帮 C/C++ 程序员彻底了解链接器

2015/12/18 · C/C++, 开发 · 1 评论 · C语言, 链接器

英文出处：David Drysdale。欢迎加入翻译组。

本文旨在帮助 C/C++ 程序员们了解链接器到底完成了些什么工作。多年来，我给许多同事解释过这一原理，因此我觉得是时候把它写下来了，这样不仅可以供更多人学习，也省去我一遍遍讲解。

[2009年3月更新，内容包括：增加了 Windows 系统中链接过程可能遇到的特殊问题，以及对某条定义规则的澄清。]

促使我写下这篇文章的起因是某次我帮人解决了一个链接错误，具体是这样的：

```
g++ -o test1 test1a.o test1b.o
test1a.o(.text+0x18): In function `main':
: undefined reference to `findmax(int, int)'
collect2: ld returned 1 exit status
```

如果你认为这是“几乎可以肯定是因为漏写了 extern “C””，那你很可能已经掌握了本文的全部内容。

目录

各部分的命名：看看 C 文件中都包含了哪些内容

本章，我们将快速回忆一下 C 文件中包含的几大部分。如果你认为自己已经完全明白下文示例程序中的内容，那么你可以跳过本章，直接阅读下一章。

我们首先要弄清的是声明和定义的区别。定义（definition）是指建立某个名字与该名字的实现之间的关联，这里的“实现”可以是数据，也可以是代码：

- 变量的定义，使得编译器为这个变量分配一块内存空间，并且还可能为这块内存空间填上特定的值
- 函数的定义，使得编译器为这个函数产生一段代码

声明（declaration）是告诉 C 编译器，我们在程序的别处——很可能在别的 C 文件中——以某个名字定义了某些内容（注意：有些时候，定义也被认为是声明，即在定义的同时，也在此处进行了声明）。

对于变量而言，定义可以分为两种：

- 全局变量（global variables）：其生命周期存在于整个程序中（即静态范围（static extent）），可以被不同的模块访问
- 局部变量（local variables）：生命周期只存在于函数的执行过程中（即局部范围（local extent）），只能在函数内部访问

澄清一点，我们这里所说的“可访问（accessible）”，是指“可以使用该变量在定义时所起的名字”。

以下是几个不太直观的特殊情况：

- 用 static 修饰的局部变量实际上是全局变量，因为虽然它们仅在某个函数中可见，但其生命周期存在于整个程序中
- 同样，用 static 修饰的全局变量也被认为是全局的，尽管它们只能由它们所在的文件内的函数访问

当我们谈及“static”关键字时，值得一提的是，如果某个函数（function）用 static 修饰，则该函数可被调用的范围就变窄了（尤其是在同一个文件中）。

无论定义全局变量还是局部变量，我们可以分辨出一个变量是已初始化的还是未初始化的，分辨方法就是这个变量所占据的内存空间是否预先填上了某个特殊值。

最后要提的一点是：我们可以将数据存于用 malloc 或 new 动态分配的内存中。这部分内存空间没法通过变量名来访问，因此我们使用指针（pointer）来代替—指针也是一种有名字的变量，它用来保存无名动态内存空间的地址。这部分内存空间最终可以通过使用 free 和 delete 来回收，这也是为什么将这部分空间称为“动态区域”（dynamic extent）。

让我们来总结一下吧：

	代码	数据				
		全局		局部		动态
		已初始化	未初始化	已初始化	未初始化	
声明	int fn(int x);	extern int x;	extern int x;	N/A	N/A	N/A
定义	int fn(int x) { … }	int x = 1; (作用域：文件)	int x; (作用域：文件)	int x = 1; (作用域：函数)	int x; (作用域：函数)	(int* p = malloc(sizeof(int));)

以下是一个示例程序，也许是一种更简便的记忆方法：

```c
/* 这是一个未初始化的全局变量的定义 */
int x_global_uninit;

/* 这是一个初始化的全局变量的定义 */
int x_global_init = 1;

/* 这是一个未初始化的全局变量的定义，尽管该变量只能在当前 C文件中访问 */
static int y_global_uninit;

/* 这是一个初始化的全局变量的定义，尽管该变量只能在当前 C文件中访问 */
static int y_global_init = 2;

/* 这是一个存在于程序别处的某个全局变量的声明 */
extern int z_global;

/* 这是一个存在于程序别处的某个函数的声明（如果你愿意，你可以在语句前加上 "extern"关键字，但没有这个必要） */
int fn_a( int x, int y);

/* 这是一个函数的定义，但由于这个函数前加了 static限定，因此它只能在当前 C文件内使用 */
static int fn_b(int x)
{
    return x +1;
}

/* 这是一个函数的定义，函数参数可以认为是局部变量 */
int fn_c( int x_local)
{
    /* 这是一个未初始化的局部变量的定义 */
    int y_local_uninit ;
    /* 这是一个初始化的局部变量的定义 */
    int y_local_init = 3 ;

    /* 以下代码通过局部变量、全局变量和函数的名字来使用它们 */
    x_global_uninit = fn_a (x_local, x_global_init);
    y_local_uninit = fn_a (x_local, y_local_init);
    y_local_uninit += fn_b (z_global);
    return (x_global_uninit + y_local_uninit);
}
```

C 编译器都做了些什么

C 编译器的任务是把我们人类通常能够读懂的文本形式的 C 语言文件转化成计算机能明白的内容。我们将编译器输出的文件称为目标文件（object file）。在UNIX平台上，这些目标文件的后缀名通常为.o，在Windows平台上的后缀名为.obj。目标文件本质上包含了以下两项内容：

- 代码：对应着 C 文件中函数的定义（definitions）
- 数据：对应着 C 文件中全局变量的定义（definitions）（对于一个已初始化的全局变量，它的初值也存于目标文件中）。

以上两项内容的实例都有相应的名字与之相关联——即定义时，为变量或函数所起的名字。

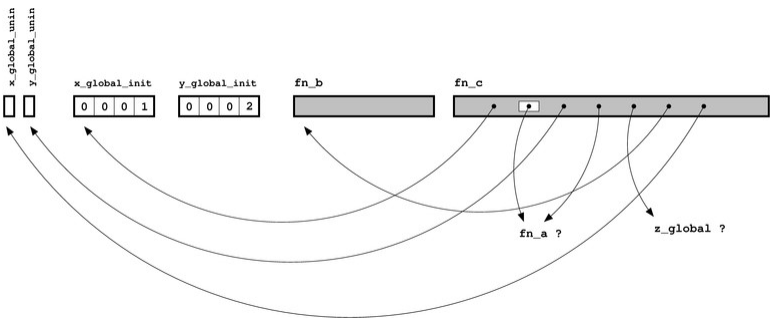
目标代码（object code）是指将程序员写成的 C 代码——所有的那些if，while，甚至goto都包括在内—经过适当编码生成对应的机器码序列。所有的这些指令都用于处理某些信息，而这些信息都得有地方存放才行—这就是变量的作用。另外，我们可以在代码中引用另一段代码—说得具体些，就是去调用程序中其它的 C 函数。

无论一段代码在何处使用某个变量或者调用某个函数，编译器都只允许使用已经声明（declaration）过的变量和函数——这样看来，声明其实就是程序员对编译器的承诺：向它确保这个变量或函数已经在程序中的别处定义过了。

链接器（linker）的作用则是兑现这一承诺，但反过来考虑，编译器又如何在产生目标文件的过程中兑现这些承诺呢？

大致说来，编译器会留个空白（blank），这个“空白”（我们也称之为“引用”（reference））拥有与之相关联的一个名字，但该名字对应的值还尚未可知。

在熟悉了以上知识后，我们大致可以勾画出上一节示例代码所对应目标文件的样子了：



目前为止，我们仅仅只从宏观的角度进行讨论，因此，接下来我们很有必要研究一下之前介绍的理论在实际中都是怎么工作的。这里我们需要用到一个很关键的工具，即命令：nm，这是一条UNIX平台上使用的命令，它可以提供目标文件的符号（symbols）信息。在Windows平台上，与其大致等价的是带 `/symbols` 选项的 dumpbin 命令；当然，你也可以选择安装 Windows 版的 GNU binutils 工具包，其中包含了 `nm.exe`。

我们来看看运行nm命令后，上文的 C 代码所产生的目标文件是什么结构：

```
c_parts.o 中的符号如下：

Name                  Value   Class        Type         Size     Line  Section

fn_a                  |       |   U  |           NOTYPE|        |     |*UND*
z_global              |       |   U  |           NOTYPE|        |     |*UND*
fn_b                  |00000000|   t  |             FUNC|00000009|     |.text
x_global_init         |00000000|   D  |           OBJECT|00000004|     |.data
y_global_uninit       |00000000|   b  |           OBJECT|00000004|     |.bss
x_global_uninit       |00000004|   C  |           OBJECT|00000004|     |*COM*
y_global_init         |00000004|   d  |           OBJECT|00000004|     |.data
fn_c                  |00000009|   T  |             FUNC|00000055|     |.text
```

不同平台的输出内容可能会有些许不同（你可以用 man 命令来查看帮助页面，从中获取某个特定版本更多的相关信息），但它们都会提供这两个关键信息：每个符号的类型，以及该符号的大小（如果该符号是有效的）。符号的类型包括以下几种（译者注[1]）：

- U: 该类型表示未定义的引用（undefined reference），即我们前文所提及的“空白”（blanks）。对于示例中的目标文件，共有两个未定义类型：“fn_a” 和 “z_global”。（有些 nm 的版本还可能包括 section（译注：即宏汇编中的区，后文直接使用section而不另作中文翻译）的名字，section的内容通常为 *UND* 或 UNDEF）
- t/T: 该类型指明了代码定义的位置。t 和 T 用于区分该函数是定义在文件内部（t）还是定义在文件外部（T）——例如，用于表明某函数是否声明为 static。同样的，有些系统包括 section ，内容形如.text
- d/D: 该类型表明当前变量是一个已初始化的变量，d 指明这是一个局部变量，D 则表示全局变量。如果存在 section ，则内容形如 .data
- b/B: 对于非初始化的变量，我们用 b 来表示该变量是静态（static）或是局部的（local），否则，用 B 或 C 来表示。这时 section 的内容可能为.bss 或者 *COM*

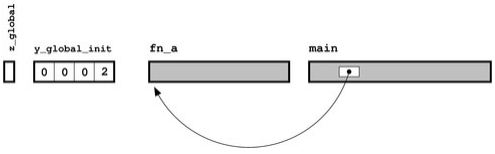
我们也很可能会看到一些不属于原始 C 文件的符号，我们可以忽略它们，因为这一般是由编译器“邪恶”的内部机制导致的，这是为了让你的程序链接在一起而额外产生的内容。

链接器都做了些什么（1）

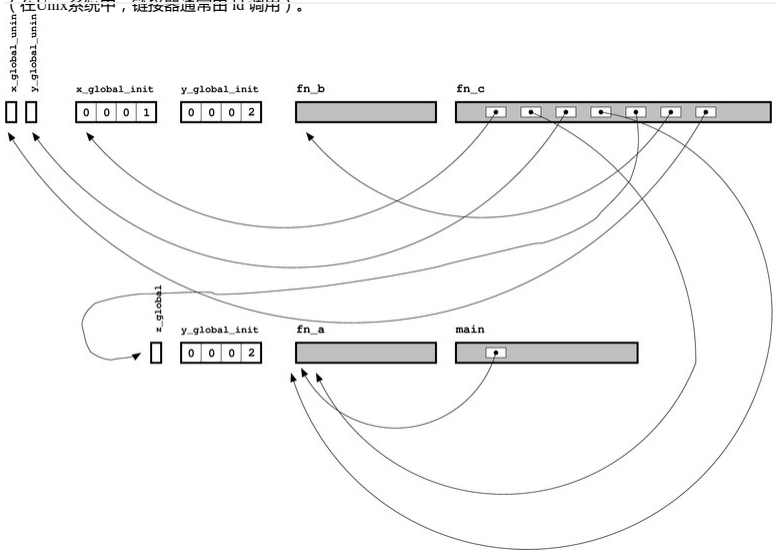
我们在上文提到过，一个函数或变量的声明，实际上就是在向 C 编译器承诺：这个函数或变已在程序中的别处定义了，而链接器的工作就是兑现这一承诺。根据上文提供的目标文件结构图，现在，我们可以开始着手“填充图中的空白”了。

为了更好地进行说明，我们给之前的 C 文件添个“伴儿”：

```c
/* 初始化的全局变量 */
int z_global = 11;
/* 另一个命名为y_global_init的全局变量 , 但它们都是static的 */
static int y_global_init = 2;
/* 声明另一个全局变量 */
extern int x_global_init;

int fn_a(int x, int y)
{
  return(x+y);
}

int main(int argc, char *argv[])
{
  const char *message = "Hello, world";

  return fn_a(11,12);
}
```



有了这两张图，我们现在可以将这图中所有的节点都互相连通了（如果不能连通，那么链接器在链接过程中就会抛出错误信息）。一切各就各位，如下图所示，链接器可以将空白都填补上了（在Unix系统中，链接器通常由 ld 调用）。



至于目标文件，我们可以使用 nm 命令来检查生成的可执行文件：

```
samples1.exe中的符号列表：

Name                  Value   Class        Type         Size     Line  Section

_Jv_RegisterClasses |        |   w  |            NOTYPE|        |     |*UND*
__gmon_start__      |        |   w  |            NOTYPE|        |     |*UND*
__libc_start_main@@GLIBC_2.0|        |   U  |              FUNC|000001ad|     |*UND*
_init               |08048254|   T  |              FUNC|        |     |.init
_start              |080482c0|   T  |              FUNC|        |     |.text
__do_global_dtors_aux|080482f0|   t  |              FUNC|        |     |.text
frame_dummy         |08048320|   t  |              FUNC|        |     |.text
fn_b                |08048348|   t  |              FUNC|00000009|     |.text
fn_c                |08048351|   T  |              FUNC|00000055|     |.text
fn_a                |080483a8|   T  |              FUNC|0000000b|     |.text
main                |080483b3|   T  |              FUNC|0000002c|     |.text
__libc_csu_fini     |080483e0|   T  |              FUNC|00000005|     |.text
__libc_csu_init     |080483f0|   T  |              FUNC|00000055|     |.text
__do_global_ctors_aux|08048450|   t  |              FUNC|        |     |.text
_fini               |08048478|   T  |              FUNC|        |     |.fini
_fp_hw              |08048494|   R  |            OBJECT|00000004|     |.rodata
_IO_stdin_used      |08048498|   R  |            OBJECT|00000004|     |.rodata
__FRAME_END__       |080484ac|   r  |            OBJECT|        |     |.eh_frame
__CTOR_LIST__       |080494b0|   d  |            OBJECT|        |     |.ctors
__init_array_end    |080494b0|   d  |            NOTYPE|        |     |.ctors
__init_array_start  |080494b0|   d  |            NOTYPE|        |     |.ctors
__CTOR_END__        |080494b4|   d  |            OBJECT|        |     |.ctors
__DTOR_LIST__       |080494b8|   d  |            OBJECT|        |     |.dtors
__DTOR_END__        |080494bc|   d  |            OBJECT|        |     |.dtors
__JCR_END__         |080494c0|   d  |            OBJECT|        |     |.jcr
__JCR_LIST__        |080494c0|   d  |            OBJECT|        |     |.jcr
_DYNAMIC            |080494c4|   d  |            OBJECT|        |     |.dynamic
_GLOBAL_OFFSET_TABLE_|08049598|   d  |            OBJECT|        |     |.got.plt
__data_start        |080495ac|   D  |            NOTYPE|        |     |.data
data_start          |080495ac|   W  |            NOTYPE|        |     |.data
__dso_handle        |080495b0|   D  |            OBJECT|        |     |.data
p.5826              |080495b4|   d  |            OBJECT|        |     |.data
x_global_init       |080495b8|   D  |            OBJECT|00000004|     |.data
y_global_init       |080495bc|   d  |            OBJECT|00000004|     |.data
z_global            |080495c0|   D  |            OBJECT|00000004|     |.data
y_global_init       |080495c4|   d  |            OBJECT|00000004|     |.data
__bss_start         |080495c8|   A  |            NOTYPE|        |     |*ABS*
_edata              |080495c8|   A  |            NOTYPE|        |     |*ABS*
completed.5828      |080495c8|   b  |            OBJECT|00000001|     |.bss
y_global_uninit     |080495cc|   b  |            OBJECT|00000004|     |.bss
x_global_uninit     |080495d0|   B  |            OBJECT|00000004|     |.bss
_end                |080495d4|   A  |            NOTYPE|        |     |*ABS*
```

这个表格包含了两个目标文件中的所有符号，显然，之前所有“未定义的引用”都已消失。同时，所有符号都按类型重新排了序，还加入了一些额外的信息以便于操作系统更好地对可执行程序实行统一处理。

输出内容中还有相当多复杂的细节，看上去很混乱，但你只要把以下划线开头的内容都过滤掉，整个结构看上去就简单多了。

重复的符号

上文提到，当链接器试图为某个符号产生连接引用时却找不到这个符号的定义，链接器将抛出错误信息。那么，在链接阶段，如果同一个符号定义了两次又该如何处理呢？

在C++中这种情况很容易处理，因为语言本身定义了一种称为一次定义法则（one definition rule）的约束，即链接阶段，一个符号有且只能定义一次（参见 C++ 标准第3.2章节，这一章节还提及了后文中我们将讲解的一些异常信息）。

对于 C 语言而言，事情就稍稍复杂一些了。C语言明确说明了，对于任何的函数或者已经初始化的全局变量，都有且只能有一次定义，但未初始化的全局变量的定义可以看成是一种临时性定义（a tentative definition）。C 语言允许（至少不禁止）同一个符号在不同的源文件中进行临时性定义。

然而，链接器还得对付除 C/C++ 以外的其它语言，对于那些语言来说，“一次定义法则”并非总是适用。例如，以 Fortran 语言的正态模式（normal model）为例，实际应用中，每个全局变量在其被引用的任何文件中都存在一个复本。此时，链接器需要从多个复本中选择一个（如果大小不同，就选最大的那个），并将剩余复本丢弃。（这种模式有时又称为链接时的“通用模式（common model）”，前头需要加上Fortran关键字： COMMON ）

因此，UNIX 系统上的链接器不会为符号的重复定义——或者说不会为未初始化全局变量的重复符号——抛出任何信息，这种情况相当正常（有时，我们将这种情况称为链接时的“松引用/定义模式（relaxed ref/def mode）”模式）。如果你为此感到苦恼（你也完全有理由苦恼），那么你可以查看你所使用的编译器和链接器的相关文档，里面通常会提供一个 –work-properly 选项，用于“收紧”链接器的检测规则。例如，GNU 工具包里提供了 -fno-common 选项，可以让编译器强行将未初始化变量存放于 BSS 段，而不是存于 common 段。

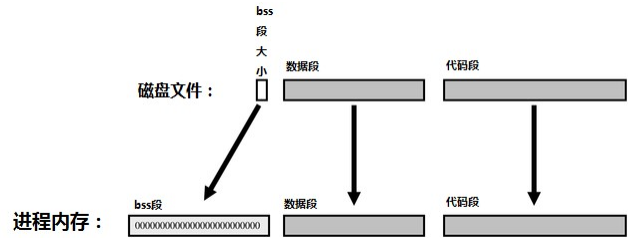
目前为止，链接器产生了可执行文件，文件中所有符号都与其合适的定义相关联。接下来，我们要休息一会儿，插播一则小知识：当我们运行这个程序时，操作系统都做了些什么？

程序的运行显然需要执行机器代码，因此操作系统无疑需要把硬盘上的可执行文件转换成机器码，并载入内存，这样CPU才能从中读取信息。程序所占用的这块内存，我们称之为代码段（code segment），或者文本段（text segment）.

没有数据，再好的代码也出不来——因此，所有全局变量也得一并载入内存。不过已初始化变量和未初始化变量有些不同。初始化变量已经提前赋予了某个特定的初值，这些值同时保存于目标文件和可执行文件中。当程序开始运行时，操作系统将这些值拷贝至内存中一块名为数据段（data segment）的区域。

对未初始化变量，操作系统假设其初值均为0， 因此没有必要对这些值进行拷贝，操作系统保留一部分全为0内存空间，我们称其为 bss 段（bss segment）。

这就意味着可执行文件可以节省这部分存储空间：初始化变量的初始值必须保存于文件中，但对于未初始化变量我们只需要计算出它们占用的空间大小即可。

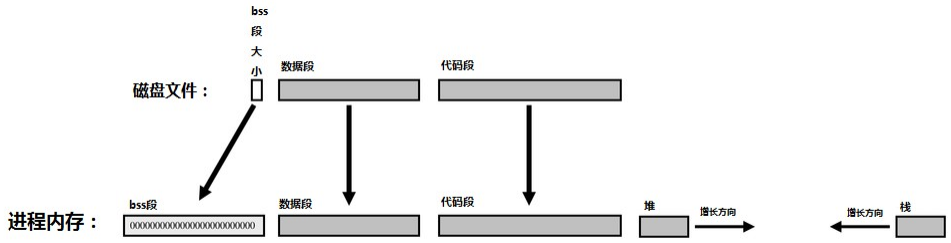


你可能已经注意到目前我们关于目标文件和链接器的所有讨论都只围绕着全局变量，完全没有作何关于上文提及的局部变量和动态分配内存的介绍。

事实上，这类数据的处理完全无需链接器介入，因为它们的生命周期只存在于程序运行之时——这与链接器进行链接操作还离了十万八千里呢。不过，从文章完整性的角度来考虑，我们还是快速过一下这部分知识点吧：

- 局部变量被存于内存的“栈”区（stack），栈区的大小随着不同函数的调用和返回而动态地增长或减小。
- 动态分配的内存而处于另一块空间，我们称之为“堆”（heap），malloc 函数负责跟踪这块空间里还有哪些部分是可用的。

我们将这部分内存空间也添加上，这样，我们就得到了一张完整的程序运行时的内存空间示意图。由于堆和栈在程序运行过程中都会动态地改变大小，通常的处理方式是让栈从一个方向向另一个方向增长，而堆则从另一端增长。也就是说，当二者相遇之时就是程序内存耗尽之日了（到那时，内存空间就被占用得满满当当啦！）。



链接器都做了些什么（2）

现在我们已经对链接器的基础知识有了一定的了解，接下来我们将开始刨根纠底，挖出它更为复杂的细节——大体上，我们会按照链接器每个特性加入的时间顺序来一一介绍。

影响链接器特性的最主要的一个现象是：如果有很多不同的程序都需要做一些相同的操作（例如将输出打印到屏幕上，从硬盘读取文件等），那么显然，一种合理的做法是将这些功能编写成通用的代码，供所有不同的程序使用。

在每个程序的链接阶段去链接相同的目标文件这种方法显然完全可行，但是，想象这么一种方法：把所有相关的目标文件集合都统一存放在一个方便访问的地方——这样我们在使用的时候会觉得生活更加简单美好了~我们将其称为“库”（library）。

（未谈及的技术问题：本节不涉及链接器“重定位(relocation)”这一重要特性的介绍。不同的程序大小也不同，因此，当动态库在不同程序中使用时，将被映射成不同的地址空间，也就是说库中所有的函数和变量在不同的程序中有不同的地址。如果所有访问该地址之处，都使用相对地址（如“向后偏移1020字节”）而不是绝对地址（固定的某个地址值，如 0x102218BF），那这也不是个事儿，可现在我们要考虑的问题在于，现实并不总这么尽如人意，当这种情况出现时，所有绝对地址都必须加上一个合适的偏移量——这就是重定位的概念。由于这一概念对C/C++程序员来说几乎是完全透明的，并且链接中报的错误也几乎不可能由重定位问题导致，因此下文将不会对此赘述。）

静态库

静态库（static library）是“库”最典型的使用方式。前文中提到使用重用目标文件的方法来共享代码，事实上，静态库本质上并不比这复杂多少。

在UNIX系统中，一般使用 ar 命令生成静态库，并以 .a 作为文件扩展名，”lib” 作为文件名前缀，链接时，使用”-l”选项，其后跟着库的名称，用于告诉链接器链接时所需要的库，这时无需加前缀和扩展名（例如，对于名为”libfred.a”的静态库，传递给链接器参数为”-lfred”）。

（过去，为了生成静态库文件，我们还需要使用另一个名为 ranlib 的工具，该工具的作用是在库的起始处建立符号索引信息。如今这一功能已经被整合到 ar 命令中了。）

在Windows平台上，静态库的扩展名为 .LIB，可用 .LIB 工具生成，但由于“导入库”（它只包含了DLL中所需要的基本信息列表，具体介绍可见下文 Windows DLLs也同样使用 .LIB 作为扩展名，因此二者容易产生混淆。

链接器在将所有目标文件集链接到一起的过程中，会为所有当前未解决的符号构建一张“未解决符号表”。当所有显示指定的目标文件都处理完毕时，链接器将到“库”中去寻找“未解决符号表”中剩余的符号。如果未解决的符号在库里其中一个目标文件中定义，那么这个文件将加入链接过程，这跟用户通过命令行显示指定所需目标文件的效果是一样一样的，然后链接器继续工作。

我们需要注意从库中导入文件的粒度问题：如果某个特定符号的定义是必须的，那么包含该符号定义的整个目标文件都要被导入。这就意味着“未解决符号表”会出现长短往复的变化：在新导入的目标文件解决了某个未定义引用的同时，该目标文件自身也包含着其他未定义的引用，这就要求链接器将其加入“符号表”中继续解决。

另一个需要注意的重要细节是库的处理顺序。链接器按命令行从左到右的顺序进行处理，只有前一个库处理结束了，才会继续处理下一个库。换句话说，如果后一个库中导入的目标文件依赖于前一个库中的某个符号，那么链接器将无法进行自动关联。

下面这个例子应该可以帮助大家更好的理解本节内容。我们假设有下列几个目标文件，并且通过命令行向链接器传入：a.o, b.o, -lx, -ly.

	a.o	b.o	libx.a	liby.a

目标文件	a.o	b.o	x1.o	x2.o	x3.o	y1.o	y2.o	y3.o
定义的变量	a1, a2, a3	b1, b2	x11, x12, x13	x21, x22, x23	x31, x32	y11, y12	y21, y22	y31, y32
未定义的引用	b2, x12	a3, y22	x23, y12	y11		y21		x31

当链接器开始链接过程时，可以解决 a.o 目标文件中的未定义引用 b2，以及 b.o 中的 a3，但 x12 和 y22 仍然处于未定义状态。此时，链接器在第一个库 libx.a 中查找这两个符号，并发现只要将 x1.o 导入，就可以解决 x12 这一未定义引用，但导入 x1.o 同时也不得不引入新的未定义引用：x23 和 y12，因此，此时未定义引用的列表里包含了三个符号：y22, x23, y12。

因为此时链接器还在处理 libx.a，所以就优先处理 x23 了，即从 libx.a 中导入 x2.o，然而这又引入了新的未定义引用——如今列表变成了y22, y12, y11，这几个引用都不在在 libx.a 中，因此链接器开始继续处理下一个库：liby.a。

接下来，同样的处理过程也发生在 liby.a 中，链接器导入 y1.o 和 y2.o：链接器在导入 y1.o 后首先将 y21 加入未定义引用列表中，不过由于 y22 的存在，y2.o 无论如何都必须导入，因此问题就此轻松搞定了。整个复杂的处理过程，目的在于解决所有未定义引用，但只需要将库中部分目标文件加入到最终的可执行文件中，避免导入库中所有目标文件。

需要注意的一点是，如果我们假设 b.o 中也使用了 y32 ，那么情况就有些许不同了。这种情况下，对 libx.a 的链接处理不变，但处理 liby.a 时，y3.o 也将被导入，这将带来一个新问题：又加入了一个新的未定义引用 x31 ，链接失败了——原因在于，链接器已经处理完了 libx.a， 但由于 x3.o 未导入，链接器无法查找到 x31 的定义。

（补充说明：这个例子展示了 libx.a 和 liby.a 这两个库之间出现循环依赖的问题，这是个典型的错误，尤其当它出现Windows系统上时）

共享库

对于像 C 标准库（libc）这类常用库而言，如果用静态库来实现存在一个明显的缺点，即所有可执行程序对同一段代码都有一份拷贝。如果每个可执行文件中都存有一份如 printf, fopen 这类常用函数的拷贝，那将占用相当大的一部分硬盘空间，这完全没有必要。

另一个不那么明显的缺点则是，一旦程序完成静态链接后，代码就永久保持不变了，如果万一有人发现并修复了 printf 中的某个bug，那么所有使用了printf的程序都不得不重新链接才能应用上这个修复。

为了避开所有这些问题，我们引入了共享库（shared libraries），其扩展名在 Unix 系统中为 .so，在 Windows 系统中为 .dll，在Mac OS X系统中为 .dylib。对于这类库而言，通常，链接器没有必要将所有的符号都关联起来，而是贴上一个“我欠你（IOU）”这样的标签，直到程序真正运行时才对贴有这样标签的内容进行处理。

这可以归结为：当链接器发现某个符号的定义在共享库中，那么它不会把这个符号的定义加入到最终生成的可执行文件中，而是将该符号与其对应的库名称记录下来（保存在可执行文件中）。

当程序开始运行时，操作系统会及时地将剩余的链接工作做完以保证程序的正常运行。在 main 函数开始之前，有一个小型的链接器（通常名为 ld.so，译者注[2]）将负责检查贴过标签的内容，并完成链接的最后一个步骤：导入库里的代码，并将所有符号都关联在一起。

也就是说，任何一个可执行文件都不包含 printf 函数的代码拷贝，如果 printf 修复了某些 bug，发布了新版本，那么只需要将 libc.so 替换成新版本即可，程序下次运行时，自然会载入更新后的代码。

另外，共享库与静态库还存在一个巨大的差异，即链接的粒度（the granularity of the link）。如果程序中只引用了共享库里的某个符号（比如，只使用了 libc.so 库中的 printf），那么整个共享库都将映射到程序地址空间中，这与静态库的行为完全不同，静态库中只会导入与该符号相关的那个目标文件。

换句话说，共享库在链接器链接结束后，可以自行解决同一个库内不同对象（objects）间符号的相互引用的问题（ar 命令与此不同，对于一个库它会产生多个目标文件）。这里我们可以再一次使用 nm 命令来弄清静态库和共享库的区别：对于前文给出的目标文件和库的例子，对于同一个库，nm 命令只能分别显示每个目标文件的符号清单，但如果将 liby.so 变成共享库，我们只会看到一个未定义符号 x31。同样，上一节提到的由静态库处理顺序引起的问题，将不会共享库中出现：即使 b.o （译者注[3]）中使用了 y32，也不会有任何问题，因为 y3.o 和 x3.o 都已全部导入了。

> 顺便推荐另一个超好用的命令： ldd，该命令是Unix平台上用于显示一个可执行程序（或一个共享库）依赖的共享库，同时还可以显示这些被依赖的共享库是否找得到——为了使程序正常运行，库加载工具需要确保能够找到所有库以及所有的依赖项（一般情况下，库加载工具会在 LD_LIBRARY_PATH 这个环境变量指定的目录列表中去搜寻所需要的库）。

```
/usr/bin:ldd xeyes
        linux-gate.so.1 =>  (0xb7efa000)
        libXext.so.6 => /usr/lib/libXext.so.6 (0xb7edb000)
        libXmu.so.6 => /usr/lib/libXmu.so.6 (0xb7ec6000)
        libXt.so.6 => /usr/lib/libXt.so.6 (0xb7e77000)
        libX11.so.6 => /usr/lib/libX11.so.6 (0xb7d93000)
        libSM.so.6 => /usr/lib/libSM.so.6 (0xb7d8b000)
        libICE.so.6 => /usr/lib/libICE.so.6 (0xb7d74000)
        libm.so.6 => /lib/libm.so.6 (0xb7d4e000)
        libc.so.6 => /lib/libc.so.6 (0xb7c05000)
        libXau.so.6 => /usr/lib/libXau.so.6 (0xb7c01000)
        libxcb-xlib.so.0 => /usr/lib/libxcb-xlib.so.0 (0xb7bff000)
        libxcb.so.1 => /usr/lib/libxcb.so.1 (0xb7be8000)
        libdl.so.2 => /lib/libdl.so.2 (0xb7be4000)
        /lib/ld-linux.so.2 (0xb7efb000)
        libXdmcp.so.6 => /usr/lib/libXdmcp.so.6 (0xb7bdf000)
```

共享库之所以使用更大的链接粒度是因为现代操作系统已经相当聪明了，当你想用静态库的时候，他为了节省一些硬盘空间，就采用小粒度的链接方式，但对于共享库来说，不同的程序运行时共用同一个代码段（但并不共同数据段和 bss 段，因为毕竟不同的程序使用不同的内存空间）。为了做到这一点，必须对整个库的内容进行一次性映射，这样才能保证库内部的符号集中保存在一片连续的空间里——否则，如果某个进程导入了 a.o 和 c.o， 另一个进程导入的是 b.o 和 c.o，那么就没什么共同点可以供操作系统利用了。

Windows DLLs

虽然 Unix 和 Windows 平台的共享库原理大体上一致，但有一些细节如果不注意的话，还是很容易犯错的。

导出符号

两个平台之间最大的区别在于 Windows 的共享库不会自动导出程序中的符号。在 Unix 上，每一个目标文件中所有与共享库关联的符号，对用户而言都是可见的，但在 Windows 上，为了使这些符号可见，程序员必须做一些额外的操作，例如，将其导出。

从 Windows DLL 中导出符号信息的方法一共有三种（这三种方法可以同时用于同一个库中）

- 在源代码中为符号声明关键字declspec(dllexport)，例如：

```
__declspec(dllexport) int my_exported_function(int x, double y);
```

- 使用链接器 LINK.EXE 提供的选项： /export:_symbol_to_export

```
LINK.EXE /dll /export:my_exported_function
```

- 使用链接器的 /DEF:_def_file_ 这一选项，它可用于导入模块定义文件（module definition (.DEF) file）。该文件中有一部分名为 EXPORTS，它包含了你想导出的符号信息。

```
EXPORTS
  my_exported_function
  my_other_exported_function
```

对于以上三种方法而言，第一种方法最为简便，因为编译器会自行为你考虑命名改写（name mangling）的问题。

.LIB 以及其它与库相关的文件

Windows 的这一特性（符号不可见）导致了 Windows 库的第二重复杂性：链接器在将各符号链接到一起时所需要的导出符号信息，并不包含在 DLL 文件中，而是包含在与之相对应的 .LIB 文件中。

与某个 DLL 库关联的 .LIB 文件列出了该 DLL 库中（导出的）符号以及符号地址。所有使用这个 DLL 库的程序都必须同时访问它的 .LIB 文件才能保证所有符号正常链接。

有件经常把人弄糊涂的事：静态库的扩展名也是 .LIB！

事实上，与 Windows 库有关的文件类型简直千姿百态，除了上文件提及的 .LIB 文件和（可选的）.DEF 文件外，以下列出了你可能遇到的所有与 Windows 库有关的文件。

- 链接输出文件：
  - library.DLL: 库的实现代码，它可实时导入每个使用该库的可执行程序。
  - library.LIB: “导入库”文件，给定了 DLL 文件中的符号及地址列表。只有当 DLL 导出某些符号时才会产生这个文件，如果没有符号导出，.LIB 文件也就没有存在的必要了。所有使用该库的程序在链接阶段都必需用到该文件。
  - library.EXP: 这是动态库处在链接期时的一个“导出文件”，当链接中二进制文件出现循环依赖时，该文件就派上用场了。
  - library.ILK: 如果链接时指定了 /INCREMENTAL 选项这就意味着开启了增量链接功能，该文件保存着增量链接时的相关状态，以供该动态库下次增量链接时使用。
  - library.PDB: 如果链接时指定了 /DEBUG 选项，将生成程序数据库，包含了整个库的所有调试信息。
  - library.MAP: 如果链接时指定了 /MAP 选项，将生成描述整个库内部布局信息的文件。
- 链接输入文件：
  - library.LIB: “导入库”文件，给定了链接时所需的 DLL 文件中的符号及地址列表。
  - library.LIB: 这是一个静态库文件，包含了链接时所需的系统目标文件集。请注意：使用 .LIB 文件时，需要区分是静态库还是“导入库”。
  - library.DEF: 这是一个“模块定义”文件，该文件对链接库的各种细节都给予了控制权，其中包括符号导出([译者注4])。
  - library.EXP: 这是动态库处于链接期时的一个“导出文件”，它提前运行一个与库文件对应的 LIB.EXE 工具（[译者注5]），并提前生成对应的 .LIB 文件。当链接中的二进制文件出现循环依赖时，该文件就派上用场了。
  - library.ILK: 增量链接状态文件，详见上文。
  - library.RES: 资源文件，包含了执行过程中所需的各种GUI部件信息，这些信息都将包含在最终的二进制文件中。

这与Unix正好相反，Unix中这些外部库所需的大部分信息一般情况下全都包含在库文件里了。

导入符号

正如上文所提，Windows 要求 DLL 显示地声明需要导出的符号，同样，使用动态库文件的程序必须显示地声明它们想导入的符号。这是一个可选功能，但对于16位 Windows 里的一些古老功能来说，这个选项可以实现运行速度的优化。

我们所要做的是在源代码里加上这么一句话：declare the symbol as __declspec(dllimport) ，看上去就像这样：

```
__declspec(dllimport) int function_from_some_dll(int x, double y);
__declspec(dllimport) extern int global_var_from_some_dll;
```

这一方法看似稀松平常，但由于 C 语言里所有函数以及全局变量都在且仅在头文件中声明一次，这会让我们陷入一个两难的境地：DLL 中包含了函数和变量的定义的代码需要进行符号导出，但 DLL 以外的代码需进行符号导入。

一般采取的回避方式是在头文件中加上一个预处理宏（preprocessor macro）：

```
#ifdef EXPORTING_XYZ_DLL_SYMS
#define XYZ_LINKAGE __declspec(dllexport)
#else
#define XYZ_LINKAGE __declspec(dllimport)
#endif

XYZ_LINKAGE int xyz_exported_function(int x);
XYZ_LINKAGE extern int xyz_exported_variable;
```

DLL 中的包含函数和变量定义的 C 文件可以确保它在引用这个头文件之前就已经定义（#defined）了预处理宏EXPORTING_XYZ_DLL_SYMS，对于符号的导出也是如此。任何引用了该文件的其他代码，都无需定义这一符号也无需指示符号的导入。

循环依赖

动态链接库的终级难题在于 Windows 比 Unix 严厉，它要求每个符号在链接期都必须是“已解决符号”。在 Unix 中，链接一个包含链接器不认识的“未解决符号”的动态库是可行的。在 Windows 中，任何使用引用了共享库的代码都必须提供库中的符号，否则程序将加载失败，Windows 不允许任何形式的松懈。

在大部分系统中，这不算个事儿，可执行程序依赖于高级库，高级库依赖于低级库，所有的一切都通过层层反向链接关联到一起：从低级库开始，再到高级库，最终到依赖它们的可执行文件。

然而，一旦两个二进制文件存在着相互依赖关系，事情就变得诡异起来。如果 X.DLL 使用了 Y.DLL 中的符号，而 Y.DLL 又反过来需要 X.DLL 中的符号，于是就出现了“先有鸡还有先有蛋”的问题：无论先链接哪个库，都无法找到另一个库的符号。

Windows提供了一种绕过这一问题的方法，大致过程如下：

- 首先，生成一个库 X 的假链接。运行 LIB.EXE（不是 LINK.EXE）来生成 X.LIB 文件，这跟用 LIB.EXE 生成的一模一样。这时不会生成 X.DLL 文件，取而代之的是 X.EXP 文件。
- 以正常的方式进行库 Y 的链接：使用上一步中生成的X.LIB，导出 Y.DLL 和 Y.LIB。
- 最后以合适的方式链接库 X，这跟正常的链接方式几乎没什么差别，唯一不同的是额外需要第一步生成的 X.EXP 文件。之后采用正常的方式，导入上一步生成的 Y.LIB，并生成 X.DLL。与正常方式不同之处在于，链接时将不再生成 X.LIB 文件，因为第一步已经生成过了（这在 .EXP 文件中有标记指示）

当然，更好的解决方法是去重构这些库来消除这种循环依赖……。

C++ 在 C 的基础上提供了更多额外的功能，这些功能中有很大一部分需要与链接器的操作进行交互。这并不符合最初的设计——最初 C++ 实现的目的是作为 C 编译器的前端，因此作为后端的链接器并不需要任何改变——但随着 C++ 功能日趋复杂，链接器也不得不加入对这些功能的支持。

函数重载和命名改编

C++ 的第一个改变是允许函数重载，即程序中允许存在多个不同版本的同名函数，当然它们的类型不同（即函数签名不同）。

```c
int max(int x, int y)
{
  if (x>y) return x;
  else return y;
}
float max(float x, float y)
{
  if (x>y) return x;
  else return y;
}
double max(double x, double y)
{
  if (x>y) return x;
  else return y;
}
```

这一做法显然给链接器出了一个难题：当其它代码调用 max 函数时，它到底是想调用哪一个呢？

链接器采用一种称为“命名改写（name mangling）”的方法来解决这一问题，之所以使用“mangling”是因为这个词有损坏、弄糟之意，与函数签名相关的信息都被“损坏”了，变成一种文本形式，成为链接器眼中符号的实际名称。不同的函数签名将被“损坏”成不同的名称，这样就解决了函数名重复的问题。

我不打算深入讲解“命名改写”的具体规则，因为不同编译平台有不同的改编规则，但我们通过查看事例代码所对应的目标文件结构，可以对“命名改写”规则有一个直观的认识（记诠住, nm 命令绝对是您不可或缺的好伙伴！）：

```c
fn_overload.o中的符号：

Name                  Value   Class        Type         Size     Line  Section

__gxx_personality_v0|        |   U  |            NOTYPE|        |     |*UND*
_Z3maxii            |00000000|   T  |              FUNC|00000021|     |.text
_Z3maxff            |00000022|   T  |              FUNC|00000029|     |.text
_Z3maxdd            |0000004c|   T  |              FUNC|00000041|     |.text
```

从上图中，我们可以看出，三个名为 max 的函数，在目标文件中的名称并不相同。聪明的你应该能够猜得出来 max 的后两个字母来自各自的参数类型：i表示int, f表示float，d表示double（如果把类、命名空间、模板，以及操作符重载都加入命名改编，情况将更为复杂）。

需要注意的是，如果你希望能够在链接器可识别的名称(the mangled names)和用户可识别的名称(the demangled names)之间相互转化，则需要另外单独使用别的程序（如 c++filt）或者加入命令行选项（对于 GNU 的 nm 命令，可以加 –demangle 选项），这样你就可以得到如下信息：

```c
fn_overload.o中的符号：

Name                  Value   Class        Type         Size     Line  Section

__gxx_personality_v0|        |   U  |            NOTYPE|        |     |*UND*
max(int, int)       |00000000|   T  |              FUNC|00000021|     |.text
max(float, float)   |00000022|   T  |              FUNC|00000029|     |.text
max(double, double) |0000004c|   T  |              FUNC|00000041|     |.text
```

命名改写机制最常见的“坑”就是当 C 和 C++ 代码混在一起写的时候，C++ 编译器生成的符号名称都经过了改编处理，而 C 编译器生成的符号名称就是它在源文件中的名称。为了避免这一问题，C++ 采用 extern “C” 来声明和定义 C 语言函数，其目的在于告诉 C++ 编译器这个函数名不能被改变，既可能因为相关的 C 代码需要调用 C++ 函数的定义，也可能因为相关的 C++ 代码需要调用 C 函数。

回到本文最初的例子，现在我们很容易能看出这很可能是因为某人将 C 和 C++ 链接到一起却忘了加 extern “C” 声明。

```c
g++ -o test1 test1a.o test1b.o
test1a.o(.text+0x18): In function `main':
: undefined reference to `findmax(int, int)'
collect2: ld returned 1 exit status
```

这条错误信息中最明显的提示点是那个函数签名——它不仅仅是在抱怨你没定义 findmax ，换句话说，C++ 代码实际上想找的是形如 “_Z7findmaxii” 的符号，可只找到 “findmax”，因此链接失败了。

顺便提一句，注意 extern “C” 的链接声明对成员函数无效（见 C++ 标准文档的7.5.4章节）

静态初始化

C++ 比C 多出的另一个大到足以影响链接器行为的功能是对象的构造函数（constructors）。构造函数是用于初始化对象内容的一段代码。就其本身而言，它在概念上等同于一个变量的初始值，但关键的区别在于，它初始化的不是一个变量，而是一整块代码。

让我们回想一下前文所学内容：一个全局变量可以给定一个特殊的初值。在 C 语言中，为全局变量设定一个初始是件轻而易举的事：在程序即将运行之时，将可执行文件中数据段所存的值拷贝至内存对应的地址即可。

在 C++ 中，构造过程所需完成的操作远比“拷贝定值”复杂得多：在程序开始正常运行之前，类层次体系中各种构造函数里的代码都必须提前执行。

为了处理好这一切，编译器在每一个C++文件的目标文件中都保存了一些额外信息，例如，保存了某个文件所需的构造函数列表。在链接阶段，链接器把所有列表合成一张大表，通过一次次扫描该表来调用每个全局对象对应的构造函数。

请注意，所有这些全局对象的构造函数的调用顺序并未定义——因此，这完全取决于链接器的实现。（更多细节可以参看 Scott Meyers 的 Effective C++ 一书，第二版的条款47和 href=”http://www.amazon.com/gp/product/0321334876”>第三版的条款4有相应的介绍）

我们同样可以使用 nm 命令来查看这些列表信息。以下面这段 C++ 代码为例：

```c
class Fred {
```

```
private:
  int x;
  int y;
public:
  Fred() : x(1), y(2) {}
  Fred(int z) : x(z), y(3) {}
};

Fred theFred;
Fred theOtherFred(55);
```

这段代码的 nm 输出如下（已经进行了反命名改编处理）：

```
global_obj.o中的符号：

Name                  Value   Class        Type         Size     Line  Section

__gxx_personality_v0|        |   U  |            NOTYPE|        |     |*UND*
__static_initialization_and_destruction_0(int, int)|00000000|   t  |             FUNC|00000039|     |.text
Fred::Fred(int)        |00000000|   W  |             FUNC|00000017|     |.text._ZN4FredC1Ei
Fred::Fred()        |00000000|   W  |             FUNC|00000018|     |.text._ZN4FredC1Ev
theFred             |00000000|   B  |           OBJECT|00000008|     |.bss
theOtherFred        |00000008|   B  |           OBJECT|00000008|     |.bss
global constructors keyed to theFred  |0000003a|   t  |             FUNC|0000001a|     |.text
```

这段输出内容给了很多信息，但我们感兴趣的是 Class 列为 W 的那两项（W 在这里表示弱符号 [译者注6]），它们的 Section 列形如".gnu.linkonce.t.stuff"，这些都是全局对象构造函数的特征，我们可以从 "Name" 这一列看出些端倪——在不同情况下使用两个构造函数中的一个。

模板

上文中，我们给了三个不同 max 函数的例子，在这个例子中，每个 max 函数带有不同的参数，但函数体的代码实际上完全相同，作为程序员，我们得为这种"复制粘贴"完全相同的代码感到可耻。

于是 C++ 引入了模板(templates)这一概念来避免这种情况——只需一份代码来完全所有工作。我们先创建一个只含有一个 max 函数代码的头文件 max_template.h ：

```
template &lt;class T>
T max(T x, T y)
{
  if (x>y) return x;
  else return y;
}
```

然后将该头文件应用到 C++ 代码中，并使用这个模板函数：

```
#include "max_template.h"

int main()
{
  int a=1;
  int b=2;
  int c;
  c = max(a,b);  // 编译能自动识别出当前需要调用的是 max<int>(int,int)
  double x = 1.1;
  float y = 2.2;
  double z;
  z = max&lt;double>(x,y); // 编译器无法识别，强制调用 max<double>(double,double)
  return 0;
}
```

这个例子中的C++文件调用了两种类型的 max(int,int) 和 max(double,double)，而对于另一个 C++ 文件，可能会调用该模板的其他实例化函数：比如max(float,float)，甚至还有可能是更复杂的 max(MyFloatingPointClass,MyFloatingPointClass)。

模板的每一个实例化函数执行时使用的都是不同的机器码，因此在程序的链接阶段，编译器和链接器需要确保程序调用的每个模板实例函数都扩展出相应类型的程序代码（但对于未被调用的其他模板实例函数而言，不会有任何多余的代码生成，这样可以避免程序代码过度膨胀）。

那么编译器和链接器是如何做到这一切换呢？一般来说，有两种实现方案：一种是将每个实例函数代码展开，另一种是将实例化操作延迟到链接阶段（我喜欢将这两种方法分别称作"普通方法"（the sane way）和 "Sun方法"（the sane way）（译注：之所以取这个名字，是因为Solaris系统下的编译器采用这样的方法，而Solaris是当年Sun公司旗下最著名的操作系统。））。

对于第一种方法，即将每个实例函数代码展开，每个目标文件中都会包含它所调用的所有模板函数的代码，以上文的 C++ 文件为例，目标文件内容如下：

```
max_template.o的符号：

Name                                Value     Class        Type         Size     Line  Section

__gxx_personality_v0                |        |   U  |            NOTYPE|        |     |*UND*
double max<double>(double, double)   |00000000|   W  |              FUNC|00000041|     |.text._Z3maxIdET_S0_S0_
int max<int>(int, int)              |00000000|   W  |              FUNC|00000021|     |.text._Z3maxIiET_S0_S0_
main                                |00000000|   T  |              FUNC|00000073|     |.text
```

我们可以从中看出目标文件中即包含了 max(int,int) 也包含了 max(double,double)。

目标函数将这两个函数的定义标记成"弱符号"（weak symbos），这表示当链接器最终生成可执行程序时，将只留下所有重复定义的其中之一，剩余的定义都将弃之不用（如果设计者愿意，那么可以将链接器设计成检查所有的重复定义，它们含有几乎完全相同的代码）。这种方法最显著的缺点是每个目标文件都将占用更多的磁盘空间。

另一种方法通常是 Solaris 系统中的 C++ 编译器所使用的方法，它不会在目标文件中包含任何跟模板相关的代码，只将这些符号标记成"未定义"。等到了链接阶段，链接器将所有模板实例化函数对应的未定义符号收集在一起，然后为它们生成相应的机器码。

这种方法可以节省每个目标文件所占的空间大小，但其缺点在于链接器必须跟踪头文件所包含的源代码，还必须在链接阶段调用C++编译器，这会减慢链接速度。

动态载入库

接下来我们将讨论本文最后一个 C++ 特性：共享库的动态加载。前文介绍了如何使用共享库，这意味着最终的链接操作可以延迟到程序真正运行的时刻。在现代操作系统中，甚至还可以再往后延迟。

这需要通过一对系统调用来实现，分别是：dlopen 和 dlsym (Windows里大致对应的调用分别是LoadLibrary 和 GetProcAddress)。前者获取共享库的名称，并将其载入运行程序的地址空间。当然，载入的这个共享库本身也可能存在未定义符号，因此，调用 dlopen 很可能同时触发多个其他共享库的载入。

dlopen 为使用者提供了两种选择，一种是一次性解决导入库的所有未定义符号（RTLD_NOW），另一种是按遇到的顺序一个个解决未定义符号（RTLD_LAZY）。第一种方法意味着调用一次dlopen需要等待相当长的时间，而第二种方法则可能需要冒一定的风险，即在程序运行过程中，突然发现某个未定义符号无法解决，将导致程序崩溃终止。

如果你想从动态库中找出符号对应的名字显然不可能。但正如以往的编程问题一样，这很容易通过添加额外的间接寻址方式解决，即使用指针而不是用引用来指向该符号。dlsym 调用时，需要传入一个 string 类型的参数，表示要查找的符号的名称，返回该符号所在地址的指针（如果没找到就返回 NULL）。

动态载入与C++特性的交互

这种动态载入的功能让人觉得眼前一亮，但它是如何与影响链接器行为的各种 C++ 特性进行交互的呢?

首当其冲的棘手问题是修改（mangled）后的变量名。当调用 dlsym 时，它接收一个包含符号名的字符串，这里的符号名必须是链接器可识别的名字，换句话说，即修改后的变量名。

由于命名改编机制随着平台和编译器的变化而变化，这意味着你想进行跨平台动态定位 C++ 符号几乎完全不可能。即使你乐意花大把的时间在某个特定的编译器上，并钻研其内部机制，仍然还有更多的问题在前方等着你——这些问题超出了普通类 C 函数的范围，你还必须要把虚表（vtables）这种类型的问题纳入到你考虑的范畴。

总而言之，一般来说最好的办法是只使用唯一一个常用的入口点 extern “C”，它可以已经调用过dlsym了。这个入口点可以是一个工厂函数，返回一个指向 C++ 对象的指针，它允许访问所有的 C++ 精华。

在一个已经调用过 dlopen 的库中，编译器可以为全局目标选出构造函数，因为库中可以定义各种特殊符号，这样链接器无论在加载还是运行时，只要库需要动态地加载或者取消，都可以调用这些符号，因此所有需要用到的构造函数和析构函数都可以放到里面。在 Unix 系统中，将这两种函数称为 _init 和 _fini，而对于使用 GNU 工具链的各种现代操作系统中，则是所有标记为 __attribute__((constructor)) 和 __attribute__((destructor)) 的函数。在 Windows 中，相应的函数是带有 reason 或者 DLL_PROCESS_ATTACH，再或者 DLL_PROCESS_DETACH 参数的 DllMain 函数。

最后，动态加载可以很好地例用 “折叠重复”（fold duplicated）的方法来进行模板实例化，但对于“链接时编译模板”（compile templates at link time）这一方法则要棘手得多——因为在这种情况下，“链接期”（link time）发生在程序运行之后（而且很可能不是在当初写源代码的机器上运行）。你需要查看编译器和链接器的手册来避免这一问题。

参考资料

本文有意跳过了许多链接器内部实现机制的细节，因为我认为针对程序员们日常工作时所遇到与链接器有关的问题，本文所介绍的内容已经覆盖了其中的95%。

如果你想进行更多的深入了解，可以参考下列文章：

- John Levine，链接器和加载器：本书对链接器和加载器的工作原理给出了非常非常详细的介绍，我所略过的所有细节都包含在本书中。本文有一个在线版本供大家翻看（或者说是出版本前的草稿）
- 在Max操作系统 OS X 上，关于Mach-O([译者注7])格式的二进制文件有一篇超好的文章 [27-Mar-06 更新]
- Peter Van Der Linden, C 专家编程：本书详细介绍了如何将 C 语言代码转换成可执行程序，关于这方面内容的书，我再没看过写得比本书更牛的了。
- Scott Meyers, More Effective C++: 本书一共有34条条款，覆盖了用 C 和 C++ 共同写出的程序中的存在的各种陷阱（无论是否与链接器有关）。
- Bjarne Stroustrup, C++ 语言的设计和演化: 本书的第11.3节讨论了 C++ 的链接以及链接产生的缘由。
- Margaret A. Ellis & Bjarne Stroustrup, 带注释的C++参考手册: 本书的7.2c一节中，介绍了一种命名改编机制。
- ELF格式的相关参考文献[PDF版]
- 特别推荐两篇很有趣有文章，分别是：creating tiny Linux executables和minimal Hello World
- “How to Write Shared Libraries” [PDF版]: 由 Ulrich Drepper 所写，本文对ELF和重定位给出了更为详细的介绍。

非常感谢Mike Capp和Ed Wilson为本文提出的宝贵建议。

译者注：

[1]

- .bss: BSS全称为Block Started by Symbol（或者block storage segment）。在采用段式内存管理的架构中，BSS 段（bss segment）通常是指用来存放程序中未初始化的全局变量的一块内存区域。BSS段属于静态内存分配。
- .data: 表示数据段（data segment），通常用来存放程序中已初始化的全局变量的一块内存区，也属于静态内存分配
- .text: 表示代码段（text segment），通常用来存放程序执行代码的一块内存区，这部分区域的大小在程序运行前就已经确定，并且内存区属于只读，代码段中也可能包含少量的只读常数变量，例如字符串常量等。
- COM: 全称common段。在《程序员的自我修养》一书中，指出，如果全局变量初始化的值不为0，则保存在data段，值为0，则保存在bss段，如果没有初始化，则保存在common段。当变量为static，且未初始化时放在bss段，否则放在com段
- 以上内容参考自: 《.bss .data .text 区别》和 《通过未初始化全局变量，研究BSS段和COMMON段的不同 》

[2] ld.so 是 Unit 系统上的动态链接器，常见的变体有两个：ld.so 针对 a.out 格式的二进制可执行文件，ld-linux.so 针对 ELF 格式的二进制可执行文件。当应用程序需要使用动态链接库里的函数时，由 ld.so 负责加载。搜索动态链接库的顺序依此是：环境变量LD——LD_BRARY_PATH（a.out格式），LD_LIBRARY_PATH（ELF格式）；在Linux中，LD_PRELOAD 指定的目录具有最高优先权。 缓存文件 /etc/ld.so.cache。此为上述环境变量指定目录的二进制索引文件。更新缓存的命令是 ldconfig。 默认目录，先在 /lib 中寻找，再到 /usr/lib 中寻找。（以上来自wiki百科）

[3] b.o: 这里原文是b.c，想来是作者的笔误

[4] def文件(module definition file模块定义文件）是用来创建dll和对应的导出库的。来自：http://www.fx114.net/qa-71-109424.aspx
def模块定义文件，用来创建dll和对应的lib def文件中，可以指定dll将会导出哪些符号给用户使用，链接器会根据def文件的说明来生成dll和lib。 在def文件中使用exports语句，可以让dll内部符号可见（默认不可见）

[5] exp：导出文件。当生成了两个dll：a.dll, b.dll，二者需要互相调用对方中的函数（循环依赖），这里存在的问题是：生成a.dll时需要b.lib，生成b.dll需要a.lib，这就变成死锁了，微软的解决办尘埃 是使用exp文件，在两个dll生成之前，使用lib.exe(library manager tool库管理工具)来创建一个DLL对应的.lib和.exp 即先生成a.lib, a.exp，然后利用a.lib去生成b.dll和b.lib，这时再用b.lib来生成a.dll。a.exp文件中缓存了a.dll的导出信息，linker加载a.exp中的信息。

[6] 对于C语言来说，编译器默认函数和初始化了的全局变量为强符号，未初始化的全局变量为弱符号(C++并没有将未初始化的全局符号视为弱符号)。我们也可以通过GCC的”__attribute((weak))”来定义任何一个强符号为弱符号。注意，强符号和弱符号都是针对定义来说的，不是针对符号的引用。 来自：http://blog.csdn.net/astrotycoon/article/details/8008629

[7] Mach不是Mac，Mac是苹果电脑Macintosh的简称，而Mach则是一种操作系统内核。Mach内核被NeXT公司的NeXTSTEP操作系统使用。在Mach上，一种可执行的文件格是就是Mach-O（Mach Object file format）。1996年，乔布斯将NeXTSTEP带回苹果，成为了OS X的内核基础。所以虽然Mac OS X是Unix的“后代”，但所主要支持的可执行文件格式是Mach-O。来自：http://www.molotang.com/articles/1935.html 和 http://www.amazon.com/gp/product/0321334876

相关文章

- 用 C 语言写一个简单的 Unix Shell（1）
- 谷歌大牛的 C 语言编程建议和技巧 · 3
- 通过这 9 本开源好书学习 C 语言
- 自写信息管理系统——C 实现 · 2
- C 中 static 的常见作用

可能感兴趣的话题

- 选择困难，要不要去外包 · 2
- FreeeNG v0.1.4重磅来袭!
- 怎么转型机器学习？ · 3
- JavaScript闭包，只学这篇就够了
- 有没有人记得之前有个 FM APP 为了保持用户在线，在后台搞了"宙斯... · 4
- 2017前端开发手册三：前端职位描述

登录后评论 新用户注册

直接登录

最新评论

夜尽天明_long（ 3 ） 2015/12/18

很好的，谢谢分享

赞 回复

- 本周热门文章
- 本月热门文章
- 热门标签

0 程序员的自我修养：温故而知新

1 cp 命令两个高效的用法

2 150 多个 ML、NLP 和 Pytho...

3 编写高质量代码的思考

4 用神经网络训练一个文本分类器

5 如何做有效的Code Review？我有这...

6 CoreOS，一款 Linux 容器发行版

7 6 个理由，为什么 GNOME 仍然是...

8 决策树算法及实现

9 PHP实现定时任务（非linux-shell...

业界热点资讯 更多 »

Go 语言如果按这样改进，能不能火过 Java？
22 小时前 · 6

GitHub 新支持在 issue 插入代码段
22 小时前 · 3

Ubuntu 17.10 的 Dock 曝光：基于 Dash to Doc...
1 天前 · 5

微软禁止 IE10 等旧版浏览器访问其企业商店
21 小时前 · 2

每秒 50GB！科学家研究高频传输比现有 WiFi 快百倍
3 天前 · 13

精选工具资源 更多资源 »

Whitewidow：SQL 漏洞自动扫描工具
数据库 · 2

Caffe：一个深度学习框架
机器学习

静态代码分析工具清单：公司篇
静态代码分析

HotswapAgent：支持无限次重定义运行时类与资源
开发流程增强工具

静态代码分析工具清单：开源篇（各语言）
静态代码分析

关于伯乐在线博客

在这个信息爆炸的时代，人们已然被大量、快速并且简短的信息所包围。然而，我们相信：过多"快餐"式的阅读只会令人"虚胖"，缺乏实质的内涵。伯乐在线内容团队正试图以我们微薄的力量，把优秀的原创文章和译文分享给读者，为"快餐"添加一些"营养"元素。

快速链接
网站使用指南 »
问题反馈与求助 »
加入我们 »
网站积分规则 »
网站声望规则 »

关注我们

新浪微博：@伯乐在线官方微博
RSS：订阅地址
推荐微信号


程序员的那些事


UI设计达人


极客范

合作联系
Email：bd@Jobbole.com
QQ： 2302462408 （加好友请注明来意）

小组 – 好的话题、有启发的回复、值得信赖的圈子
头条 – 分享和发现有价值的内容与观点
相亲 – 为IT单身男女服务的征婚传播平台
资源 – 优秀的工具资源导航
翻译 – 翻译传播优秀的外文文章
文章 – 国内外的精选文章
设计 – UI,网页，交互和用户体验
iOS – 专注iOS技术分享
安卓 – 专注Android技术分享
前端 – JavaScript, HTML5, CSS
Java – 专注Java技术分享
Python – 专注Python技术分享

