood 专栏九月讲拦截 API 时的一张图，译者注）

图 2 A.OBJ，B.OBJ，和 C.OBJ

记住了这两个规则，再理解链接器在 MS-DOS 和 16 位 Windows 上是如何工作的就容易了。但是，Win32 链接器又在刚才讲的内容上加了几条规则。首先就是包含$字符的节名规则。如果一个节名中包含$字符（例如，.idata$4），$字符及其后续字符在可执行文件中都被移除了。然而，在链接器修改这些名字之前，它是以$字符之前的字符作为节名来进行组合的。“$”字符之后的字符用以对 OBJ 文件的节进行排序以产生最终的可执行文件。这些节是按“$”字符之后的字符的字母顺序来存储的。例如，三个叫做 foo$c，foo$a，和 foo$b 的节在最终的可执行文件中将被组合成一个叫做 foo 的节。这个节中最前面的是 foo$a 中的数据，接着是 foo$b 中的数据，最后是 foo$c 中的数据。这种名字中包含$字符的节的自动组合有多种用途。后面我讨论导入函数时，你会看到一个例子。它也被用于创建 C++构造函数和析构函数静态初始化时所需的数据表。

除了$字符这个组合规则外，Win32 链接器还有一些其它规则。拥有代码属性的节有特别的优先权，它们被放在可执行文件的最前面。紧接着代码的是由在编译时未指定初始值的全局数据（例如，在 C++中 int i;语句声明的全局变量）组成的未初始化数据节。接下来是已初始化的数据（包含在.data 节中），以及链接器产生的数据节，例如.reloc 节。

未初始化的数据通常被编译器放在一个叫做.bss 的节中。现在很少能在可执行文件中看到.bss 节。Microsoft 链接器把.bss 节合并到了.data 节中，而.data 节是被编译器使用的主要的已初始化的数据节。但是请注意，这是针对希望运行于非 Posix 子系统，并且子系统版本大于

3.5 的可执行文件来说的。其它未初始化数据的节由链接器单独处理（也就是说，它们并未被合并）。

现在倒着来看可执行文件。如果在 OBJ 文件中有.debug 节，它被放在文件的最后。如果没有，链接器就把.reloc 节放在最后，因为在大多数情况下，Win32 加载器不需要读取重定位信息。减少需要读取的可执行文件的内容可以减少加载时间。关于重定位的内容将在后面讨论。

Win32 下另外一个不符合两个基本规则的是可移除节。这些节存在于 OBJ 文件中，但是链接器并不把它们复制到可执行文件中。这些节通常有 LINK_REMOVE 和 LINK_INFO 属性（见 WINNT.H 文件），并且被命名为.drectve。Microsoft 编译器之所以产生它们是为了向链接器传递信息。如果你看一下由 Visual C++ 编译后产生的 OBJ 文件，你会看到在.drectve 节中的数据就像下面这个样子：

```
-defaultlib:LIBC -defaultlib:OLDNAMES
```

如果你怀疑这些数据是传递到链接器的命令行参数，那你是正确的。当你使用 C++的 __declspec(dllexport)修饰符时，你会看到更多这方面的证据。例如：

```
void __declspec(dllexport) ExportMe( void ){...}
```

将导致.drectve 节包含：

```
-export:_ExportMe
```

如果你看一下 LINK 的命令行参数列表，绝对能看到-export 也在其中。

修正和重定位

为什么编译器不直接由源文件产生可执行文件，从而省略链接器呢？主要原因是，大部分程序并不是仅包含一个源文件。编译器专注于由单个的源文件产生等价的机器代码。由于一个源文件可能引用其它源文件中代码或数据，而编译器不能精确地产生调用那个函数或访问那个变量的正确代码。编译器惟一的选择就是在它产生的文件中包含描述外部代码或数据的额外信息。这个对外部代码或数据的描述就是修正（Fixup）。说得明白一点就是，编译器产生的访问外部函数或变量的代码是不正确的，必须在后面修正。

设想一下在 C++中调用一个名为 Foo 的函数：

```
//...

Foo();

//...
```

由 32 位 C++编译器产生的精确代码应该是：

```
E8 00 00 00 00
```

0xE8 是 CALL 指令的机器码。接下来的 DWORD 应该是 Foo 函数的偏移（相对于 CALL 指令）。很明显，Foo 函数相对于 CALL 指令的偏移不是 0 字节。如果你执行这段代码，它并不会按你原本期望的方式运行。产生的这段代码有错误，它需要被修正。

在上面的例子中，链接器需要把 CALL 指令的机器码后面的 DWORD 替换成 Foo 函数的正确地址。在可执行文件中，链接器将用 Foo 函数的相对地址改写这个 DWORD。链接器怎么知道它需要被修正呢？是修正记录（Fixup Record）告诉它的。链接器是怎么知道 Foo 函数的地址的？链接器知道可执行文件中的所有符号，因为正是它负责排列和组合可执行文件中的各个部分的。

现在来看一下修正记录。对于基于 Intel 的 OBJ 文件来说，通常遇到的修正记录有三种。第一种是 32 位相对修正，也就是 REL32 修正。（它相应于 WINNT.H 中的 IMAGE_REL_I386_REL32 这个#define 定义。）在上面的例子中，对 Foo 函数的调用应该有一个 REL32 类型的修正记录，并且这个记录中应该包含一个 DWORD 类型的偏移，链接器需要用合适的值覆盖这个偏移处的内容。如果你对由上面的代码产生的 OBJ 文件运行

```
DUMPBIN /RELOCATIONS
```

你会看到类似下面的内容：

```
Offset    Type          Applied To         Symbol Index     Symbol Name
--------  ------------  ---------------    --------------   -------------
00000004  REL32                 00000000                 7    _Foo
```

这个修正记录表示链接器需要计算函数 Foo 的相对偏移，并把它写到这个节内的偏移 0x00000004 处。这个修正记录仅在链接器创建可执行文件之前需要，之后它就被丢弃了，并不会出现在可执行文件中。既然这样，那为什么大部分可执行文件中还一个.reloc 节呢？这正是第二种类型的修正记录发挥作用的地方。设想以下程序：

```
int i;

int main()

{

i = 0x12345678;

}
```

Visual C++ 在可执行文件中为赋值语句生成以下指令：

```
MOV DWORD PTR [00406280],12345678
```

真正有趣的是指令中的[00406280]这一部分。它引用的是内存中的一个固定位置，并且假定包含变量 i 的那个 DWROD 在可执行文件的默认加载地址 0x400000 之上的 0x6280 字节处。现在，想象

一下，如果可执行文件不能被加载在默认加载地址，那会怎么样呢？假设 Win32 加载器把它加载到默认加载地址 2M 之上的地方（也就是，被加载在地址 0x600000 处）。如果是这样，指令中的[00406280]这一部分应该被调整为[00606280]。

这正是 DIR32（Direct32）大显身手的时候。它们能够表示哪里需要相对于实际地址（直接地址）做一些修改。这也意味着可执行文件的加载地址很重要。当创建可执行文件时，加载器利用 OBJ 文件中的 DIR32 类型的修正记录来创建.reloc 节。在 OBJ 文件上运行 DUMPBIN /RELOCATIONS，会出现类似下面的内容：

```
Offset    Type          Applied To         Symbol Index     Symbol Name
--------  ------------  ---------------    -------------    ------------
00000005  DIR32                 00000000                4    _i
```

这个修正记录表示链接器需要计算变量 i 的 32 位绝对地址，并且把它写到节内偏移 0x00000005 处。

可执行文件中的.reloc 节基本上就是可执行文件中的一系列地址，这些地址是默认地址与实际加载地址不同，需要被修正的地方。默认情况下，Win32 加载器并不需要.reloc 节。然而，当 Win32 加载器需要加载可执行文件到一个不同于其首选地址的地址时，.reloc 节允许那些使用代码与数据的绝对地址的指令获得更正。

第三种类型在 Intel 平台上的 OBJ 文件中比较常见，它就是 DIR32NB（Direct32，No Base），供调试信息使用。链接器的次要工作之一就是创建调试信息，这种信息中包含函数和变量名称以及它们的地址。由于只有链接器知道函数和变量到哪里结束，所以 DIR32NB 修正记录被用来指示调试信息中需要函数或变量地址的地方。DIR32 和 DIR32NB 的关键区别在于 DIR32NB 中的修正值不包含可执行文件的默认加载地址。

库

在一些情况下，把两个或多个 OBJ 文件组合成单个文件，然后送往链接器更有价值。这方面的经典例子就是 C++运行时库（RTL）。C++ RTL 是由许多源文件被编译之后，产生的所有 OBJ 文件被组合成的一个库。对 Visual C++ 来说，标准的单线程静态运行时库是 LIBC.LIB。RTL 库还有其它版本，诸如调试版（例如，LIBCD.LIB）和多线程版（LIBCMT.LIB）。

库文件通常以.LIB 为扩展名。它们由库文件头和包含在 OBJ 中的原始数据组成。库文件头用于告诉链接器哪个符号（函数和变量）可以在它里面的 OBJ 文件中找到，同时也指明这个符号存在于哪个 OBJ 中。你可以通过使用 DUMPBIN /LINKERMEMBER 选项来观察库文件的内容。不管是什么原因，你会发现如果你指定选项:1 或:2，DUMPBIN 的输出会更具可读性。例如，用 Visual C++ 5.0 的 PENTER.LIB 文件，使用以下命令行

```
"DUMPBIN /LINKERMEMBER:1 PENTER.LIB"
```

它的部分输出结果如下：

```
6 public symbols

180 _DumpCAP@0

180 _StartCAP@0

180 _StopCAP@0

180 _VERSION

180 __mcount

180 __penter
```

每个符号前面的 180 表示那个符号（例如，_DumpCAP@0）可以在从库文件开头算起的 0x180 字节处的 OBJ 文件中找到。如你所见，PENTER.LIB 里面只有一个 OBJ 文件。更复杂的 LIB 文件有多个 OBJ 文件，因此符号前面的偏移会有所不同。

不像在命令行上传递 OBJ 文件那样，链接器并非必须把一个库中的所有 OBJ 文件都链接到最终的可执行文件中。事实上，正好相反。链接器不包含库中 OBJ 文件中的任何代码或数据，除非从那个 OBJ 文件中至少引用了一个符号。换句话说，链接器命令行中明确指定的 OBJ 文件总是被链接到最终的可执行文件中，而 LIB 文件中的 OBJ 文件只有在被引用过时才被链接进去。

库中的符号可以以三种方式被引用。首先，直接访问明确指定的 OBJ 文件中的符号。例如，如果在我的一个源文件中调用 C++的 printf 函数，在我的 OBJ 文件中将会产生一个引用（和修正）。当创建可执行文件时，链接器会搜索它的 LIB 文件以查找包含 printf 代码的 OBJ 文件，并且链接相应的 OBJ 文件。

第二，可能存在一个间接引用。“间接”意味着通过第一种方法包含的 OBJ 引用了库中另外一个 OBJ 文件中的符号。而这第二个 OBJ 可能又引用了库中第三个 OBJ 文件中的符号。链接器的艰苦工作之一就是，即使符号通过 49 级间接引用，它也必须跟踪并且包含引用的每一个 OBJ 文件。

当查找符号时，链接器按它的命令行上遇到的LIB文件的顺序进行搜索。然而，一旦在某个库中找到一个符号，那个库就变成了首选库，首先在它里面搜索所有其它符号。一旦某个符号在这个库中找不到，这个库就失去了它的首选地位。此时，链接器搜索它的列表中的下一个库。（要获得更详细的技术信息，请参考Microsoft知识库文章Q31998，网址为 http://www.microsoft.com/kb。）

现在，我们把目光转向导入库。在结构上，导入库与普通库并无区别。在解析符号（resolve symbol）时，链接器并不知道导入库与普通库的区别。它们的关键区别在于，导入库中的 OBJ 并没有相应的编译单元（例如，并没有相应的源文件）。实际上，是链接器自己基于正在创建的可执行文件所导出的符号产生导入库的。换句话说，链接器在创建可执行文件的导出表的同时也创建了相应的导入库来引用这些符号。这引出了下一个主题—导入表。

创建导入表

Win32 最基础的特性之一就是能够从其它可执行文件中导入函数。关于导入的 DLL 和函数的所有信息都存在于可执行文件中的一个称为导入表（Import Table）的表中。当它单独成节时，这个节的名字叫.idata。

导入对 Win32 可执行文件来说是至关重要的，因此，如果说链接器并不知道导入表方面的专业知识，那真是令人难以置信。但事实的确如此。换句话说，链接器并不知道，也不关心你调用的函数是在另外的 DLL 中，还是在这个文件中。链接器在这一点表现得特别聪明。它仅仅简单地依照上面描述的节组合规则和符号解析规则就创建了导入表，看起来好像它并没有意识到这个表的重要性。

让我们看一下一些导入库的片段，看一看链接器是怎样出色地完成这个任务的。图 2 是对 USER32.LIB 导入库运行 DUMPBIN 时的部分输出结果。假设你调用了 ActivateKeyboardLayout 这个 API。在你的 OBJ 文件中可以找到一个_ActivateKeyboardLayout@8 的修正记录。从 USER32.LIB 的头部，链接器知道这个函数可以在文件偏移 0xEA14 处的 OBJ 中找到。因此，链接器忠实地在最终的可执行文件中包含了上述指定的内容（见图 3）。

图 3　导入表

```
1121 public symbols

EA14 _ActivateKeyboardLayout@8

...

Archive member name at EA14: USER32.dll/

...

SECTION HEADER #2

.text name

RAW DATA #2

00000000 FF 25 00 00 00 00 .%....

...

SECTION HEADER #4

.idata$5 name

RAW DATA #4

00000000 00 00 00 00 ....
```

```
...

SECTION HEADER #5

.idata$4 name

RAW DATA #5

00000000 00 00 00 00 ....

...

SECTION HEADER #6

.idata$6 name

RAW DATA #6

00000000 00 00 41 63 74 69 76 61 | 74 65 4B 65 79 62 6F 61 ..Activa|teKeyboa

00000010 72 64 4C 61 79 6F 75 74 | 00 00 rdLayout|..

...

COFF SYMBOL TABLE

...

003 00000000 SECT2 notype () External | _ActivateKeyboardLayout@8
```

从图 3 可以看出，牵涉到了 OBJ 文件中的许多节，包括.text，.idata$5，.idata$4 和.idata$6。在.text 节中是一个 JMP 指令（机器码 0xFF 0x25）。从图 3 最后的 COFF 符号表可以看出，_ActivateKeyboardLayout@8 被解析到了.text 节的这个 JMP 指令上。因此，链接器把你对 ActivateKeyboardLayout 的调用转换成了对导入库的 OBJ 中的.text 节的 JMP 指令的调用。

在可执行文件中，链接器把所有的.idata$XXX 节组合成单个的.idata 节。现在回忆一下链接器组合节名中带有$字符的节时要遵守的规则。如果从 USER32.LIB 导入的还有其它函数，它们的.idata$4，.idata$5 和.idata$6 这些节也要放入其中。结果就形成了所有的.idata$4 节组成了一个数组，所有的.idata$5 节组成了另一个数组。如果你熟悉“导入地址表（Import Address Table，IAT）”的话，这实际上就是它的创建过程。

最后，注意节.idata$6 的原始数据中包含了字符串“ActivateKeyboardLayout”。这就是导入地址表中有被导入函数的名字的原因。重要的一点是，对链接器来说，创建导入表并非难事。它只是依照我前面描述的规则，做它自己的工作而已。

创建导出表

除了为可执行文件创建导入表外，链接器还负责创建导出表。它的工作难易参半。在第一遍中，链接器的任务是收集关于所有导出符号的信息并创建导出函数表。在此期间，链接器创建导出表，并且把它写入到 OBJ 文件中一个叫.edata 的节中。这个 OBJ 文件除了扩展名是.EXP 而不是.OBJ 外，其它地方都符合标准。你使用 DUMPBIN 检查一下这些 EXP 文件的内容就知道了。

在第二遍中，链接器的工作就很轻松了。它只是把 EXP 文件当作普通的 OBJ 文件来对待。这也意味着 OBJ 文件中的节.edata 应该被包含到可执行文件中。如果你在可执行文件看到.edata 节，它就是导出表。但是，近来很少能找到.edata 节。看起来好像是如果可执行文件使用 Win32 控制台或 GUI 子系统，链接器就自动合并.edata 节和.rdata 节，如果其中一个存在的话。

总结

很明显链接器做的工作要比我这里描述的多得多。例如，生成某种类型的调试信息（例如，CodeView 信息）是链接器全部工作中的重要部分。但是，生成调试信息并不是必须的，因此我没有花什么时间来描述它。同样，链接器应该能够创建 MAP 文件，这种文件包含可执行文件中公共符号的列表，但是它同样不是必须的功能。

虽然我提供了许多复杂的背景知识，但是链接器的核心是简单地把多个编译单元组合成可执行文件。第一个基本功能是组合节；第二个功能是解析节之间的相互引用（修正）。联系一下诸如导出表之类系统特定的数据结构方面的知识，你就基本上掌握了这个功能强大且重要的工具。

译者：SmartTech　电子信箱：zhzhtst@163.com