

# 链接器和加载器

Beta 2

Beta 2, 2006年11月7日

仅供学术研究用，版权属于原书出版社

翻译维护：colyli@gmail.com

# 内容目录

<b>第 0 章 引子.....</b>	<b>10</b>
献给.....	10
介绍.....	10
本书的目标读者是哪些人? .....	10
章节摘要.....	11
项目.....	12
致谢.....	12
联系我们.....	13
<b>第 1 章 链接和加载.....</b>	<b>14</b>
链接器和加载器做什么? .....	14
地址绑定：从历史的角度.....	14
链接与加载.....	16
两遍链接.....	17
目标代码库.....	18
重定位和代码修改.....	19
编译器驱动.....	20
链接器命令语言.....	21
链接：一个真实的例子.....	22
练习.....	26
<b>第 2 章 体系结构的问题.....</b>	<b>27</b>
应用程序二进制接口.....	27
内存地址.....	27
字节顺序和对齐.....	27
地址构成.....	29
指令格式.....	29
过程调用和可寻址性.....	30
过程调用.....	30
数据和指令引用.....	32
IBM 370.....	32
SPARC.....	34
SPARC V8.....	34
SPARC V9.....	35
Intel x86.....	36
分页和虚拟内存.....	37

程序地址空间.....	39
映射文件.....	40
共享库和程序.....	41
位置无关代码.....	41
Intel 386 分段.....	42
嵌入式体系结构.....	43
怪异的地址空间.....	44
非一致性内存.....	44
内存对齐.....	44
练习.....	44
<b>第3章 目标文件.....</b>	<b>47</b>
目标文件中都有什么?.....	47
设计一个目标文件格式.....	47
空目标文件格式: MS-DOS 的.COM 文件.....	48
代码区段: Unix a.out 文件.....	48
a.out 头部.....	50
与虚拟内存的交互.....	51
重定位: MS-DOS EXE 文件.....	54
符号和重定位.....	56
可重定位的 a.out 格式.....	56
重定位项.....	58
符号和字串.....	59
a.out 格式小结.....	60
Unix ELF 格式.....	60
可重定位文件.....	62
ELF 可执行文件.....	66
ELF 格式小结.....	68
IBM 360 目标格式.....	68
ESD 记录.....	69
TXT 记录.....	70
RLD 记录.....	70
END 记录.....	71
小结.....	71
微软可移植可执行体格式.....	72
PE 特有区段.....	76
运行 PE 可执行文件.....	77
PE 和 COFF.....	78
PE 文件小结.....	78
Intel/Microsoft 的 OMF 文件格式.....	78

OMF 记录.....	79
OMF 文件的细节.....	80
OMF 格式小结.....	82
不同目标格式的比较.....	82
项目.....	83
练习.....	84
<b>第 4 章 存储空间分配.....</b>	<b>85</b>
段和地址.....	85
简单的存储布局.....	85
多种段类型.....	87
段与页面的对齐.....	88
公共块和其它特殊段.....	88
公共块.....	89
C++重复代码消除.....	90
初始化和终结.....	92
IBM 伪寄存器.....	93
特殊的表.....	94
X86 分段的存储分配.....	95
链接器控制脚本.....	96
嵌入式系统的存储分配.....	97
实际中的存储分配.....	98
Unix a.out 链接器的存储分配策略.....	98
ELF 中的存储分配策略.....	99
Windows 链接器的存储分配策略.....	101
练习.....	102
项目.....	103
<b>第 5 章 符号管理.....</b>	<b>104</b>
绑定和名字解析.....	104
符号表格式.....	104
模块表.....	106
全局符号表.....	108
符号解析.....	109
特殊符号.....	110
名称修改.....	110
简单的 C 和 Fortran 名称修改.....	110
C++类型编码：类型和范围.....	111
链接时类型检查.....	113
弱外部符号和其它类型符号.....	113
维护调试信息.....	113

行号信息.....	114
符号和变量信息.....	114
实际的问题.....	115
练习.....	115
项目.....	116
<b>第6章 库.....</b>	<b>117</b>
库的目的.....	117
库的格式.....	117
使用操作系统.....	117
UNIX 和 Windows 的 Archive 文件.....	118
扩展到 64 位.....	120
Intel OMF 库文件.....	120
建立库文件.....	121
搜索库文件.....	122
性能问题.....	123
弱外部符号.....	123
练习.....	124
项目.....	124
<b>第7章 重定位.....</b>	<b>126</b>
硬件和软件重定位.....	126
链接时重定位和加载时重定位.....	127
符号和段重定位.....	127
符号查找.....	128
基本的重定位技术.....	128
指令重定位.....	129
ECOFF 段重定位.....	131
ELF 重定位.....	132
OMF 重定位.....	132
可重链接和重定位的输出格式.....	132
其它重定位格式.....	133
以链表形式组织的引用.....	133
以位图形式组织的引用.....	134
特殊段.....	134
特殊情况的重定位.....	135
练习.....	135
项目.....	135
<b>第8章 加载和重叠.....</b>	<b>137</b>
基本加载.....	137
带重定位的基本加载.....	138

位置无关代码.....	138
TSS/360 位置无关代码.....	138
例程指针表.....	140
目录表.....	141
ELF 位置无关代码.....	141
位置无关代码的开销和得益.....	143
自举加载.....	144
树状结构的覆盖.....	145
定义覆盖.....	147
覆盖的实现.....	149
覆盖的其它细节.....	150
覆盖技术小结 .....	151
练习.....	151
项目.....	151
<b>第 9 章 共享库.....</b>	<b>153</b>
绑定时间.....	154
实际的共享库.....	155
地址空间管理.....	155
共享库的结构.....	156
创建共享库.....	156
创建跳转表.....	157
创建共享库.....	158
创建空占位库.....	158
版本命名.....	159
使用共享库链接.....	159
使用共享库运行.....	160
malloc hack 和其它共享库问题.....	160
练习.....	163
项目.....	163
<b>第 10 章 动态链接和加载.....</b>	<b>165</b>
ELF 动态链接.....	165
ELF 文件内容.....	165
加载一个动态链接的程序.....	168
启动动态链接器.....	169
库的查找.....	169
共享库的初始化.....	170
使用 PLT 的惰性过程链接(lazy procedure linkage).....	171
动态链接的其它特性.....	172
静态的初始化.....	172

库的版本.....	173
运行时的动态链接.....	173
Microsoft 动态链接库.....	174
PE 文件中的输入/输出符号 ( imported and exported symbols ) .....	174
惰性绑定.....	177
DLL 库和线程.....	178
OSF/1 伪静态共享库.....	178
让共享库快一些.....	179
几种动态链接方法的比较.....	179
练习.....	180
项目.....	181
<b>第 11 章 高级技术.....</b>	<b>182</b>
C++的技术.....	182
试验链接.....	183
消除重复代码.....	184
借助于数据库的方法.....	185
增量链接和重新连接.....	185
链接时的垃圾收集.....	187
链接时优化.....	188
链接时代码生成.....	189
链接时统计和工具.....	190
链接时汇编.....	190
加载时代码生成.....	190
Java 链接模式.....	191
加载 Java 类.....	192
练习.....	193
项目.....	194



# 第 0 章 引子

\$Revision: 2.2 \$

\$Date: 1999/06/09 00:48:48 \$

## 献给

我的家人， 托妮亚和莎拉。

## 介绍

几乎从有计算机以来，链接器和加载器就是软件开发工具包中的一部分，因为他们是允许使用模块（而不是一个单独的大文件）来构建程序的关键工具。

早在 1947 年，程序员们就开始使用原始的加载器：将程序的例程存储在多个不同的磁带上，并将他们合并、重定位为一个程序。在上世纪 60 年代早期，这些加载器就已经发展的相当完善了。由于那时内存很贵且容量有限，计算机的速度（以今天的标准）很慢，为了创建复杂的内存覆盖策略（将大容量的程序加在到少量的内存中），以及重复编辑之前链接过的文件（节省重新创建程序的时间），这些链接器都包含了很多复杂的特性。

上世纪 70 到 80 年代，链接技术几乎没有进展。链接器趋向于更加简单，虚拟内存技术将应用程序和覆盖机制中的大多数存储管理工作都转移给了操作系统，越来越快的计算机和越来越大的磁盘也使得重新链接一个程序或替换个别模块比仅仅链接改变过的地方更加容易了。从上世纪 90 年代起，由于增加了诸如动态链接共享库和 C++ 的诸多现代特性，链接器又开始变得复杂起来。像 IA64 这样具有长指令字和编译时访存调度等特性的先进处理器架构，也需要将一些新的特性加入到链接器中以确保在被链接的程序中可以满足代码的这些复杂需求。

## 本书的目标读者是哪些人？

本书预计供下述几类读者使用。

- **学生：**由于链接过程看起来似乎是微不足道和显而易见的，编译器构建和操作系统课程通常对链接和加载都缺乏重视。这对于以前讨论 Fortan, Pascal, C，和不使用内存映射或共享库的操作系统而言可能是对的，但是现在就不那么正确了。  
C++, Java 和其它的面向对象语言需要更加完善的链接环境。使用内存映射的可执行程序，共享库，和动态链接影响了一个操作系统的很多部分，一个忽略链接问题的操作系统设计者将承担巨大的风险。
- **实习程序员**也需要知道链接器都做了什么，尤其是对现代语言。C++ 在链接器中放置了不少独特的特性，而大型 C++ 程序容易发生难以诊断的 bug 也是由于在链接时发生了不可预料的事情（最常见的情况是静态构造函数没有按照程序员预计的顺序执行）。当正确使用时，诸如共享库和动态链接此类的链接器特性将（给程序员

的工作)带来很大的灵活性和强大支持。

- **语言设计者和开发人员**应该在构建语言和编译器时了解链接器会做什么和能做什么。由于可以由链接器处理某些细节,那些手工进行了30多年的编程任务今天在C++中可以自动处理了(想一想在C语言中为了获取和C++中的模板相同的功能,或为了保证在程序主体执行之前使成百个C源文件中的初始化例程可以执行,程序员不得不做的那堆事情)。有了功能更强大的链接器的支持,未来的语言将更加自动化而不仅限于程序范畴内的常规任务。由于链接是编译过程中将整个程序的代码放在一起处理并可对程序作为一个整体施加影响的唯一阶段,因此链接器还将被加入更多的全局程序优化功能。

(编写链接器的人员当然都需要本书。但是全球所有的链接器设计者大概能坐满一个房间,而且其中有半数因为审阅手稿已经拥有本书了。)

## 章节摘要

第1章,链接和加载,对链接的过程进行了一个简短的历史回顾,并讨论链接过程中的各个阶段,最后以一个可由输入目标文件产生显示“Hello, world”程序的简单却完整的链接器实例结束本章。

第2章,体系结构问题,以链接器设计的角度来回顾计算机体系结构。这里会以典型的精简指令集体系结构的SPARC,古老而富有活力的寄存器-内存体系结构的IBM 360/370,以及自成一派的Intel x86体系结构为例。在每种体系结构中,会讨论包括内存架构、程序寻址架构和地址布局等重要方面。

第3章,目标文件,探究了目标文件和可执行文件的内部结构。本章从最简单的MS-DOS .COM文件开始,进而分析包括DOS的EXE,Windows的COFF和PE(EXE和DLL),UNIX的a.out和ELF,以及Intel/Microsoft的OMF等更加复杂的文件。

第4章,存储分配,涵盖了链接过程的第一个阶段,即为被链接程序的各段分配存储空间,跟以实际的链接器来举例。

第5章,符号管理,涵盖符号的绑定和解析,这是一个将某个文件中引用的另一个文件的符号解析为机器地址的过程。

第6章,库,涵盖目标代码库的创建和使用,以及库的结构和性能问题。

第7章,重定位,涵盖地址重定位,即将程序内的目标代码调整为可以反映其运行时实际地址的过程。这里同样谈到了位置无关代码(PIC),这种代码是通过无需重定位的方法建立的,并讨论了这种方法的优势和代价。

第8章,加载和覆盖,涵盖了加载过程,即将程序从文件中取出装入计算机内存以运行。本章还讨论了一种古老而有效的节省存储空间的树结构覆盖技术。

第9章,共享库,讨论了在不同程序中共享一个库代码的单一拷贝需要做哪些工作。本章将注意力放在静态链接的共享库上。

第10章,动态链接和加载,将第9章的讨论延伸至动态链接的共享库。本章详细说明了两个实例,Win32的动态链接库(DLLs),和Unix/Linux的ELF共享库。

第11章,高级技术,着眼于成熟的现代链接器所做的一些变化。讨论C++需要的新特

性，包括编译命名格式(name mangling)，全局构造与析构函数，模板扩充，和冗余代码的消除。此外还包括如增量链接，链接时垃圾收集，链接时代码生成和优化，加载时代码生成，统计和性能监视。本章对 Java 的链接模式进行了简要阐述，相对于本书涉及的其它链接器它被认为更具有语义上的复杂性。

第 12 章，参考文献，这是一个带有说明的参考书目。

## 项目

从第 3 章到第 11 章有一个持续的实践项目，使用 perl 语言开发一个小但可用的链接器。尽管 perl 不那么像是实现产品级链接器时所使用的编程语言，但对一个学期内要完成的项目而言是很好的选择。Perl 处理了很多在 C/C++ 编程中会拖延编程速度的许多低层编程细节，好让学生把精力集中在当前项目的算法和数据结构上。在当前很多的计算机上，Perl 是免费的，包括 Windows 95/98 和 NT，UNIX 和 Linux，并且还有很多非常优秀的书籍指导新手使用 Perl（可参看 12 章书目中的建议）。

第 3 章最初的项目构建了一个可以以简单却完整的目标代码格式读取和写入文件的链接器框架。后继章节不断的向它添加功能，直到最后变成一个支持共享库并可产生动态可链接目标代码的功能完善的链接器。

Perl 颇能处理二进制文件和数据结构，因此如果愿意的话，该项目的链接器可以用来处理它自己的目标代码格式。

## 致谢

有许多许多人慷慨地牺牲了自己的时间来阅读和评论本书的手稿，包括出版商的评论家和 comp.compilers usenet 新闻组中阅读和评注本书在线版本的读者们。下面以姓氏字母顺序列出他们的名字：Mike Albaugh, Rod Bates, Gunnar Blomberg, Robert Bowdidge, Keith Breinholt, Brad Brisco, Andreas Buschmann, David S. Cargo, John Carr, David Chase, Ben Combee, Ralph Corderoy, Paul Curtis, Lars Duening, Phil Edwards, Oisin Feeley, Mary Fernandez, Michael Lee Finney, Peter H. Froehlich, Robert Goldberg, James Grosbach, Rohit Grover, Quinn Tyler Jackson, Colin Jensen, Glenn Kasten, Louis Krupp, Terry Lambert, Doug Landauer, Jim Larus, Len Lattanzi, Greg Lindahl, Peter Ludemann, Steven D. Majewski, John McEnerney, Larry Meadows, Jason Merrill, Carl Montgomery, Cyril Muerillon, Sameer Nanajkar, Jacob Navia, Simon Peyton-Jones, Allan Porterfield, Charles Randall, Thomas David Rivers, Ken Rose, Alex Rosenberg, Raymond Roth, Timur Safin, Kenneth G Salter, Donn Seeley, Aaron F. Stanton, Harlan Stenn, Mark Stone, Robert Strandh, Bjorn De Sutter, Ian Taylor, Michael Trofimov, Hans Walheim, Roger Wong。

本书中正确部分是他们努力的结果。而本书中错误部分都由作者负责（如果您发现了后者，请用下面的地址联系本人以便在后续版本中更正这些错误）。

我特别要感谢出版商Morgan-Kaufmann的两位编辑，Tim Cox 和Sarah Luger，他们容忍我写作过程中无限期的拖延，并把本书中支离破碎的章节拼凑在一起。

## 联系我们

本书有一个支持网站 <http://linker.iecc.com>。上面包含了本书的样章，工程项目的 perl 代码和目标文件示例，还有本书的更新和勘误。

您可以通过 [linker@iecc.com](mailto:linker@iecc.com) 给作者发送 e-mail。作者会阅读所有的来信，但可能因收信量过大而无法及时回复所有的问题。

# 第1章 链接和加载

\$Revision: 2.3 \$

\$Date: 1999/06/30 01:02:35 \$

## 链接器和加载器做什么？

任何一个链接器和加载器的基本工作都非常简单：将更抽象的名字与更底层的名字绑定起来，好让程序员使用更抽象的名字编写代码。也就是说，它可以将程序员写的一个诸如 `getline` 的名字绑定到“`iosys` 模块内可执行代码的 612 字节处”或者可以采用诸如“这个模块的静态数据开始的第 450 个字节处”这样更抽象的数字地址然后将其绑定到数字地址上。

## 地址绑定：从历史的角度

一个有助于深入理解链接器和加载器做了什么的方法就是看看他们在计算机编程系统的发展中承担了什么角色。

最早的计算机完全是用机器语言进行编程的。程序员需要在纸质表格上写下符号化程序，然后手工将其汇编为机器码，通过开关、纸带或卡片将其输入到计算机中（真正的高手可以在开关上直接编码）。如果程序员使用符号化的地址，那他就得手工完成符号到地址的绑定。如果后来发现需要添加或删除一条指令，那么整个程序都必须手工检查一遍并将所有被添加或删除的指令影响的地址都进行修改。

这个问题就在于名字和地址绑定的过早了。汇编器通过让程序员使用符号化名字编写程序，然后由程序将名字绑定到机器地址的方法解决了这个问题。如果程序被改变了，那么程序员必须重新汇编它，但是地址分配的工作已经从程序员推给计算机了。

代码的库使得地址分配工作更加复杂。由于计算机可以执行的基本操作极其简单，有用的程序都是由那些可以执行更高级、更复杂操作的子程序组成的。计算机在安装时都带有一些预先编写好、调试好的子例程库，程序员可以将它们用在自己写的新程序中，而不需编写所有的子程序。然后程序员可以将这些子例程加载到主程序中以构成一个完整的可以工作的程序。

程序员们甚至在使用汇编语言之前就使用子程序库了。在 1947 年，领导 ENIAC 项目的 John Mauchly，就写文章描述了如何将主程序和磁带中一系列选定的子程序一起加载到计算机中，并通过将子程序代码重定位以反映实际被加载的地址。鉴于 Mauchly 认为程序和子程序都是由机器语言编写的，因此我们可能会惊奇的发现，甚至在汇编语言出现之前，链接器的两个基本功能重定位和库查询就已经出现了。可重定位的加载器允许子例程的作者或用户在编写子例程时认为它们都起始于地址 0，并将实际的地址绑定延迟到这些例程被链接到某个特定的程序中时。

随着操作系统的出现，有必要将可重定位的加载器从链接器和库中分离出来。在有操作系统之前，一个程序可以支配机器所有的内存，由于知道计算机中所有的地址都是可用的，

因此它能以固定的内存地址来汇编和链接。但是有了操作系统以后，程序就必须和操作系统甚至其它程序共享计算机的内存。这意味着在操作系统将程序加载到内存之前是无法确定程序运行的确切地址的，并将最终的地址绑定从链接时推延到了加载时。现在链接器和加载器已经将这个工作划分开了，链接器对每一个程序的部分地址进行绑定并分配相对地址，加载器完成最后的重定位步骤并赋予的实际地址。

随着计算机系统变得越来越复杂，链接器被用来做了更多、更复杂的名字管理和地址绑定的工作。Fortran 程序使用了多个子程序和公共块（被多个子程序共享的数据区域），而它是由链接器来为这些子程序和公共数据块进行存储布局和地址分配的。逐渐地链接器还需要处理目标代码库。包括用 Fortran 或其它语言编写的应用程序库，并且编译器也支持那些可以从被编译好的处理 I/O 或其它高级操作的代码中隐含调用的库。

由于程序很快就变得比可用的内存大了，因此链接器提供了覆盖技术，它可以让程序员安排程序的不同部分来分享相同的内存，当程序的某一部分被其它部分调用时可以按需加载。上世纪 60 年代在硬盘出现后覆盖技术在大型主机系统上得到了广泛的应用，一直持续到 70 年代中期虚拟内存技术出现。然后重新以几乎相同的形式在 80 年代早期的微型机算机上出现，并在 90 年代 PC 上采用虚拟内存后逐渐没落。现在它们仍被应用在内存受限的嵌入式环境中，并且当程序员或者编译器为了提高性能而精确的控制内存使用时可能会再次出现。

随着硬件重定位和虚拟内存的出现，每一个程序可以再次拥有整个地址空间，因此链接器和加载器变得不那么复杂了。由于硬件（而不是软件）重定位可以对任何加载时重定位进行处理，程序可以按照被加载到固定地址的方式来链接。但是具有硬件重定位功能的计算机往往不止运行一个程序，而且经常会运行同一个程序的多个副本。当计算机运行一个程序的多个实例时，程序中的某些部分在所有的运行实例中都是相同的（尤其是可执行代码），而另一些部分是各实例独有的。如果不变的部分可以从发生改变的部分中分离出来，那么操作系统就可以只使用一份不变部分的副本，节省相当可观的存储空间。编译器和汇编器可以被修改为在多个段内创建目标代码，为只读代码分配一个段，为别的可写数据分配其它段。链接器必须能够将相同类型的所有段都合并在一起，以使得被链接程序的所有代码都放置在一个地方，而所有的数据放在另一个地方。由于地址仍然是在链接时被分配的，因此和之前相比并不能延迟地址绑定的时机，但更多的工作被延迟到了链接器为所有段分配地址的时候。

即使多个不同的程序运行在一个计算机上时，这些不同的程序实际上仍会共享很多公共代码。例如，几乎每一个 C 语言的程序都会使用诸如 fopen 和 printf 这样的例程，数据库应用程序都会使用一个巨大的访问库来连接数据库，运行在诸如 X Window, MS Windows, 或 Macintosh 这样的图形用户界面下的应用程序会使用到部分的图形用户界面库。多数系统现在都会提供共享库给应用程序使用，这样所有使用某个库的程序可以仅共享一份副本。这既提升了不少运行时的性能也节省了大量磁盘空间：在小程序中通用库例程会占用比主程序本身更多的空间。

在较简单的静态共享库中，每个库在创建时会被绑定到特定的地址，链接器在链接时将程序中引用的库例程绑定到这些特定的地址。由于当静态库中的任何部分变化时程序都需要被重新连接，而且创建静态链接库的细节也是非常冗长乏味的，因此静态链接库实际上很麻烦死板。故又出现了动态链接库，使用动态链接库的程序在开始运行之前不会将所用库中的段和符号绑定到确切的地址上。有时这种绑定还会更为延迟：在完全的动态链接中，被调

用例程的地址在第一次调用前都不会被绑定。此外在程序运行过程中也可以加载库并进行绑定。这提供了一种强大且高性能的扩展程序功能的方法。尤其是微软 Windows 广泛的使用了运行时加载共享库（如我们所知的 DLL，Dynamiclly Linked Libraries）对程序进行构建和扩展。

## 链接与加载

链接器和加载器完成几个相关但概念上不同的动作。

- 程序加载：将程序从辅助存储设备（自 1968 年后这就意味着磁盘）拷贝到主内存中准备运行。在某些情况下，加载仅仅是将数据从磁盘拷入内存；在其他情况下，还包括分配存储空间，设置保护位或通过虚拟内存将虚拟地址映射到磁盘内存页上。
- 重定位：编译器和汇编器通常为每个文件创建程序地址从 0 开始的目标代码，但是几乎没有计算机会允许从地址 0 加载你的程序。如果一个程序是由多个子程序组成的，那么所有的子程序必须被加载到互不重叠的地址上。重定位就是为程序不同部分分配加载地址，调整程序中的数据和代码以反映所分配地址的过程。在很多系统中，重定位不止进行一次。对于链接器的一种普遍情景是由多个子程序来构建一个程序，并生成一个链接好的起始地址为 0 的输出程序，各个子程序通过重定位在大程序中确定位置。当这个程序被加载时，系统会选择一个加载地址，而连接好的程序会作为整体被重定位到加载地址。
- 符号解析：当通过多个子程序来构建一个程序时，子程序间的相互引用是通过符号进行的；主程序可能会调用一个名为 sqrt 的计算平方根例程，并且数学库中定义了 sqrt 例程。链接器通过标明分配给 sqrt 的地址在库中来解析这个符号，并通过修改目标代码使得 call 指令引用该地址。

尽管有相当一部分功能在链接器和加载器之间重叠，定义一个仅完成程序加载的程序为加载器，一个仅完成符号解析的程序为链接器是合理的。他们任何一个都可以进行重定位，而且曾经也出现过集三种功能为一体的链接加载器。

重定位和符号解析的划分界线是模糊的。由于链接器已经可以解析符号的引用，一种处理代码重定位的方法就是为程序的每一部分分配一个指向基址的符号，然后将重定位地址认为是对该基址符号的引用。

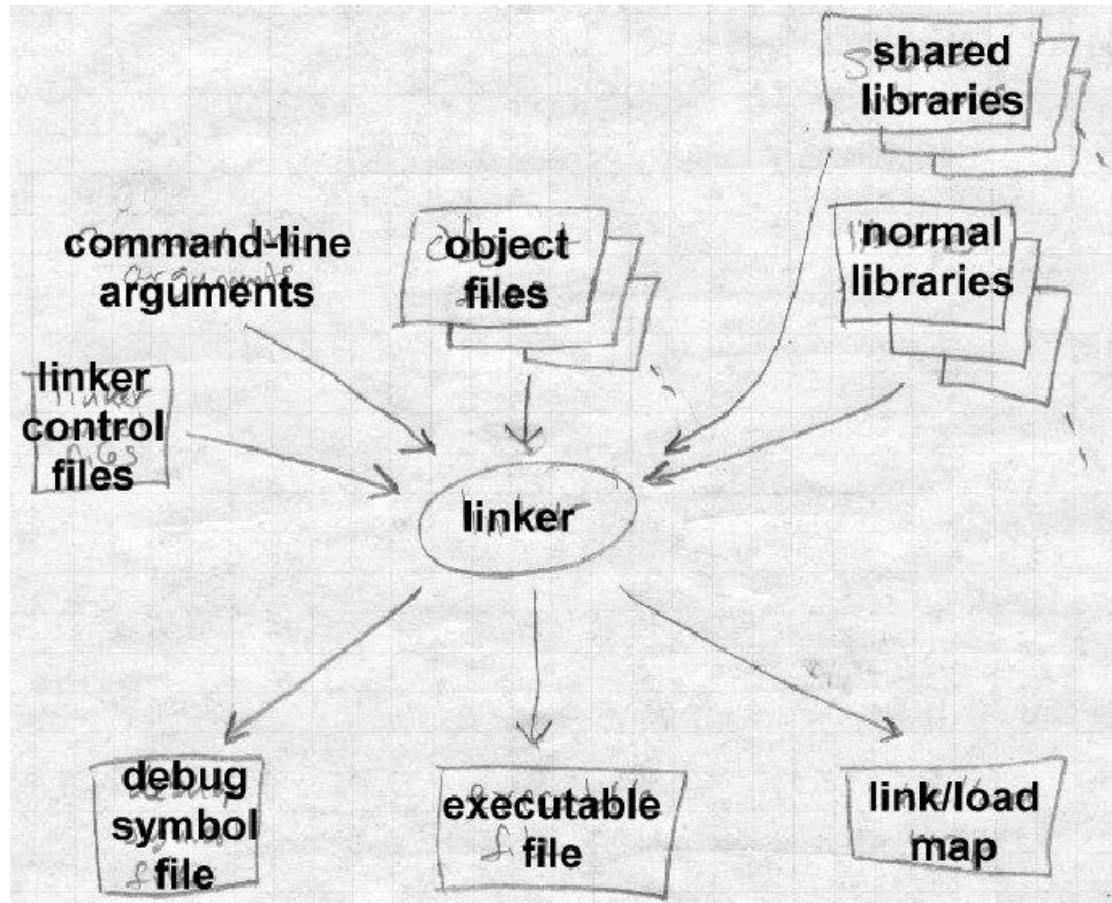
链接器和加载器共有的一个重要特性就是他们都会修改目标代码，他们也许是唯一比调试程序在这方面应用更为广泛的程序。这是一个独特而强大的特性，而且细节非常依赖于机器的规格，如果做错的话就会引发令人困惑的 bug。

## 两遍链接

现在我们来看看链接器的普遍结构。就象编译或汇编一样，链接基本上也是一个两遍的过程。链接器将一系列的目标文件、库、及可能的命令文件作为它的输入，然后将输出的目标文件作为产品结果，此外也可能有诸如加载映射信息或调试器符号文件的副产品。图 1。

图 1.1 链接过程

链接器接受输入文件，产生输出文件，有时候还有其他的文件。



每个输入文件都包含一系列的段 (segments)，即会被连续存放在输出文件中的代码或数据块。每一个输入文件至少还包含一个符号表 (symbol table)。有一些符号会作为导出符号，他们在当前文件中定义并在其他文件中使用，通常都是可以在其它地方被调用的当前文件内例程的名字。其它符号会作为导入符号，在当前文件中使用但不在当前文件中定义，通常都是在该文件中调用但不存在于该文件中的例程的名字。

当链接器运行时，会首先对输入文件进行扫描，得到各个段的大小，并收集对所有符号的定义和引用。它会创建一个列出输入文件中定义的所有段的段表，和包含所有导出、导入符号的符号表。

利用第一遍扫描得到的数据，链接器可以为符号分配数字地址，决定各个段在输出地址空间中的大小和位置，并确定每一部分在输出文件中的布局。

第二遍扫描会利用第一遍扫描中收集的信息来控制实际的链接过程。它会读取并重定位目标代码，为符号引用替换数字地址，调整代码和数据的内存地址以反映重定位的段地址，并将重定位后的代码写入到输出文件中。通常还会再向输出文件中写入文件头部信息，重定位的段和符号表信息。如果程序使用了动态链接，那么符号表中还要包含运行时链接器解析动态符号时所需的信息。在很多情况下，链接器自己将会在输出文件中生成少量代码或数据，例如用来调用覆盖中或动态链接库中的例程的“胶水代码”，或在程序启动时需要被调用的指向各初始化例程的函数指针数组。

不论程序是否使用了动态链接，文件中都会保存一个程序本身不会使用的重链接或调试用符号表，但是这些信息可以被处理输出文件的其它程序所使用。

有些目标代码的格式是可以重链接的，也就是一次链接器运行的输出文件可以作为下次链接器运行的输入。这要求输出文件要包含一个像输入文件中那样的符号表，以及其它会出现在输入文件中的辅助信息。

几乎所有的目标代码格式都预备有调试符号，这样当程序在调试器控制下运行时，调试器可以使用这些符号让程序员通过源代码中的行号或名字来控制程序。根据目标代码格式细节的不同，调试符号可能会与链接器需要的符号混合在一个符号表中，也可能独立于链接器需要的符号表为链接器建立单独建立一个有些许冗余的符号表。

有很少的一些链接器可以在一次扫描中完成工作。他们是通过在链接过程中将输入文件的部分或全部缓冲在内存或磁盘中，并稍后读取被缓冲的信息的方法来实现的。由于这是一个并不影响链接过程两边扫描实质的实现技巧，因此这里我们不再赘述。

## 目标代码库

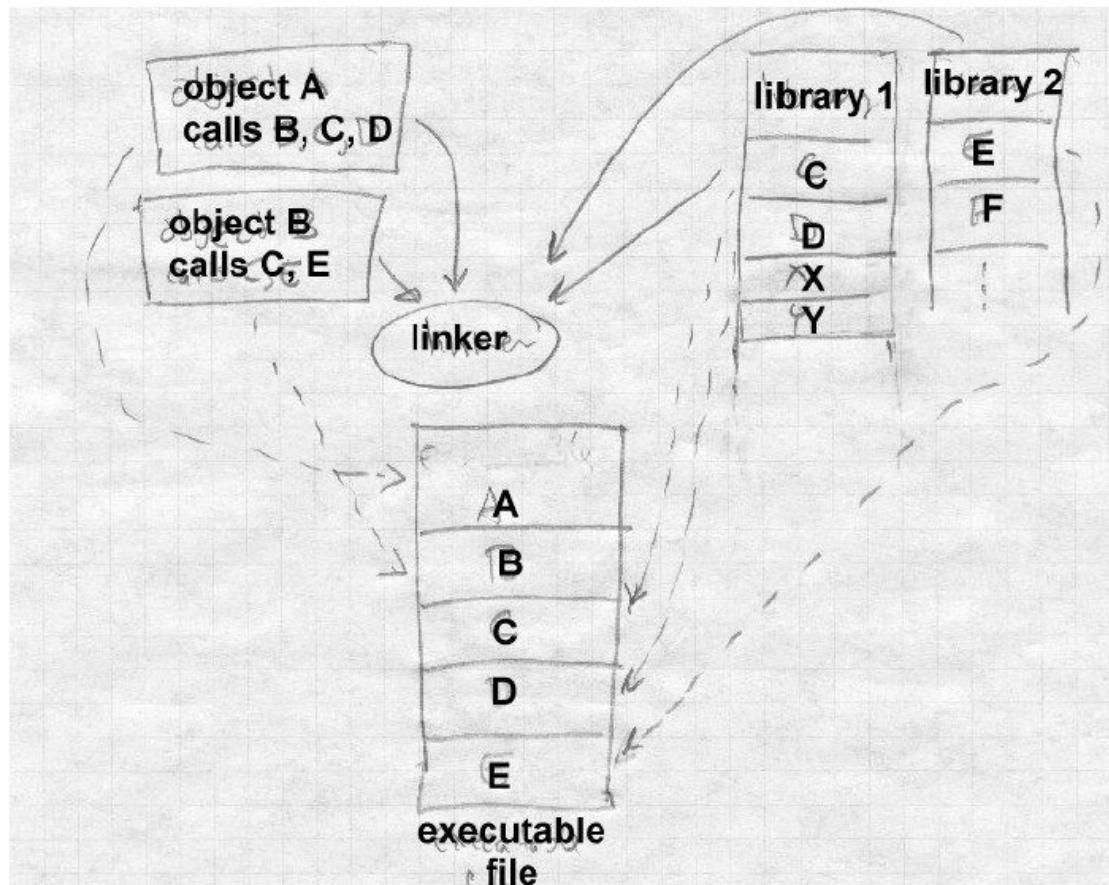
所有的链接器都会以这样或那样的形式来支持目标代码库，同时它们中的大多数还会支持各种各样的共享库。

目标代码库的基本原则是很非常简单的，如图 2 示。一个库不比一些目标代码文件的集合复杂多少（实际上，在某些系统上你可以直接将一些目标代码文件链接在一起，将结果作为链接库来使用）。当链接器处理完所有规则的输入文件后，如果还存在未解析的导入名称（imported name），它就会查找一个或多个库，然后将输出这些未解析名字的库中的任何文件链接进来。

---

图 1-2 目标代码库

链接器读入目标文件，再跟上很多库文件。



由于链接器将部分工作从链接时推迟到了加载时，使这项任务稍微复杂了一些。在链接器运行时，链接器会识别出解析未定义符号所需的共享库，但是链接器会在输出文件中标明用来解析这些符号的库名称，而不是在链接时将他们链入程序，这样可以在程序被加载时进行共享库绑定。详细内容见第 9 和第 10 章。

## 重定位和代码修改

链接器和加载器的核心动作是重定位和代码修改。当编译器或汇编器产生一个目标代码文件时，它使用文件中定义的未重定位代码地址和数据地址来生成代码，对于其它地方定义的数据或代码通常就是 0。作为链接过程的一部分，链接器会修改目标代码以反映实际分配的地址。例如，考虑如下这段将变量 a 中的内容通过寄存器 eax 移动到变量 b 的 x86 代码片段。

```
mov a, %eax
mov %eax, b
```

如果 a 定义在同一文件的位置 0x1234，而 b 是从其它地方导入的，那么生成的代码将会是：

```
A1 34 12 00 00 mov a, %eax
A3 00 00 00 00 mov %eax, b
```

每条指令包含了一个字节的操作码和其后 4 个字节的地址。第一个指令有对地址 1234

的引用（由于 x86 使用从右向左的字节序，因此这里是逆序），而第二个指令由于 b 的位置是未知的因此引用位置为 0。

现在想象链接器将这段代码进行连接，a 所属段被重定位到了 0x10000，b 最终位于地址 0x9A12。则链接器会将代码修改为：

```
A1 34 12 01 00 mov a, %eax  
A3 12 9A 00 00 mov %eax, b
```

也就是说，链接器将第一条指令中的地址加上 0x10000，现在它所标识的 a 的重定位地址就是 0x11234，并且也补上了 b 的地址。虽然这些调整影响的是指令，但是目标文件中数据部分任何相关的指针也必须修改。

在稍老一些的地址空间很小、直接寻址的计算机系统上，由于只有一到两种链接器需要处理的地址格式，因此代码修改的过程相当简单。对于现代计算机，包括所有的 RISC 架构，都需要进行复杂的多的代码修改。没有一条指令有足够的空间容纳一个直接地址，因此编译器和链接器不得不才用复杂的寻址技巧来处理任意地址上的数据。某些情况下，使用两到三条指令来组成一个地址都是有可能的，每个指令包含地址的一部分，然后使用位操作将它们组合为一个完整的地址。在这种情况下，链接器不得不对每个指令都进行恰当的修改，将地址中的某些位插入到每一个指令中。其它情况下，一个例程或一组例程使用的所有地址都被放置在一个作为“地址池”的数组中，初始化代码将某一个机器寄存器指向这个数组，当需要时，代码会将该寄存器作为基址寄存器从地址池中加载所需指针。链接器需要由被程序使用的所有地址来创建这个数组，并修改各指令使它们可以关联到正确的地址池入口处。我们将在第 7 章讨论这个话题。

有些系统需要无论加载到什么位置都可以正常工作的位置无关代码。链接器需要提供额外的技巧来支持位置无关代码，与程序中无法做到位置无关的部分隔离开来，并设法使这两部分可以互相通讯（详见第 8 章）。

## 编译器驱动

很多情况下，链接器所进行的操作对程序员是几乎或完全不可见的，因为它会做为编译过程的一部分自动进行。多数编译系统都有一个可以按需自动执行编译器各个阶段的编译器驱动。例如，若一个程序员有两个 C 源程序文件（简称 A, B），那么在 UNIX 系统上编译器驱动将会运行如下一系列的程序：

- C 语言预处理器处理 A，生成预处理的 A
- C 语言编译预处理的 A，生成汇编文件 A
- 汇编器处理汇编文件 A，生成目标文件 A
- C 语言预处理器处理 B，生成预处理的 B
- C 语言编译预处理的 B，生成汇编文件 B
- 汇编器处理汇编文件 B，生成目标文件 B
- 链接器将目标文件 A、B 和系统 C 库链接在一起

也就是说，编译器驱动首先会将每个源文件编译为汇编语言，然后转换为目标代码，接着链接器会将目标代码链接器一起，并包含任何需要的系统 C 库例程。

编译器驱动通常要比这聪明的多，他们会比较源文件和目标代码文件的时间，仅编译那些被修改过的源文件（UNIX make 程序就是典型的例子）。尤其是在编译 C++ 和其它面向对象语言时，编译器驱动会使用各种各样的技巧来克服链接器或目标代码格式的局限。例如，C++ 模板定义了一个数量可能不限的相关例程的集合，这样就可以找到程序实际使用的数目有限的模板例程集合，编译器驱动可以在没有模板代码时将程序的目标文件链接在一起，然后读取链接器的错误信息查看未定义东西，再调用 C++ 编译器为需要的模板例程生成目标代码并再次链接。我们将在第 11 章讨论这些问题。

## 链接器命令语言

每个链接器都有某种形式的命令语言来控制链接过程。最起码链接器需要记录所链接的目标代码和库的列表。通常都会有一大长串可能的选项：在哪里放置调试符号，在哪里使用共享或非共享库，使用哪些可能的输出格式等。多数链接器都允许某些方法来指定被链接代码将要绑定的地址，这在链接一个系统内核或其它没有操作系统控制的程序时就会用到。在支持多个代码和数据段的链接器中，链接器命令语言可以对链接各个段的顺序、需要特殊处理的段和某些应用程序相关的选项进行指定。

有四种常见技术向链接器传送指令：

- 命令行：多数系统都会有命令行（或相似功能的其它程序），通过它可以输入各种文件名和开关选项。这对于 UNIX 和 Windows 链接器是很常用的方法。对于那些命令行长度有限制的系统，常用的办法是让链接器从文件中读取命令并在命令行上那样对待他们。
- 与目标文件混在一起：有些链接器，如 IBM 主机系统的链接器，从一个单个输入文件中接受替换的目标文件及链接器命令。这种方式来源于卡片输入的年代，那时程序员需要把目标代码卡片摞起来和手工打制的命令卡片一起送到读卡器中。
- 嵌入在目标文件中：有一些目标代码格式，特别是微软的，允许将链接器命令嵌入到目标文件中。这就允许编译器将链接一个目标文件时所需要的任何选项通过文件自身来传递。例如 C 编译器将搜索标准 C 库的命令嵌入到文件中（来传递给链接过程）。
- 单独的配置语言：极少有链接器拥有完备的配置语言来控制链接过程。可以处理众多目标文件类型、机器体系架构和地址空间规定的 GNU 链接器，拥有可以让程序员指定段链接顺序、合并相近段规则、段地址和大量其它选项的一套复杂的控制语言。其它链接器一般拥有诸如支持程序员可定义的重叠技术等特性的稍简单一些的配置语言。

## 链接：一个真实的例子

我们已经通过一个简小的链接实例来结束对链接过程的介绍。图 3 所示为一对 C 语言源代码文件，m.c 中的主程序调用了一个名为 a 的例程，而调用了库例程 strlen 和 write 的 a

例程在 a.c 中。

---

图 1-3 源程序

源程序 m.c

```
extern void a(char *);  
int main(int ac,  char **av)  
{  
    static char string[] = "Hello,  world!\n";  
    a(string);  
}
```

源程序 a.c

```
#include <unistd.h>  
#include <string.h>  
void a(char *s)  
{  
    write(1,  s,  strlen(s));  
}
```

---

如图 4 所示，主程序 m.c 在我的 Pentium 机器上用 gcc 编译成一个典型 a.out 目标代码。格式长度为 165 字节的目标文件。该目标文件包含一个固定长度的头部，16 个字节的“文本”段，包含只读的程序代码，16 个字节的数据段，包含字符串。其后是两个重定位项，其中一个标明 pushl 指令将字符串 string 的地址放置在栈上为调用例程 a 作准备，另一个标明 call 指令将控制转移到例程 a。符号表分别导出和导入了\_main 与\_a 的定义，以及调试器需要的其它一系列符号（每一个全局符号都会下划线的前缀，第 5 章中将会讲述原因）。注意由于字符串 string 在同一个文件中，pushl 指令引用了 string 的临时地址 0x10，而由于\_a 的地址是未知的所以 call 指令引用的地址为 0x0。

---

图 1-4 m.o 的目标代码

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
-----	------	------	-----	-----	----------	------

0	.text	00000010	00000000	00000000	00000020	2**3
---	-------	----------	----------	----------	----------	------

1	.data	00000010	00000010	00000010	00000030	2**3
---	-------	----------	----------	----------	----------	------

Disassembly of section .text:

00000000 <\_main>:

0: 55	pushl %ebp
-------	------------

1: 89 e5	movl %esp, %ebp
----------	-----------------

```

3: 68 10 00 00 00 pushl $0x10
4: 32          .data
8: e8 f3 ff ff ff call 0
9: DISP32 _a
d: c9 leave
e: c3 ret
...

```

---

如图 5 所示，子程序文件 a.c 编译成一个长度为 160 字节的目标文件，包括头部， 28 字节的文本段，无数据段。两个重定位项标记了对 strlen 和 write 的 call 指令，符号表中导出\_a 并导入了\_strlen 和\_write。

---

图 1-5 a.c 的目标代码

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn	
0	.text	0000001c	00000000	00000000	00000020	2**2	
			CONTENTS,	ALLOC,	LOAD,	RELOC,	CODE
1	.data	00000000	0000001c	0000001c	0000003c	2**2	
			CONTENTS,	ALLOC,	LOAD,	DATA	

Disassembly of section .text:

```

00000000 <_a>:
0: 55          pushl %ebp
1: 89 e5        movl %esp, %ebp
3: 53          pushl %ebx
4: 8b 5d 08      movl 0x8(%ebp), %ebx
7: 53          pushl %ebx
8: e8 f3 ff ff ff call 0
9: DISP32 _strlen
d: 50          pushl %eax
e: 53          pushl %ebx
f: 6a 01        pushl $0x1
11: e8 ea ff ff ff call 0
12: DISP32 _write
16: 8d 65 fc      leal -4(%ebp), %esp
19: 5b popl       %ebx
1a: c9 leave
1b: c3 ret

```

---

为了产生一个可执行程序，链接器将这两个目标文件，以及一个标准的 C 程序启动初始化例程，和必要的 C 库例程整合到一起，产生一个部分如图 6 所示的可执行文件。

---

图 1-6 可执行程序的部分代码

Sections:

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00000fe0	00001020	00001020	00000020	2**3
1	.data	00001000	00002000	00002000	00001000	2**3
2	.bss	00000000	00003000	00003000	00000000	2**3

Disassembly of section .text:

00001020 <start-c>:

```
...
1092: e8 0d 00 00 00 call 10a4 <_main>
...
```

000010a4 <\_main>:

```
10a4: 55          pushl %ebp
10a5: 89 e5        movl %esp, %ebp
10a7: 68 24 20 00 00 pushl $0x2024
10ac: e8 03 00 00 00 call 10b4 <_a>
10b1: c9          leave
10b2: c3          ret
...
```

000010b4 <\_a>:

```
10b4: 55          pushl %ebp
10b5: 89 e5        movl %esp, %ebp
10b7: 53          pushl %ebx
10b8: 8b 5d 08      movl 0x8(%ebp), %ebx
10bb: 53          pushl %ebx
10bc: e8 37 00 00 00 call 10f8 <_strlen>
10c1: 50          pushl %eax
10c2: 53          pushl %ebx
10c3: 6a 01        pushl $0x1
10c5: e8 a2 00 00 00 call 116c <_write>
10ca: 8d 65 fc      leal -4(%ebp), %esp
10cd: 5b          popl %ebx
10ce: c9          leave
```

```
10cf: c3           ret  
...  
  
000010f8 <_strlen>:  
...  
  
0000116c <_write>:  
...
```

---

链接器将每个输入文件中相应的段合并在一起，故只存在一个合并后的文本段，一个合并后的数据段和一个 bss 段（两个输入文件不会使用的，被初始化为 0 的数据段）。由于每个段都会被填充为 4K 对齐以满足 x86 的页尺寸，因此文本段为 4K（减去文件中 20 字节长度的 a.out 头部，但逻辑上它并不属于该段），数据段和 bss 段每个同样也是 4K 字节。

合并后的文本段包含名为 start-c 的库启动代码，由 m.o 重定位到 0x10a4 的代码，重定位到 0x10b4 的 a.o，以及被重定位到文本段更高地址从 C 库中链接来的例程。数据段，没有显示在这里，按照和文本段相同的顺序包含了合并后的数据段。由于\_main 的代码被重定位到地址 0x10a4，所以这个代码要被修改到 start-c 代码的 call 指令中。在 main 例程内部，对字符串 string 的引用被重定位到 0x2024，这是 string 在数据段最终的位置，并且 call 指令中地址修改为 0x10b4，这是\_a 最终确定的地址。在\_a 内部，对\_strlen 和\_write 的 call 指令也要修改为这两个例程的最终地址。

可执行程序中仍然有很多其它的 C 库例程，没有显示在这里，它们由启动代码和\_write（在稍后例子中的出错处理例程）直接或间接的调用。由于可执行程序的文件格式不是可以重链接的，且操作系统从已知的固定位置加载它，因此它不包含重定位数据。它带有一个有助于调试器（debugger）工作的符号表，尽管这个程序没有使用这个符号表并且可以将其删除以节省空间。

在这个例子中，从库中链接的代码明显要多于程序本身的代码。这是很正常的，尤其当程序使用大的图形库或窗口库，这就促进了共享库的出现，详见第 9 章和第 10 章。这个链接好的程序大小为 8K，但若使用共享库链接则同样的程序大小仅为 264 字节。当然这是一个像玩具一样的例子，但真实程序经常也会采用同样的方法节省空间。

## 练习

将链接器和加载器分成独立的程序有什么好处？在哪些情况下一个整合的链接加载器是有用的？

在过去 50 年中，几乎每个编程系统都包含一个链接器，为什么？

这一章中，我们讨论了链接和加载被汇编或编译后的机器代码。对于一个直接解释源代码的纯解释系统，链接器和加载器是否依然有用？如果一个解释系统直接将源程序变成一种像 P-code 或 Java 虚拟机那样的中间代码呢？

# 第2章 体系结构的问题

\$Revision: 2.3 \$

\$Date: 1999/06/15 03:30:36 \$

链接器和加载器，以及编译器和汇编器，与体系结构的细节密切相关，这包括硬件体系结构和操作系统对目标计算机在体系结构方面的约定。本章中我们将会涉及足够的计算机体系结构知识以理解链接器必须做的工作。本章所有对计算机体系结构的描述是经过考虑而有所删减的，例如浮点和 I/O 这些不影响链接器的部分就去掉了。

硬件体系结构的两个方面影响到链接器：程序寻址和指令格式。链接器需要做的事情之一就是对数据和指令中的地址及偏移量都要进行修改。两种情况下链接器都必须确保所做的修改符合计算机使用的寻址方式；当修改指令时还需要进一步确保修改结果不能是无效指令。

在本章结尾，我们还会讨论地址空间架构，即程序运行时需要处理的地址集合。

## 应用程序二进制接口

每个操作系统都会为运行在该系统下的应用程序提供**应用程序二进制接口**（Application Binary Interface）。ABI 包含了应用程序在这个系统下运行时必须遵守的编程约定。ABI 总是包含一系列的系统调用和使用这些系统调用的方法，以及关于程序可以使用的内存地址和使用机器寄存器的规定。从一个应用程序的角度看，ABI 既是系统架构的一部分也是硬件体系结构的重点，因此只要违反二者之一的条件约束就会导致程序出现严重错误。

在很多情况下，链接器为了遵守 ABI 的约定需要进行一些重要的工作。例如，ABI 要求每个应用程序包含一个该程序中各例程使用的所有静态数据的地址表，链接器通常会通过收集所有链接到程序中的模块的地址信息来创建这个表。ABI 经常会影响链接器的是对标准过程调用的定义，本章稍后会讨论这个话题。

## 内存地址

计算机系统都有主存储器。主存总是表现为一块连续的存储空间，每一个存储位置都有一个数字地址。这个地址从 0 开始，并逐渐增长为某个较大的数字（由地址中的位数决定）。

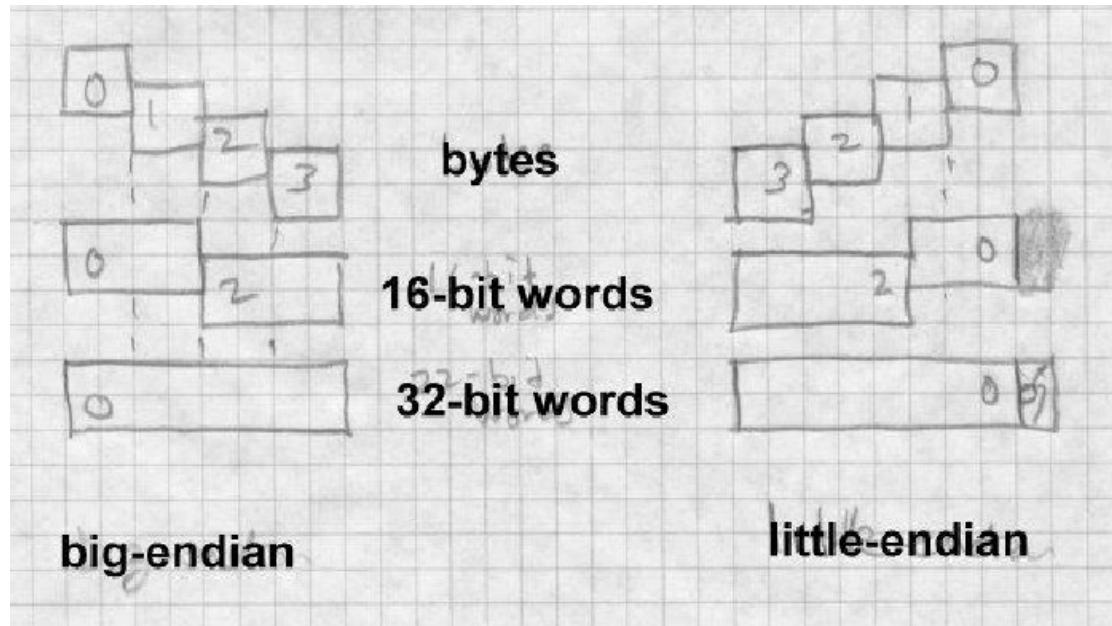
## 字节顺序和对齐

每个存储位置都是由固定数量的数位（bit）组成的。经过将近 50 年的历史，由 64 位到 1 位组成的计算机存储位置都被设计过，但现在几乎所有产品化的计算机都使用 8 位（1 字节）的地址。由于计算机处理的大多数数据，尤其是程序地址，都是大于 8 位的，所以通过将相邻的字节合为一组，计算机同样可以很好的处理 16 位、32 位、64 位或 128 位的数据。

在某些计算机上，尤其是 IBM 和 Motorola，多字节数据的第一个字节（数字地址最低）是高位字节（most significant byte），在其它诸如 DEC 和 Intel 的机器上，第一个字节是低位字节（least significant byte），如图 1 所示。沿用“格列佛游记”中的典故，将 IBM/Motorola 的字节序策略称为 big-endian，而 DEC/Intel 的字节序策略称为 little-endian。

图 2-1 内存地址定位

内存地址的图示



由两种方案的优缺点引起的激烈讨论已经持续了很多年了。由于在两台字节序相同的机器间移植程序要比不同字节序的机器要容易的多，所以实际中对字节序选择的主要考虑来自于对旧系统的兼容。新近的很多芯片设计可以支持任何一种字节序，这可以在芯片布线时进行选择，也可以在系统引导时通过编程选择，甚至某些情况下可以针对每个应用程序进行不同选择。（在这些可切换的芯片上，被加载和存储指令处理的数据的字节序会发生变化，但是被编码到指令中的常量的字节序是不变的，这是一些可以让链接器作者的工作变得很有趣的细节）

多字节数据通常会被对齐到一些“天生”的边界上。就是说，4 字节的数据必须对齐到 4 字节的边界上，2 字节要对齐到 2 字节的边界上，并以此类推。另一种想法就是任何 N 字节数据的地址至少要有  $\log_2(N)$  个低位为 0。在某些系统上（Intel X86, DEC VAX, IBM 370/390），引用未对齐数据会付出性能降低的代价，在另外一些系统上（多数 RISC 芯片），这会导致程序故障。即使在那些引用未对齐数据不会导致故障的系统上，性能的损失也是非常大的，以至于值得我们花费精力来尽可能保持地址的对齐。

很多处理器同样要求程序指令的对齐。多数 RISC 芯片要求指令必须对齐在 4 字节的边界上。

每种体系结构都定义了一系列的寄存器，这是可以由程序指令直接引用的数量很少的固定长度高速存储区域。各种体系结构的寄存器数量是变化的，从 x86 架构的 8 个到某些 RISC 设计的 32 个，寄存器的容量几乎都是和程序地址的大小相同，就是说在一个 32 位地址的

系统中寄存器是 32 位的，而在具有 64 位地址的系统上，寄存器就是 64 位的了。

## 地址构成

当计算机程序执行时，它会根据中程序的指令从内存中读取或向内存。程序的指令本身也存储在内存中，但通常和程序的数据位于内存中不同的部分。

指令在逻辑上是按照存储的顺序被执行的，但通过指定程序中新的地址来执行的跳转指令是例外（有些体系结构会用名词“分支（branch）”来指代某些或者全部的跳转，但是我们在这里把它们都称为跳转）。每个指令中引用的数据内存地址，每个跳转指令引用的地址，要被加载或存储的数据的地址，或指令要跳转到的地址等，计算机们具有一系列的指令格式和地址构成需要链接器在重定位指令中的地址时予以处理。

尽管计算机设计者们在这些年中提出了无数不同而复杂的寻址策略，但现在大多数产品化的计算机都使用一套类似的简单寻址策略（设计者发现很难设计出高速的复杂体系结构，而且编译器也很少能够充分利用复杂寻址特性）。我们以三种架构为例：

- IBM 360/370/390(这里统称为 370)。尽管这是仍在使用的最古老的架构之一，并在过去 35 年中不断增加新特性，但其相对简洁的设计仍然能够很好的工作，并且能够实现在可以与现代 RISC 性能相当的芯片中。
- SPARC V8 和 V9。一个流行的 RISC 架构，具有相当简单的寻址策略。V8 使用 32 位的寄存器和地址，V9 扩充了 64 位的寄存器和地址。SPARC 的设计与其它诸如 MIPS 和 Alpha 这样的 RISC 架构相似。
- Intel 386/486/Pentium(这里通称为 x86)。仍在使用的最无规律和无法理解的架构之一，但不可否认它是最流行的。

## 指令格式

每种体系结构都有一些不同的指令格式。我们将只探讨与程序和数据寻址相关的格式细节，因为这些是影响链接器的主要细节。370 在数据引用和跳转中使用相同的指令格式，SPARC 使用不同的指令格式，而 Intel 的有些格式相同，有些格式不同。

每条指令都包含一个操作码，它决定了指令做什么，此外还有一个操作数。操作数可以被编码到指令本身（立即操作数），或者放置在内存中。内存中每个操作数的地址总要经过一些计算。有时地址包含在指令中（直接寻址）。更经常的是地址存储在某一个寄存器中（寄存器间接寻址），或通过将指令中的一个常量加上寄存器中的内容计算得来。如果寄存器中的值是一个存储区域的地址，而指令中的常量是存储区域中想要访问的数据的偏移量，这种策略称为基址寻址。如果二者调换过来，并且寄存器中保存的是偏移量，那这种策略就是索引寻址。基址寻址与索引寻址之间的区别不那么好定义，而且很多体系结构都将他们混在一起了。例如，370 有一种寻址模式会将两个寄存器和指令中的常量加在一起，并强制的将一个寄存器称作基址寄存器，另一个为索引寄存器，虽然他们都是相同对待的。

还有其它更为复杂的地址计算方法也仍在使用中，但是由于它们不包含链接器需要调整的域，因此链接器的多数组成部分都不需为此担心。

一些体系结构使用固定长度的指令，而另一些使用变长指令。所有的 SPARC 指令都是 4 字节长，并对齐到 4 字节边界。IBM 370 的指令可以是 2、4 或 6 个字节长，指令的第一个字节的头 2 位确定了指令的长度和格式。Intel x86 的指令格式随时都可以是 1 到 14 个字节长。这里的编码方式颇为复杂，部分是由于 x86 最初是为内存受限环境设计的紧凑指令编码，另外也是在 286、386 和后继芯片上的新加指令不得不被硬塞到已存在指令集未使用的位模式中。幸运的是，从链接器作者的角度来看，链接器需要调整的地址和偏移量都是以字节为边界的，所以通常不需要考虑指令编码问题。

## 过程调用和可寻址性

在最早的计算机中，内存很小，指令中的地址域足够容纳计算机任何一个内存位置的地址，现在我们称这种策略为直接寻址。在上世纪 60 年代早期，可寻址内存已经变得相当大使得如果指令集中每个指令都包含整个地址将占用太多仍然宝贵的内存。为了解决这个问题，计算机的架构师们在地址引用指令中部分或彻底的放弃了直接寻址，使用索引和基址寄存器来提供寻址所需的大部分或全部地址位。这可以让指令短一些，但与之而来的代价是编程更复杂了。

在没有采用直接寻址的体系结构中，包括 IBM 370 和 SPARC，程序在进行数据寻址时存在一个“自举”的问题：一个例程要使用寄存器中的基址来计算数据地址，但是将基址从内存中加载到寄存器中的标准方法是从存有另一个基址的寄存器中寻址。

自举问题就是如何在程序开始时将第一个基址载入到寄存器中，随后再确保每一个例程都拥有它需要的基址来寻址它要使用的数据。

## 过程调用

每种 ABI 都通过将硬件定义的调用指令与内存、寄存器的使用约定组合起来定义了一个标准的过程调用序列。硬件的调用指令保存了返回地址（调用执行后的指令地址）并跳转到目标过程。在诸如 x86 这样具有硬件栈的体系结构中返回地址被压入栈中，而在其它体系结构中它会被保存在一个寄存器里，如果必要软件要负责将寄存器中的值保存在内存中。具有栈的体系结构通常都会有一个硬件的返回指令将返回地址推出栈并跳转到该地址，而其它体系结构则使用一个“跳转到寄存器中地址”的指令来返回。

在一个过程的内部，数据寻址可分为 4 类：

- 调用者可以向过程传递参数。
- 本地变量在过程中分配，并在过程返回前释放。
- 本地静态数据保存在内存的固定位置中，并为该过程私有。
- 全局静态数据保存在内存的固定位置中，并可被很多不同过程引用。

为每个过程调用分配的一块栈内存称为“栈框架（stack frame）”。图 2 显示了一个典型的栈框架。

---

图 2-2 栈框架内存布局  
一个栈框架的图示。

---

参数和本地变量通常在栈中分配空间，某一个寄存器可以作为栈指针，它可以基址寄存器来使用。SPARC 和 x86 中使用了该策略的一种比较普遍的变体，在一个过程开始的时候，会从栈指针中加载专门的框架指针或基址指针寄存器。这样就可以在栈中压入可变大小的对象，将栈指针寄存器中的值改变为难以预定的值，当仍使过程的参数和本地变量们仍然位于相对于框架指针在整个过程执行中都不变的固定偏移量处。如果假定栈是从高地址向低地址生长的，而框架指针指向返回地址保存在内存中的位置，那么参数就位于框架指针较小的正偏移量处，本地变量在负偏移量处。由于操作系统通常会在程序启动前为其初始化栈指针，所以程序只需要在将输入压栈或推栈时更新寄存器即可。

对于局部和全局静态数据，编译器可以为一个例程引用的所有静态变量创建一个指针表。如果某个寄存器存有指向这个表的指针，那么例程可以通过使用表指针寄存器将对象在表中的指针读取出来，加载到另一个使用表指针寄存器作为基址的寄存器中，并将第二个寄存器做为基址寄存器来寻址任何想要访问的静态目标。因此，关键技巧是表的地址存入到第一个寄存器中。在 SPARC 上，例程可以通过带有立即操作数的一系列指令来加载表地址，同时在 SPARC 或者 370 上例程可以通过一系列子例程调用指令将程序计数器（保存当前指令地址的寄存器）加载到一个基址寄存器，虽然后面我们还会讨论这种方法在对待库代码时会遇到问题。一个更好的解决方法是将提取表指针的工作交给例程的调用者，因为调用者已经加载了自己的表指针，并可以从自己的表中获取被调用例程的表的指针。

图 3 所示为一个典型的例程调用序列。Rf 是框架指针，Rt 是表指针，Rx 是临时寄存器。调用者将自己的表指针保存到自己的栈框架中，然后将被调用例程的地址和它的指针表地址载入到寄存器中，再进行调用。被调用的例程可以通过 Rt 中的表指针找到它需要的所有数据，包括它随后要调用的例程的地址和表指针。

---

图 2-3 理想的调用过程

... 将参数压入堆栈 ...

```
store Rt      xxx(Rf) ; save caller's table pointer in caller's stack frame
load Rx      MMM(Rt) ; load address of called routine into temp register
load Rt      NNN(Rt) ; load called routine's table pointer
call (Rx) ; call routine at address in Rx
load Rt      xxx(Rf) ; restore caller's table pointer
```

---

有一些优化方法经常是可能有用的。很多情况下，在一个模块中的所有例程会共享一个指针表，这时模块内的调用不需要改变表指针。SPARC 的约定是整个模块共享一个由链接器创建的表，这样表指针寄存器可以在模块内调用时保持不变。同一模块内的调用可以通过一个将被调用例程的偏移量编码到指令中的调用指令实现，这就不再需要再将被调用例程的地址加载到寄存器中了。在所有这些优化中，同一模块中对某个例程的调用序列缩减为一个单

独的调用指令。

又回到地址自举的问题了，这个表指针的链最初是怎么开始的呢？如果每一个例程都从前面例程中获取它的表指针，那么最初的例程从哪里获得呢？答案不是固定的，但是总会涉及到一些特殊代码。主例程的表可能存储在一个固定的位置，或初始指针值被标注在可执行文件中这样操作系统可以在程序开始前加载它。无论使用的是什么技术，都是需要链接器的帮助的。

## 数据和指令引用

我们再更具体的看看程序在我们讨论的 3 种体系结构中寻址数据的方法。

### IBM 370

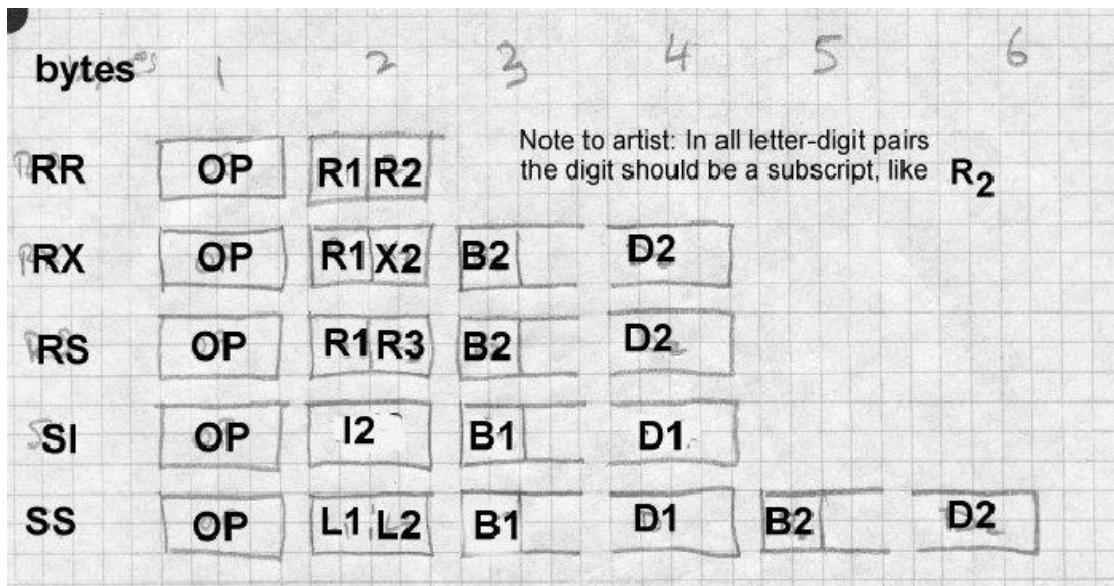
60 年代的老式 System/360 使用一种非常简洁的数据寻址策略，随着它发展到 370 和 390，就逐渐变得更加复杂了。每个引用数据内存的指令通过将指令中 12 位无符号的偏移量与基址寄存器（也可能是索引寄存器）相加来计算地址。有 16 个通用寄存器，每个均为 32 位，编号从 0 到 15。除了一个之外其余的都可以用作索引寄存器。如果寄存器 0 在地址计算中被指定，那么将使用数值 0 而非寄存器本身的内容（寄存器 0 存在且可以用于计算，但不能用于寻址）。在那些需要从寄存器中读取跳转目标地址的指令而言，寄存器 0 意味着不跳转。

图 4 所示为主要的指令格式。RX 指令包含一个寄存器操作数和一个内存操作数，后者的地址通过将指令中的偏移量与基址寄存器和索引寄存器相加得来。多数情况下索引寄存器的数值为 0，这时地址恰好是基址加偏移量。在 RS，SI 和 SS 格式中，12 位的偏移量会与基址寄存器相加。RS 指令有一个内存操作数，此外还有一到两个操作数在寄存器中。SI 指令有一个内存操作数，另一个操作数是在指令自身中的 8 位立即数。SS 指令有两个内存操作数，所以是存储到存储的操作。RR 格式有两个寄存器操作数而没有内存操作数，虽然一些 RR 指令会将一个或所有的寄存器解释为指向内存的指针。370 和 390 在这些格式上稍加了一点变化，但数据寻址格式没有变化。

---

图 2-4 IBM 370 指令格式

IBM 的 RX，RS，SI，SS 指令格式的图示



指令可以通过将基寄存器设置为 0 来直接寻址内存中最低的 4096 个位置。这是底层系统软件所具有的能力，但是从不在应用软件中使用，因为它们都要使用基址寄存器寻址。

注意在所有的三类指令格式中，12 位的偏移量永远存储在一个 16 位对齐的半个字的低 12 位中。这就可以在修改目标文件中特定的地址偏移量时不需考虑任何对指令格式的引用，因为偏移量格式都是相同的。

最初的 360 系统采用 24 位寻址，地址保存在内存或寄存器的 32 位字的低 24 位，忽略高 8 位。370 扩展到 31 位寻址。不幸的是，包括最流行的操作系统 OS/360 在内的很多程序，都使用内存中的 32 位地址字的高字节来存放标志或其它数据，这就不可能在仍然支持现存目标代码的同时将按先前的方法将寻址能力扩展到 32 位。作为替换方案，系统具有 24 位和 31 位的模式，并且在任何时刻 CPU 都可以解释 24 位或 31 位的地址。一个由软硬件结合的规定指明，如果一个地址字的高位置 1，则该地址字其它各位保存的是 31 位地址；如果高位置 0，则该地址保存 24 位地址。由于依赖于某个特定例程是多久以前写的，程序可能而且肯定会进行模式切换，因此链接器必须能够同时处理 24 位和 31 位地址。由于历史原因，早期 360 产品线的低端型号中经常只有 64K 或更小的内存而且程序使用加载和存储半个字的指令来操作地址值，所有 370 的链接器也会处理 16 位地址。

370 和 390 的后期型号增加了与 x86 系列相似的分段地址空间。该特性可以让操作系统通过非常复杂的访问控制和地址空间切换规则来定义可供程序寻址的多个 31 位地址空间。据我所知，目前还没有编译器或链接器可以支持这些特性，该特性主要引用在高性能数据库系统上，所以我们今后将不对其进一步讨论。

370 的指令寻址也是相当简洁的。在最初的 360 中，跳转指令（经常是指分支指令）都是 RR 或 RX 格式的。在 RR 格式的跳转中，第二个寄存器操作数存有跳转目标，如果是寄存器 0 意味着不进行跳转。在 RX 格式的跳转中，内存操作数是跳转的目标。过程调用是“分支”和“链接”（对 31 位寻址会被稍后的“分支”和“存储”所取代），它将返回地址存储在特定的寄存器，若是 RR 格式则跳转到第二个寄存器中的地址，若是 RX 格式则跳转到第二个操作数中的地址。

对于一个例程内的跳转，例程需要建立“可寻址性”，即一个可供 RX 指令使用的指向（或至少接近）例程开始位置的基址指针寄存器。按照约定，寄存器 15 保存着例程的入口地址，可被当作基址指针使用。“分支和链接”或者“分支和存储”且第二个寄存器为寄存器 0 的 RR 格式跳转，会将后继指令的地址保存在第一个操作数寄存器中但不进行跳转，如果先前寄存器的内容是未知的那么这可以用来设置基址寄存器。鉴于 RX 指令具有 12 位的偏移量域，一个单独的基址寄存器可以负责 4K 的内存块。如果一个例程的大小超过这个，那就需要多个基址寄存器来覆盖该例程所有的代码。

390 为这些跳转都增加了相对地址的格式。在这些新格式的指令中，包含一个需要被逻辑上左移一位的（由于指令都是偶数字节对齐的）有符号 16 位偏移量，将其加上该指令的地址就可以得到跳转目标的地址。这种新的格式不需要寄存器来计算地址，并且允许在 $+/-6$  4K 内的跳转，除了最大的那些例程外应付例程内跳转是足够了。

## SPARC

SPARC 就像它的名字一样是精简指令集的处理器，由于到现在这种架构已经发展了九个版本了，最初的简单设计到现在已经变得有些复杂了。SPARC 直到 V8 版本都是 32 位架构，SPARC V9 扩展为 64 位架构。

### *SPARC V8*

SPARC 有四种主要指令格式和 31 种次要指令格式。图 5 所示为 4 种跳转格式和 2 种数据寻址模式。

在 SPARC V8 中有 31 个通用寄存器，每个 32 位，从 1 到 31 编号。寄存器 0 是一个数值总为 0 的伪寄存器。

这里会采用一种不常见的“寄存器窗口”策略来试图将过程调用和返回过程中寄存器存储和恢复的次数最小化。窗口策略对于链接器几乎没有影响，所以将不再对其进行讨论（寄存器窗口是针对 SPARC 所继承的 Berkely RISC 设计的）。

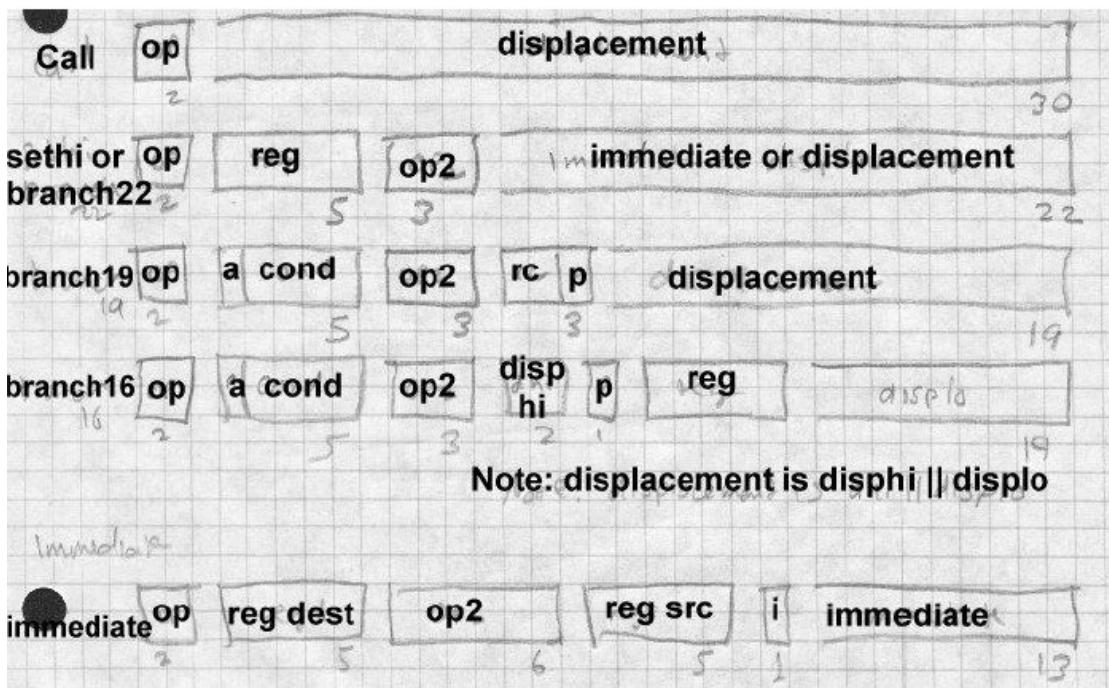
数据引用会使用两种寻址模式中的二者之一。一种模式通过将两个寄存器的数值加在一起计算地址（如果一个寄存器中已经有想要的目的地址，那么另一个寄存器可以是寄存器 0）。另一种模式会将指令中的 13 位有符号偏移量与一个基址寄存器相加。

SPARC 的汇编器和链接器通过一个双指令序列支持一种伪直接寻址策略。这两个指令是 SETHI，它将一个 22 位的立即数装入一个寄存器的高 22 位并使其低 10 位为 0，接着是一个 R 指令将自己的 13 位立即数通过“或运算”加入到寄存器的低 13 位。汇编器和链接器会设法将 32 位寄存器的高部分和低部分放入这两个指令中。

---

图 2-5 SPARC

30 位调用 22 位分支和 SETHI 19 位分支 16 位分支（仅用于 V9），R+R 操作和 R+I13 操作。



过程调用指令和多数条件跳转指令（在 SPARC 的术语中称为分支）使用分支偏移量从 6 位到 30 位不等的相对寻址。无论偏移量大小是多少，跳转指令会将偏移量左移 2 位（因为所有的指令地址都是 4 字节对齐的），然后将结果带符号的扩展为 32 位或 64 位，将这个数值加上跳转或调用指令的地址就可以得到目标地址。调用指令 call 在这里使用 30 位的偏移量，这意味着它可以到达 32 位 V8 地址空间的任何地址。调用指令将返回地址保存在寄存器 15。不同的跳转指令会使用 16、19 或 22 位的偏移量，这对任何大小合乎常理的例程是足够了。16 位偏移量格式的跳转指令会将偏移量分割为高的 2 位和低的 14 位存放在指令字的不同部分，但是这不会给链接器带来任何大麻烦。

SPARC 也有一个“跳转和链接”指令，可以按照与数据引用指令相同的方法计算目标地址，即将两个源寄存器或者一个源寄存器和一个常量偏移量加在一起。它同样可以将返回地址保存在某个目标寄存器中。

过程调用可以使用 Call 指令或“跳转和链接”指令，这会将返回地址保存在寄存器 15，然后跳转到目标地址。过程返回使用 JMP8[r15]，以返回 call 后面的 2 个指令的位置（SPARC 的调用和跳转指令是“被延迟”的，可以在跳转或调用之前选择执行后面的指令）。

### SPARC V9

SPARC V9 将所有的寄存器扩展为 64 位，使用寄存器的低 32 位来兼容旧的 32 位程序。所有已存在的指令仍然按以前的方式工作，除了寄存器已经由 32 位扩展为 64 位了。新的加载和存储指令处理的是 64 位数据，新的分支指令可以测试前一个指令的 32 位或 64 位结果。SPARC V9 没有为合成一个全 64 位地址增加新的指令，也没有增加新的调用指令。完整的 64 位地址可以通过一个冗长的过程来生成：使用 SETHI 和 OR 从独立的寄存器中创建两个 32 位的半地址，然后将高位的 32 位向左移，并通过 OR 操作将高位和低位两部分合在一起。实际上，64 位地址是从指针表中加载的，模块内的调用会从表中加载目标例程的地址到某个寄存器，然后使用“跳转和链接”来实施调用。

## Intel x86

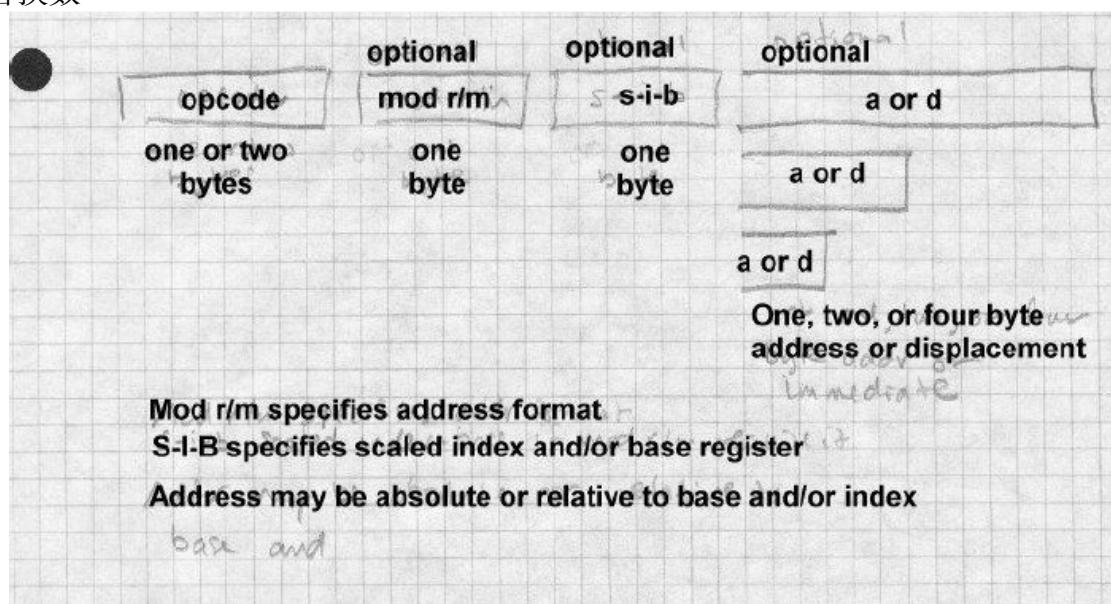
Intel x86 是目前我们讨论的三种体系结构中最复杂的。它具有一个不对称的指令集和分段的地址空间。有 6 个 32 位通用寄存器名为 EAX、EBX、ECX、EDX、ESI 和 EDI，同时还有两个主要用于寻址的寄存器，EBP 和 ESP，六个专门的 16 位段寄存器 CS、DS、ES、FS、GS 和 SS。每个 32 位的寄存器的低半部可以当做 16 位寄存器使用，称为 AX、BX、CX、DX、SI、DI、BP 和 SP。从 AX 到 DX 寄存器的低 8 位和高 8 位又可以作为 8 位寄存器，称为 AL、AH、BL、BH、CL、CH、DL 和 DH。在 8086、186 和 286 中，很多指令需要操作数放在特定的寄存器中，但在 386 和之后的芯片中，多数（并不是全部）需要特定寄存器的指令都被统一为可以使用任何寄存器。ESP 是硬件栈指针，总是保存着当前栈的地址。EBP 通常做为框架寄存器来使用，指向当前栈框架的基址。（指令集建议但不要求这样）

任何时候 x86 处理器都会运行在以下三种模式之一：模仿最初 16 位 8086 的实模式，286 中加入的 16 位保护模式，386 中加入的 32 位保护模式。保护模式中涉及到 x86 声名狼藉的段机制，但我们这里对其暂不考虑。

多数对内存中数据地址进行寻址的指令都会使用一个通用的指令格式，如图 2-6 所示（那些不使用特定架构定义的寄存器的指令，例如 PUSH 和 POP 指令总是使用 ESP 来对栈寻址）。通过将一个指令中有符号的 1、2 或 4 字节的可替换数、一个可以为任意 32 位寄存器的基址寄存器和一个可选的可以使用除 ESP 外任意 32 位寄存器的索引寄存器，这三者中的任一个或者全部加起来，可以计算得到地址。索引可以逻辑左移 0、1、2 或 3 位，这样对多字节值数组的索引会更容易一些。

图 2-6 一般 x86 指令格式

一个或两个指令码字节，可选的 R/M 模字节，可选的 s-i-b 字节，可选的 1、2 或 4 字节可替换数



虽然将可替换数、基址和索引都放入一个指令中是可能的，但是直接寻址时多数只使

用一个 32 位的可替换数，或在栈寻址和指针引用时跟一到两个字节可替换数一起使用基址。从链接器的观点看，直接寻址允许指令或数据的地址被嵌入到程序的任何 1 字节对齐的位置。

条件跳转、无条件跳转或例程调用通常都使用相对寻址。任何的跳转指令都可以具有 1、2 或 4 字节的偏移量可以将其与该指令后一条指令的地址相加得到跳转目的地址。调用指令或者包含一个 4 字节的绝对地址，或者使用任何一种通常的寻址模式来引用存有目标地址的内存。这就允许跳转和调用指令出现在当前 32 位地址空间的任何位置。无条件跳转和调用指令也可以使用上面描述的计算完全数据地址的方法来计算目标地址，这多数会在对存储在寄存器中的地址进行跳转或调用时被用到。调用指令会将返回地址压入由 ESP 指向的栈中。

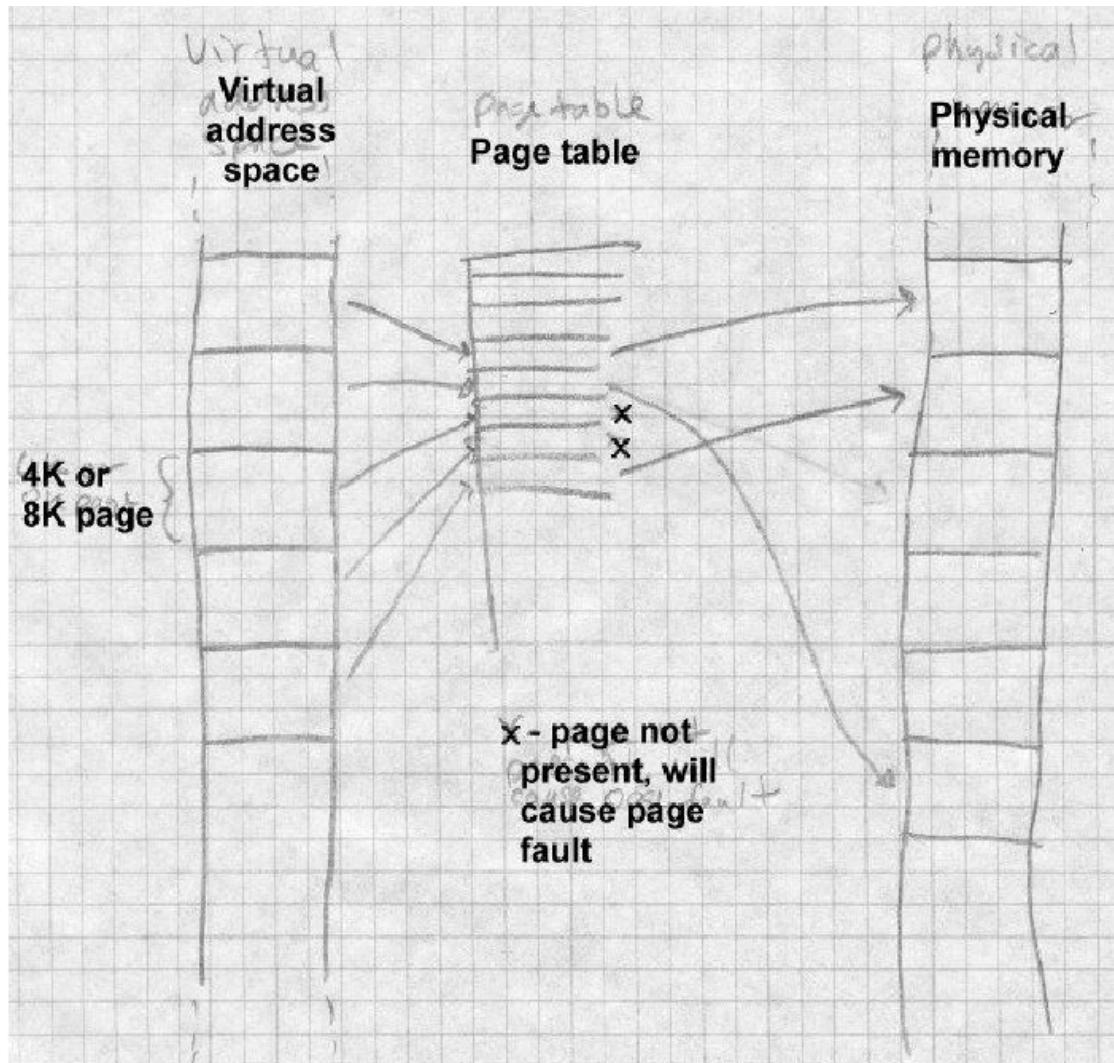
无条件跳转和调用也可以在指令中使用完全 6 字节的“段/偏移量”地址，或计算“段/偏移量”地址中所存储的地址。这些调用指令会将返回地址和调用者的段号都压入栈中，这就可以进行段间调用和返回。

## 分页和虚拟内存

在多数现代计算机系统中，每个程序都可以寻址数量巨大的内存，在一个典型的 32 位系统中这通常是 4GB。很少有机器有那么大的内存，即使有它也需要将其在多个程序之间共享。分页硬件将一个程序的地址空间划分为大小固定的页，典型的大小是 2K 或 4K，同时将计算机的物理内存划分为同样大小的页框。硬件包含了由地址空间中各个页对应的页表项组成的多个页表，如图 7 所示。

---

图 2-7 页映射  
将页和实际页框通过一个大页表对应起来



一个页表项可以包含针对某个页的实际内存页框，或通过标志位标注该页“不存在”。当应用程序尝试使用一个不存在的页时，硬件会产生一个由操作系统处理的“页失效”错误。操作系统可以将页的内容从磁盘上复制到一个空闲的内存页框中，并让应用程序继续运行。通过按需将页在内存和磁盘之间移动，操作系统可以提供“虚拟内存”的功能，这样从应用程序看来使用的是比实际大的多的内存。

但是虚拟内存也会带来开销。执行一条指令只需要不到1微妙的时间，但由于页失效导致随后的调入或调出页操作（将页从磁盘传送到主存，或相反的过程）由于涉及到磁盘传输所以需要若干毫秒。应用程序产生的页失效越多，它就运行的越慢，最坏的情况会导致“页抖动”，这时页失效对程序的有效运行没有任何帮助。程序使用的页越少，它可能产生的页失效也就越少。如果链接器可以将有关联的例程挤压到一个页或者少量的几个页，就会提高分页的性能。

如果页可以被标注为只读，那么也会提升性能。由于只读页可以重新加载因此它们不需要调出页的操作。如果某个页逻辑上出现在多个地址空间中（这通常会在运行相同程序的多个实例时），一个单独的物理页就可以满足所有的地址空间。

对于32位寻址和使用4K页的x86，需要一个具有 $2^{20}$ 个项的页表来覆盖整个地址空间。由于每个页表项通常为4字节，这会使页表的大小变成不切实际的4MB。因此，可分页的架

构会通过将高层次页表指向那些最终映射到虚拟地址所对应的物理页框的低层次页表来实现对页表的再次分页。在 370 上，高层次页表（被称为段表）的每一项映射 1MB 的地址空间，这样段表在 31 位地址模式时可以包含 2048 项。如果整个段都不存在的话，那么段表中的每一项都可以是空，否则就会指向将页映射到那个段上的低层次页表。每一个低层次页表共有 256 个页表项，每一个对应段中 4K 的内存块。虽然对齐的边界略有差别，但 x86 使用类似的方式划分它的页表。每一个高层次页表（称为目录）映射 4MB 的地址空间，这样高层次页表共有 1024 项。每一个低层次的页表同样包含 1024 项去映射和该页表对应的 4MB 地址空间中的 1024 个 4K 页。SPARC 架构将页大小定义为 4K，并有三级页表而非两级。

两级或三级的页表对应用程序是透明的，但有一个重要的例外：操作系统可以通过修改高层次页表的某一项改变对一大块地址空间（在 370 上是 1MB，在 x86 上是 4MB，在 SPARC 上是 256KB 或 16MB）的映射，因此由于效率的原因，地址空间经常通过替换单独的第二级页表项来按照这个尺寸倍数来管理，而不是在进程切换时重新加载整个页表。

## 程序地址空间

每个程序都运行在一个由计算机硬件和操作系统共同定义的地址空间中。链接器和加载器需要生成与这个地址空间匹配的可运行程序。

最简单的地址空间是由 PDP-11 版本的 UNIX 提供的。该地址空间为从 0 开始的 64K 字节。程序的只读代码从位置 0 加载，可读写的数据跟在代码的后面。PDP-11 具有 8K 的页，所以数据从代码后 8K 对齐的地方开始。栈向下生长，从 64K-1 的地方开始，随着栈和数据的增长，对应的区域会变大：当它们相遇时程序就没有可用的地址空间了。接着 PDP-11 出现的 VAX 版本的 UNIX，使用了相似的策略。每一个 VAX 的 UNIX 程序的头两个字节都是 0（这是一个表明不保存任何东西的寄存器保存掩码）。因此，一个全 0 的空指针总是有效的，并且如果一个 C 程序将空值作为一个字串指针，那么位置 0 的零字节将会当作空字串对待。由于这个原因，上世纪 80 年代的 UNIX 由于空指针的原因包含有很多难以发现的 bug，经过多年后，由于易于发现和修正所有的空指针 bug，移植到其它体系结构上的 UNIX 在位置 0 提供了零字节。

Unix 系统将每个程序都放置在单独的地址空间中，而操作系统运行在与应用程序在逻辑上隔离的地址空间中。那些将多个程序放在相同地址空间的操作系统，由于程序的实际加载地址只有在程序运行时才能确定，因此就使得链接器和加载器（尤其是加载器）的工作更为复杂。

x86 上的 MS-DOS 系统不使用硬件保护，所以系统和应用程序共享同一个地址空间。当系统运行一个程序的时，它会查找最大的空闲内存块（可能会位于地址空间的任何位置），将程序加载到其中，然后运行它。IBM 的大型主机操作系统所做的跟这差不多，也是将程序加载到有效地址空间的可用内存块中。在这两种情况下，要么是程序加载器，要么某些时候是程序本身需要调整程序被加载的位置。

MS Windows 采用了一种特殊的加载策略。每个程序按照被加载到一个标准开始地址的方式来链接，但是在可执行程序中带有重定位信息。当 Windows 加载这个程序时，如果可能

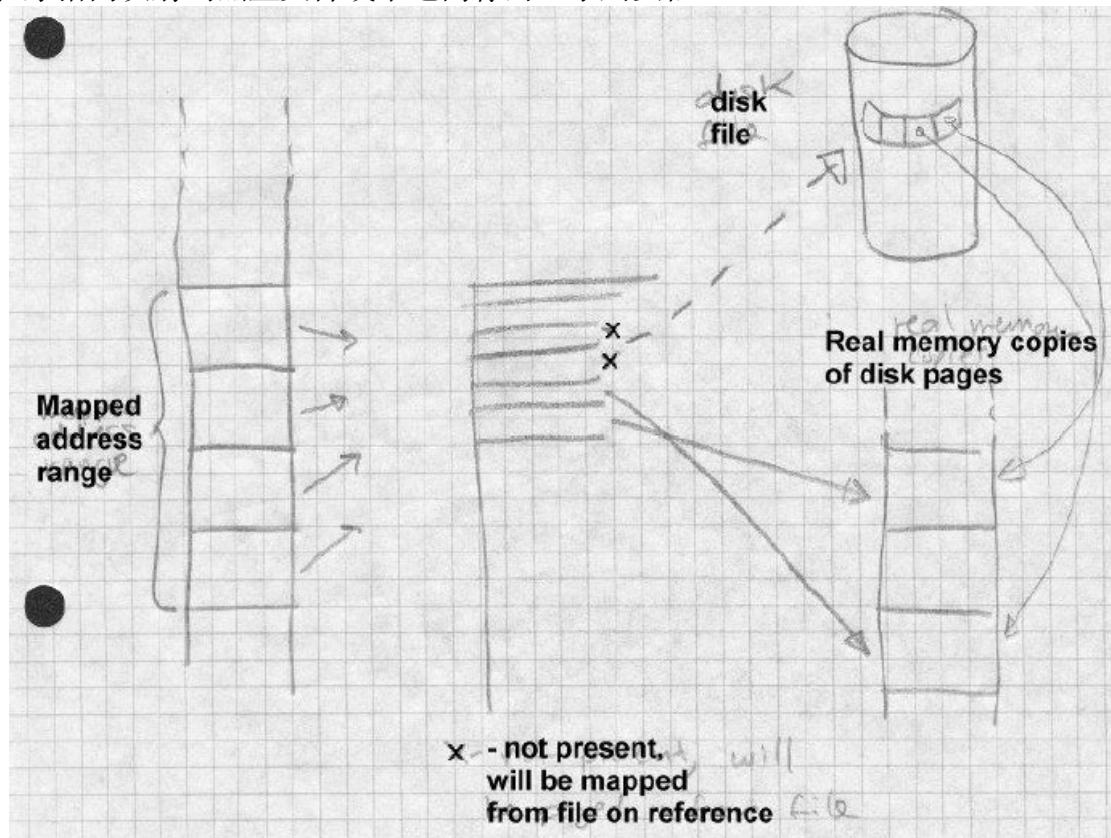
的话它就将程序放置在这个起始地址处，但如果这个地址不可用那就会将它加载到其它地方。

## 映射文件

虚拟内存系统在真实内存和硬盘之间来回移动数据，当数据无法保存在内存中时就会将它交换到磁盘上。最初，交换出来的页面都是保存在独立于文件系统名字空间的单独匿名磁盘空间上的。换页发明之后不久，设计者们发现通过让换页系统读写命名的磁盘文件可以将换页系统和文件系统统一起来。当一个应用程序将一个文件映射到程序的部分地址空间时，操作系统将那部分地址空间对应的页设置为“不存在”，然后将该文件像这部分地址空间对应的页交换磁盘那样来使用，如图 8 所示。程序可以通过引用这部分地址空间的方法来读取文件，这时换页系统会从磁盘加载所需的页。

图 2-8 映射文件

程序指向映射到磁盘文件或本地内存的一系列页框



处理对映射文件的写操作有三种不同的方法。最简单的办法是将文件以只读方式 (RO) 映射，任何对映射文件存储数据的操作都会失败，这通常会导致程序终止。第二种方法是将文件以可读写方式 (RW) 映射，这样对映射文件在内存中副本的修改会在取消映射的时候写回磁盘上。第三种方法是将文件以写时复制方式 (COW) 映射。这种情况下操作系统会对该页面做一个副本，这个副本会被当作没有映射的私有页来对待。在应用程序看来，由于本程序所做的修改仅对自己可见而对其它程序不可见，因此以 COW 的方式映射文件与分配一块匿

名的新内存并将文件内容读入其中很类似。

## 共享库和程序

在几乎所有能够同时运行多个程序的系统中，每个程序都有一套独立的页面，使各自都有一个逻辑上独立的地址空间。由于错误或恶意的程序无法破坏或窃取其它程序信息，这就使得系统更加的健壮，但也会带来性能问题。如果单一的程序或单一的程序库在多于一个的地址空间中被使用，若能够在多个地址空间中共享这个程序或程序库的单一副本，那将节省大量的内存。对于操作系统实现这个功能是相当简捷的——只需要将可执行程序文件映射到每一个程序的地址空间即可。不可重定位的代码和只读的数据以 R0 方式映射，可写的数据以 COW 方式映射。操作系统还可以让所有映射到该文件的进程之间共享 R0 和尚未被写的 COW 数据对应的物理页框（如果代码在加载时需要重定位，重定位过程会修改代码页，那他们就必须被当作 COW 对待，而不是 R0）。

要完成这种共享工作需要链接器予以相当多的支持。在可执行程序中，链接器需要将所有的可执行代码聚集起来形成文件中可以被映射为 R0 的部分，而数据是可以被映射为 COW 的另一部分。每一个段的开始地址都需要以页边界对齐，这既针对逻辑上的地址空间也包括实际的被映射文件。当多个不同程序使用一个共享库时，链接器需要做标记，好让程序启动时共享库可以被映射到它们各自的地址空间中。

## 位置无关代码

当一个程序在多个不同的地址空间运行时，操作系统通常可以将程序加载到各地址空间的相同位置。这样可以让链接器将程序中所有的地址绑定到固定的位置且在程序加载时不需要进行重定位，因此链接器的工作简单了很多。

共享库使情况变得相当复杂。在一些简单的共享库设计中，每一个库会在系统引导时或库被建立时分配一个全局唯一的内存地址。这可以让每一个库放置在固定的位置上，但由于库内存地址的全局列表需要由系统管理员维护这就给共享库的管理带来了严重的瓶颈。再进一步，如果一个库的新版本比之前的版本尺寸大且无法保存在先前分配的位置，那么整个的共享库，以及引用这些库的程序都需要被重新链接。

有一个替代的办法就是允许不同的程序将库映射到各自地址空间的不同位置。这会使库的管理容易一些，但是这需要编译器、链接器和程序加载器的配置，好让库可以在工作的时候忽略掉它被加载到地址空间的什么位置。

一个简单的方法是在库中包含标准的重定位信息，在库被映射到各个地址空间时，加载器可以修改程序中的任何重定位地址以反映库被加载的位置。不幸的是，修改的过程会导致对库的代码和数据的修改，这意味着若它是按照 COW 方式映射的则对应的页不能再被共享，或它是按照 R0 方式映射的则会导致程序的崩溃。

为了避免这种情况，共享库使用了位置无关代码（PIC: Position Independnet Code），这是无论被加载到内存中的任何位置都可以正常工作的代码。共享库中的代码通常都是位置

无关代码，这样代码可以以 R0 方式映射。数据页仍然带有需要被重定位的指针，但由于数据页将以 COW 方式映射，因此这里对共享不会有损失。

对于大部分计算机系统，位置无关代码是非常容易创建的。本章中讨论的三种体系结构都使用相对跳转，因此例程中的跳转指令无需重定位。对栈上的本地数据引用是基于基址寄存器的相对寻址，因此也不需重定位。仅有的挑战在于对那些不在共享库中的例程的调用，以及对全局数据的引用。直接数据寻址和 SPARC 的高位/低位寄存器加载技术是不能使用的，因为他们都需要运行时重定位。幸运的是，还有很多方法可以用来处理库间调用和全局数据引用。当我们在第 9 章和第 10 章讨论共享库的时候再对其详细讨论。

## Intel 386 分段

本章最后的话题是关于 Intel 架构中声名狼藉的分段系统。除了一些遗留下来的 ex-Burrroughs Unisys 大型主机系统外，x86 系列是唯一仍在普遍使用的分段架构。但是由于它非常的流行，我们不得不处理它。但是就像我们将要简要讨论的那样，32 位操作系统不怎么使用分段机制，在老一些的系统和流行的 16 位嵌入式 x86 系列中分段机制被广泛的引用。

最初的 8086 是 Intel 颇为流行的 8 位 8080 和 8085 微处理器的后继版本。8080 具有 16 位地址空间，这使得 8086 的设计者在保持对 16 位地址空间的兼容和提供更大的地址空间之间进退维谷。他们最终妥协了，方法是提供多个 16 位的地址空间。每个 16 位的地址空间就是我们知道的段。

一个运行中的 x86 程序由四个段寄存器定义了四个活动段。CS 寄存器定义了代码段，用来指示指令获取的位置。DS 寄存器定义了数据段，用来指示多数数据被读取和存储的位置。SS 寄存器定义了栈段，这对 PUSH 和 POP 指令的操作数是有用的，程序的返回地址会通过 call 和 return 指令被压入和弹出栈中，以 EBP 或 ESP 为基址寄存器可以完成对任何数据的引用。ES 寄存器定义了扩展段，会被一些新的字串操作指令所用到。386 和之后的芯片还定义了 2 个新的段寄存器 FS 和 GS。通过段覆盖，对任何数据的引用都可以定向到某一个特定的段。例如，指令 MOV EAX, CS:TEMP 可以从代码段（而不是数据段）的位置 TEMP 获取一个数值。FS 和 GS 段仅在段“覆盖（segment override）”时使用。

段值可以有相同的。多数程序会将 DS 和 SS 设置为相同的数值，这样对栈变量和全局变量的指针可以交换使用。有一些小程序会将四个段寄存器设置成相同的值，这样可以提供我们所知道被称为“微小（tiny）”模式的地址空间。

在 8086 和 186 上，体系结构定义了一种段数值与内存地址之间的固定映射方法，即将段数值左移 4 位。例如段数值为 0x123 的段从内存地址 0x1230 处开始。这种简单的寻址方式也被称为“实模式”。程序员经常将一个段可以寻址的 16 字节的内存单元非正式的称为“段落（paragraphs）”。

286 引入了保护模式，操作系统可以将段映射到实际内存的任何位置，并可以通过将段标注为“不存在”而实现基于段的虚拟内存。每个段可以被标识为可执行、可读或可读写，以提供段级的保护机制。386 将保护模式扩展到 32 位，这样每个段最大可以到 4GB 而不是仅有的 64KB。

对于 16 位寻址，除了最小的程序外其它的都要处理分段的地址。修改段寄存器中的内

容是比较慢的，在 486 上相比与修改通用寄存器内容所需的 1 个时钟周期，修改段寄存器需要 9 个时钟周期。因此，程序和程序员都费了很大的周折将数据和代码挤入尽可能少的段中以尽量避免更换段寄存器中的内容。链接器通过提供“聚集”功能将相关的代码或数据收集到一个段中，来为这个过程提供帮助。代码和数据的指针，可以是仅使用偏移量的近 (near) 类型，也可以是同时需要段和偏移量的远 (far) 类型。

编译器可以为那些决定代码或数据地址缺省是近类型还是远类型的多种内存模式生成对应的代码。小 (small) 模式代码中所有的指针都是近类型，且仅有一个代码段和数据段。大 (large) 模式代码有多个代码和数据段，所有的指针缺省都是远类型。编写有效分段的代码是很有技巧的，而且已经有其它文档很好的对其阐述了。

分段的寻址会对链接器提出重要的需求。程序中的每一个地址都要有一个段址和段内偏移量。目标文件含有多个被链接器装入各种段中的代码块。运行在实模式下的程序必须将程序内的段进行标注好让程序被加载时被重定位到实际的段位置。在保护模式下执行的程序更需要标注出数据将被加载到哪个段且对每个段采取怎样的保护（代码，只读数据，可读写数据）。

虽然 386 同时支持 32 位段和 286 的 16 位段的所有特性，但多数 32 位程序根本就不使用段。386 中也加入了分页机制，可以提供分段机制的多数实用优点，并且没有性能损失，也没有编写额外段操作代码的麻烦。多数 386 操作系统在微小 (tiny) 模式下运行应用程序，由于 386 下的段已经不再那么小了所以他有一个更为人知的名字叫做扁平 (flat) 模式。他们会创建单独的代码段和数据段，每个都有 4GB 长并且都映射到完整的 32 位分页地址空间上。即使应用程序只使用一个段，这个段也可以有整个地址空间这么大。

386 可以在一个程序内同时使用 16 位和 32 位的段，对少数操作系统也可以做到，例如 Windows95 和 Windows98 就利用了这个功能。Windows95 和 Windows98 可以在一个共享的 16 位地址空间内运行旧的 Windows3.1 代码，同时新的 32 位程序运行在各自的微小模式地址空间中，并将 16 位地址空间映射过来以允许相互调用。

## 嵌入式体系结构

嵌入式系统中的链接会遇到多种在其它环境中很少遇到的问题。虽然嵌入式芯片的内存容量和性能都很有限，但由于嵌入在芯片中的嵌入式程序会安装到成千上万的设备上，因此在尽可能小的内存容量下让程序跑的尽可能快是非常重要的。一些嵌入式系统会使用通用芯片的低成本型号，例如 80186，同样也有使用诸如 Motorola56000 系列 DSP（数字信号处理器）这样专用处理器的系统。

## 怪异的地址空间

嵌入式系统具有很小且分布怪异的地址空间。一个 64K 的地址空间可能会包括高速的片内 ROM 和 RAM，低速的片外 ROM 和 RAM，片内外围设备，或片外外围设备。也可能会存在多个不连续的 ROM 或 RAM 区域。56000 系统具有 3 个 24 位字的 64K 地址空间，每一个都是由 RAM、

ROM 和外围设备组成的。

嵌入式芯片的开发都会使用带有配合处理器芯片的支持逻辑和芯片的系统开发板。即使对相同处理器的不同开发板也会具有不同的内存布局。不同型号的芯片具有不同容量的 RAM 和 ROM，所以程序员需要在努力将程序挤入更小的内存（成本更低）和使用内存容量更大的芯片（成本更高）之间进行选择。

嵌入式系统的链接器需要有办法来指明被链接程序在内存布局上的大量细节，分配特定类型的代码和数据，甚至将例程和变量分开放入特定的地址。

## 非一致性内存

对片上内存的引用要比片外内存快很多，因此在同时具有两类内存的系统中，对时间要求最严格的程序需要放在快的内存中。有时候，在链接时将程序的所有对时间敏感的代码放入快速内存是可能的。但此外将数据或代码从慢速内存复制到快速内存也是很有用的，这样多个例程可以在不同时间中共享快速内存。对于这种技巧，如果能够告诉链接器“将这段代码放在位置 XXXX 但将它像在位置 YYYY 那样链接”那将是非常有用的，这样就可以在将代码从低速内存的 XXXX 位置复制到高速内存的 YYYY 位置后程序不会出错了。

## 内存对齐

DSP 对某些的数据结构有非常严格的内存对齐要求。例如在 56000 系列上，有一种可以非常有效的处理循环缓冲区的寻址模式，但需要缓冲区的基址要对齐在至少有缓冲区那么大的 2 的幂次大小的边界上（例如对于 50 个字大小的缓冲区就需要对齐在 64 字节的边界上）。快速傅立叶变化（FFT），一个在信号处理中极其重要的运算，是依赖于地址位操作的，这也同样需要 FFT 操作的数据要对齐在 2 的幂次边界上。与传统的体系结构不同，这里对边界对齐的要求依赖于数据块的大小，因此高效率的将他们装入可用内存中是需要技巧和耐心的。

## 练习

1. 一个 SPARC 程序包含这些指令（这并不是一个可用的程序，而是用来示例的）。

Loc	Hex	Symbolic
1000	40 00 03 00	CALL X
1004	01 00 00 00	NOP; no operation, for delay
1008	7F FF FE ED	CALL Y
100C	01 00 00 00	NOP
1010	40 00 00 02	CALL Z
1014	01 00 00 00	NOP
1018	03 37 AB 6F	SETHI r1, 3648367 ; set high 22 bits of r1
101C	82 10 62 EF	ORI r1, r1, 751; OR in low 10 bits of r1

1a. 在一个 CALL 指令中，高 2 位是指令代码，低 30 位是一个有符号字(不是字节)偏移量。X、Y 和 Z 的地址是多少？

1b. 在 1010 处的对 Z 的调用都做了什么？

1c. 在 1018 和 101C 处的两个指令将一个 32 位地址调入寄存器 1。SETHI 将指令的低 22 位调入寄存器的高 22 位，ORI 还会将指令的低 13 位通过“或”运算放入寄存器，寄存器 1 中保存的地址是什么？

1d. 如果链接器将 X 移到 0x2504 处但不改变例子中的代码的位置，需要将 1000 中的指令如何修改使其仍然指向 X？

2. 若一个奔腾程序包含如下指令（不要忘记 x86 是 little-endian 的）：

Loc Hex Symbolic

1000 E8 12 34 00 00 CALL A

1005 E8 ?? ?? ?? ?? CALL B

100A A1 12 34 00 00 MOV %EAX, P

100F 03 05 ?? ?? ?? ?? ADD %EAX, Q

2a. 例程 A 和数据字 P 的位置是什么？(提示：在 x86 系统上，相对地址是根据指令下一个字节的地址计算的)

2b. 如果例程 B 是在地址 0x0f00，数据字 Q 是在地址 0x3456，例子中的??字节是什么？

3. 链接器和加载器是否需要“理解”目标体系结构指令集中的每个指令？如果一个目标系统新的型号增加了新的指令，是否需要修改链接器来支持？如果象 386 对于 286 那样在现有指令下增加新的寻址模式呢？

4. 回到计算机的黄金时代，那个时代程序员需要在半夜工作因为只有那个时候他们才能得到机时，而不是因为他们半夜起床，很多计算机系统使用字而不是字节地址。例如，PD P-6 和 10 使用 36 位字和 18 位寻址方式，每个指令都是一个字，操作数地址在字的下半部。(程序也可以在数据字的上半部分存储地址，尽管没有直接指令集支持)链接一个对字寻址的体系结构和一个对字节寻址的体系结构相比有多少不同？

5. 编写一个可修改目标系统的链接器有多复杂呢（就是只需要修改很少的特定部分的源代码就可以对不同目标架构提供支持的链接器）？对于可以处理不同体系结构的多目标链接器呢(尽管不是同一个链接器完成工作的)？

# 第3章 目标文件

\$Revision: 2.6 \$

\$Date: 1999/06/29 04:21:48 \$

编译器和汇编器创建了目标文件（包含由源程序生成的二进制代码和数据）。链接器将多个目标文件合并成一个，加载器读取这些目标文件并将它们加载到内存中（在一个集成编程环境中，当用户告诉它建立一个程序时，编译器、汇编器、链接器会在后台运行，但是它们确实是存在于“盖子”下面的）。在本章中，我们将深入到目标文件格式和内容的细节之中。

## 目标文件中都有什么？

一个目标文件包含五类信息。

- 头信息：关于文件的整体信息，诸如代码大小，翻译成该目标文件的源文件名称，和创建日期等。
- 目标代码：由编译器或汇编器产生的二进制指令和数据。
- 重定位信息：目标代码中的一个位置列表，链接器在修改目标代码的地址时会对它进行调整。
- 符号：该模块中定义的全局符号，以及从其它模块导入的或者由链接器定义的符号。
- 调试信息：目标代码中与链接无关但会被调试器使用到的其它信息。包括源代码文件和行号信息，本地符号，被目标代码使用的数据结构描述信息（如C语言数据结构定义）。（某些目标文件甚至包含比这更多的信息，但上面这些对于我们本章所需关注的已经足够了）

并不是所有的目标文件格式都包含这几类信息，一个很有用的目标文件格式很少或不包含以上任何信息，都是可能的。

## 设计一个目标文件格式

对一个目标文件格式的设计实际上是对目标文件所处的各种用途导致的折衷方案。一个文件可能是可链接的，能够作为链接编辑器或链接加载器的输入；它也可能是可执行的，可以加载到内存中作为一个程序运行；或者是可加载的，作为库同程序一起被加载到内存中；或者它是以上几种情况的组合。某些格式只支持上面的一到两种用法，而另一些格式则支持所有的用法。

一个可链接文件还包含链接器处理目标代码时所需的扩展符号和重定位信息。目标代码经常被划分为多个会被链接器区别对待的小逻辑段。一个可执行程序中会包含目标代码（为了能让文件被映射到地址空间中它通常是页对齐的），但是可以不需要任何符号（除非它要进行运行时动态链接）以及重定位信息。目标代码可以是一个单独的大段，或反映了硬件执行环境的一组小段（多数是只读或可读写的页）。根据系统运行时环境细节的不同，一

个可加载文件可以仅包含目标代码，或为了进行运行时链接还包含了完整的符号和重定位信息。

在应用中会存在某些冲突。面向逻辑的可链接段分组策略很少能够与面向硬件的可执行段分组策略相匹配。尤其是在一些较小的计算机上，链接器每次只会对可链接文件的一小片进行读写，但可执行程序会被整体的加载到内存中。这中区别在 MS-DOS 中尤为明显，因为它的可链接 OMF 格式与可执行 EXE 格式是完全不同的。

这里我们将会涉及到一系列常用的格式，从最简单的开始，一直到最复杂的。

## 空目标文件格式：MS-DOS 的. COM 文件

碰到一个仅有可运行二进制代码而没有其它信息的能够使用的目标代码文件是可能的。MS-DOS 的. COM 就是最有名的例子。一个. COM 文件中除了二进制代码外没有别的。当操作系统运行一个. COM 文件时，它只需将文件的内容加载到一块空闲内存中，从偏移量 0x100 处开始执行（0-0xFF 存放的是程序的命令行参数和其它参数，称为程序段前缀 PSP），将所有的 x86 段寄存器设置为指向 PSP，将 SP（栈指针）寄存器指向该段的末尾（由于栈是向下生长的），然后跳转到被加载程序的入口处。

x86 的分段架构使得这种文件格式可以工作。因为所有的 x86 程序地址都被解释为是相对于当前段基址的，所有的段寄存器都指向该段的基址，而程序总是以相对段位置为 0x100 的方式被加载。因此，对于可以放入单个段的程序而言，由于段相对地址可以在链接时确定而不需要再进行调整。

对于那些不能放入单一段的程序来说，对地址的调整工作是程序员的事情。而且确实存在一些程序是在启动时读取某个段寄存器然后将它的值与保存在程序中的某个地方的段值相加。当然这类繁琐的工作趋向于由链接器和加载器来自动完成，MS-DOS 通过. EXT 文件来完成这些（在本章稍后部分会讲述到）。

## 代码区段：Unix a.out 文件

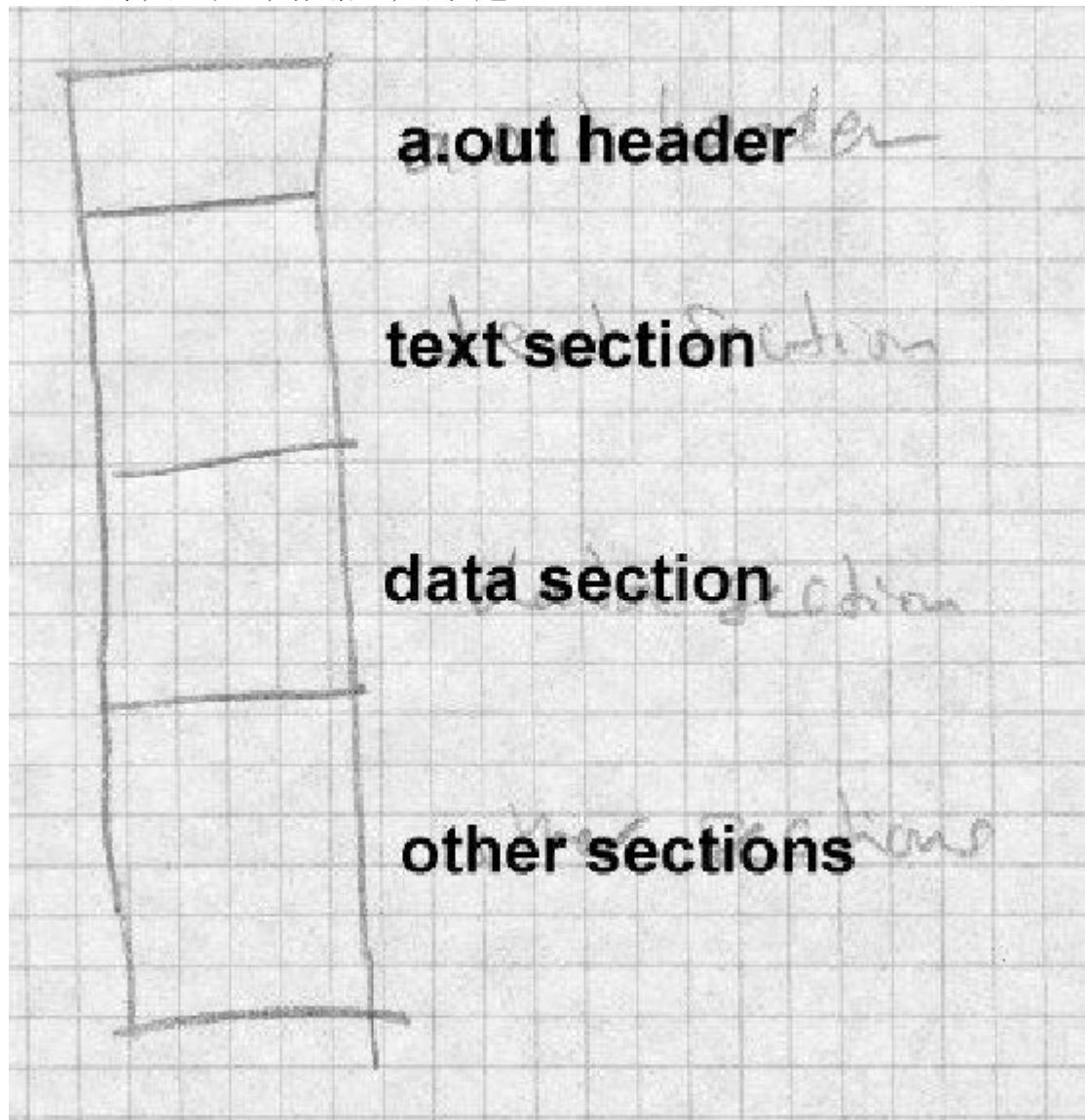
具有硬件内存重定位部件的计算机系统（今天几乎所有的计算机都有）通常都会为新运行的程序创建一个具有空地址空间的新进程，这种情况下程序就可以按照从某个固定地址开始的方式被链接，而不需要加载时的重定位。UNIX 的 a.out 目标文件格式就是针对这种情况的。

最简单的情况下，一个 a.out 文件包含一个小文件头，后面接着是可执行代码（由于历史的原因被称为文本段），然后是静态数据的初始值，如图 1 所示。PDP-11 只有 16 位寻址，将程序的地址空间限制为 64K。这个限制很快就变得太小了，所以 PDP-11 产品线的后续型号为代码（称为指令空间 I）和数据（称为数据空间 D）提供了独立的地址空间，这样一个程序可以拥有 64K 的代码空间和 64K 的数据空间。为了支持这个特性，编译器、汇编器、链接器都被修改为可以创建两个段的目标文件（代码放入第一个段中，数据放入第二个段中，程序加载时先将第一个段载入进程的 I 空间，再将第二个段载入进程的 D 空间）。

---

图 3-1：简化的 a.out

a.out 头部，文本段，数据段，和其它段



独立的 I 和 D 空间还有另一个性能上的优势：由于一个程序不能修改自己的 I 空间，因此一个程序的多个实体可以共享一份程序代码的副本。在诸如 UNIX 这样的分时系统上，she ll（命令解释器）和网络服务进程具有多个副本是很普遍的，共享程序代码可以节省相当可观的内存空间。

现在唯一通用的仍然为代码和数据进行单独寻址的计算机就是 286（或处于 16 位保护模式的 386）。即使在地址空间巨大的现代计算机上，操作系统可以通过虚拟内存来更有效的（相比于可读/写页的方式）处理只读代码页的共享，因此所有的现代加载器都支持它们。这意味着链接器创建的格式中至少要标识出只读和可读写的段来。实际中，多数链接器支持的格式中都具有多种类型的段，诸如只读数据，供后继链接操作使用的符号和重定位信息段，调试符号，和共享库信息（UNIX 的惯例令人混淆的将文件区段`section`称为段`segment`，所以我们在讨论 UNIX 的文件格式时也使用这个术语）。

## a.out 头部

a.out 的头部根据 UNIX 版本的不同而略有变化，但最典型的是 BSD UNIX 的版本，如图 2 所示（在本章的示例中，int 类型为 32 位，short 类型为 16 位）。

图 3-2: a.out 头部

```
int a_magic; // 幻数
int a_text; // 文本段大小
int a_data; // 初始化的数据段大小
int a_bss; // 未初始化的数据段大小
int a_syms; // 符号表大小
int a_entry; // 入口点
int a_trsize; // 文本重定位段大小
int a_drsiz; // 数据重定位段大小
```

幻数 a\_magic 说明了当前可执行文件的类型<sup>1</sup>。不同的幻数告诉操作系统的程序加载器以不同的方式将文件加载到内存中；我们将在下面讨论这些区别。文本和数据段大小 a\_text 和 a\_data 以字节为单位标识了头部后面的只读代码段和可读写数据段的大小。由于 UNIX 会自动将新分配的内存清零，因此初值无关紧要或者为 0 的数据不必在 a.out 文件中存储。未初始化数据大小 a\_bss 说明了在 a.out 文件中的可读写数据段后面逻辑上存在多少未初始化的数据（实际上是被初始化为 0）。

a\_entry 域指明了程序的起始地址，同时 a\_syms, a\_trsize 和 a\_drsiz 说明了在文件数据段后的符号表与重定位信息的大小。已经被链接好可以运行的程序中既不需要符号表也不需要重定位信息，所以除非链接器为了调试器加入符号信息，否则在可运行文件中这些域都是 0。

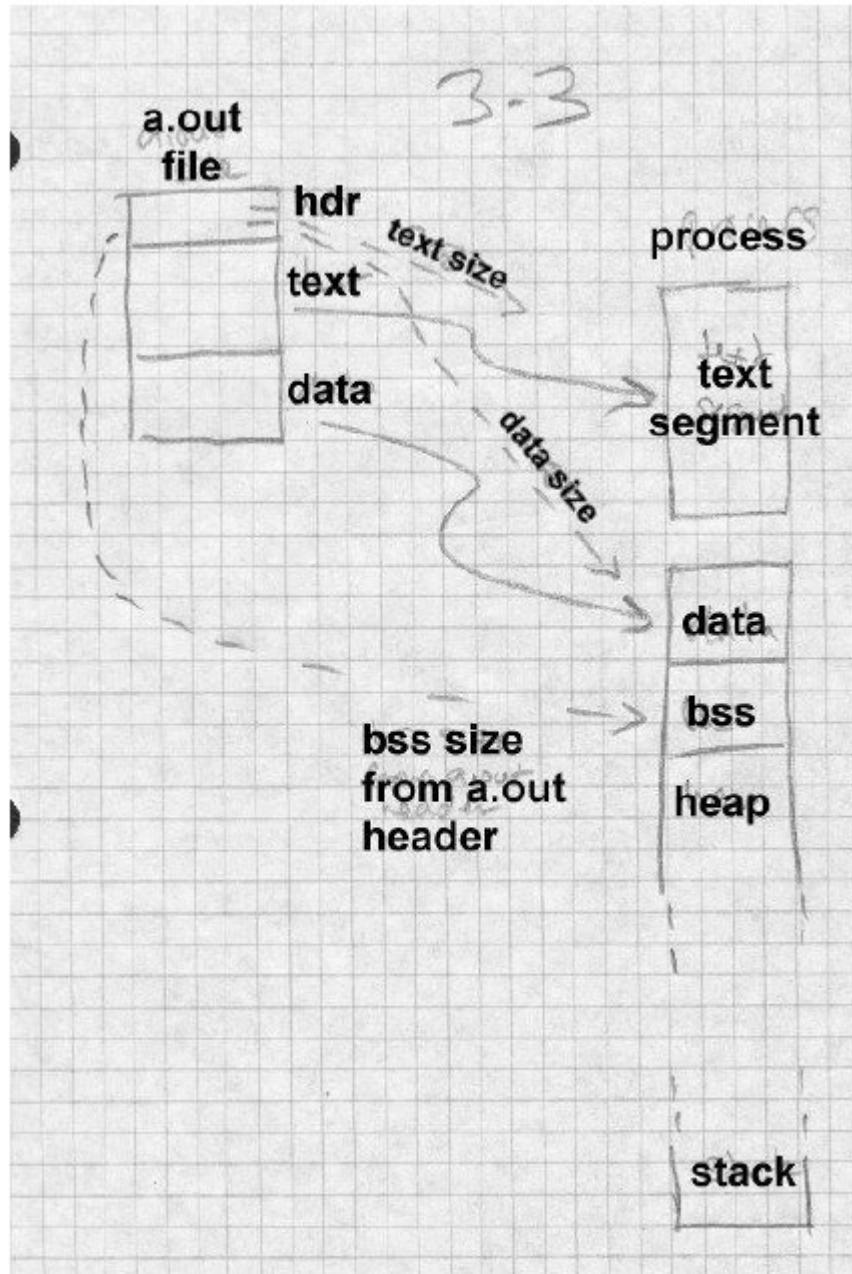
## 与虚拟内存的交互

操作系统加载和启动一个简单的双段文件的过程非常简单，如图 3 所示：

图 3-3: 加载一个 a.out 文件到一个进程中

文件和段构成的图，箭头表明数据流向

<sup>1</sup> 历史上最初的 PDP-11 上的幻数是八进制的 407，这是一个分支指令，可以越过它后面属于头部的 7 个字 (word) 而跳转到文本段的起始位置。这就构成了位置无关代码的最初形式。一个引导加载器可以将包括头部在内的整个可执行程序加载到内存中（通常在位置 0），然后跳转到被加载文件的入口位置运行程序。尽管只有少量程序用到了这个特性，但幻数 407 在 25 年后仍然与我们同在。



- 读取 a.out 的头部获取段的大小。
- 检查是否已存在该文件的可共享代码段。如果是的话，将那个段映射到该进程的地址空间。如果不是，创建一个并将它映射到地址空间中，然后从文件中读取文本段放入这个新的内存区域。
- 创建一个足够容纳数据段和 BSS 的私有数据段，将它映射到进程的地址空间中，然后从文件中读取数据段放入内存中的数据段并将 BSS 段对应的内存空间清零。
- 创建一个栈的段并将其映射到进程的地址空间（由于数据堆和栈的增长方向不同，因此栈段通常是独立于数据段的）。将命令行或者调用程序传递的参数放入栈中。
- 适当的设置各种寄存器并跳转到起始地址。

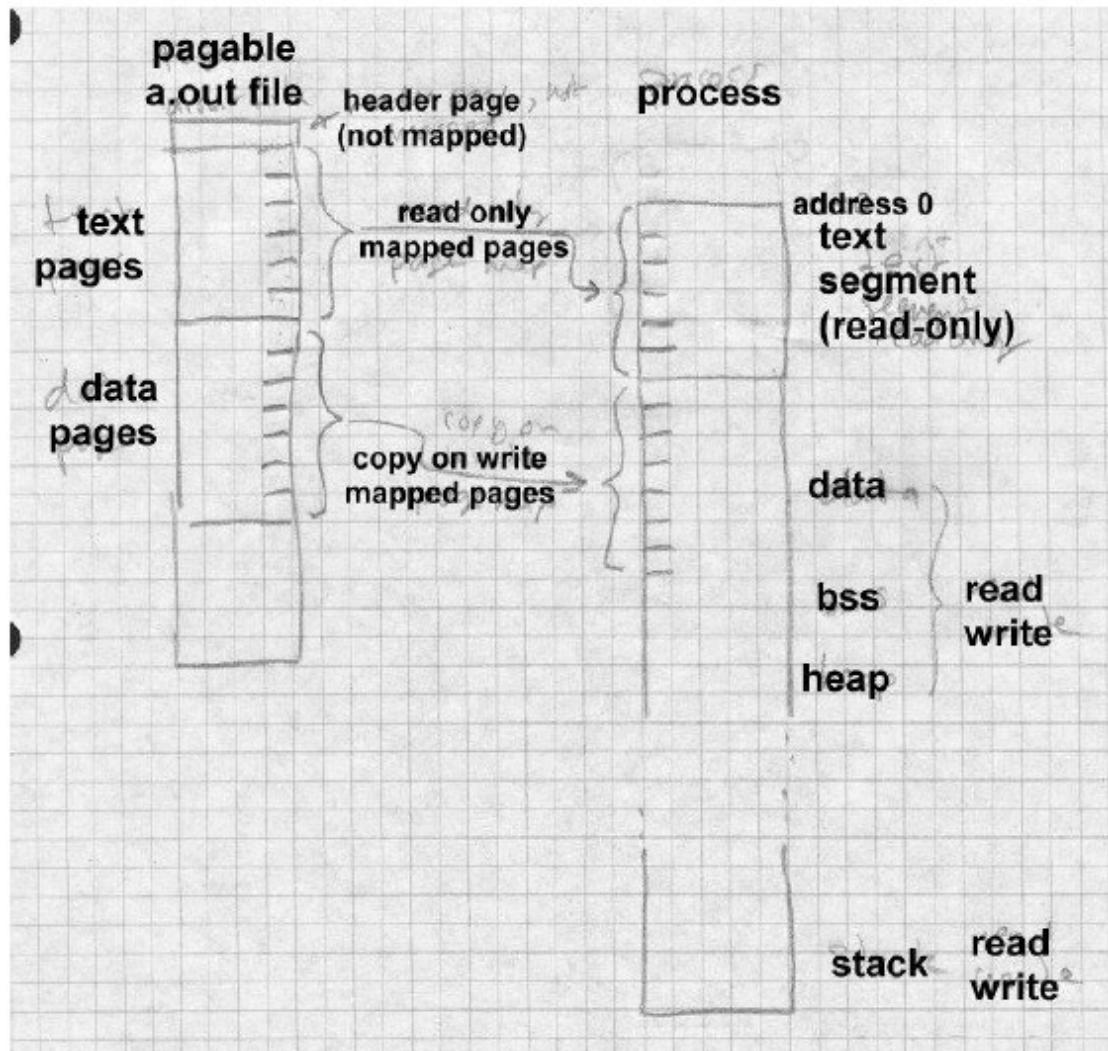
这种策略（称为 NMAGIC，N 表示 new，当然这是相对于 1975 年那时而言）相当有效，早期的 PDP-11 和 VAX UNIX 系统在很多年中都将它应用于所有的目标文件，而且可链接文件直到 90 年代在整个 a.out 格式的生命周期内都在使用这种策略。当 UNIX 系统采用虚拟内存后，

对这种简单策略的些许改进还进一步加速了程序加载的速度并节省了相当可观的内存。

在一个分页系统中，上述的简单机制会为每一个文本段和数据段分配新的虚拟内存。由于 a.out 文件已经存储在磁盘中了，所以目标文件本身可以被映射到进程的地址空间中。虚拟内存只需要为程序写入的那些页分配新的磁盘空间，这样可以节省磁盘空间。并且由于虚拟内存系统只需要将程序确实需要的那些页从磁盘加载到内存中（而不是整个文件），这样也加快了程序启动的速度。

对 a.out 文件格式进行少许修改就可以做到这一点，如图 4 所示，这就够成了被称为 Z MAGIC 的格式。这些变化将目标文件中的段对齐到页的边界。在页大小为 4K 的系统上，a.out 头部扩展为 4K，文本段的大小也要对齐到下一个 4K 的边界。由于 BSS 段逻辑上跟在数据段的后面并在程序加载时被清零，所以没有必要对数据段进行页边界对齐的填充。

图 3-4：将 a.out 文件映射到进程中  
文件和段构成的图，将页框映射到段中。

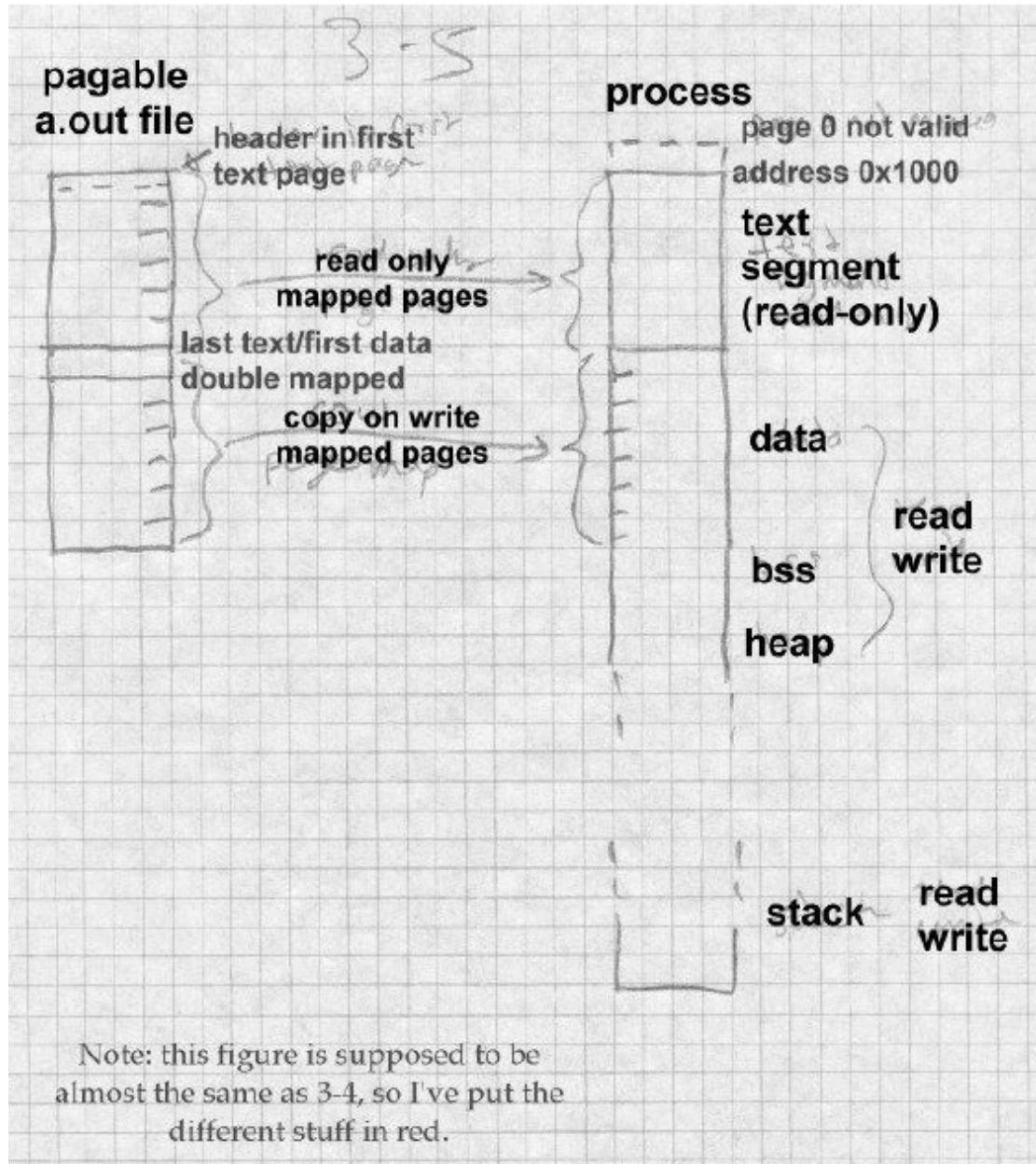


ZMAGIC 格式的文件减少了不必要的换页，对应付出的代价是浪费了大量的磁盘空间。a.out 的头部仅有 32 字节长，但是仍需要分配 4K 磁盘空间给它。文本和数据段之间的空隙平均浪费了 2K 空间，即半个 4K 的页。上述这些问题都在被称为 ZMAGIC 的压缩可分页格式中被

修正了。

由于并没有什么特别的原因要求文本段的代码必须从地址 0 处开始运行，因此压缩可分页文件将 a.out 头部当成是文本段的一部分（实际上由于未初始化的指针变量经常为 0，位置 0 绝对不是一个程序入口的好地方）。代码紧跟在头部的后面，并将整个页映射为进程的第二个页，而不映射进程地址空间的第一个页，这样对位置 0 的指针引用就会失败，如图 5 所示。它也产生了一个无害的副作用就是将头部映射到进程的地址空间中了。

图 3-5：将一个压缩的 a.out 文件映射到进程中  
文件和段构成的图，将页框映射到段中。



QMAGIC 格式的可执行文件中文本和数据段都各自扩充到一个整页，这样系统就可以很容易的将文件中的页映射到地址空间中的页。数据段的最后一页由值为零的 BSS 数据填充补齐；如果 BSS 数据大于可以填充补齐的空间，那么 a.out 的头部中会保存剩余需要分配的 BSS 空间大小。

尽管 BSD UNIX 将程序加载到位置 0 (或 QMAGIC 格式的 0x1000) 处，其它版本的 UNIX 会将程序加载到不同的位置。例如 Motorola 68K 系列上的系统 5 (System V) 会将程序加载到 0x80000000 处，在 386 上会加载到 0x8048000 处。只要地址是页对齐的，并且能够与链接器和加载器达成一致，加载到哪里都没有关系。

## 重定位：MS-DOS EXE 文件

对于那些可以为每一个进程分配新的地址空间让每个程序都可以加载到相同逻辑地址的系统而言，a.out 格式是足够了。但是很多系统就没有那么幸运了。有一些系统会将所有的程序加载到相同的地址空间。还有一些系统虽然会为程序分配自己的地址空间，但是并不总是将程序加载到相同的地址 (32 位 Windows 系统就属于这最后一类)。

在这些情况下，可执行程序会包含多个 (通常被称为 fixups 的) 重定位项，它们指明了程序中需要在被加载时进行修改的地址位置。具有 fixups 的最简单的格式之一就是 MS-DOS EXE 格式。

如我们上面所见的 .COM 格式，DOS 将程序载入到一块连续的可用实模式内存中。如果一个 64K 的段无法容纳整个程序，就需要使用明确的段基址对程序和数据进行寻址，并在程序加载时必须调整程序中的段基址以匹配程序实际加载的位置。文件中的段基址是按照程序将被加载到位置 0 来存储的，所以修正的动作就是将程序实际被加载到的段地址与存储的段基址相加。就是说，如果程序实际被加载到位置 0x5000，即段基址为 0x500，那么文件中对段基址 0x12 的引用将会重定位为 0x512。由于程序是作为一个整体被重定位的，段内偏移量不会改变，所以加载器不需要修正除段基址之外的其它内容。

每个 .EXE 文件都是以图 6 所示的头部结构开始的。跟在头部后面的是变量长度相关的额外信息 (采用重叠技术的加载器，自解压文件和其它与应用程序相关的技巧都可能会用到) 和一个 segment:offset 格式的 32 位修正地址列表。修正地址是程序基地址的相对地址，所以这些修正地址本身也需要被重定位以寻找那些程序中需要被修改的地址。在修正地址列表后的是程序代码。在代码的后面，也许还有会被程序加载器忽略的额外信息。(在下面的例子中，far 类型指针为 32 位，其中 16 位段基址和 16 位段内偏移量)

---

图 3-6: .EXE 文件头部格式

```
char signature[2] = "MZ"; //幻数
short lastsize; //最后一个块使用的字节数
short nblocks; //512 字节块的个数
short nreloc; //重定位项个数
short hdrsize; //以 16 字节段为单位的文件头部尺寸
short minalloc; //需额外分配的最小内存量
short maxalloc; //需额外分配的最大内存量
void far *sp; //初始栈指针
short checksum; //文件校验和
void far *ip; //初始指令指针
```

```
short relocpos; //重定位修正表位置  
short noverlay; //重叠的个数，程序为0  
char extra[]; //重叠所需的额外信息等  
void far *relocs[]; //重定位项，从 relocpos 开始
```

---

加载. EXE 文件只比加载. COM 文件复杂一点点。

- 读入文件头部，验证幻数是否有效。
- 找一块大小合适的内存区域。minalloc 和 maxalloc 域说明了在被加载程序末尾后需额外分配的内存块的最大和最小尺寸（链接器总是缺省的将最小尺寸设置为程序中类似 BSS 的未初始化数据的大小，将最大尺寸设置为 0xFFFF）。
- 创建一个程序段前缀（Program Segment Prefix），即位于程序开头的控制区域。
- 在 PSP 之后读入程序的代码。nblocks 和 lastsize 域定义了代码的长度。
- 从 relocpos 处开始读取 nreloc 个修正地址项。对每一个修正地址，将其中的基址与程序代码加载的基址相加，然后将这个重定位后的修正地址作为指针，将程序代码的实际基址与这个指针指向的程序代码中的地址相加。
- 将栈指针设置为重定位后的 sp，然后跳转到重定位后的 ip 处开始执行程序。

除了与分段寻址相关的怪异特性外，这是程序加载时的典型初始化过程。在少数情况下，程序的不同片段可以用不同的方式重定位。在 286 保护模式下（EXE 文件不支持），虽然可执行文件中的代码和数据段被加载到系统中各自独立的段，但是由于体系结构的原因段基址是不连续的。每一个保护模式的可执行程序在靠近文件开头的位置有一个表列出来程序需要的所有段。系统会创建一个表将可执行程序中的每个段与系统中实际的段址对应起来。在进行地址调整时，系统会在这个表中查找逻辑段址，并将其替换为实际的段址，相比于重定位这更类似一个符号绑定的过程。

有一些系统还允许在加载时进行符号解析，但我们将这个话题留到第 10 章。

## 符号和重定位

目前我们讨论过的目标文件格式都是可加载的，即可以加载到内存中并直接运行。多数目标文件并不是可加载的，但相当一部分是由编译器或汇编器生成传递给链接器或库管理器的中间文件。这些可链接文件比起那些可运行文件来说，要复杂的多。由于可运行文件要运行在计算机的底层硬件上因此必须要足够简单，但可链接文件的处理属于软件层面，因此可以做很多非常高级的事情。原则上，一个支持链接的加载器可以在程序被加载时完成所有链接器必须完成的功能，但由于效率原因加载器通常都尽可能的简单，以提高程序启动的速度（动态链接（我们将在第 10 章涉及），将很多工作由链接器转移到加载器（由此在性能上有一些损失），但由于现代计算机的速度足够快了，所以采用动态链接的利大于弊）。

现在我们来看看五种逐步复杂的格式：BSD UNIX 系统采用的 a.out 可重定位格式，系统五（System V）使用的 ELF 格式，IBM 360 目标文件格式，32 位 Windows 上使用的扩展的 COFF 可链接和 PE 可执行格式，以及 COFF 格式 Windows 系统之前的 OMF 可链接格式。

## 可重定位的 a.out 格式

UNIX 系统对于可运行文件和可链接文件都使用相同的一种目标文件格式，其中可运行文件省略掉了那些仅用于链接器的段。我们在图 2 中看到的 a.out 格式包含了链接器使用的一些域。文本和数据段的重定位表的大小保存在 a\_trsize 和 a\_drsiz 中，符号表的尺寸保存在 a\_syms 中。这三个段跟在文本和数据段后，如图 7 所示。

---

图 3-7：简化的 a.out

a.out 头部

文本段

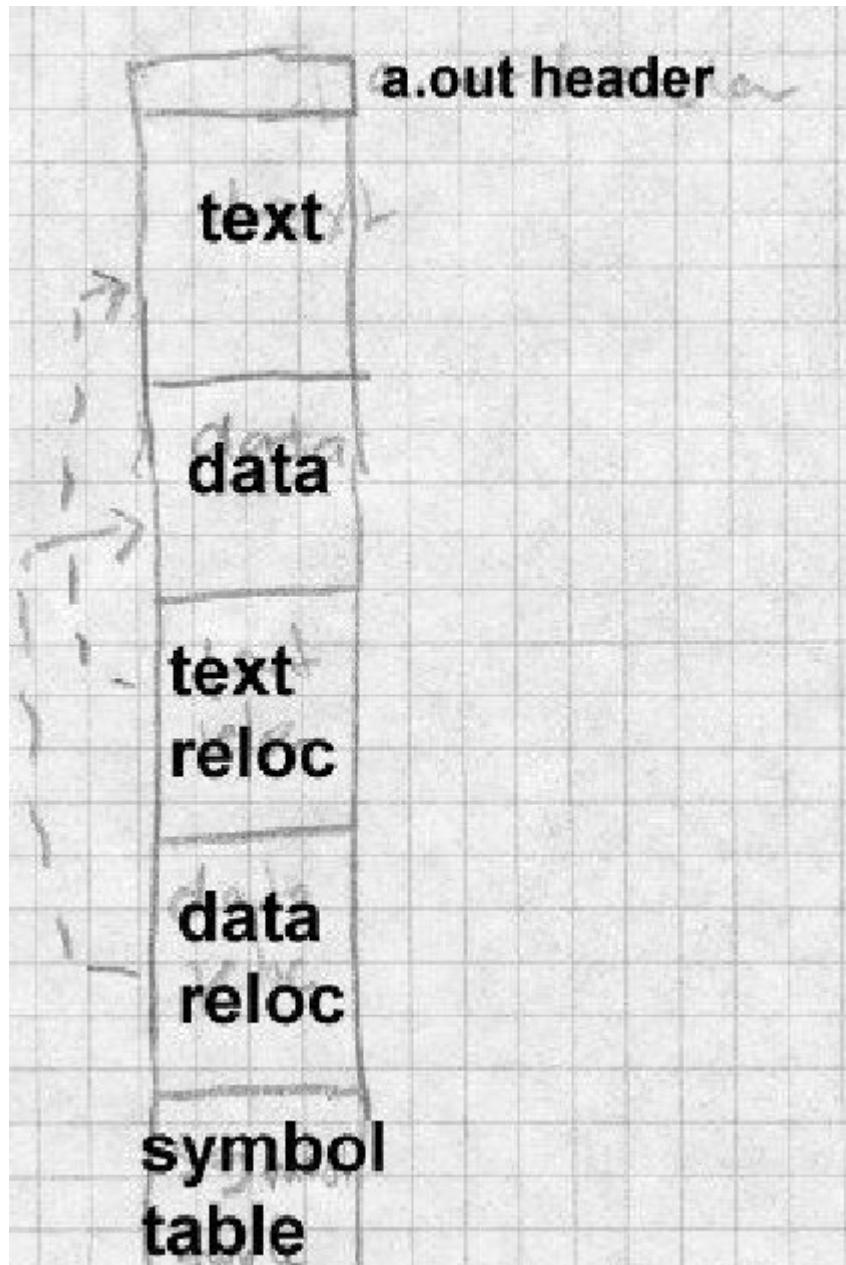
数据段

文本重定位表

数据重定位表

符号表

字串表



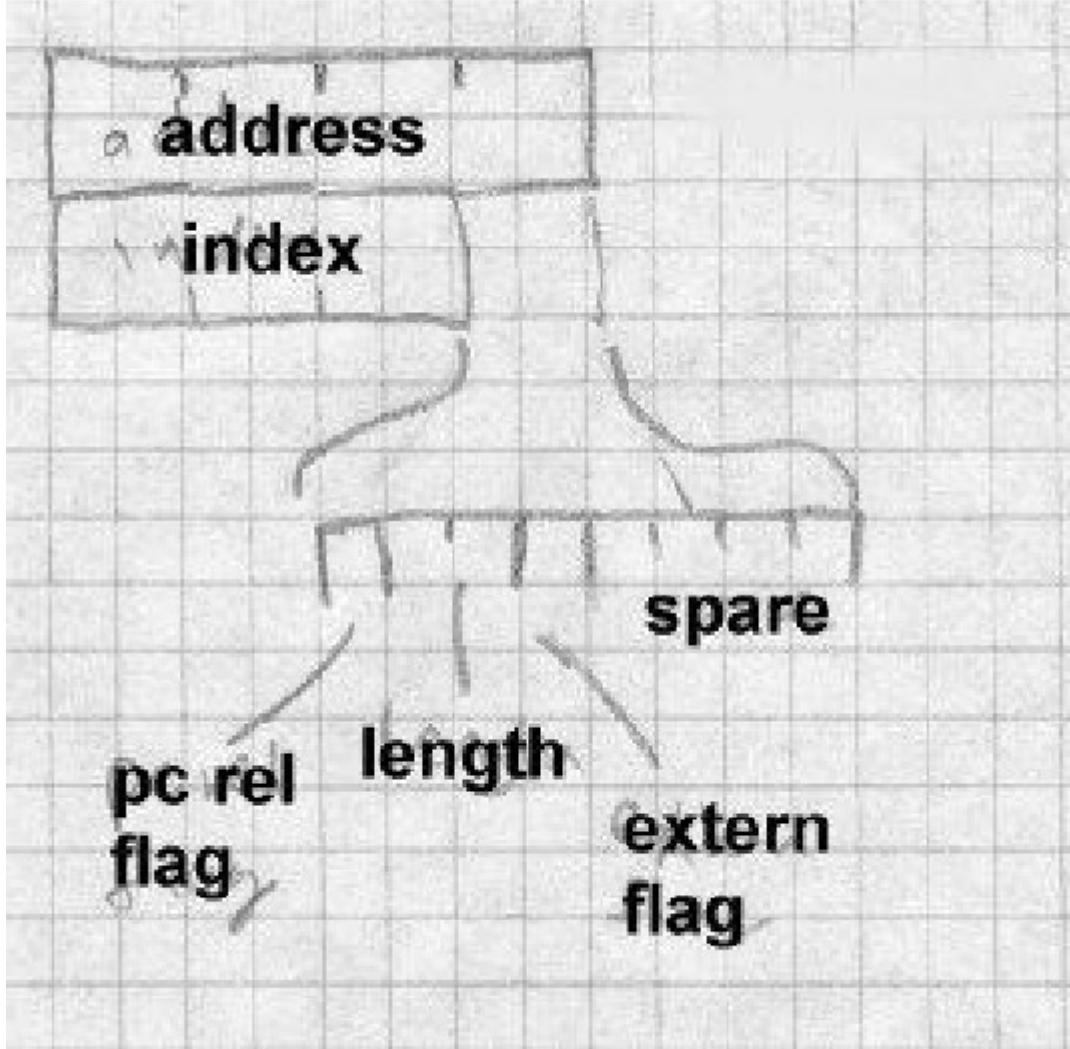
## 重定位项

重定位项有两个功能。当一个代码段被重定位到另一个不同的段基址时，重定位项标注出代码中需要被修改的地方。在一个可链接文件中，同样也有用来标注对未定义符号引用的重定位项，这样链接器就知道在最终解析符号时应当向何处补写符号的值。

图 8 展示了一个重定位项的格式。每一个重定位项包含了在文本或数据段中需被重定位的地址，以及定义了要做什么的信息。该地址是一个需要进行重定位的项目到文本段或数据段起始位置的偏移量。长度域说明了该重定位项目的长度，从 0 到 3 依次对应 1、2、4 或者（在某些体系结构上）8 个字节。pcrel 标志表示这是一个“PC（程序计数器，即指令寄存器）相对的”重定位项目，如果是的话，它会在指令中被作为相对地址使用。

图 3-8：重定位项格式

4 个 byte 的地址，3 个 byte 的索引，1 个 bit 的 pcrel 标志，2 个 bit 的长度域，1 个 bit 的外部标志，4 个 bit 的空闲位



外部标志域控制对 index 域的解释，确定该重定位项目是对某个段或符号的引用。如果外部标志为 off，那这是一个简单的重定位项目，index 就指明了该项目是基于哪个段（文本、数据或 BSS）寻址的。如果外部标志为 on，那么这是一个对外部符号的引用，则 index 是该文件符号表中的符号序号。

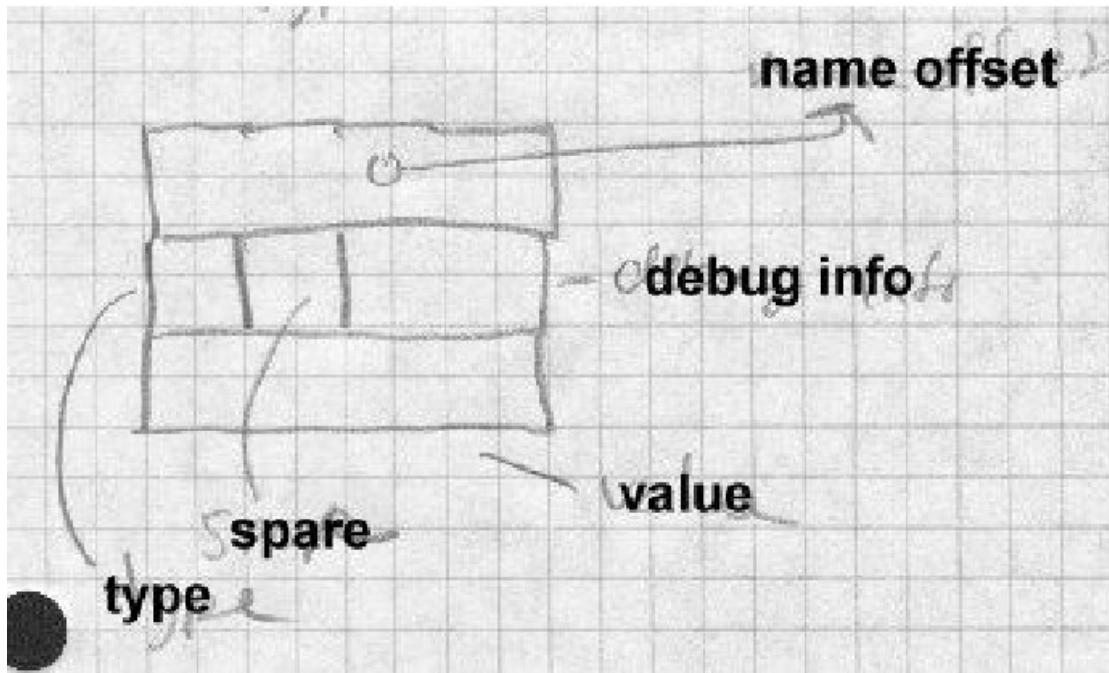
这种重定位格式对于多数硬件体系结构是足够了，但某些更复杂的架构需要额外的标志位，例如 3 字节的 IBM 370 地址常量或 SPARC 中的高、低半地址常量。

## 符号和字符串

a.out 文件的最后一个段是符号表。每个表项长度为 12 字节，描述一个符号，如图 9 所示。

图 3-9：符号格式

4个byte的名字偏移量，1个byte的类型，1个byte的空闲字节，2个byte的调试信息，4个byte的值



UNIX 编译器允许任意长度的标识符，所以名字字串全部都在符号表后面的字串表中。符号表项的第一个域是该符号以空字符结尾的名字字串在字串表中的偏移量。在类型字节中，若低位被置位则该符号是外部的（用词不当，该符号实际是可以被其它模块看到的符号，所以称为全局符号更合适）。非外部符号对于链接是没有必要的，但是会被调试器用到。其余的位是符号类型。最重要的类型包括：

- 文本、数据或 BSS：模块内定义的符号。外部标志位可能设置或没有设置。值为与该符号对应的模块内可重定位地址。
- abs：绝对非可重定位符号（absolute non-relocatable symbol）。很少在调试信息以外的地方使用。外部标志位可能设置或没有设置。值为该符号的绝对地址。
- undefined：在该模块中未定义的符号。外部标志位必须被设置。值通常为 0，但下面会讲到的“公共块技巧”中的内容是例外。这些符号类型对于诸如 C、Fortran 这样的老一些语言是足够了，但对于 C++ 等而言，几乎不够。

作为一种特例，编译器可以使用一个未定义的符号来要求链接器为该符号的名字预留一块存储空间。如果一个外部符号的值不为零，则该值是提示链接器程序希望该符号寻址存储空间的大小。在链接时，若该符号的定义不存在，则链接器根据其名字在 BSS 中创建一块存储空间，大小为所有被链接模块中该符号提示尺寸中的最大值。如果该符号在某个模块中被定义了，则链接器使用该定义而忽略提示的空间大小。这种“公共块技巧（common block hack）”支持 Fortran 公共块和未初始化的 C 外部数据的典型用法（尽管是非标准的）。

## a.out 格式小结

对于相对简单的分页系统，a.out 格式是简单而有效的。之所以被淘汰出主流，主要是因为它不能很容易的支持动态链接。并且，a.out 格式不支持 C++ 语言，因为 C++ 语言对所有的初始化和终结代码都需要特殊的处理。

## Unix ELF 格式

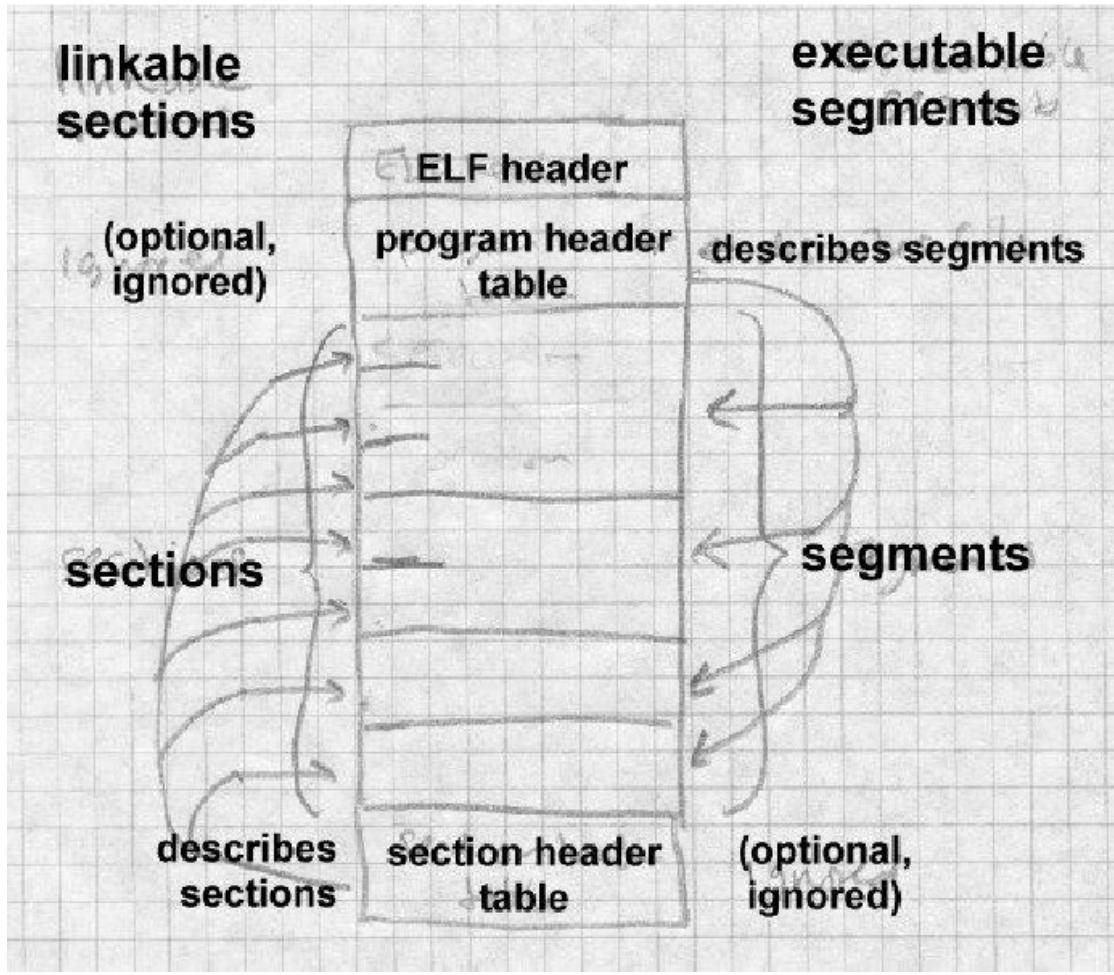
传统的 a.out 格式为 UNIX 社区服务了超过 10 年，但是在 UNIX 系统 5 (System V) 推出时，AT&T 认为需要一些更好的东西来支持交叉编译、动态链接以及其它的现代系统特性。早期的系统 5 采用 COFF 格式，即通用目标文件格式 (Common Object File Format)，它最初是为交叉编译的嵌入式系统设计的，由于其无扩展版本不支持 C++ 和动态链接，因此并不能在所有的分时系统上都很好的工作。在系统 5 的后期版本中，COFF 被 ELF 格式替代，即可执行和链接格式 (Executable and Linking Format)。ELF 同样被流行的自由软件 Linux 和 BSD 等类 UNIX 系统所采用。ELF 有一个关联的调试格式称为 DWARF，我们将在第五章看到。在这里我们只讨论 32 位的 ELF。也存在通过简单的方法将数据尺寸和地址扩展到 64 的 ELF 格式 64 位变体。

ELF 格式有三个略有不同的类型：可重定位的，可执行的，和共享目标 (shared objects)。可重定位文件由编译器和汇编器创建，但在运行前需要被链接器处理。可执行文件完成了所有的重定位工作和符号解析（除了那些可能需要在运行时被解析的共享库符号），共享目标就是共享库，即包括链接器所需的符号信息，也包括运行时可以直接执行的代码。

ELF 格式具有不寻常的双重特性，如图 10 所示。编译器、汇编器和链接器将这个文件看作是被区段 (section) 头部表描述的一系列逻辑区段的集合，而系统加载器将文件看成是由程序头部表描述的一系列段 (segment) 的集合。一个段 (segment) 通常会由多个区段 (section) 组成。例如，一个“可加载只读”段可以由可执行代码区段、只读数据区段和动态链接器需要的符号组成。可重定位文件具有区段表，可执行程序具有程序头部表，而共享目标文件两者都有。区段 (section) 是用于链接器后续处理的，而段 (segment) 会被映射到内存中。

---

图 3-10：一个 ELF 文件的两种视角



ELF 文件都是以 ELF 头部起始的，如图 11 所示。头部被设计为即使在那些字节顺序与文件的目标架构不同的机器上也可以被正确的解码。头 4 个字节是用来标识 ELF 文件的幻数，接下来的 3 个字节描述了头部其余部分的格式。当程序读取了 class 和 byteorder 标志后，它就知道了文件的字节序和字宽度，就可以进行相应的字节顺序和数据宽度的转换。其它的域描述了区段头部或程序头部的大小和位置（如果它们存在的话）。

图 3-11：ELF 头部

```

char magic[4] = "\177ELF";//幻数
char class;      //地址宽度, 1 = 32 位, 2 = 64 位
char byteorder; //字节序, 1 = little-endian, 2 = big-endian
char hversion;   //头部版本, 总是 1
char pad[9];     //填充字节
short filetype; //文件类型: 1 = 可重定位, 2 = 可执行,
                 //3 = 共享目标, 4 = 转储镜像 (core image)
short archtype; //架构类型, 2 = SPARC, 3 = x86, 4 = 68K, 等等.
int fversion;    //文件版本, 总是 1
int entry;       //入口地址 (若为可执行文件)
int phdrpos;     //程序头部在文件中的位置 (不存在则为 0)

```

```
int shdrpos; //区段头部在文件中的位置（不存在则为0）
int flags; //体系结构相关的标志，总是0
short hdrsize; //该 ELF 头部的大小
short phdrent; //程序头部表项的大小
short phdrcnt; //程序头部表项个数（不存在则为0）
short shdrent; //区段头部表项的大小
short phdrcnt; //区段头部表项的个数（不存在则为0）
short strsec; //保存有区段名称字串的区段的序号
```

---

## 可重定位文件

一个可重定位或共享目标文件可以看成是一系列在区段头部表中被定义的区段的集合，如图 12 所示。每个区段只包含一种类型的信息，可以是程序代码、只读数据或可读写数据，重定位项，或符号。在模块中定义的符号都是以段的相对地址定义的，因此一个过程（procedure）的入口点也是由包含该过程代码的程序代码区段的相对地址来定义的。此外还存在两个伪段，SHN\_ABS（数字 0xffff1）逻辑上包含了绝对不可重定位符号（absolute non-relocatable symbols），SHN\_COMMON（数字 0xffff2）包含未初始化的数据块，这是从 a.out 的公共块技术中继承下来的。区段 0 总是空的，其对应的区段表项也为全零。

---

图 3-12：区段头部

```
int sh_name; //名称，可在字串表中索引到
int sh_type; //区段类型
int sh_flags; //标志位，见下
int sh_addr; //若可加载则为内存基址，否则为0
int sh_offset; //区段起始点在文件中的位置
int sh_size; //区段大小（字节为单位）
int sh_link; //相关信息对应的区段号，若没有则为0
int sh_info; //区段相关的更多信息
int sh_align; //移动区段时的对齐粒度
int sh_entsize; //若该区段为一个表时其中表项的大小
```

---

区段类型包括：

- PROGBITS：程序内容，包括代码，数据和调试器信息。
- NOBITS：类似于 PROGBITS，但在文件本身中并没有分配空间。用于 BSS 数据，在程序加载时分配空间。
- SYMTAB 和 DYNSYM：符号表，后面会有更加详细的描述。SYMTAB 包含所有的符号并用于普通的链接器，DYNSYM 包含那些用于动态链接的符号（后一个表需要在运行时被加载到内存中，因此要让它尽可能的小）。

- STRTAB: 字串表，与 a.out 文件中的字串表类似。与 a.out 文件不同的是，ELF 文件能够而且经常为不同的用途创建不同的字串表，例如全段名称、普通符号名称和动态链接符号名称。
- REL 和 RELA: 重定位信息。REL 项将其中的重定位值加到存储在代码和数据中的基地址值，而 RELA 将重定位需要的基地址也保存在重定位项自身中（由于历史原因，x86 目标文件使用 REL 重定位类型，68k 使用 RELA 重定位类型）。每种体系结构下都有多种重定位类型，但它们类似于（也起源于）a.out 的重定位类型。
- DYNAMIC 和 HASH: 动态链接信息和运行时符号 hash 表。这里用到了 3 个标志位：ALLOC，意味着在程序加载时该区段要占用内存空间；WRITE 意味着该区段被加载后是可写的；EXECINSTR 即表示该区段包含可执行的机器代码。

一个典型的可重定位可执行程序会有十多个区段。很多区段的名称对于链接器在根据它所支持的区段类型来进行特定的处理（同时根据标志位将不支持的区段忽略或原封不动的传递下去）时，都是有意义的。

区段的类型包括：

- .text 是具有 ALLOC 和 EXECINSTR 属性的 PROGBITS 类型区段。相当于 a.out 的文本段。
- .data 是具有 ALLOC 和 WRITE 属性的 PROGBITS 类型区段。对应于 a.out 的数据段。
- .rodata 是具有 ALLOC 属性的 PROGBITS 类型区段。由于是只读数据，因此没有 WRITE 属性。
- .bss 是具有 ALLOC 和 WRITE 属性的 NOBITS 类型区段。BSS 区段在文件中没有分配空间，因此是 NOBITS 类型，但由于会在运行时分配空间，所以具有 ALLOC 属性。
- .rel.txt, .rel.data 和 .rel.rodata 每个都是 REL 或 RELA 类型区段。是对应文本或数据区段的重定位信息。
- .init 和 .fini，都是具有 ALLOC 和 EXECINSTR 属性的 PROGBITS 类型区段。与 .text 区段相似，但分别为程序启动和终结时执行的代码。C 和 Fortran 不需要这个，但是对于具有初始和终结函数的全局数据的 C++ 语言来说是必须的。
- .symtab 和 .dynsym 分别是 STMTAB 和 DNNSYM 类型的区段，对应为普通的和动态链接器的符号表。动态链接器符号表具有 ALLOC 属性，因为它需要在运行时被加载。
- .strtab 和 .dynstr 都是 STRTAB 类型的区段，这是名称字串的表，要么是符号表，要么是段表的段名称字串。.synstr 区段保存动态链接器符号表字串，由于需要在运行时被加载所以具有 ALLOC 属性。此外还有一些特殊的区段诸如 .got 和 .plt，分别是全局偏移量表（Global Offset Table）和动态链接时使用的过程链接表（Procedure Linkage Table），PLT 将在第 10 章中涉及。.debug 区段包含调试器所需的符号，.line 区段也是用于调试器的，它保存了从源代码的行号到目标代码位置的映射关系。而 .comment 区段包含着文档字串，通常是版本控制中的版本序号。

还有一个特殊的区段类型 .interp，它包含解释器程序的名字。如果这个区段存在，系统不会直接运行这个程序，而是会运行对应的解释器程序并将该 ELF 文件作为参数传递给解释器。例如 UNIX 上多年以来都有可以解释型的自运行文本文件，只需要在文件的第一行加上：

```
#!/path/to/interpreter
```

ELF 将这种功能扩展到那些可以运行非文本程序的解释器上。实际使用中它被用来调用运行时动态链接器以加载程序并将任何需要的共享库链接进来。

ELF 符号表与 a.out 符号表相似，包含一个由表项组成的数组，如图 13 所示。

图 3-13：ELF 符号表

```
int name;      //名称字串在字串表中的位置  
int value;     //符号值，在可重定位文件中是段相对地址，  
               //在可执行文件中是绝对地址  
int size;      //目标或函数的大小  
char type:4;   //符号类型：数据目标，函数，区段，或特殊文件  
char bind:4;   //符号绑定类型：局部，全局，或弱符号  
char other;    //空闲  
short sect;    //段基址，ABS，COMMON 或 UNDEF
```

ELF 符号表增加了少许新的域。size 域指明了数据目标（尤其是未定义的 BSS，又使用了公共块技巧）的大小，一个符号的绑定可以是局部的（仅模块内可见），全局的（所有地方均可见），或者弱符号。

弱符号是半个全局符号：如果存在一个对未定义的弱符号的有效定义，则链接器采用该值，否则符号值缺省为 0。

符号的类型通常是数据或者函数。对每一个区段都会有一个区段符号，通常都是使用该区段本身的名字，这对重定位项是有用的（ELF 重定位项的符号都是相对地址，因此就需要一个段符号来指明某一个重定位项目是相对于文件中的哪一个区段）。文件入口点是一个包含源代码文件名称的伪符号。

区段号（即段基址）是相对于该符号的定义所在的那个段的，例如函数入口点都是相对于 .text 段定义的。这里还可以看到三个特殊的伪区段，UNDEF 用于未定义符号，ABS 用于不可重定位绝对符号，COMMON 用于尚未分配的公共块（COMMON 符号中的 value 域提供了所需的对齐粒度，size 域提供了尺寸最小值。一旦被链接器分配空间后，COMMON 符号就会被转移到 .bss 区段中）。

如图 14 所示，是一个典型的完整的 ELF 文件，包含代码、数据、重定位信息、链接器符号、和调试器符号等若干区段。如果该文件是一个 C++ 程序，那可能还包含 .init、.fini、.rel.init 和 .rel.fini 等区段。

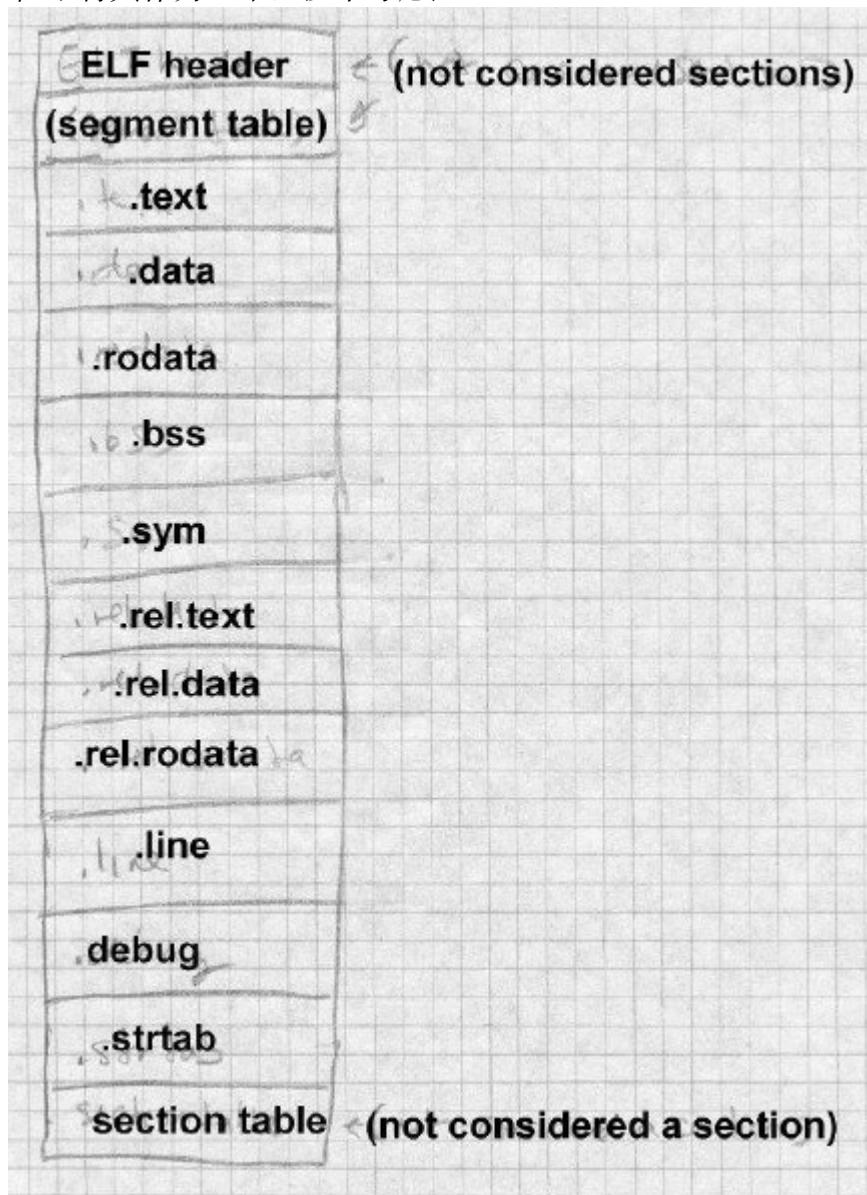
图 3-14：可重定位 ELF 文件示例

#### ELF 文件头部

```
.text  
.data  
.rodata  
.bss
```

.sym  
.rel.text  
.rel.data  
.rel.rodata  
.line  
.debug  
.strtab

(区段表，但不将其作为一个区段来考虑)



---

## ELF 可执行文件

一个 ELF 可执行文件具有与可重定位 ELF 文件相同的通用格式，但对数据部分进行了调整以使得文件可以被映射到内存中并运行。文件中会在 ELF 头部后面存在程序头部。程序头

部定义了要被映射的段。如图 15 所示为程序头部，是一个由段描述符组成的数组。

图 3-15：ELF 程序头部

```
int type;      //类型：可加载代码或数据，动态链接信息，等  
int offset;    //段在文件中的偏移量  
int virtaddr;  //映射段的虚拟地址  
int physaddr;  //物理地址，未使用  
int filesize;  //文件中的段大小  
int memsize;   //内存中的段大小（如果包含 BSS 的话会更大些）  
int flags;     //读，写，执行标志位  
int align;     //对齐要求，根据硬件页尺大小不同有变动
```

一个可执行程序通常只有少数几种段，如代码和数据的只读段，可读写数据的可读写段。所有的可加载区段都归并到适当类型的段中以便系统可以通过少数的一两个操作就可以完成文件映射。

ELF 格式文件进一步扩展了 QMAGIC 格式的 a.out 文件中使用的“头部放入地址空间”的技巧，以使得可执行文件尽可能的紧凑，相应付出的代价就是地址空间显得凌乱了些。一个段可以开始和结束于文件中的任何偏移量处，但是段的虚拟起始必须和文件中起始偏移量具有低位地址模对齐的关系，例如，必须起始于一页的相同偏移量处。系统必须将段起始所在页到段结束所在页之间整个的范围都映射进来，哪怕在逻辑上该段只占用了被映射的第一页和最后一页的一部分。图 16 所示为一个典型的段分布方式。

图 3-16：ELF 可加载段

	文件偏移量	加载地址	类型
ELF 头部	0	0x8000000	
程序头部	0x40	0x8000040	
只读文本 (尺寸为 0x4500)	0x100	0x8000100	可加载 可读，可执行
可读/写数据 (文件中尺寸为 0x2200) 内存中尺寸为 0x3500	0x4600	0x8005600	可加载 可读，可写 可执行
不可加载信息和 可选的区段头部			

-----+-----+-----+-----+

---

被映射的文本段包括 ELF 头部，程序头部，和只读文本，这样 ELF 头部和程序头部都会在文本段开头的同一页中。文件中仅有的可读写数据段紧跟在文本段的后面。文件中的这一页会同时被映射为内存中文本段的最后一页和数据段的第一页（以 copy-on-write 的方式）。如果计算机具有 4K 的页，并在可执行文件中文本段结束于 0x80045ff，然后数据段起始于 0x8005600。文件中的这一页（即同时存有文本和数据段的页）在内存 0x8004000 处被映射为文本段的最后一页（头 0x600 个字节包含文本段中 0x8004000 到 0x80045ff 之间的内容），并在 0x8005000 处被映射为数据段（0x600 以后的部分包含数据段从 0x8005600 到 0x80056ff 的内容）。

BSS 段也是在逻辑上也是跟在数据段的可读写区段后，在本例中长度为 0x1300 字节，即文件中尺寸与内存中尺寸的差值。数据段的最后一页会从文件中映射进来，但是在随后操作系统将 BSS 段清零时，copy-on-write 系统会该段做一个私有的副本。

如果文件中包含 .init 或 .fini 区段，这些区段会成为只读文本段的一部分，并且链接器会在程序入口点处插入代码，使得在调用主程序之前会调用 .init 段的代码，并在主程序返回后调用 .fini 区段的代码。

ELF 共享目标包含了可重定位和可执行文件的所有东西。它在文件的开头具有程序头部表，随后是可加载段的各区段，包括动态链接信息。在构成可加载段的各区段之后的，是重定位符号表和链接器在根据共享目标创建可执行程序时需要的其它信息，最后是区段表。

## ELF 格式小结

ELF 是一种较为复杂的格式，但它的表现和预期的一样好。它既是一个足够灵活的格式（可以支持 C++），又是一种高效的可执行格式（对于支持动态链接的虚拟内存系统），同时也可以很方便的将可执行程序的页直接映射到程序的地址空间。它还允许从一个平台到另一个平台的交叉编译和交叉链接，并在 ELF 文件内包含了足以识别目标体系结构和字节序的信息。

## IBM 360 目标格式

IBM 360 系统上的目标文件格式是在 60 年代设计的，但一直沿用至今。它最初是为 80 列的穿孔卡设计的，但后来被现代系统的磁盘文件所采用。每个目标文件包含一系列的控制区段 (csect)，即单独的可重定位代码或数据块的另一种可选命名。一个源代码例程通常会被编译到一个 csect 中，或将代码编入一个 csect，数据编入另一个 csect。如果一个控制区段有名字的话，它可以用来作为寻址该控制区段起始地址的符号；其它类型的符号包括控制区段内定义的符号、未定义的外部符号、公共块和少数其它类型。每一个在某个目标文件中定义或使用的符号都有一个很小的整数型的标识符，称为外部符号 ID (ESID: External Symbol ID)。一个目标文件就是由长度为 80 字节的通用格式的记录组成的序列，如图 17 所

示。每一个记录的第 1 个字节均为 0x02，它将该记录标识为目标文件的一部分（起始为空格的记录会被当作链接器的命令来对待）。第 2 个到第 4 个字节是记录的类型，程序代码或文本的类型为 TXT，定义了符号和 ESD 的外部符号目录的类型为 ESD，重定位目录的类型为 RLD，最后一个记录（同时定义了起始点）的类型为 END。接下来一直到第 72 个字节是由记录类型决定的。第 73 到 80 字节被忽略，在现实中的穿孔卡上他们实际就是卡序号。

一个目标文件由若干定义控制区段 (csect) 和符号的 ESD 记录开始，然后依次是 TXT 记录，RLD 记录和 END 记录。这些记录的顺序相当灵活。若干 TXT 记录可以对单个位置的内容重复定义，而文件中的最后一个会胜出。这就可以在卡片盒子的再追加打孔卡作为“补丁”，而不是重新汇编或编译。

---

图 3-17：IBM 目标记录格式

```
char flag = 0x2;  
char rtype[3]; // 3 字节记录类型  
char data[68]; // 格式特定数据  
char seq[8]; // 忽略，通常是序号
```

---

## ESD 记录

每个文件都是以 ESD 记录开头的，如图 18 所示，定义了文件中使用的控制区段 (csect) 和符号，并为它们分配了 ESD。

每个目标文件都以 ESD 开始，如图 18 所示，定义了文件中使用的 csect 和符号，并为它们都分配 ESD。

---

图 3-18：ESD 格式

```
char flag = 0x2; // 字节 1  
char rtype[3] = "ESD"; // 字节 2-4, 3 字节类型  
char pad1[6];  
short nbytes; // 字节 11-12, 信息中的字节数: 16, 32 或 48  
char pad2[2];  
short esid; // 字节 15-16, 第一个符号的 ESD  
{ // 字节 17-72, 共计 3 个符号  
    char name[8]; // 由空白补齐的符号名  
    char type; // 符号类型  
    char base[3]; // 控制区段源或标签偏移量  
    char bits; // 属性位  
    char len[3]; // 目标长度或控制区段 ID  
}
```

---

每条记录可定义 ESID 连续的 3 个符号，符号可以容纳 8 个 EBCDIC 字符。符号类型有：

- Sd 和 PC：区段定义（Section Definition）或私有代码（Private Code），定义了一个控制区段（csect）。该控制区段的起始地址（base）是区段的逻辑起始地址，通常为零，长度就是 csect 本身的高度。属性字节包含了指明 csect 使用 24 位或 3 位程序寻址、需要被加载到 24 位或 31 位地址空间的标志。PC 是名字空白的控制区段；控制区段的名字在程序中必须是唯一的，但可以存在多个未命名的 PC 区段。
- LD：标签定义（Label Definition）。基地址（base）是标签在所属控制区段中的偏移量，高度域为该控制段的 ESID。无属性位。
- CM：公共块。高度就是该公共块的高度，其它域忽略。
- Er 和 WX：外部引用（External Reference）或弱外部（Weak External）。均为其它地方定义的符号。链接器会对一个未在程序中其它地方定义的 ER 类符号报告一个错误，但对 WX 类符号而言这不是错误。
- PR：伪寄存器，一个在链接时定义但由加载时分配的小存储区域。由属性位提供所需的对齐要求（1—8 字节），高度就是这段区域的大小。
- PR：精灵寄存器，一个在链接时定义但在运行时分配的小的存储区域。属性位给出相应的对齐要求，1 对齐到 8 字节，len 是这个区域的大小。

## TXT 记录

接下来是文本记录，如图 19 所示，其中包含了程序代码和数据。每个文本记录定义了一个控制段中连续的 56 个字节。

图 3-19：TXT 记录格式

```
char flag = 0x2;          //字节 1
char rtype[3] = "TXT"; //字节 2-4，三个字母的类型字串
char pad;
char loc[3];           //字节 6-8，控制区段在文本中的相对地址
char pad[2];
short nbytes;           //字节 11-12，信息的字节数
char pad[2];
short esid;              //字节 15-16，该控制区段 ESID
char text[56];           //字节 17-72，数据
```

## RLD 记录

文本记录后面是 RLD 记录，如图 20 所示，其中包含了一系列的重定位项。

图 3-20: RLD 格式

```
char flag = 0x2;          //字节 1
char rtype[3] = "TXT"; //字节 2-4, 3 字节的类型字串
char pad[6];
short nbytes;           //字节 11-12, 信息的字节数
char pad[7];
{           //字节 17-72, 4 或 8 字节的重定位项
    short t_esid; //目标, 被引用的 csect 或 symbol 的 ESID,
                    //或 CXD 的 0 (PR 定义的总尺寸)
    short p_esid; //指针, 所引用控制区段的 ESID
    char flags;   //引用的类型和尺寸
    char addr[3]; //相对于控制区段的引用地址
}
```

---

每一个重定位项都有目标和指针的 ESID，一个标志字节和指针的（控制区段）相对地址。标志字节给出了引用的类型（代码、数据、PR 或 CXD）和长度（1、2、3、4 字节），一个符号位指明是加上还是减去重定位地址，此外还有一个“相同（same）”位。如果“相同”位被设置，则下一个重定位项（忽略自己的两个 ESID）采用与当前项相同的 ESID。

## END 记录

图 21 所示为 END 记录，其中给出了程序的起始地址，它要么是某个控制区段内的地址，或者是某个外部符号的 ESID。

图 3-21: END 格式

```
char flag = 0x2;          //字节 1
char rtype[3] = "END"; //字节 2-4, 三字母的类型字串
char pad;
char loc[3];            //字节 6-8, 相对于控制区段的起始地址, 或 0
char pad[6];
short esid;              //字节 15-16, 控制区段或符号的 ESID
```

---

## 小结

尽管 80 列记录已经相当过时了，但是 IBM 的这种目标文件格式仍然惊人的简单和灵活。非常小的链接器和加载器就可以处理这种格式。在 IBM 360 系统的某一个型号上，我曾经使用过一个加载器，可以整个容纳在一张 80 列打孔卡上，能够加载一个程序，解释 TXT 和 END

记录，并忽略其它内容。

基于磁盘的系统，或者以卡镜像的方式存储目标文件，或者使用该格式的一个变种（采用记录类型相同但尺寸长得多且没有序列号的记录）。在 DOS ( IBM 为 360 开发的一种轻量级操作系统) 上的链接器会产生没有 ESID、仅由单个控制区段和缺去符号的 RLD 记录的一种简化输出格式。

在目标文件中，唯一命名的控制区段可以让程序员或链接器按自己的要求来组织程序中的模块，例如将所有的代码控制区段都放在一起。能够体现这种格式寿命的主要地方就是最长 8 个字节的符号，以及每个控制区段都没有类型信息。

## 微软可移植可执行体格式

微软 Windows Nt 系统非常混杂的继承了包括 MSDOS 和 Windows 早期版本、Digital VAX VMS (很多程序员都在上面工作过的平台)、UNIX 系统 5 (其余的很多工程师都工作过的平台) 的特性。NT 的格式是从 COFF ( UNIX 在 a.out 之后、ELF 之前使用的一种文件格式) 继承而来的。这里我们将会看看 PE 格式，以及它和微软 PE (即微软的 COFF 版本) 的区别。

Windows 是在低速处理器、有限内存容量、最初没有硬件分页的低端环境中开发的，所以共享库总是要强调整节省内存，并采用特定技巧来提升性能 (某些技巧在 PE/COFF 设计中来看都是显而易见的)。多数 Windows 可执行程序包含很多 “资源”，这是一个指代程序和 GUI 之间共享的诸如光标、图标、位图、菜单、字体等对象的通用名词。PE 格式文件会为程序代码在该文件中使用的所有资源包含一个资源目录。

PE 可执行文件是专为分页系统设计的，因此 PE 文件中的页通常可以直接被映射到内存中并运行，这与 ELF 可执行文件很相似。PE 格式文件要么是 EXE 程序，要么是 DLL 共享库 (即动态链接库)。这两种类的格式是相同的，通过一个状态位来标识这个 PE 文件属于哪一类。每一类都会包含一个可被加载到相同地址空间的其它 PE 文件使用的导出函数和数据的列表，以及一个需要在加载时从其它 PE 处解析的导入函数和数据的列表。每个文件都包含一系列与 ELF 段类似称为区段、段和目标的块 (chunk)。这里，我们将这种块称为区段，这是现在针对微软使用的名词。

如图 22 所示，一个 PE 文件，起始于一个小 DOS .EXE 文件 (打印诸如 “This program needs Microsoft Windows.” 的消息，微软对诸如此类的向后兼容问题投入了相当多的精力)。EXE 文件头部末尾一个先前未使用过的域指向了 PE 的识别符。紧跟在识别符后面的是包含有一个 COFF 区段和 “可选” 头部的文件头部 (在所有 PE 文件中忽略其名字)，再后面是区段头部的列表。区段头部描述了文件内不同种类的区段。COFF 目标文件以 COFF 头部开始并忽略可选头部。

---

图 3-22：微软 PE 和 COFF 文件格式

DOS 头部 (仅对 PE 格式)

DOS 程序部分 (仅对 PE 格式)

PE 标识符 (仅对 PE 格式)

COFF 头部

可选 header (仅对 PE 格式)

区段表

可映射区段 (pointed to from section table)

COFF 行号、符号和调试信息 (在 PE 文件中可选)

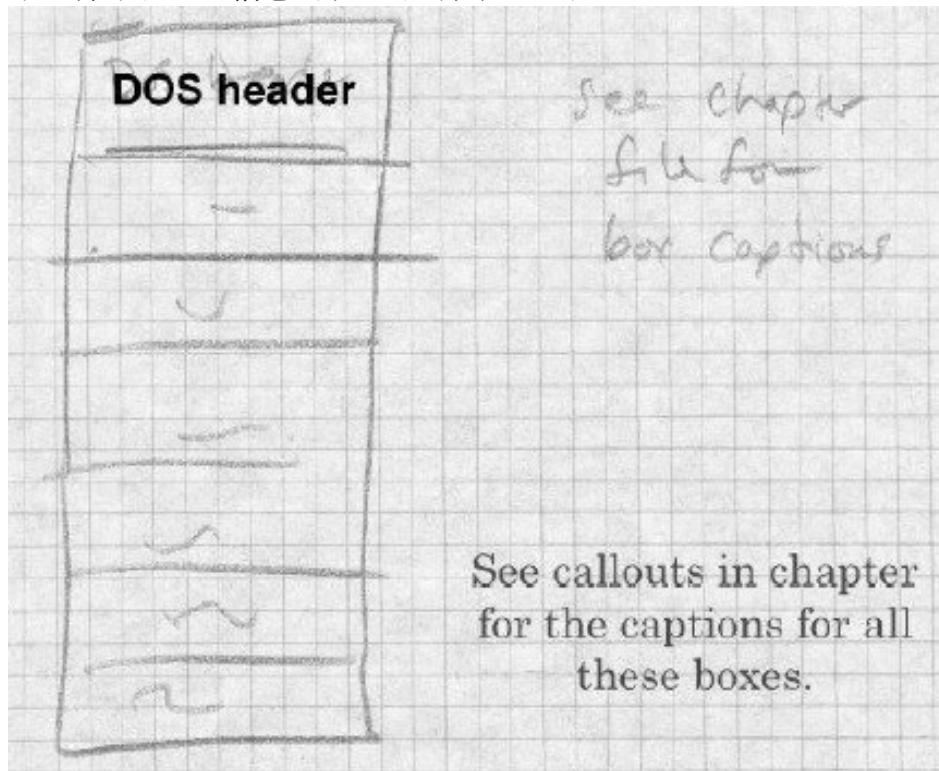


图 23 显示了 PE、COFF 和“可选”头部。COFF 头部描述了文件的内容，其中最重要的内容是区段表中的表项数目。“可选”头部中包含指向文件中若干最常用区段的指针。这里的地址都是相对于程序被加载到内存中位置的偏移量，也被称为相对虚拟地址 (RVA)。

图 3-23：PE 和 COFF 头部

PE 标识符

```
char signature[4] = "PE\0\0"; //幻数，同样也表示了字节顺序
```

COFF 头部

```
unsigned short Machine; //要求的 CPU，如 0x14C 为 80386，等等
```

```
unsigned short NumberOfSections; //区段数，无则为零
```

```
unsigned long TimeDateStamp; //创建时间，无则为零
```

```
unsigned long PointerToSymbolTable; //COFF 格式中符号表在文件  
//内的偏移量，若无则为零
```

```
unsigned long NumberOfSymbols; //COFF 符号表中表项数目，若无  
//则为零
```

```
unsigned short SizeOfOptionalHeader; //随后的“可选”头部的尺寸
```

```
unsigned short Characteristics; //特性，0x02 为可执行，0x200 为  
//不可重定位，0x2000 为 DLL
```

Pe 头部后面的可选头部，在 COFF 目标文件中没有

```
//COFF 域
unsigned short Magic; //8 进制的 413, 由 a.out 的 ZMAGIC 而来
unsigned char MajorLinkerVersion;
unsigned char MinorLinkerVersion;
unsigned long SizeOfCode; // .text 段的尺寸
unsigned long SizeOfInitializedData; // .data 段的尺寸
unsigned long SizeOfUninitializedData; // .bss 段的尺寸
unsigned long AddressOfEntryPoint; // 入口点的相对虚拟地址 RVA
unsigned long BaseOfCode; // .text 段的相对虚拟地址 RVA
unsigned long BaseOfData; // .data 段的相对虚拟地址 RVA
//附加域
unsigned long ImageBase; // 映射文件起始位置的虚拟地址
unsigned long SectionAlignment; // 区段对齐, 典型的是 4096 或 64K
unsigned long FileAlignment; // 文件页对齐, 典型是 512
unsigned short MajorOperatingSystemVersion;
unsigned short MinorOperatingSystemVersion;
unsigned short MajorImageVersion;
unsigned short MinorImageVersion;
unsigned short MajorSubsystemVersion;
unsigned short MinorSubsystemVersion;
unsigned long Reserved1;
unsigned long SizeOfImage; // 可映射镜像的尺寸总和, 按照
                           // SectionAlignment 补齐
unsigned long SizeOfHeaders; // 整个区段表中头部的尺寸总和
unsigned long CheckSum; // 常为 0
unsigned short Subsystem; // 要求的子系统: 1 为 Native, 2 为 Windows
                         // 图形用户界面, 3 为不带图形界面的
                         // Windows, 5 为 OS/2, 7 为 POSIX
unsigned short DllCharacteristics; // 何时调用初始化例程 (逐渐被
                                   // 废弃的):
                           // 1 为进程启动时, 2 为进程结束
                           // 时, 4 为线程启动时, 8 为线程
                           // 结束时
unsigned long SizeOfStackReserve; // 预留栈大小
unsigned long SizeOfStackCommit; // 初始分配的栈大小
unsigned long SizeOfHeapReserve; // 预留堆大小
unsigned long SizeOfHeapCommit; // 初始分配堆大小
```

```
unsigned long LoaderFlags;           //废弃的
unsigned long NumberOfRvaAndSizes; //随后的镜像数据目录中的目录项
                                  //个数
//下面重复的数据对，每一项对应一个目录项
{
    unsigned long VirtualAddress; //目录的相对虚拟地址
    unsigned long Size;
}
```

这些目录依次为：

- 导出目录
- 导入目录
- 资源目录
- 例外目录
- 安全目录
- 基地址重定位表
- 调试目录
- 镜像描述字串
- 机器特定数据
- 线称本地存储目录
- 加载配置目录

---

每个 PE 文件都是以系统加载器可以很容易将其映射到内存中的方法来创建的。每一个区段都和物理磁盘块边界对齐（或根据 filealign 值采用更大的对齐边界），并在逻辑上与内存页的边界对齐（x86 上为 4096）。链接器会为每个 PE 文件创建一个文件将被映射的特定目标地址（imagebase）。如果该地址所在的地址空间区域有效（几乎总是有效的），就不需要进行加载时的调整了。在少数情况下诸如旧的 win32 兼容系统目标地址不可用，则加载器必须将文件映射到其它地方，这种情况下文件必须在 .reloc 区段中包含重定位调整信息以告诉链接器要修改什么。共享的 DLL 库也是重定位的一个关注点，这是因为 DLL 映射的地址是依赖于地址空间当时的实际情况的。

在 PE 头部后面是区段表，这是有图 24 所示的数据结构组成的数组。

---

图 3-24：区段表

```
//表项构成的数组
unsigned char Name[8]; //ASCII 编码的区段名称
unsigned long VirtualSize; //映射到内存中的尺寸
unsigned long VirtualAddress; //相对于镜像基地址的内存地址
unsigned long SizeOfRawData; //物理尺寸，对齐于文件对齐要求
                           //的倍数
unsigned long PointerToRawData; //文件偏移量
```

```
//接下来的四项出现在 COFF 中，在 PE 中为 0 或没有
unsigned long PointerToRelocations; //重定位项的偏移量
unsigned long PointerToLinenumbers; //行号项的偏移量
unsigned short NumberOfRelocations; //重定位项的数目
unsigned short NumberOfLinenumbers; //行号项的数目
unsigned long Characteristics; //特性，0x20 为文本，0x40 为数据，
                                //0x80 为 BSS，0x200 为不加载，
                                //0x800 为不链接，0x10000000 为共
                                //享的，0x20000000 为可执行，
                                //0x40000000 为可读，0x80000000 为
                                //可写
```

---

每个区段都同时具有文件地址 (PointerToRawData)、尺寸 (SizeOfRawData)，以及内存地址 (VirtualAddress)、尺寸 (VirtualSize)，它们并不需相同。处理器的页尺寸经常会比磁盘块尺寸大，通常页大小为 4K，磁盘块为 512 字节，在一页中间就结束了的区段也不需要为页中剩余的空间分配磁盘块，这样可以节省少量的磁盘空间。每个区段都会根据页面对应的硬件权限来进行标识，例如对于代码标识为可读+可执行，对数据标识为可读+可写。

## PE 特有区段

Pe 文件除了像 UNIX 可执行程序那样包含 .text、.data 和 .bss (有些时候) 外，还包含有很多 Windows 特有的区段。

- 导出区段：在当前模块中定义并对其它模块可见的符号列表。可执行程序通常不导出符号，或为调试导出少数符号。DLL 会为它们所提供的例程和数据导出符号。为了保持 Windows 节省空间的传统，被导出的符号也可以通过称为导出序号的小整数来引用，就像引用符号名字那样。导出区段包含一个由被导出符号的相对虚拟地址 (RVA) 组成的数组，它还包含两个对应的数组，即符号名称组成的数组 (ASCII 字串的相对虚拟地址)，和符号导出序号组成的数组，依据字串名称排序。如果要通过名称来查找一个符号，首先要在字串名表中进行折半查找，并根据发现的名字查找对应于序号表中的表项，然后用这个序号来索引 RVA 的数组 (这比反复遍历三个字的表项要快)。当 RVA 指向一个命名在另一个库中发现的符号的字串时，导出区段也扮演了“转发者”的角色。
- 导入区段：导出表列出了所有需要在加载时从 DLL 中进行解析的符号。链接器会预先确定符号可以在哪些 DLL 中被找到，因此导入表的开始首先是导入目录，每个被引用的 DLL 对应一个目录项。每个目录项包含有 DLL 的名字，对应的数组一个标识所需的符号，另一个是在镜像文件中存储符号值的位置。第一个数组中的表项可以是序号 (若高位被置位) 或者在猜测序号之前指向名字串的指针以提高查找的速度。

第二个数组包含着存储符号值的位置；如果该符号是一个过程，那么链接器会已经调整了所有对它的调用而通过该位置进行间接调用，如果该符号是数据，则在导入模块中的引用会将这个位置作为指向实际数据的指针来使用（某些编译器会自动提供重定向，而另一些则需要显式的程序代码）。

- 资源区段：资源表以树的结构来组织。这个数据结构允许任意深度的树，但实际上这个树是三级的，对应资源类型，名称和语言（语言在这里是指自然语言，这允许将可执行程序定制为英语以外的语言）。每个资源都可以有一个名字或编号。一个典型的资源可以是类型为 DIALOG（对话框），名称为 ABOUT（程序中的“关于”框），语言为英语。于那些 ASCII 名称的符号不同，资源使用 Unicode 名称以支持英语外的语言。实际的资源是二进制数据组成的块，资源的格式取决于资源的类型。
- 线程本地存储区段：Windows 支持进程的多线程执行。每个线程可以有自己的私有存储空间，称为线程本地存储区段（Thread Local Storage 或 TLS）。该区段指向一块用来在线程启动时初始化 TLS 的镜像数据，以及若干在线程启动时调用的初始化例程的指针。该区段通常出现在 EXE 文件中而在 DLL 中没有，这是因为在在一个程序动态的链接到某个 DLL 时 Windows 不会分配 TLS 存储区域（见第十章）。
- 调整区段：如果某个可执行程序要被移动，它会被作为一个整体移动这样所有的调整项都具有相同的值，即目标地址和实际加载地址的差值。调整项表，如果存在的话，会包含一个由调整块组成的数组，每一项都包含被映射程序一个 4K 页的调整信息（没有调整表的可执行程序只能按链接的目标地址来加载）。每个调整块包含该页的相对虚拟地址（RVA）的基地址，调整项的数目，和一个由 16 位调整项组成的数组。每一项的低 12 位是需要重定位的块内偏移量，高 4 位是调整类型，例如加上 32 位的值，调整高 16 位或低 16 位（对于 MIPS 架构）。这种一块一块的策略可以是重定位表项大小压缩为 2 个字节（而不是 ELF 格式中的 8 或 12 个字节），因此在重定位表中节省了相当可观的空间。

## 运行 PE 可执行文件

启动一个 PE 可执行程序的过程是相对简单的。

- 读入文件的第一页，其中有 DOS 头部，PE 头部和区段头部等。
- 确定地址空间的目标区域是否有效，如果不可用则另分配一块区域。
- 根据各区段头部的信息，将文件中的所有区段映射到地址空间的适当位置上。
- 如果文件并没有被加载到它的目标地址中，则进行重定位。
- 遍历导入区段中的 DLL 列表，将任何未加载的库都加载（该过程可以是递归的）。
- 解析所有在导入区段中的导入符号。
- 根据 PE 头部的值创建初始的栈和堆。
- 创建初始线程并启动该进程。

## PE 和 COFF

Windows 的 COFF 可重定位目标文件具有像 PE 那样的 COFF 文件头部和区段头部，但结构与可重定位的 ELF 文件更相似。COFF 文件没有 DOS 头部和位于 PE 头部后面的可选头部。每个代码和数据区段都带有重定位和行号信息（EXE 文件中的行号信息集中在一个不被加载器处理的调试区段中）。COFF 目标文件像 ELF 文件那样，具有相对于区段的重定位（而不是 R VA 相对重定位），并总是包含一个所需符号的符号表。对于语言编译器而言 COFF 文件不包含任何的资源，资源位于由资源编译器创建的独立的目标文件中。

COFF 同样具有一些 PE 文件没有的区段类型。最值得关注的是包含链接器所需的文本命令字串的 `.directive` 区段。编译器通常通过 `.directive` 区段告诉链接器去寻找恰当的特定语言库。包括 MSVC 在内的一些编译器也带有链接器命令以便在创建 DLL 时导出代码和数据（这种将命令和目标代码混在一起的方法又会到了以前；IBM 的链接器早在 60 年代就可以接受将命令和目标代码混在一起的卡片了）。

## PE 文件小结

对于一个支持虚拟内存的线性寻址操作系统而言，PE 文件格式是相当不错格式（还具有少量从 DOS 继承而来的历史包袱）。它还包括了一些额外特性，诸如专为提高小型系统上程序加载速度的序号式的导入、导出（但其在当代 32 位系统上的效率还是有待商榷的）。针对 16 位分段可执行程序的 NE 格式早期版本要复杂得多，因此 PE 的改进是相当明显的。

## Intel/Microsoft 的 OMF 文件格式

本章中我们倒数第二要看看的格式是仍在使用的最古老的格式之一，Intel 目标模块格式（Intel Object Module Format，OMF）。Intel 最初是在 70 年代后期为 8086 定义的 OMF 格式。多年后众多厂商诸如 MS，IBM 和 Phar Lap（它们写了一套广为使用的基于 DOS 的 32 位扩展工具集），都定义了它们自己的扩展。现在的 Intel OMF 格式是最初的 Intel 规范与多数扩展的结合（除了那些与其它扩展冲突或从未被使用过的扩展）。

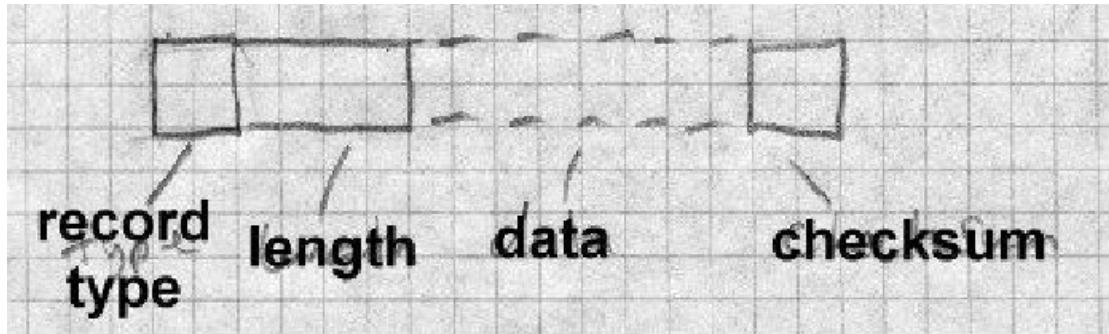
我们目前为止看到的格式都是针对那些具有随即访问磁盘和足够的内存、直接处理编译和链接的环境来设计的。OMF 出现时正值微处理器开发的早期（那时内存容量都很小而且存储都是基于打孔纸带的）。因此，OMF 将目标文件划分为一系列的短记录，如图 25 所示。每个记录包含 1 字节的类型，2 字节的长度，内容，和可以让整个记录的字节和为零的校验字节（纸带设备没有内建的检错机制，而灰尘或者粘住导致的错误并不少见）。OMF 文件设计为可以在没有大量存储空间的机器上以最少的扫描次数来让链接器完成它的工作。通常会采用一种  $1\frac{1}{2}$  遍扫描的技巧，先进行一遍不完全扫描以找到那些靠近每个文件前部的符号名称，然后再进行完全的扫描来进行链接，并产生输出。

---

图 3-25：OMF 记录格式

下图包括

- 1字节类型
- 2字节长度
- 可变长度的数据（长度为 length）
- 1字节校验和



为了满足处理 8086 分段的架构，OMF 格式非常的复杂。OMF 链接器的主要目标之一就是将代码和数据装入数目尽可能少的段或段组中。OMF 文件中的每一个代码或数据块都是和某个段对齐的，并且每一个段会属于某个段组“segment group”或者段类“segment class”（段组必须足够小以便可以在单个段内寻址，而段类可以是任意尺寸，因此段组可以用作寻址和存储管理，而段类只能用于存储管理）。代码可以通过名字引用段或者段组，也可以使用基于段或者段组基地址的相对地址来引用段内的代码。

OMF 格式还包含一些对覆盖链接（overlay linking）的支持，尽管我知道的 OMF 链接器中还没有哪个对此支持，而是使用一个单独的链接器命令文件来替代覆盖指令。

## OMF 记录

目前 OMF 至少定义了 40 种记录类型，不能一一在此列举，因此我们只会看一个简单的 OMF 文件（完整的规范可以在 Intel 的 TIS 文档中看到）。

OMF 使用了一些编码技巧使得记录尽可能的短小。所有的名称字串都是变长的，存储时首先是长度字节，然后是字串。空名称（在某些情况下是有效的）就是一个值为零的字节。OMF 模块通过名字来引用段、符号、段组等。OMF 模块将每个名字（只列一次）列在 L NAMES 类型记录中，然后依次使用名字列表的序号来定义段、段组、和符号等的名字。即第一个名字是 1，第二个名字是 2，以此类推遍历所有名字而不必担心到底有多少个 L NAMES 记录（当某些定义使用相同的名字时这会节省一点点空间）。索引 0 到 0x7f 存储为 1 个字节，索引 0 x80 到 0x7fff 存储为 2 个字节（第一个字节的高位置位表明这是一个 2 字节的序号）。奇怪的是第一个字节的低 7 位是索引值的高 7 位，第二个字节的 8 位是索引值的底 8 位，这正好与 Intel 本身的字节序相反（译者：所以作者才感到奇怪）。段、段组、和外部符号同样也是通过各自独立的索引序列号来引用。例如，某个模块列有名称 DGROUP、CODE 和 DATA，对应定义的名字索引为 1、2 和 3。然后模块定义了两个段 CODE 和 DATA，在名字列表中的索引为 2 和 3 索引。由于 CODE 是第一个被定义的，所以它对应的段索引就是 1，而 DATA 则为 2。

最初的 OMF 格式是基于 16 位 Intel 架构定义的。对于 32 位程序，由于地址尺寸的原因在 OMF 中定义了新的记录类型。由于所有的 16 位记录类型正好对代码的索引都是偶数，所以对应的 32 位记录类型会对代码采用比 16 位类型大 1 的奇数作为索引。

## OMF 文件的细节

图 26 列出了一个简单 OMF 文件中的记录。

---

图 3-26：典型的 OMF 记录序列

THEADR 程序名称

COMENT 标志和选项

LNAMES 段、段组和段类的名字列表

SEGDEF 段(每个段一个记录)

GRPDEF 段组(每个段组一个记录)

PUBDEF 全局符号

EXTDEF 未定义的外部符号(每个符号一个记录)

COMDEF 公共块

COMENT 第一遍扫描信息

LEDATA 代码或数据块(多个)

LIDATA 重复数据块(多个)

FIXUPP 重定位和外部引用修正信息，跟在所引用的 LEDATA 或 LIDATA 后面

MODEND 模块结尾

---

文件开头是 THEADR 记录，它标识了模块的开始，并以字串的形式提供了该模块对应源代码文件的名称（如果该模块是库的一部分，则它的开头是结构类似的 LHEADR 记录）。

第二个记录是（很糟糕的起错名字的）COMENT 记录，它包含着针对链接器的配置信息。每个 COMENT 记录还包含一些标志位指明在链接时是否保留注释，此外还有 1 字节的类型，以及注释文本。某些注释类型其实本身就是注释，例如编译器的版本信息和版权提示，但某些注释类型提供了必须的链接器信息，诸如使用的内存模式（从 Tiny 到 Large）、处理完该文件后需要搜索的库名称、弱外部符号的定义，以及厂商放入 OMF 格式中的其它类型的数据。

接下来是一系列的 LNAMES 记录，列出了在本模块中为段、段组、段类和覆盖所使用的所有名字。如上面所说明的那样，所有 LNAMES 中的所有名字在逻辑上可以看成是一个数组（其中第一个名字的索引为 1）。

在 LNAMES 记录后是 SEGDEF 记录，每一项对应一个在当前模块定义的段。SEGDEF 记录拥有一个对应其段名称的索引，或者是它所属段类、覆盖的名称索引（如果说有的话）。同样被包含的还有段的属性，包括对齐要求、于其它模块中同名段合并的规则，以及段的长度。

接下来是 GRPDEF 记录，如果说有的话，定义了当前模块中的段组。每个 GRPDEF 记录都具有组名的索引以及组中各段的索引。

PUBDEF 记录定义了对其他模块可见的“公共”符号。每个 PUBDEF 定义了在单个段组或段内的一或多个符号。该记录包含有段和段组的索引，以及每个符号对应在段或段组内的偏移量，它的名字，和一个单字节大小的编译器特定的类型域。

EXTDEF 记录定义了那些尚未被定义的外部符号。每个记录包含一个符号的名称，以及一到两个字节的调试器符号类型。COMDEF 记录定义了公共块，除还需要定义符号的最小尺寸外它与 EXTDEF 记录很相似。所有在模块中的 EXTDEF 和 COMDEF 记录在逻辑上组成一个数组，这样地址调整信息可以通过索引来引用他们。

接下来是可选的特殊的 COMENT 记录，它标记了第一遍扫描数据的结尾。它告诉链接器可以在链接过程的第一遍扫描中跳过文件中剩余部分。

文件的剩余部分由程序实际的代码和数据组成，其间混杂着包含重定位和外部引用信息的调整信息记录。共有 LEDATA（枚举）和 LIDATA（迭代）两类数据记录。LEDATA 只有段索引和起始偏移量，跟在数据后边并存储在那里。LIDATA 同样开头是段索引和起始偏移量，但随后可能会存在一系列嵌套的重复数据块。LIDATA 可以高效处理由如下所示 Fortran 语句生成的代码：

```
INTEGER A(20, 20) /400*42/
```

这样一个 LIDATA 可以拥有一个 2 或 4 字节的数值为 42 的块，并将其重复 400 次。

每一个需要地址调整的 LEDATA 或 LIDATA 必须马上在后面跟一个 FIXUP 记录。FIXUP 是截止现在最为复杂的记录类型。每个 FIXUP 具有三个项目：第一个是目标（target），即被引用的地址；第二个是框架（frame），即在段或段组中相对于要被计算的地址的位置；最后第三个是要被调整的位置。由于在多个调整信息中引用某个框架是非常普遍的，而在多个调整信息中引用某个目标地址也是颇为普遍的，OMF 定义了调整线索（fixups threads），作为框架或目标的简捷表达（宽度为 2 位的代码），这样通过线索号可以在任何地方定义总计 4 个框架和 4 个目标，每个线索号可以在必要时重复定义。例如，如果一个模块包含一个数据段组，该段组被模块中几乎所有的数据引用当作框架来使用，因此为该组的基址定义一个线索号可以节省大量的空间。在实际中，GRPDEF 记录后面几乎总是会跟着一个为该段组定义了一个框架线索的 FIXUP 记录。

每个 FIXUP 记录是一个子记录的序列，每个子记录要么定义了一个线索（thread），要么定义了一个调整信息（fixup）。定义线索的记录用标志位来表明它定义的是框架还是目标的线索。定义目标线索的记录包含有线索号、引用的类型（段相对引用，段组相对引用，外部相对引用）、基段或段组或符号的索引、和可选的基址偏移量。一个定义框架的记录包含线索号和引用类型（目标定义中的所有类型再加上两个常见的特殊情况：作为位置的段相同，以及作为目标的段相同）。

一旦定义了线索，调整信息的子记录就相对简单了。它包含需要调整的地址，指明调整类型的代码（16 位偏移量、16 位段、完整的“段基址：偏移量”地址、8 位相对地址，等等），以及框架和目标。框架和目标可以引用先前定义的线索，或在适当的地方被指定。

在 LEDATA、LIDATA 和 FIXUP 记录之后，模块的结尾处有一个 MODEND 记录标记，如果当前模块是程序的主例程的话该记录还能（并非必需）指定入口点。

一个真正的 OMF 文件还具有很多针对本地符号、行号和其它调试信息的记录类型，在 W

indows 环境下还要包括在目标 NE 文件（针对可分段 16 位处理器的 PE 格式）中创建导入、导出区段的信息，但模块的结构没有变化。各个记录的顺序颇为灵活，尤其是当第一遍扫描结束标志不存在时。仅有的几条强制规则就是 THEADER 和 MODEND 必须是第一个和最后一个记录，FIXUP 必须紧随相关的 LEDATA 或 LIDATA，不允许模块内的间接引用（forward references）。在实际中，还可以在定义符号、段和段组时生成对应的记录，只要它出现在引用它们的其它记录之前。

## OMF 格式小结

相比于我们已经看到的其它格式，OMF 格式是颇为复杂的。造成这种复杂性的部分是采用了压缩数据的一些技巧，部分是由于将各个模块都划分为许多小记录，部分是由于多年来累计增加的各种特性，部分是从分段的程序寻址上继承而来的。对于已分类记录的记录格式一致性是非常重要的，它既可以允许非常简单的进行扩展，也可以允许处理 OMF 文件的程序跳过那些它们不知道的记录类型。

然后，今天即使是小型的桌面电脑也可以拥有成兆字节（megabyte）的内存和大容量磁盘，OMF 将目标文件划分为很多小记录的方法越发显得弊大于利。直到 70 年代采用小记录的目标模块相当普遍，但是现在已经过时了。

## 不同目标格式的比较

本章中我们已经看到了 7 种不同的目标和可执行格式，从最简单的（.COM）到成熟的（ELF 和 PE）再到已经过时的（OMF）。诸如 ELF 这样的现代目标格式会尝试将所有的数据都收集为一种类型以更方便链接器的处理。它们在安排文件布局时还考虑到了虚拟内存的因素，这样系统加载器就可以使用尽可能少的额外工作将文件映射到程序的地址空间中去。

每种目标格式都显示了它所工作的操作系统的风格。UNIX 系统保持了简捷并且定义良好的内部接口，a.out 和 ELF 格式将这些都体现在颇为简单的结构以及缺少特例处理上。而 Windows 则相反，将进程管理和用户接口也纠缠在了一起。

## 项目

这里我们定义一种用于本书的项目作业中的简单目标格式。与其它目标格式不同，该类型完全是由一行行的 ACSII 文本构成的。这样就可以很容易的用文本编辑器来创建简单的目标文件，同样也就可以很容易的检查项目链接器的输出文件。图 27 为该格式的草图。段、符号和重定位项均由一行一行的文本组成（各个域通过空格来划分）。程序会忽略每一行末尾额外的域，数字均为 16 进制的。

图 3-27：项目目标文件格式

LINK

*nsegs nsyms nrels*

```
-- segments --
-- symbols --
-- rels --
-- data --
```

---

第一行是幻数，即单词 LINK。

第二行至少包含 3 个十进制数字，依次为文件中段的个数，符号表项的个数，和重定位项的个数。对于这个链接器的扩展版本，也可以在这三个数字后面追加其它的信息。如果不存在符号和重定位项，则对应的数字为 0。

接下来是段的定义。每一个段的定义包含段的名字，段的逻辑起始地址，以字节为单位的段长度，和一个描述该段的编码字母串。编码字母包括 R（可读）、W（可写）、P（存在于目标文件中），其实也可以使用其它字母来代替。一个类 a.out 文件中典型的段可以是：

```
.text      1000  2500  RP
.data      4000    C00        RWP
.bss       5000  1900  RW
```

各段根据它们出现的先后顺序编号，第一个段的编号为 1。

接下来是符号表，每个表项的格式如下：

```
name          value  seg          type
```

name 就是符号的名称，value 是符号的 16 进制数表示的值，seg 是于定义该符号的位置相对的段编号，如果是绝对符号或未定义符号则为 0。type 是字母串（D 表示已定义，U 表示未定义）。符号也同样根据它们被列出的顺序来依次编号，编号从 1 开始。

接下来的是重定位信息，每行对应一项：

```
loc          seg          reg          type ...
```

loc 是要被重定位的位置，seg 是可以找到该位置的段，ref 是要被重定位的段或符号的编号，type 是与体系结构相关的重定位类型。通常类型有 A4（4 字节绝对地址），R4（4 字节相对地址）。某些重定位类型在 type 后面还可能有额外的域。

重定位信息后面的是目标文件的数据。每个段的数据是以换行为结尾的十六进制数字的长串（这样在 perl 中就可以很容易的读/写区段数据了）。每对十六进制数字表示一个字节。段数据字串的顺序与段表中的顺序相同，并且对于每一个“存在于当前文件”的段都必须有一个段数字字串。十六进制数字串的长度由定义的段长度决定；如果段长度为 100 字节，则对应的那行段数据字串应该有 200 个字符，不包括结尾的换行符。

项目 3-1：写一个 perl 程序，将这种格式的目标文件读入，并将内容保存在合适的 Perl 表或数组中，然后在将它们写回到一个目标文件中。该输出文件并不要求与输入文件完全一样，但它们在语义上应当时相同的。例如，符号并不需要按照它们被读入的顺序写到文件中，由于它们是可以重新排列的，因此必须要调整重定位项以适应符号表的新顺序。

## 练习

1. 一个如该项目格式的文本目标格式是否实用呢？（提示：见 Fraser 和 Hanson 的论文

“一个机器无关的链接器” 。 )

# 第 4 章 存储空间分配

\$Revision: 2.3 \$

\$Date: 1999/06/15 03:30:36 \$

链接器或加载器的首要任务是存储分配。一旦分配了存储空间后，链接器就可以继续进行符号绑定和代码调整。在一个可链接目标文件中定义的多数符号都是相对于文件内的存储区域定义的，所以只有存储区域确定了才能够进行符号解析。

与链接的其它方面情况相似，存储分配的基本问题是简单的，但处理计算机体系结构和编程语言语义特性的细节让问题复杂起来。存储分配的大多数工作都可以通过优雅和相对架构无关的方法来处理，但总有一些细节需要特定机器的专门技巧来解决。

## 段和地址

每个目标或可执行文件都会采用目标地址空间的某种模式。通常这里的目地是目标计算机的应用程序地址空间，但某些情况下（例如共享库）也会是其它东西。在一个重定位链接器或加载器中的基本问题是确保程序中的所有段都被定义并具有地址，并且这些地址不能发生重叠（除非有意这样）。

每一个链接器输入文件都包含一系列各种类型的段。不同类型的段以不同的方式来处理。通常，所有相同类型的段，诸如可执行代码段，会在输出文件中被合并为一个段。有时候段是在其它段的基础上合并得到的（如 Fortran 的公共块），以及在越来越多的情况下（如共享库和 C++ 专有特性），链接器本身会创建一些段并将其放置在输出中。

存储布局是一个“两遍”的过程，这是因为每个段的地址在所有其它段的大小未确定前是无法分配的。

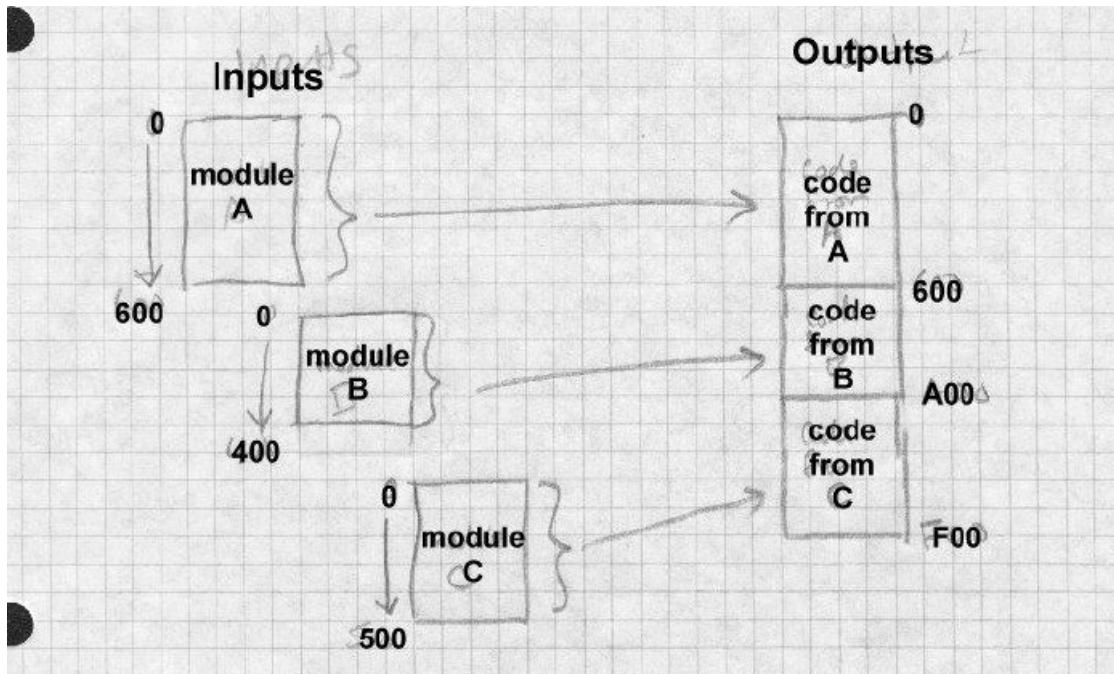
## 简单的存储布局

在一种简单而不现实的情形下，链接器的输入文件包含一系列的模块，将它们称为  $M_1, M_2, \dots, M_n$ ，每一个模块都包含一个单独的段，从位置 0 开始长度依次为  $L_1, L_2, \dots, L_n$ ，并且目标地址空间也是从 0 开始。如图 1 所示。

---

图 4-1：单独段的存储空间分配

从位置 0 开始的多个段按照一个跟着另一个的方式重定位



链接器或加载器依次检查各个模块，按顺序分配存储空间。模块  $M_i$  的起始地址为从  $L_1$  到  $L_{i-1}$  相加的总和，链接得到的程序长度为从  $L_1$  到  $L_n$  相加的总和。

多数体系结构要求数据必须对齐于字边界，或至少在对齐时运行速度会更快些。因此链接器通常会将  $L_i$  扩充到目标体系结构最严格的对齐边界（通常是 4 或 8 个字节）的倍数。

例 1：假定一个称为 main 的主程序要与三个分别称为 calif, mass 和 newyork 的子例程链接（按照地理位置划分风险投资）。每个例程的大小为（16 进制数字）：

名称	尺寸
<hr/>	
main	1017
calif	920
mass	615
newyork	1390

假定从 16 进制的地址 1000 处开始分配存储空间，并且要求 4 字节对齐，那么存储分配的结果可能是：

名称	位置
<hr/>	
main	1000 - 2016
calif	2018 - 2937
mass	2938 - 2f4c
newyork	2f50 - 42df

由于对齐的原因，2017 处的一个字节和 2f4d 处的三个字节被浪费了，但无须忧虑。

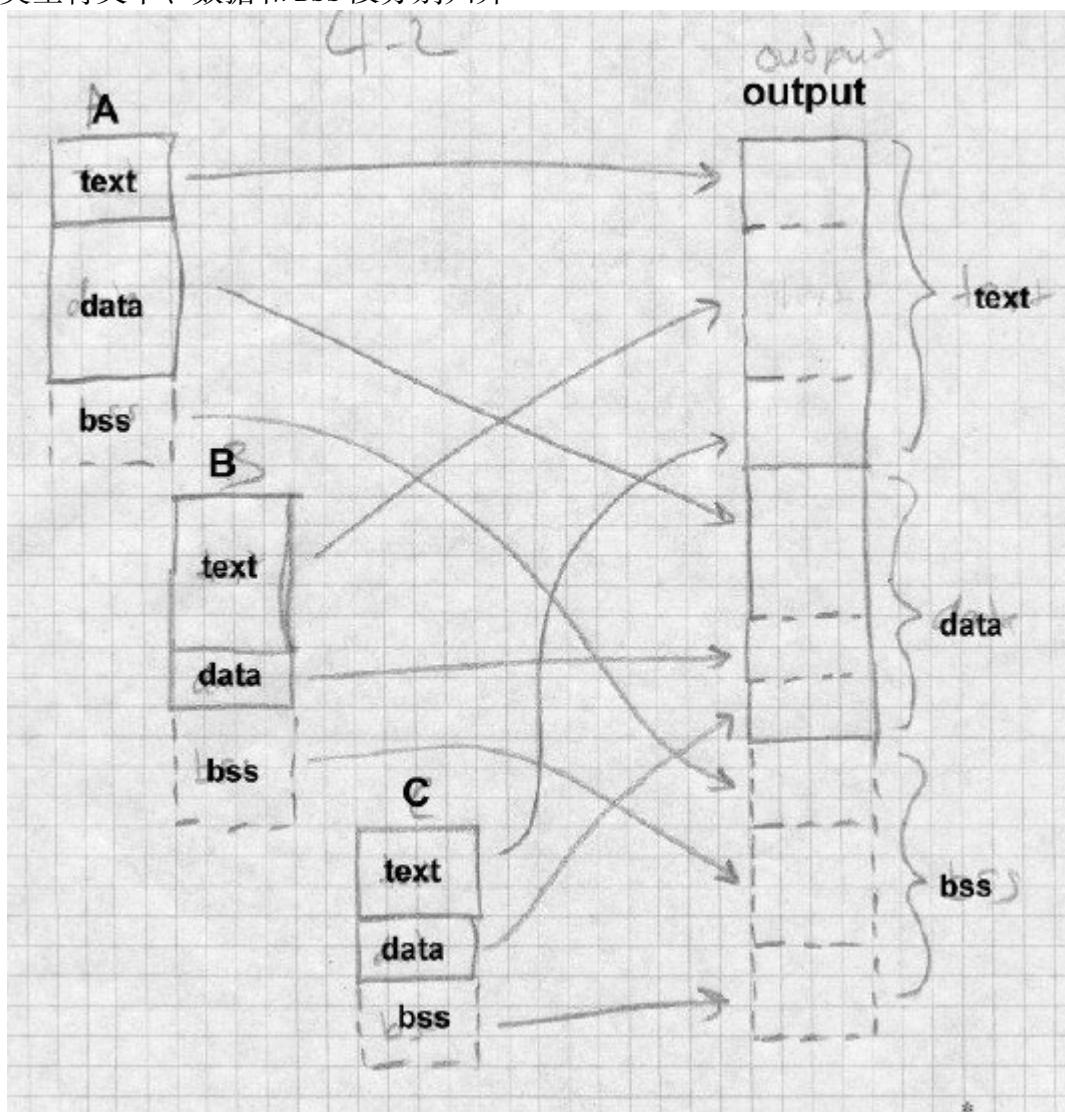
## 多种段类型

除最简单格式外所有的目标格式，都具有多种段的类型，链接器需要将所有输入模块中相应的段组合在一起。在具有文本和数据段的 UNIX 系统上，被链接的文件需要将所有的文本段都集中在一起，然后跟着的是所有的数据，在后面是逻辑上的 BSS（即使 BSS 在输出文件中不占空间，它仍然需要分配空间来解析 BSS 符号，并指明当输出文件被加载时要分配的 BSS 空间尺寸）。这就需要两级存储分配策略。

现在每一个模块  $M_i$  具有大小为  $T_i$  的文本段，大小为  $D_i$  的数据段，以及大小为  $B_i$  的 BSS 段，如图 2 所示。

图 4-2：多种段的存储分配

按类型将文本、数据和 BSS 段分别归并



在读入每个输入模块时，链接器为每个  $T_i$ ,  $D_i$ ,  $B_i$  按照（就像是）每个段都各自从位置 0 处开始的方式分配空间。在读入了所有的输入文件后，链接器就可以知道这三种段各自的大小  $T_{tot}$ ,  $D_{tot}$  和  $B_{tot}$ 。由于数据段跟在文本段之后，链接器将  $T_{tot}$  加到每一个数据段所分配的地址上，接着，由于 BSS 跟在文本和数据段之后，所以链接器会将  $T_{tot}$ 、 $D_{tot}$  的和加到每一个 BSS 段分配的地址上。

同样，链接器通常会将分配的大小按照对齐要求扩充补齐。

## 段与页面的对齐

如果文本和数据被加载到独立的内存页中，这也是通常的情况，文本段的大小必须扩充为一个整页，相应的数据和 BSS 段的位置也要进行调整。很多 UNIX 系统都使用一种技巧来节省文件空间，即在目标文件中数据紧跟在文本的后面，并将那个（文本和数据共存的）页在虚拟内存中映射两次，一次是只读的文本段，一次是写时复制（copy-on-write）的数据段。这种情况下，数据段在逻辑上起始于文本段末尾紧接着的下一页，这样就不需扩充文本段，数据段也可对齐于紧接着文本段后的 4K（或者其它的页尺寸）页边界。

例 2：我们将例 1 扩展，使得每个例程都有文本、数据和 BSS 段。字对齐要求还是 4 个字节，但页大小为 0x1000 字节。

名称	文本段	数据段	BSS 段
main	1017	320	50
calif	920	217	100
mass	615	300	840
newyork	1390	1213	1400
（均为 16 进制数字）			

链接器首先分配文本段，然后是数据段，接着是 BSS。注意这里数据段起始于页边界 0x5000，但 BSS 紧跟在数据的后面，这是因为在运行时数据和 BSS 在逻辑上是一个段。

名称	文本段	数据段	BSS 段
main	1000–2016	5000–531f	695c–69ab
calif	2018–2937	5320–5446	69ac–6aab
mass	2938–2f4c	5448–5747	6aac–72eb
newyork	2f50–42df	5748–695a	72ec–86eb

在 0x42e0 到 0x5000 之间的页结尾处浪费了一些空间。虽然 BSS 段的结束位置在页面中部的 0x86eb 处，但程序们普遍都会紧跟其后分配“堆”空间。

## 公共块和其它特殊段

上面这种简单的段分配策略在链接器处理的 80% 的存储分配中都工作的很好，但剩下的那些情况就需要用特殊的技巧来处理了。这里我们来看看比较常见的几个。

### 公共块

公共块存储是一个可以追溯到 50 年代 Fortran I 时的特性。在最初的 Fortran 系统中，

每一个子程序（主程序、函数或者子例程）都有各自局部声明和分配的标量和数组变量。同时还有一个各例程都可以使用的存储标量和数组的公共区域。公共块存储被证明是非常有用的，并且在后续 Fortran 中单一的公共块（就是我们现在知道的空白公共块，即它的名称是空白的）已经普及为多个可命名的公共块，每一个子程序都可以声明它们所用的公共块。

在最初的 40 年中，Fortran 不支持动态存储分配，公共块是 Fortran 程序用来绕开这个限制的首要工具。标准 Fortran 允许在不同例程中声明不同大小的空白公共块，其中最大的尺寸最终生效。Fortran 系统们无一例外的都将它扩展为允许以不同的大小来声明所有类型的公共块，同样还是最大的尺寸最终生效。

大型的 Fortran 系统经常会超过它们所运行系统的内存容量限制，在没有动态内存分配时，程序员不得不频繁的重新创建软件包，压缩尺寸来解决软件包遇到的此类问题。在一个软件包中除一个之外的其它子程序都将公共块声明为只有一个元素的数组。剩下的那个子程序声明所有公共块的实际大小，并在程序启动时将这些尺寸都保存在其余软件包可以使用的（在另一个公共块中的）变量中。这样就可以通过修改和重新编译定义这些公共块的一个例程，来调整公共块的尺寸，然后再重新链接。

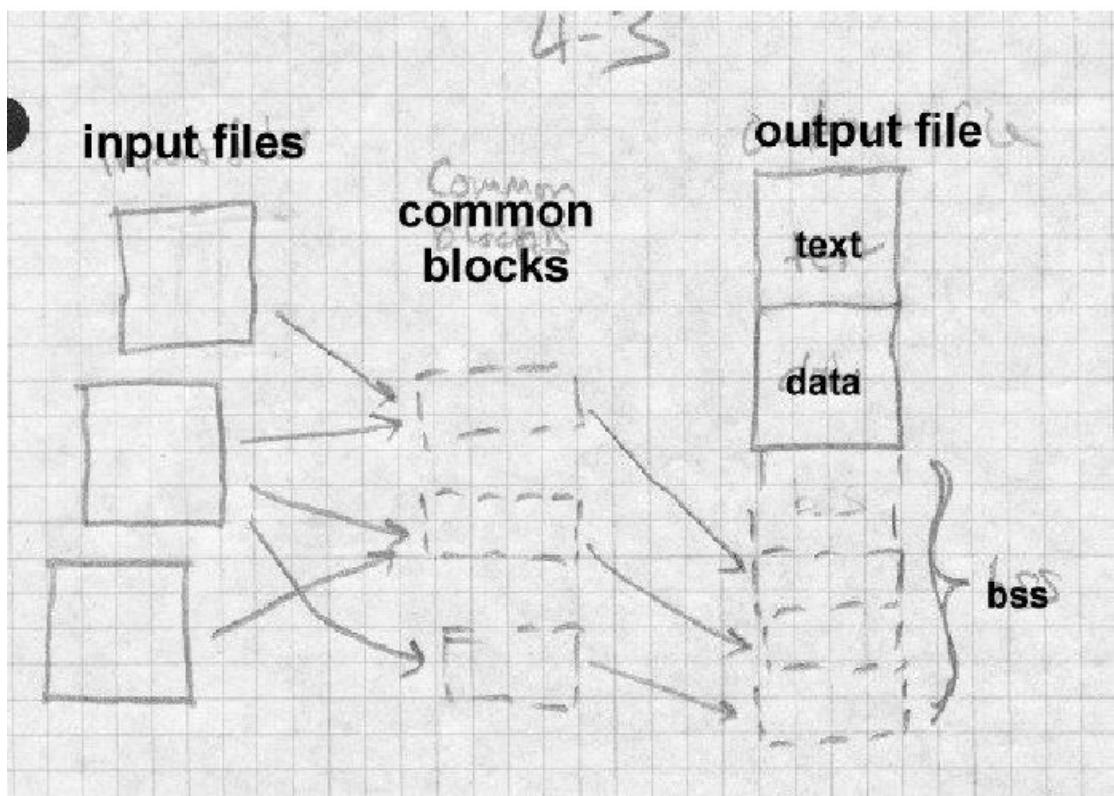
从 60 年代开始 Fortran 增加了 BLOCK DATA 数据类型来为任意公共块（空白公共块除外，这是为数不多的限制）的部分或全部来指明局部初始数据值，这在某种程度上更复杂了。通常用来初始化公共块的在 BLOCK DATA 中的公共块尺寸，也在链接时被用来当作该公共块的实际大小。

在处理公共块时，链接器会将输入文件中声明的每个公共块当作一个段来处理，但并不会将这些段串联起来，而是将相同名称的公共块重叠在一起。这里会将声明的最大的尺寸作为段的大小，除非在某一个输入文件中存在该段的已初始化的版本。在某些系统上，已初始化的公共块是一个单独的段类型，而在另一些系统上它可能只是数据段的一部分。

UNIX 链接器总是一贯支持公共块，甚至从最早版本的 UNIX 都具有一个 Fortran 子集的编译器，并且 UNIX 版本的 C 语言传统上会将未初始化的全局变量作为公共块对待。但在 ELF 之前的 UNIX 目标文件只有文本、数据和 BSS 段，没有办法直接声明一个公共块。作为一个特殊技巧，链接器将未定义但具有非零初值的符号当作是公共块，而该值就是公共块的尺寸。链接器将遇到的此类符号中最大的数值作为该公共块的尺寸。对于每一个公共块，它在输出文件的 BSS 段中定义了相应的符号，在每一个符号的后面分配所需要的空间。

---

图 4-3: Unix 公共块  
在 BSS 末尾的公共块



## C++重复代码消除

在某些编译系统中，C++编译器会由于虚函数表、模板和外部 inline 函数而产生大量的重复代码。这些特性的设计是隐含的期望那种程序所有部分都可以被运行的环境。一个虚函数表（通常简称为 vtbl）包含一个类的所有虚函数（可以被子类覆盖的例程）的地址。每个带有任何虚函数的类都需要一个 vtbl。模板本质上就是以数据类型为参数的宏，并能够根据特定的类型参数集可以扩展为特定的例程。确保是否存在一个对普通例程的引用可供调用是程序员的责任，就是说对如 hash(int) 和 hash(char \*) 每一类 hash 函数都有确定的定义，hash(T) 模板可以根据程序中使用 hash 函数时不同的参数数据类型创建对应的 hash 函数。

在每个源代码文件都被单独编译的环境中，最简单的方法就是将所有的 vtbl 都放入到每一个目标文件中，扩展所有该文件用到的模板例程和外部 inline 函数，这样做的结果就是产生大量的冗余代码。

最简单的方法就是在链接时仍然将那些重复代码保留着。那么得到的程序肯定可以正确的工作，但代码会膨胀的比理想尺寸大三倍或者更多。

在那些使用简单链接器的系统上，某些 C++ 系统使用了一种迭代链接的方法，并采用独立的数据库来管理将哪些函数扩展到哪些地方，或者添加 pragma（向编译器提供信息的程序源代码）向编译器反馈足够的信息以仅仅产生必须的代码。我们将在第 11 章涉及这些。

最近的很多 C++ 系统已经正面解决了这个问题，要么是让链接器更聪明一些，要么就是将链接器整合到程序开发环境的其它部分中（后一种方法我们在第 11 章还会涉及到）。链接器的方法是让编译器在每个目标文件中生成所有可能的重复代码，然后让链接器来识别和

消除重复的代码。

MS Windows 链接器为代码区段定义了 COMDAT 标志来告诉链接器忽略除明确命名区段外的所有重复区段。编译器会根据模板给每个区段命名，名字中包含了参数类型，如图 4 所示。

---

图 4-4: Windows

IMAGE_COMDAT_SELECT_NODUPPLICATES	1 Warn if multiple identically named sections occur.
IMAGE_COMDAT_SELECT_ANY	2 Link one identically named section, discard the rest.
IMAGE_COMDAT_SELECT_SAME_SIZE	3 Link one identically named section, discard the rest. Warn if a discarded section isn't the same size.
IMAGE_COMDAT_SELECT_EXACT_MATCH	4 Link one identically named section, discard the rest. Warn if a discarded section isn't identical in size and contents. (Not implemented.)
IMAGE_COMDAT_SELECT_ASSOCIATIVE	5 Link this section if another specified section is also linked.

---

GNU 链接器是通过定义一个“link once”类型的区段（与公共块很相似）来解决这个模板的问题的。如果链接器看到诸如.gnu.linkonce.name 之类的区段名称，它会将第一个明确命名的此类区段保留下来并忽略其它冗余区段。同样编译器会将模板扩展到一个采用简化模板名称的.gnu.linkonce 区段中。

这种策略工作的相当不错，但它并不是万能的。例如，它不能保护功能上并不完全相同的 vtbl 和扩展模板。一些链接器尝试去检查被忽略的和保留的区段是否是每个字节都相同。这种方法是很保守的，但是如果两个文件采用了不同的优化选项，或编译器的版本不同，就会产生报错信息。另外，它也不能尽可能多的忽略冗余代码。在多数 C++ 系统中，所有的指针都具有相同的内部表示，这意味着一个模板的具有指向 int 类型指针参数的实例和指向 float 类型指针参数的实例会产生相同的代码（即使它们的 C++ 数据类型不同）。某些链接器也尝试忽略那些和其它区段每个字节都相同的 link-once 区段，哪怕它们的名字并不是完全的相同，但这个问题仍然没有得到满意的解决。

虽然我们在这里只是讨论了模板的问题，但相同的问题也会发生在外部 inline 函数，缺省构造、复制和赋值例程中，也可以采用相同的方法处理。

## 初始化和终结

另一个问题并不仅限于 C++，但在 C++ 上尤为严重，就是初始化和终结代码（initializers and finalizers）。一般来说，如果它们可以在程序启动的时候可以运行一个初始化例

程，并在程序结束的时候运行一个终结例程，那把它们写成库会更容易些。C++允许静态变量。如果一个变量的类具有构造函数，那这个构造函数在程序启动时会被调用来对初始化变量，同样如果一个变量的类具有析构函数，那析构函数也会在程序退出时被调用。有很多办法可以在不需要链接器支持的情况下做到这一点，我们将会在第 11 章讨论到，但现代链接器通常都会直接支持该特性。

通常的方法是将每个目标文件中的初始化代码都放入一个匿名的例程中，然后将指向该例程的指针放置在名为. init（或其它相近名字）的段中。链接器将所有的. init 段串联在一起，因此就创建了一个指向所有这些初始化例程的指针列表。程序的初始化部分只需要遍历该列表依次调用所有例程即可。退出时的代码可以采用相同方法，只是段的名字改为了. fini。

实践证明这种方法也不是完全令人满意的，因为有一些初始化代码要求比另外一些更早的运行。C++定义指出应用程序级的构造函数运行顺序是不确定的，但 I/O 和其它系统库的构造函数需要在应用程序自己的构造函数之前执行。完美的方法应当是让每一个初始化例程都精确的列出它们的依赖关系，并在此基础上进行拓扑排序。BeOS 操作系统的动态链接器就是这么做的，使用到了库的引用依赖关系（如果库 A 依赖于库 B，那么库 B 的初始化代码就可能需要先运行）。

一个更简单的近似方法是设置多个用于初始化的段，如. init 和. ctor，这样启动程序首先为所有库级初始化调用. init 中的例程，然后为 C++ 的构造函数调用. ctor 中的例程。同样的问题出现在程序结束时，对应的段为. dtor 和. fini。有一个系统甚至还允许程序员设置优先级编号，0 至 127 为用户代码，128 至 255 是系统库，链接器在合并代码之前会先将初始化和终结代码按优先级编号排序，最高优先级的初始化代码最先运行。但这仍不能令人完全满意，因为构造函数之间会存在顺序依赖关系，从而产生非常难以调试的错误，但在这里 C++ 将避免这些错误的责任交给了程序员。

该策略的一个变种是将实际的初始化代码放在. init 段中，当链接器合并它们的时候该段会成为完成所有初始化工作的 inline 代码。只有少量系统进行了这种尝试，但在不支持直接寻址的计算机上是很难让它工作的，因为从每个目标文件中提取出来的代码块还要能够对它们原本文件中的数据进行寻址，通常这都需要寄存器来指向可以指向寻址数据的表。匿名例程采用和其它例程相同的方式来初始化它们的寻址过程，借助已有的方案来减少寻址的问题。

## IBM 伪寄存器

IBM 主机系统的链接器提供了一种称为“外部模拟（external dummy）”区段或“伪寄存器（pseudo-registers）”的有趣特性。360 是较早的无直接寻址的主机架构之一，这就意味着实现小数据区域共享要付出昂贵的开销。每一个引用全局对象的例程都需要一个 4 字节的指针指向该对象，如果这个对象只有开头 4 个字节那么大的话，这将是相当大的开销。例如 PL/1 程序对每一个打开的文件和其它全局对象都需要一个指针（虽然 PL/1 应用程序的程序员无法访问伪寄存器，但它是唯一使用伪寄存器的高级语言。它使用伪寄存器指向打开

文件的控制块这样应用程序就可以包括进那些对 I/O 系统的 inline 调用）。

一个相关的问题是 OS/360 不支持我们现在所说的那种称为进程/任务级本地存储的东西，并且对共享库只提供非常有限的支持。如果两个作业运行同样的程序，或者这个程序被标注为可重入（这时它们共享整个程序、代码和数据），或者标注为不可重入（这时不共享任何东西）。所有的程序都被加载到相同的地址空间，因此相同程序的多个实例必须标注出实例本身数据的范围（360 系统不具备硬件内存重定位功能，尽管 370 支持了，但也知道 OS/VS 操作系统的若干个版本之后系统才提供进程独立的地址空间）。

伪寄存器可以帮助解决这些问题，如图 5 所示。每一个输入文件都可以声明（多个）伪寄存器，也称为外部模拟区段（360 系统的汇编语言中，它与结构体的声明很相似）。每个伪寄存器都有名字、长度和对齐要求。在链接时，链接器将所有的伪寄存器都收集到一个逻辑段中，将最大的尺寸和最严格的对齐要求施加于每个伪寄存器，并为它们分配在该逻辑段中不会相互重叠的偏移量。

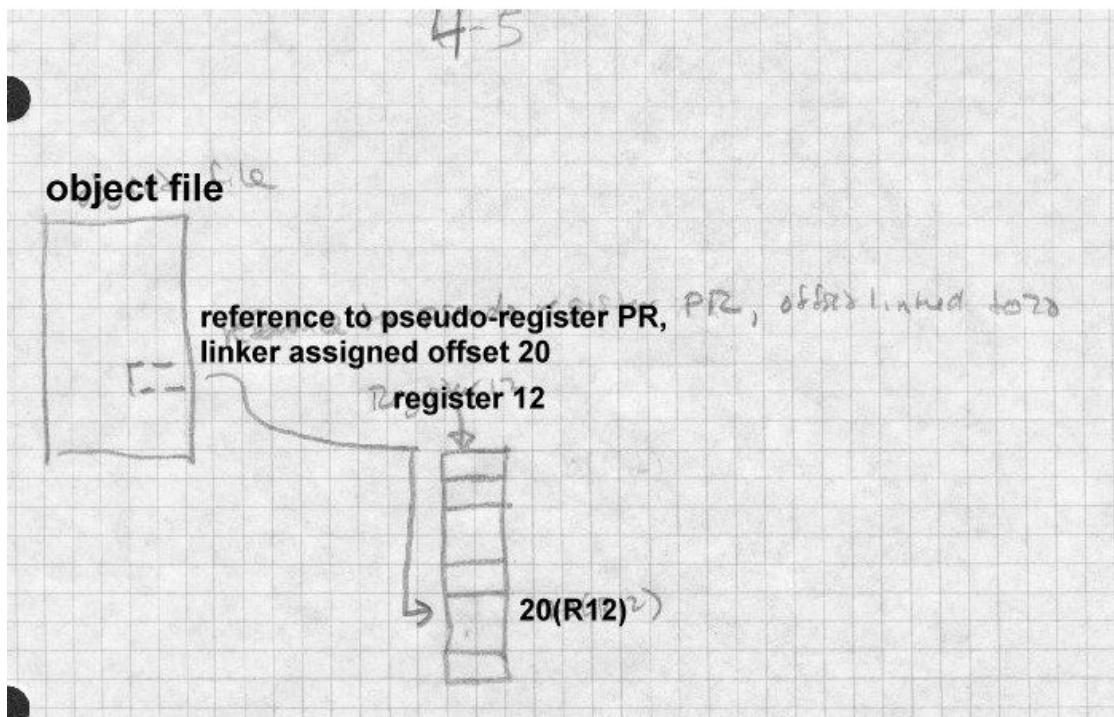
但链接器不会为伪寄存器段分配空间。它只是计算该段的大小，并将其存储在程序的数据段中以特殊的 CXD ( cumulative external dummy，即重定位项) 标识的位置。当引用一个伪寄存器时，程序代码还需要另一个特殊的 XD ( external dummy )，它是用来指示将偏移量放置在哪一个该伪寄存器所属逻辑段内的重定位类型。

程序的初始化代码为伪寄存器动态的分配空间，使用 CXD 可以知道需要多大的空间，并按惯例将这个空间的地址存放在寄存器 12 中，在整个程序运行期间都不会改变。程序中的任何一部分都可以通过将寄存器 12 的值与某个伪寄存器对应的 XD 的值相加得到该伪寄存器的地址。一般都是通过 load 和 store 指令来完成的，将 R12 ( 寄存器 12 ) 作为索引寄存器与嵌入到指令的地址替换域中的 XD 项相加（地址替换域只有 12 位，但由于 XD 将 16 位半字的高 4 位保持为 0，即基址寄存器为 0，所以仍然可以产生正确的结果）。

---

#### 图 4-5：精灵寄存器

通过 R12 指向一串地址块。各种例程通过偏移量引用它们。



这样的结果就是程序的所有部分都可以 load、store 和其它 RX 格式指令来直接访问所有的伪寄存器。如果一个程序存在多个活动的实例，每个实例就可以通过采用不同的 R12 值来分配独立的空间。

尽管最初引用伪寄存器的原因现在大多数都被废弃了，但为链接器提供可以高效访问线程本地地址的方法确实一个非常好的思想，并且仍然出现在很多现代操作系统中，其中最著名的就是 Windows。同样，现代的 RISC 机器也分享了 369 系统有限的寻址范围，因此需要使用内存指针表来寻址任意的内存地址。在很多 RISC UNIX 系统上，编译器为每个模块创建两个数据段，一个是通常的数据段，另一个是“小 (small)”数据段，即大小低于某一个尺寸阀值的静态对象。链接器将所有的小数据段收集在一起，然后让程序的启动代码将合并的小数据段的地址放入一个保留的寄存器中。这样就可以通过和这个寄存器相关的基址寻址来直接引用这些小数据。要注意，与伪寄存器不同，小数据的存储空间既会被链接器分配，也会被链接器放置到输出中，在每个程序中只有一份小数据。某些 UNIX 系统支持线程，但线程级的存储是特定的程序代码完成的，不需要链接器的特殊帮助。

## 特殊的表

链接器分配存储的最后一个资源是链接器本身。尤其是当应用程序使用共享库或者重叠技术时，链接器会创建由指针、符号或其它别的数据构成的多个段来在运行时支持库或者重叠。一旦这些库被建立了，链接器会按照对待任何其它段的方式来为它们分配存储空间。

## X86 分段的存储分配

8086 和 80286 的分段内存寻址的怪癖要求导致了少量特殊的东西。x86 OMF 目标文件给每个段都有一个名字和可选的类别。所有具有相同名字的段，会根据由编译器或者汇编器设置的一些标志位来合并到一个大的段中，并且所有类别相同的段都会被连续的分配在一个块中。编译器或汇编器使用类别名来标注段的类型（诸如代码或静态数据），因此链接器可以将给定类别的所有的段分配在一起。当某个类别的所有段总长小于 64K 时，它们可以被当作使用一个段寄存器的单独寻址“组”来对待，这样可以节省客观的时间和空间。

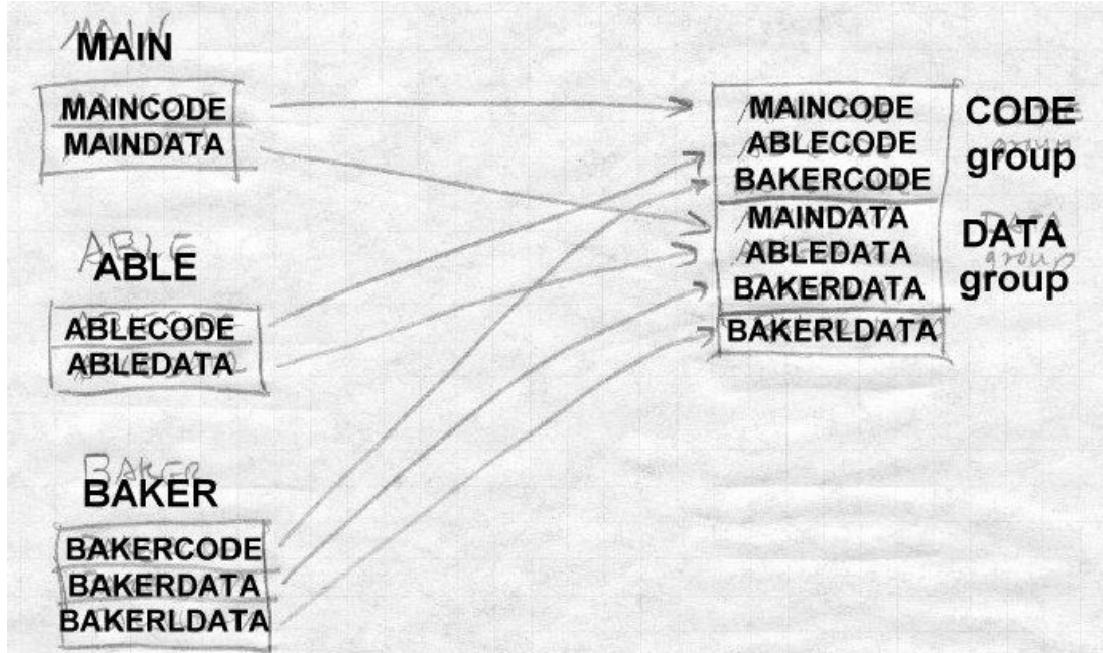
图 6 所示为一个由三个输入文件连接而成的程序，三个输入文件依次为 main、able 和 baker。文件 main 中包含段 MAINCODE 和 MAINDATA，able 中包含段 ABLECODE 和 ABLEDATA，baker 中包含段 BAKERCODE、BAKERDATA 和 BAKERLDATA。每一个代码段都是 CODE 类别，数据段都是 DATA 类别，但“大数据”BAKERLDATA 不赋予类别。在链接好的程序中，假定 CODE 段最大 64K，它们在运行时可以当作单独的段来对待，可以使用 short（而不是 far）调用和跳转指令，以及一个不变的 CS 段代码寄存器。同样如果所有的 DATA 段可以装在 64K 中，则它们也可以当作单一的段来对待，使用 short 的内存引用指令和一个不变的 DS 数据段寄存器。BAKERLDATA 段在运行时作为一个独立的段处理，程序代码会加载一个段寄存器（通常是 ES）来指向它。

图 4-6: X86

CODE 类别的 MAINCODE、ABLECODE 和 BAKERCODE 段

DATA 类别的 MAINDATA、ABLEDATA 和 BAKERDATA 段

单独的 BAKERLDATA 段



实模式和 286 保护模式的程序几乎是以相同的方式来链接的。主要的不同在于链接器一

且在保护模式程序中生成链接好的段，链接器就完成工作了，只有在程序加载时才会赋予实际的内存地址和段号。在实模式中，链接器还有额外的一步就是为段分配线性地址，并相对于程序起始位置为这些段分配段落（paragraph）号。然后在加载的时候，程序加载器必须调整实模式程序中所有的段落号（paragraph number）或者保护模式程序中所有的段号（segment number）以反映程序被加载的实际位置。

## 链接器控制脚本

传统上链接器可以允许用户对输出数据进行有限的控制。由于链接器已经开始要面对内存组织非常复杂的目标环境，诸如众多的嵌入式处理器和目标环境，因此就非常必要对目标地址空间和输出文件中的数据提供更加精确的控制。具有一系列固定段的简单链接器通常具有可以指定各个段基地址的开关参数，这样程序就可以被加载到非标准的应用环境中（操作系统内核通常会用到这些开关参数）。有一些链接器具有数量庞大的命令行开关参数，由于系统经常会限制命令行的长度，因此经常将这些命令行逻辑上连续的放置在一个文件中。例如，微软的链接器在文件中为每个区段设置特性时最多可以采用大约 50 个命令行开关选项，包括输出的基地址和一系列其它输出相关的细节。

其它的链接器定义了可以控制链接器输出的脚本语言。GNU 链接器，也定义了这么一种具有一长串命令行参数的语言。图 7 所示为可以在系统 5 版本 3.2（System V Release 3.2）的系统上（如 SCO UNIX）产生 COFF 可执行程序的一个简单链接脚本示例。

---

图 4-7：生成 COFF 可执行程序的 GNU 链接器控制脚本

```
OUTPUT_FORMAT("coff-i386")
SEARCH_DIR(/usr/local/lib);
ENTRY(_start)
SECTIONS
{
    .text SIZEOF_HEADERS : {
        *(.init)
        *(.text)
        *(.fini)
        etext = .;
    }
    .data 0x400000 + (. & 0xffc00fff) : {
        *(.data)
        edata = .;
    }
    .bss SIZEOF(.data) + ADDR(.data) :
    {
        *(.bss)
```

```

*(COMMON)
end = . ;
}
.stab 0 (NOLOAD) :
{
[ .stab ]
}
.stabstr 0 (NOLOAD) :
{
[ .stabstr ]
}
}

```

---

开始的几行描述了输出的格式（必须是编译进链接器的格式表中存在的），查找目标代码库的位置，和缺省入口点的名称（本示例中为\_start）。然后它列出了输出文件中的区段。在区段名后面是一个指明区段开始地址的可选数值。因此可以看出，.text 区段紧跟在文件头部后面，输出文件中的.text 区段包含了所有输入文件中的.init 区段、所有的.text 区段和所有的.fini 区段。链接器定义了符号 etext 作为.fini 区段后面的地址。然后脚本设置了.data 区段，强制将其起始地址设置为文本区段后面的 4KB 对齐的地址 0x400000，该区段中包含了所有输入文件中的.data 区段，并紧跟其后定义了edata 符号。然后是紧跟在数据段后面的.bss 区段，它包括了所有输入文件中的.bss 区段和公共块，并将 bss 段的末尾用符号 end 标识（COMMON 是该连接语言的一个关键字）。在那之后的两个区段是从输入文件相应位置收集的众多符号表项，但只有调试器会查看这些符号，因此在运行时不会被加载。链接器脚本语言比这个简单的例子要复杂得多，足以描述从简单的 DOS 可执行程序到 Windows PE 可执行程序以至到复杂的重叠管理的各种类型。

## 嵌入式系统的存储分配

嵌入式系统的存储分配与我们到现在为止已看到的策略相近，只是由于程序所运行的复杂的地址空间而复杂了一些。嵌入式系统链接器提供的脚本语言可以让程序员地址空间的区域，并将特定的段或目标文件分配到这些区域中，并可以指明各区域中每个段的地址对齐要求。

诸如 DSP 这样的专用处理器的链接器还要支持各处理器的特殊特性。例如，Motorola 5600X DSP 系列支持循环缓冲区必须对齐在不小于缓冲区大小的 2 的幂次的地址上。56K 目标格式为这些缓冲区有一个特殊的段类型，链接器会自动的将它们分配到正确的边界上，并尽量减小（**作者笔误？**）未使用的空间。

## 实际中的存储分配

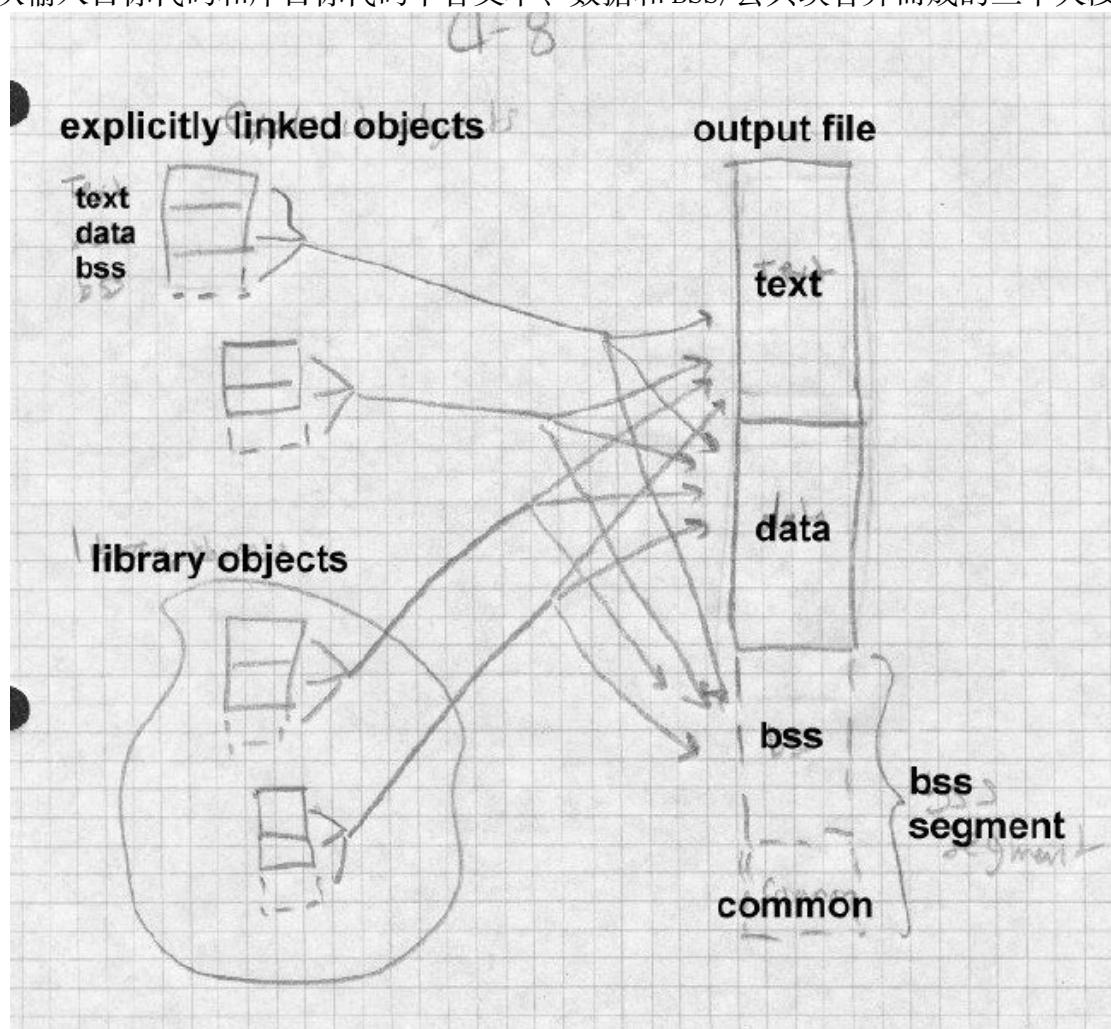
现在我们看看几种流行链接器的存储分配策略，作为本章的结束。

## Unix a.out 链接器的存储分配策略

ELF 之前的 UNIX 链接器的存储分配策略只比本章开头的理想实例稍微复杂一点，这是因为各个段在链接之前已经知道了，如图 8 所示。每个输入文件具有文本、数据和 BSS 段，也可能有伪装为外部符号的公共块。链接器从每个输入文件和库目标文件中收集文本、数据和 BSS 的大小。在读取了所有的目标文件之后，任何未解析的具有非零值外部符号都被放入公共块中，并在 BSS 尾部分配空间。

图 4-8: a.out 链接

从输入目标代码和库目标代码中各文本、数据和 BSS/公共块合并而成的三个大段



这里，链接器可以为各个段直接赋予地址。文本段根据所创建的不同 a.out 格式起始于一个固定的位置，或者是 0 位置（最老的格式），或者是 0 位置的下一页（NMAGIC 格式），或者是一页再加上 a.out 头部（QMAGIC）。数据段可以直接跟在文本段后面（旧的非共享 a.out 格式），或起始于文本段后下一页的边界处（NMAGIC 格式）。在每种格式中，BSS 都紧跟在数据段后面。在每一个段内部，将各输入文件中的段排列在前一个段后面字对齐的边界

处。

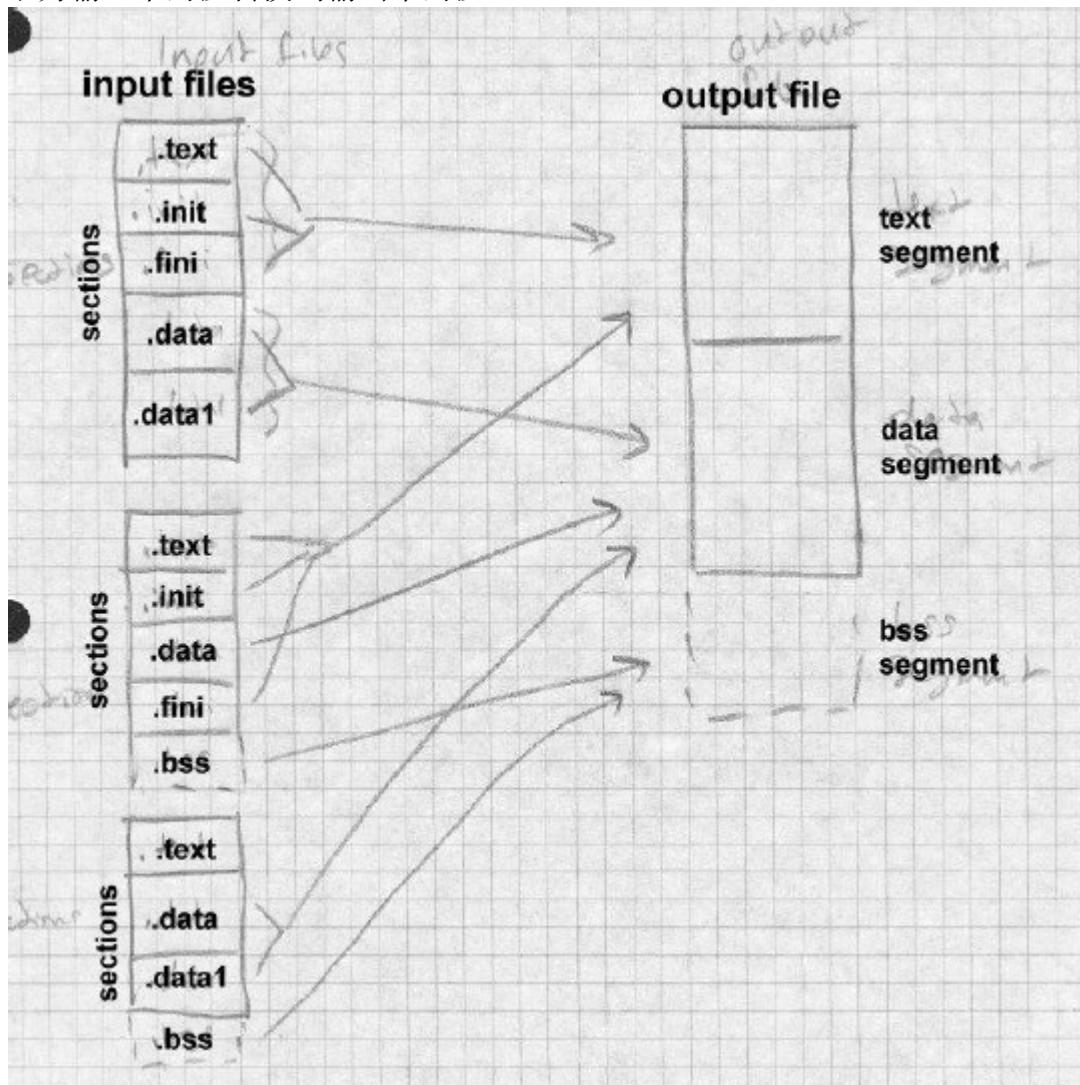
## ELF 中的存储分配策略

ELF 链接要比 a.out 复杂一些，因为输入文件中的各个段可以是任意大小的，链接器必须将输入段（ELF 术语中的段）转换为可加载的段（ELF 术语中的段）。链接器还要创建程序加载器需要的程序头部，和动态链接所需的一些特殊区段，如图 9 所示。

图 4-9：ELF 链接

摘自 TIS ELF 文档页 2-7 和 2-8

所示为输入中的段转换到输出中的段



ELF 目标文件具有传统的文本、数据和 BSS 区段，现在拼写为 .text、.data 和 .bss。经常还会包含 .init 和 .fini（启动和退出时的代码），和其它一些琐碎的东西。.rodata 和 .data1 在某些编译器中被用来表示只读数据和 out-of-line 数据（有些编译器也有对应只读 out-of-line 数据的 .rodata1 区段）。在诸如 MIPS 这样地址偏移量受限的 RISC 系统中，还有 .

`sbss` 和 `scommon` 区段，即小的 BSS 和公共块，有利于小的对象组合到单个可以直接寻址的区域，就像我们在上面讨论伪寄存器时说到的那样。在 GNU C++ 系统中，还可以会有可以被插入文本、只读数据和数据段中的 `linkonce` 区段。

如果不考虑众多的区段类型，那么链接过程都是一样的。链接器将各个输入文件和库目标文件中的同类型区段收集在一起。链接器还会标注出哪些符号会在运行时从共享库中解析，并创建 `.interp`、`.got`、`.plt` 和符号表区段来支持运行时链接（我们将细节的讨论推迟到第 9 章）。一旦这些都完成了，链接器会按照传统的顺序来分配空间。与 `a.out` 不同，ELF 格式不会从 0 位置加载任何东西，而是从地址空间的中间部位来加载，这样栈可以在文本段以下向下增长，堆可以在数据段末尾以上向上增长，以更加紧凑的利用地址空间。在 386 系统上，文本的基地址是 0x08048000，这样既可以允许位于文本以下的合理的栈空间，同时将 0x08000000 以上的空间留出来，允许多数程序将它用来创建单一的二级页表（回想一下在 386 上，每一个二级页表可以映射大小为 0x00400000 的地址空间）。ELF 使用 QMAGIC 的技巧将头部包括到文本段内，所以实际的文本段起始于 ELF 头部和程序头部表之后，典型的位于文件偏移量 0x100 处。然后再将 `.interp`（动态链接器的逻辑链接，需要首先被运行）、动态链接器符号表区段、`.init`、`.text`，以及 `link-once` 文本和只读数据分配到文本段中。

接下来是数据段，逻辑上起始于文本段末尾的下一个页（因为在运行时该页会同时被映射为文本段的最后一页和数据段的第一页）。链接器分配各种 `.data` 区段和 `link-once` 数据和 `.got` 区段，以及一些平台上会用到的 `.sdata` 小数据和 `.got` 全局偏移量表。

最后是 BSS 区段，逻辑上紧跟在数据的后面，由 `.sbss` 开始（如果有的话，将它放在 `.sdata` 和 `.got` 的后面），然后是 BSS 段和公共块。

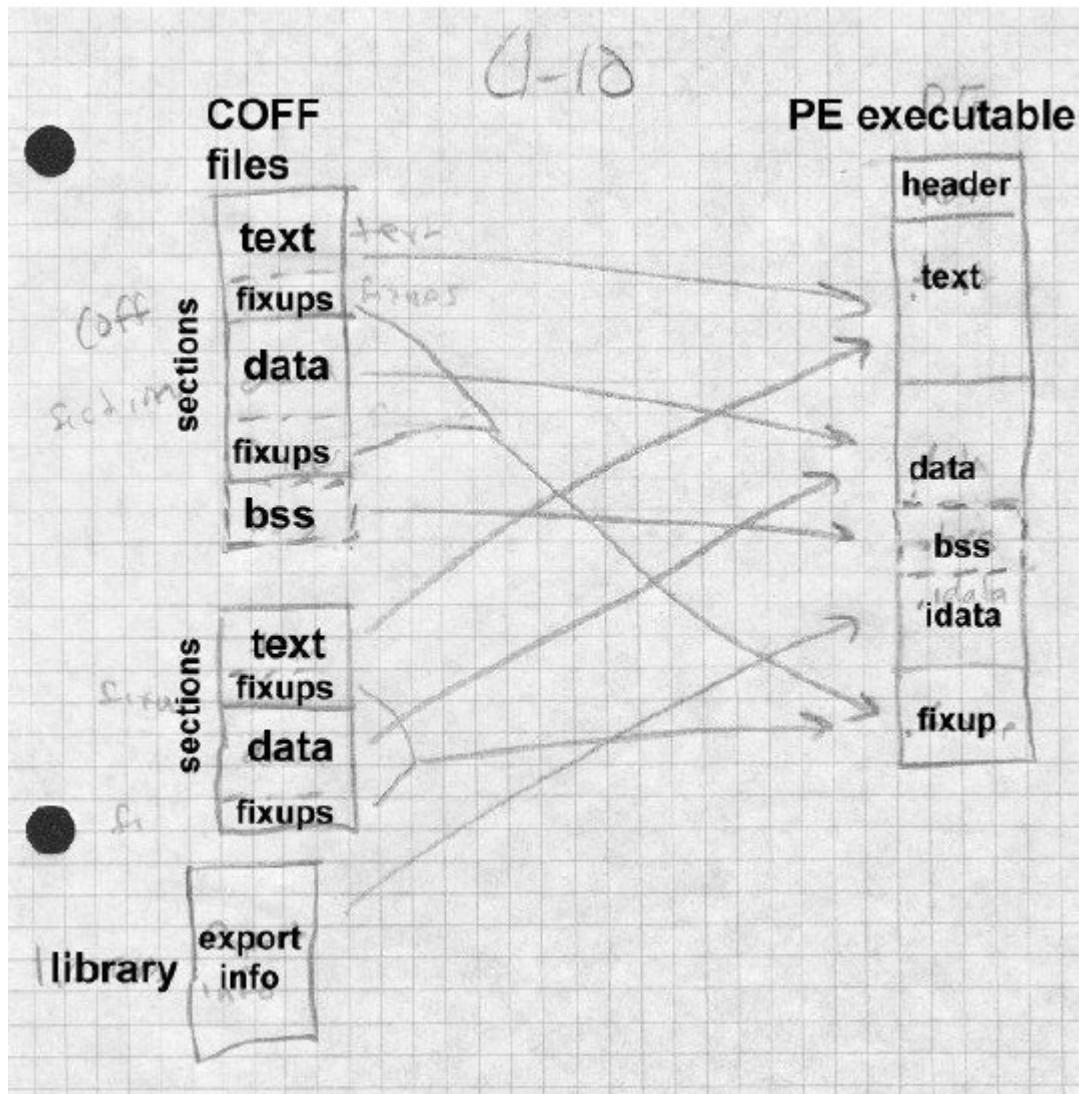
## Windows 链接器的存储分配策略

Windows Pe 文件的存储分配策略比 ELF 文件还要简单一点，这是因为 PE 的动态链接模式需要链接器的支持较少，作为代价编译器承担了更多的工作，如图 10 所示。

---

图 4-10：PE 文件的存储分配

摘自微软网站



Pe 可执行文件传统上从地址 0x400000 处加载，即文本段开始的位置。文本段包括输入文件中的文本段，还有初始化和终结代码段。接下来是数据段，对齐于逻辑磁盘块的边界（磁盘块通常比内存页要小，在 Windows 下通常是 512 字节或 1KB，而非 4KB）。接下来是 BSS 和公共块，.rdata 重定位调整信息（为那些通常不能直接加载到预期目标地址的 DLL 库），动态链接用的导入和导出表，以及其它段（诸如 Windows 资源）。

一个特殊的段类型是. tls，线程本地存储（thread local storage）。Windows 进程可以并且经常在并发活动控制中采用多线程。PE 文件中的. tls 数据会分配给每一个线程。它包括要初始化的数据块和线程启动/结束时要调用的函数表。

## 练习

1. 为什么链接器要将各个段都混在一起，让同类型的段一个跟一个的排列？保持它们原先的顺序不是更简单些吗？
2. 在什么时候（如果有的话），链接器为例程分配存储空间的顺序会对结果产生影响？在我们的例子中，如果链接器按照 newyork、mass、calif、main 的顺序分配空间而不是 mai

n、calif、mass、newyork 的顺序，这会有什么不同？（后面当我们讨论到覆盖和动态链接时会再次问到这个问题，所以你可以先忽略那些）

3. 多数情况下链接器将相同类型的段顺序分配，例如，calif、mass 和 newyork 的文本段会一个跟着一个。但对公共块段时链接器会将同名公共块重叠摞在一起分配。为什么？

4. 允许在不同的输入文件中声明同名而不同大小的公共块是否是一个好主意？为什么是，或不是？

5. 在例子 1 中，假定程序员重写了 calif 例程而使得目标代码现在的长度为 0x1333。重新计算要分配的段地址。在例子 2 中，进一步假定由于重写了 calif 后数据和 BSS 段的大小变成了 0x975 和 0x120，重新计算要分配的段地址。

## 项目

项目 4-1：扩展项目 3-1 中的链接器框架来进行简单的 UNIX 风格存储分配。假定在输出文件中只有 .text、.data 和 .bss 段，文本起始于 0x1000，数据起始于文本后下一个 0x1000 的倍数，BSS 起始于数据后面 4 字节对齐处，你的链接器需要写出一个不完全的目标文件，其中要有输出文件的段定义（现在你还不需要考虑符号、重定位和数据）。在你的链接器中，要确保你有数据结构可以让你确定各输入文件中的各段都被赋予了什么地址，因为你在后续章节的项目中会用到它。使用例子 2 中的简单例程来测试你的分配器。

项目 4-2：实现 UNIX 风格的公共块。即扫描符号表中具有非零值的未定义符号，并将适当的尺寸增加的 .bss 段中。不要考虑调整符号表项的事情，那是下一章要做的。

项目 4-3：扩展 4-1 中的分配器以处理输入文件中的任意段，合并所有名字相同的段。一个可行的分配策略是将具有 RP 属性的段放在 0x1000 处，将具有 RWP 属性的段放在下一个 0x1000 对齐的起始处，然后是 4 字节边界的 RW 属性段。将 .bss 中的公共块紧跟在 RW 属性段后面分配。

扩展 4-3 中的分配器来处理输入文件中的任意段，将所有有相同名字的段合并。一个合理的分配策略是在多个 1000 字节前分配带有 RP 属性的段，在下一个 1000 字节的边界开始分配带有 RWP 属性的段，然后在 4 字节边界放 RW 属性的段。在 .bss 段中分配带有 RW 属性的公共块。

# 第 5 章 符号管理

\$Revision: 2.2 \$

\$Date: 1999/06/30 01:02:35 \$

符号管理是链接器的关键功能。如果没有某种方法来进行模块之间的引用，那么链接器的其它功能也就没有什么太大的用处了。

## 绑定和名字解析

链接器要处理各种类型的符号。所有的链接器都要处理各模块之间符号化的引用。每个输入模块都有一个符号表。其中的符号包括：

当前模块中被定义（和可能被引用）全局符号。

在被模块中被引用但未被定义的全局符号（通常成为外部符号）。

段名称，通常被当作定义在段起始位置的全局符号。

非全局符号，调试器或崩溃转储（crash dump）分析通常会用到它们。这些符号几乎不会被链接过程用到，但有时候它们经常会和全局符号混在一起，所以链接器至少要能够跳过它们。在另一些情况中它们会在文件中一个单独的表中，或在一个单独的调试信息文件中（可选的）。

链接器读入输入文件中所有的符号表，并提取出有用的信息，有时就是输入的信息，通常都是关于需要链接哪些东西的。然后它会建立链接时符号表并使用该表来指导链接过程。根据输出文件格式的不同，链接器会将部分或全部的符号信息放置在输出文件中。

某些格式会在一个文件中存在多个符号表。例如 ELF 共享库会有一个动态链接所需信息的符号表，和一个单独的更大的用来调试和重链接的符号表。这个设计不见得糟糕。动态链接器所需的表比全部的表通常要小得多，将它独立出来可以加快动态链接的速度，毕竟调试或重链接一个库的机会（相比运行这个库）还是很少的。

## 符号表格式

链接器中的符号表与编译器中的相近，由于链接器中用到的符号一般没有编译器中的那么复杂，所以符号表通常也更简单一些。在链接器内，有一个列出输入文件和库模块的符号表，保留了每一个文件的信息。第二个符号表处理全局符号，即链接器需要在输入文件中进行解析的符号。第三个表可以处理模块内调试符号，尽管少数情况下链接器也会为调试符号建立完整的符号表，但通常都只需将输入的调试符号传递到输出文件。

在链接器本身内部，符号表通常以表项组成的数组形式来保存，并通过一个 hash 函数来定位表项，或者是由指针组成的数组，并通过 hash 函数来索引，相同 hash 的表项以链表的形式来组织。当需要在表中定位一个符号时，链接器根据符号名计算 hash 值，将该值用桶的个数来取模，以定位某一个 hash 桶（图中的 symhash[h%NBUCKET]，h 为 hash 值），然

后遍历其中的符号链表来查找符号。

传统上，链接器仅支持短名称，从 IBM 主机系统的 8 个字符和多数 DEC 系统上早期 UNIX 系统的 6 个字符到一些消亡中的小型计算机的 2 个字符。现代链接器支持的名称要长得多，这既是由于程序员使用了更长的名称（或诸如在 Cobol 中，不再通过修改名称使它们的头八个字符唯一），也是因为编译器将名称进行了修改添加了额外的类型信息编码字符。

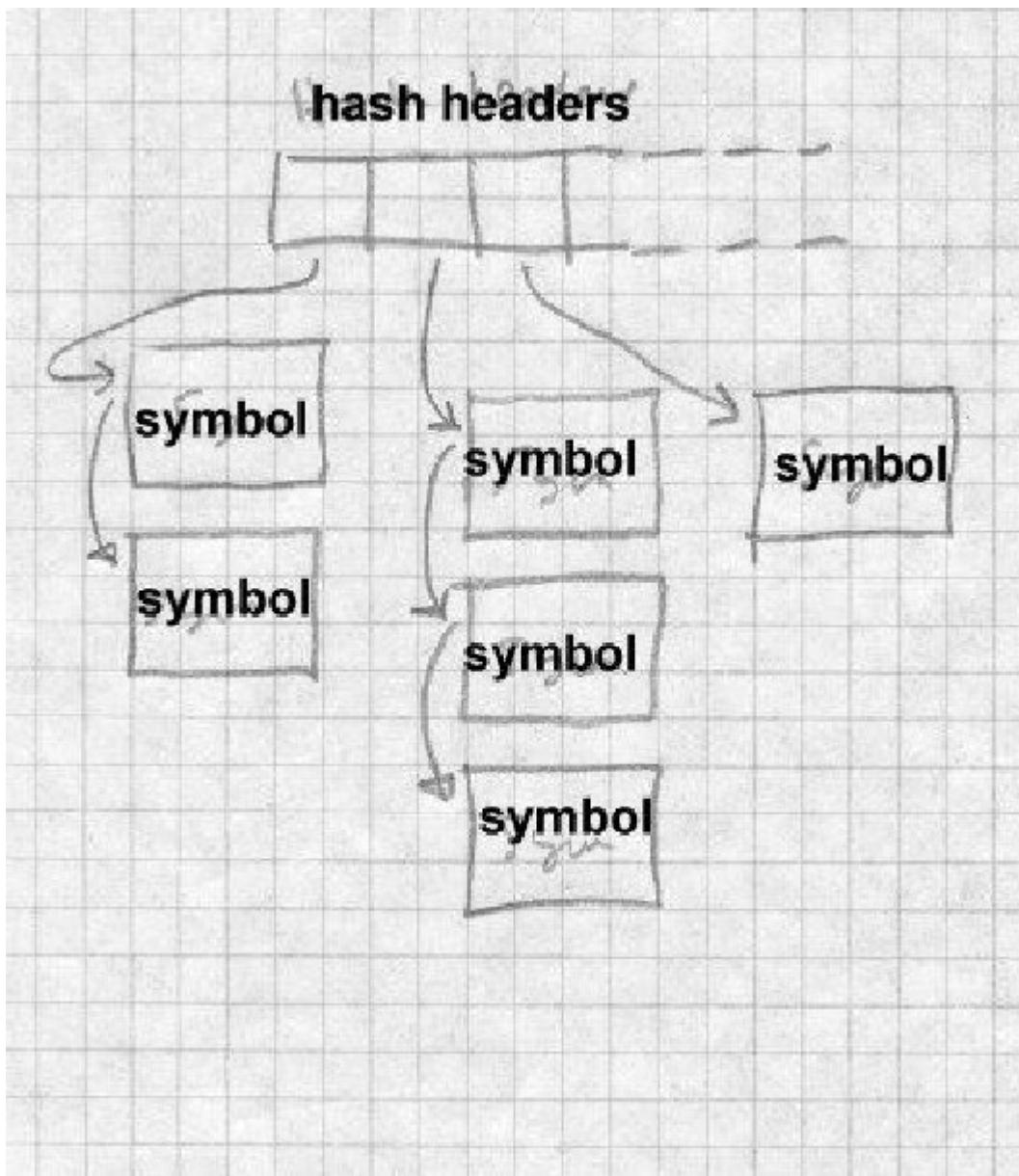
更早的名称长度受限的链接器会在查找 hash 链中对每一个符号名称进行字符串比较，直到找到匹配项或遍历完毕。现在的程序经常会有很多长符号在最后几个字符之前都是相同的（在 C++ 修改名称的情况下这经常发生），这就带来很大的字符串比较开销。一个简单的解决办法是将所有的 hash 值都保存在符号表中，并且只在 hash 值相同的时候才进行字符串比较。根据上下文情况的不同，如果一个符号没有被找到，链接器可能会将它加入相应的链中，也可能会报一个错误。

---

### 图 5-1：符号表

具有 hash 或者 hash 符号队列的典型符号表

```
struct sym *symhash[NBUCKET];  
struct sym {  
    struct sym *next;  
    int fullhash; /*全 hash 值*/  
    char *symname;  
    ...  
};
```



## 模块表

链接器需要跟踪整个链接过程中出现的每一个输入模块，即包括明确链接的模块，也包括从库中提取出来的模块。图 2 所示可以产生 a.out 目标文件的 GNU 链接器的简化版模块表结构。由于每个 a.out 文件的关键信息大部分都在文件头部中，该表仅仅是将文件头部复制过来。

图 5-2：模块表

```
/* 该文件名称 */
char *filename;
/* 符号名字串起始地址 */
char *local_sym_name;
```

```

/* 描述文件内容的布局 */
/* 文件的 a.out 头部 */
struct exec header;
/* 调试符号段在文件内的偏移量，如果没有则为 0 */
int symseg_offset;
/* 描述从文件中加载到内核的数据 */
/* 文件的符号表 */
struct nlist *symbols;
/* 字串表大小，以字节为单位 */
int string_size;
/* 指向字串表的指针 */
char *strings;
/* 下面两个只在 relocatable_output 为真，或输出未定义引用的行号时使用 */
/* 文本和数据的重定位信息 */
struct relocation_info *textrel;
struct relocation_info *datarel;
/* 该文件的段与输出文件的关系 */
/* 该文件中文本段在输出文件核心镜像中的起始地址 */
int text_start_address;
/* 该文件中数据段在输出文件核心镜像中的起始地址 */
int data_start_address;
/* 该文件中 BSS 段在输出文件核心镜像中的起始地址 */
int bss_start_address;
/* 该文件中第一个本地符号在输出文件中符号表中的偏移量，以字节为单位 */
int local_syms_offset;

```

---

该表中还包含了指向符号表、字串表（在一个 a.out 文件中，符号名称字串是在符号表外另一个单独的表中）和重定位表在内存中副本的指针，同时还有计算好的文本、数据和 BSS 段在输出中的偏移量。如果该文件是一个库，每一个被链接的库成员还有它自己的模块表表项（细节在此略去）。

第一遍扫描中，链接器从每一个输入文件中读入符号表，通常是将它们一字不差的复制到内存中。在将符号名放入单独的字串表的符号格式中，链接器还要将符号表读入，并且为了后续处理更容易一些，还要遍历符号表将每一个的名称字串偏移量转换为指向内存中名称字串的指针。

## 全局符号表

链接器会保存一个全局符号表，在任何输入文件中被引用或者定义的符号都会有一个

表项，如图 3 所示。每次链接器读入一个输入文件，它会将该文件中所有的全局符号加入到这个符号表中，并将定义或引用每个符号的位置用链表组织起来。当第一遍扫描完成后，每一个全局符号应当仅有一个定义，0 或多个引用（这里稍微简化了一些，因为 UNIX 目标文件会将公共块伪装成具有非零值的未定义符号，但这只是一个链接器很容易处理的特殊情况）。

---

图 5-3：全局符号表

```
/* 摘自 GNU ld a.out */
struct glosym
{
    /* 指向该符号所在 hash 桶中下一个符号的指针 */
    struct glosym *link;
    /* 该符号的名称 */
    char *name;
    /* 作为全局符号的符号值 */
    long value;
    /* 该符号在文件中的外部 nlist 链表，包括定义和引用 */
    struct nlist *refs;
    /* 非零值则意味该符号被定义为公共块，该数值即各公共块中的最大尺寸 */
    int max_common_size;
    /* 非零意味着该全局符号是存在的。库程序不能根据该数值加载 */
    char defined;
    /* 非零则意味着一个确信被加载的文件中引用了该全局符号。大于 1 的数值是该
       符号定义的 n_type 编码
    */
    char referenced;
    /* 1 表示该符号具有多个定义
       2 表示该符号具有多个定义，其中一些是集合元素，并且有一个已经被打印出
          来了
    */
    unsigned char multiply_defined;
}
```

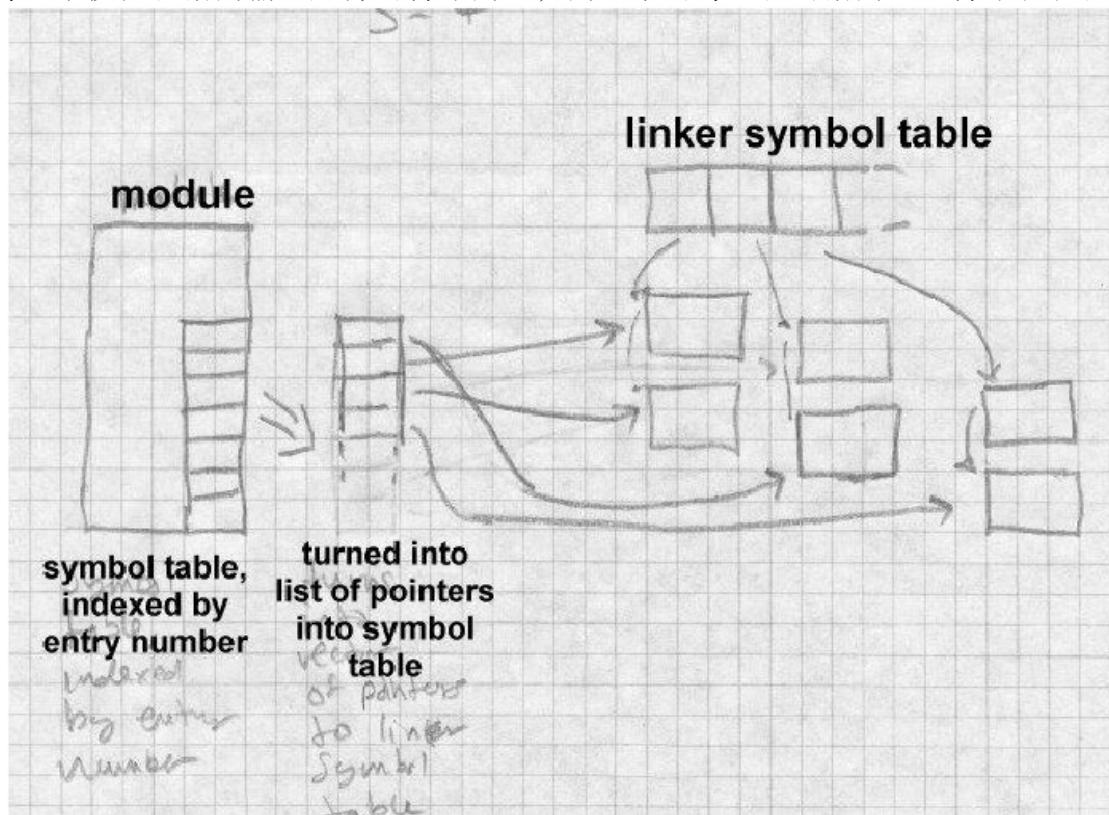
---

由于每个输入文件中的全局符号都被加入到全局符号表中，链接器会将文件中每一个项链接到它们在全局符号表中对应的表项中，如图 4 所示。重定位项一般通过索引模块自己的符号表来指向符号，因此对于每一个外部引用，链接器必须要对此很清楚，例如模块 A 中的符号 15 名为 fruit，模块 B 中的符号 12 同样名为 fruit，也就是说，它们是同一个符号。每一个模块都有自己的索引集，相应也要用自己的指针向量。

---

图 5-4：通过全局符号表来解析文件中的符号

每一个模块项指向输入文件的符号向量，向量中的每一项均指向全局符号表表项。



## 符号解析

在链接的第二遍扫描过程中，链接器在创建输出文件时会解析符号引用。解析的细节与重定位（见第七章）是有相互影响的，这是因为在多数目标格式中，重定位项标识了程序中对符号的引用。在最简单的情况下，即链接器使用绝对地址来创建输出文件（如 UNIX 链接器中的数据引用），解析仅仅是用符号地址来替换符号的引用。如果符号被解析到地址 20486 处，则链接器会将相应的引用替换为 20486。

实际情况要复杂得多。诸如，引用一个符号就有很多种方法，通过数据指针，嵌入到指令中，甚至通过多条指令组合而成。此外，链接器生成的输出文件本身经常还是可以再次链接的。这就是说，如果一个符号被解析为数据区段中的偏移量 426，那么在输出中引用该符号的地方要被替换为可重定位引用的 [数据段基址+426]。

输出文件通常也拥有自己的符号表，因此链接器还要新创建一个在输出文件中符号的索引向量，然后将输出重定位项中的符号编号映射到这些新的索引中。

## 特殊符号

很多系统还会使用少量链接器自己定义的特殊符号。所有的 UNIX 系统都要求链接器定义 etext、edata 和 end 符号依次作为文本、数据和 BSS 段的结尾。系统调用 sbrk() 将 end 的

地址作为运行时内存堆的起始地址，所以堆可以连续的分配在已经存在的数据和 BSS 的后面。

对于具有构造和析构例程的程序，很多链接器会为每一个输入文件创建指向这些例程的指针表，并通过链接器创建的诸如 `_CTOR_LIST_` 这样的符号让该语言的启动代码可以找到这个表并依次调用其中所有的例程。

## 名称修改

在目标文件符号表和链接中使用的名称，与编译目标文件的源代码程序中使用的名称往往是有差别的。主要原因有 3：避免名称冲突，名称超载，和类型检查。将源代码中的名称转换为目标文件中的名称的过程称为名称修改（name mangling）。本节讨论 C、Fortran 和 C++ 程序中典型的名称修改。

### 简单的 C 和 Fortran 名称修改

在较早的（也许是 1970 年之前）目标格式中，编译器将源代码中的名称直接用作目标文件中的名称，可能会依据名称长度的限制将名称截短。这种方法还算不错，但有时会与编译器或库中预留的名称冲突。例如进行格式化 I/O 的 Fortran 程序会隐含的调用库中的例程来进行读和写。此外还有其它例程来处理算术错误、复杂计算和其它在程序语言中过于复杂而不能通过 `inline` 函数实现的东西。

所有这些例程的名称实际上都是保留名称，部分编程经验可以告诉我们不能使用哪些名称。作为一个惊人的实例，下面这个 Fortran 程序在若干年中都可以使 OS/369 系统崩溃：

```
CALL    MAIN  
END
```

为什么呢？OS/360 系统的编程规定要求任何一个包括主程序的例程都要有一个名称，而主程序的名称为 `MAIN`。当一个 fortran 主程序启动时，它调用操作系统来捕获一系列的算术错误陷阱（arithmetic error traps），每一个陷阱捕获调用都会在系统表中分配一些空间。但这个程序不停的递归调用自己，每次都会嵌套的创建一系列嵌套的陷阱调用，当系统表超出存储空间大小时，系统就崩溃了。OS/390 要比它 30 年前的前任健壮的多，但预留名称的问题仍然存在。在混合语言的程序中，情况甚至更糟，因为所有语言的代码都要避免使用任何其它语言运行时库中已经用到的名称。

解决预留名称问题的方法之一是用其它东西（而不是过程调用）来调用运行时库。例如在 PDP-6 和 PDP-10 上，对 Fortran I/O 包的接口是通过一系列绕回到程序（而不是操作系统）的系统调用指令实现的。这曾经是一个聪明的技巧，但是它特定于 PDP-6/10 架构，扩展性不是很好，因为没有办法在混合语言的代码中共享这些陷阱，而仅仅链接所需要的 I/O 包的最小部分实际中也是行不通的，因为没有什么容易的办法可以知道程序的输入模块中会用到哪些陷阱。

UNIX 系统采取的办法是修改 C 和 Fortran 过程的名称这样就不会因为疏忽而与库和其它例程中的名称冲突了。C 过程的名称通过在前面增加下划线来修饰，所以 `main` 就变成了 `_mai`

n。Fortran 的名称进一步被修改首尾各有一个下划线，所以 calc 就成了\_calc\_（这种独特的方法使得从 Fortran 中调用 C 中名字末尾带有下划线的例程，这样就可以用 C 编写 Fortran 的库）。这种策略唯一明显的缺点是将目标格式中允许的 C 名称长度从 8 字节减少到 7 字节，对 Fortran 则减少到 6 个字节。那时 Fortran-66 标准只要求 6 个字符的名称，所以并没有什么问题。

在其它系统上，编译器设计者们采取了截然相反的方法。多数汇编器和链接器允许在符号中使用 C 和 C++ 标识符中禁用的字符，如. 或者\$。运行库会使用带有禁用字符的名称来避免与应用程序的名称冲突，而不再是修改 C 或 fortran 程序中的名称。采用哪一种方式取决于开发人员的方便。在 1974 年 UNIX 被用 C 语言重写的时候，它的作者就已经扩展了汇编语言的库，那时比起重新开始修改已经存在的代码，修改新的 C 和 C 兼容例程的名称要来的更容易一些。而现在，已经过了 20 年了，汇编器的代码都已经被全部重写了 5 次了，而编译器（尤其是那些创建 COFF 和 ELF 目标文件的编译器），也已经不再使用前缀下划线了。

## C++类型编码：类型和范围

修改名称的另一个用处是将范围和类型信息编码，这样就可以用现存的链接器来链接使用 C++、Ada 和其它比 C、Cobol 和 Fortran 具有更复杂命名规则的语言编写的程序了。

在一个 C++ 程序中，程序员可以定义很多具有相同名称但范围不同的函数和变量，对于函数，还有参数类型。一个单独的程序可以具有一个名为 V 的全局变量和一个类中的静态成员 C::V。C++ 允许函数名重载，即一些具有相同名称不同参数的函数，例如 f(int x) 和 f(float x)。类的定义可以括入函数，括入重载名称，甚至括入重新定义了内嵌操作的函数，即一个类可以包含一个函数，它的名字实际上可以是>> 或其它内建操作符。

C++ 最初是通过名为 cfont 的翻译器来实现的，生成 C 代码并使用已有的链接器，因此它的作者对过程名称进行名称修改以在 C 编译器不察觉的情况下由链接器来处理。所有的链接器都必须将匹配相同名称的有定义和无定义符号做为自己的基本工作。从此几乎所有的 C++ 编译器都会直接生成目标代码或至少是汇编代码，但名称修改仍然被保留下来做为处理名称冲突的标准方法。虽然现代链接器已经很清楚名称修改会影响报错信息中的名称，但它们还是显示了修改过的名称。

主流的 C++ 手册都描述了 cfront 使用过的这种名称修改策略，其中的一些微小变化现在已经成为了事实标准。我们在这里对此进行描述。

C++ 类之外的数据变量名称不会进行任何的修改。一个名为 foo 的数组修改后的名称仍为 foo。与类无关的函数名称修改后增加了参数类型的编码，通过前缀\_F 后面跟表示参数类型的字母串来实现。图 5 列出了各种可能的类型表示。例如，函数 func(float, int, unsigned char) 变成了 func\_FfiUc。类的名称会被当作是各种类型来对待，编码为类名称长度数字后面跟类的名称，例如 4Pair。类还可以包含内部多级子类的名称，这种限定性 (qualified) 名称被编码为 Q，还有一个数字标明该成员的级别，然后是编码后的类名称。因此 First::Second::Third 就变成了 Q35First6Second5Third。这意味着采用两个类做为参数的函数 f(Pair, First::Second::Third) 就变成了 f\_\_F4PairQ35First6Second5Third。

图 5-5: C++中调整后的名字的类型

类型	字母
void	v
char	c
short	s
int	i
long	l
float	f
double	d
long double	r
varargs	e
<hr/>	
unsigned	U
const	C
volatile	V
signed	S
<hr/>	
pointer	P
reference	R
array of length n	An_
function	F
pointer to nth member	MnS

类的成员函数编码为：先是函数名，然后是两个下划线，接着是编码后的类名称，然后是F和参数，所以 `c1::fn(void)` 就变成了 `fn_2c1Fv`。所有的操作符都具有4到5个字符的编码后名称，诸如“\*”对应`_m1`，“|=”对应`_aor`。包括构造、析构、new 和 delete 在内的特殊函数编码为`_ct`、`_dt`、`_nw` 和 `_d1`。因此具有两个字符指针参数的类 Pair 的构造函数 `Pair(char *, char*)` 的名称就变成了`_ct_4PairFPcPc`.

最后，由于修改后的名称会变得很长，因此对具有多个相同类型参数的函数有两种简捷编码。代码 `Tn` 表示“与第 `n` 个参数类型相同”，`Nm` 表示“`n` 个参数与第 `m` 个参数的类型相同”。因此函数 `segment(Pair, Pair)` 的名称就成了 `segment_F4PairT1`，而函数 `trapezoid(Pair, Pair, Pair)` 的名称就是 `trapezoid_F4PairN31`。

名称修改可以为每一个可能的 C++ 类提供唯一的名称，相应的代价就是在错误信息和列表中会出现惊人长度和（在没有链接器和调试器支持下）难以理解的名称。尽管如此，C++ 还有一个本质上的问题就是名字空间相当巨大。任何表示 C++ 对象名称的策略都会具有和名称修改相近的冗余，而名称修改的优势在于至少还有一些人可以读懂它。

名称修改的早期用户经常会发现虽然链接器在理论上支持长名称，但实际上长名称效

果并不很好，尤其针对具有大量仅最后几个字符不同的名称的程序，性能非常糟糕。幸运的是，符号表算法是一个很好理解的方法，我们可以期望链接器通过它顺利的处理长名称。

## 链接时类型检查

虽然名称修改在 C++ 出现后才流行起来，但链接器类型检查的思想已经由来已久（我第一次碰到它是 1974 年在 Dartmouth PL/I 链接器上）。链接器类型检查的想法非常简单。多数语言都有声明了参数类型的过程，如果调用者没有将被调用过程期望的参数个数或类型传递给被调用者，那就是错误，如果调用者和被调用者在不同的文件中被编译，那这种错误是非常难以察觉的。对于链接器类型检查，每一个定义和未定义的全局符号都会有一个用字串表示的参数和返回值类型，与名称修改中的 C++ 参数类型相近。在链接器解析一个符号时，它将引用处的类型串与符号定义处的类型串进行比较，如果不匹配则报错。这个策略的好处之一就是链接器根本不需要理解类型编码的含义，仅仅比较字串是否相同就可以了。

即使在一个支持 C++ 名称修改的环境中，由于并不是所有的 C++ 类型信息都会被编码到修改的名称中，因此这种类型检查仍然非常有用。通过与此类似的策略来进行函数返回值类型、全局数据类型的检查也是非常有益的。

## 弱外部符号和其它类型符号

目前位置，我们一直认为所有链接器全局符号的工作方式都是相同的，每次提到的名称要么是定义，要么是对符号的引用。很多目标格式都会将引用分为弱或者是强。强引用必须被解析，而弱引用在存在定义则解析，如果不存在定义也不认为是错误。链接器处理弱引用的方式与强引用很相似，除了第一次扫描结束后没有定义的弱引用不会报错。通常链接器会将未定义的弱符号定义为 0，这是一个应用程序代码可以检查的数值。弱符号在连接库的时候是非常有用的，因此我们将在第 6 章进行讨论。

## 维护调试信息

现代编译器都支持源代码级的调试。即程序员可以基于源代码的函数和变量来调试目标代码，设置断点和单步跟踪。编译器通过将调试信息插入目标文件来实现的，调试信息包括源代码行号到目标代码地址的映射，并描述了程序中用到的所有函数、变量、类型和数据结构。

UNIX 使用了两种颇为不同的调试信息格式，即主要使用在 a.out、COFF 和非系统 5 (System V) ELF 文件的 stab ( 符号表的缩写 ) 格式，和定义在系统 5 的 ELF 文件中的 DWARF 格式。微软为它们的 Codeview 调试器定义了自己的格式，最新的版本是 CV4。

## 行号信息

所有基于符号的调试器都必须将程序地址和源代码行号对应起来。这样就可以通过调试器将断点放入代码的适当位置来实现用户基于源代码行号的断点设置，并可以让调试器将调用堆栈中的程序地址和错误报告中的源代码行号关联起来。

除优化编译代码外，行号信息是很简单的。优化编译的代码中会去除一些代码，导致目标文件中的代码序列与源代码行号的序列不匹配。

对于编译器生成代码所对应源代码文件中的每一行语句，编译器会产生一个行号项（包括行号和代码开始位置）。如果一个程序地址跨越了两个行号项，调试器会将两个行号中较小的报告出来。行号还需要被文件名称（包括源文件名称和头文件名称）限定。有一些格式会通过创建一个文件列表并将文件索引放入每一个行号项中来实现这一点，行号列表中的“begin include”和“end include”项，内在的维护了有行号成员组成的栈。

当编译器优化根据语句生成不连续的代码时，一些目标格式（DWARF）让编译器将每一个字节都映射回源代码中的一行，这会占用进程的大量空间，而其它格式则仅仅产生一个大概的位置。

## 符号和变量信息

编译器还要为每一个程序变量生成名称、类型和位置。调试符号信息某种程度上要比名称修改更为复杂，因为它不仅要对类型名称编码，还有定义类型时的数据结构类型，这样才能保证调试器能够正确处理一个数据结构中的所有子域的格式。

符号信息可以是一个隐式或显式的树结构。每个文件的最顶层是在最顶层定义的类型、变量和函数的列表，每一个内部是数据结构的子域，或函数内部定义的变量，诸如此类。在函数内部，包含“begin block”和“end block”的树标识了对行号的引用，这样调试器就可以指出程序中每一个变量的范围了。

符号信息中最有趣的部分是位置信息。静态变量的位置不会改变，但一个例程中的局部变量可能是静态的，可能在栈里、在寄存器里、在优化后的代码里，在例程的不同部分可能会从一个地方移动到另一个地方。在多数体系结构上，标准的例程调用序列会为每一个嵌套的例程维护保存堆栈和框指针（frame pointer）的链，每个例程中的局部栈变量存放在相对于框指针的已知偏移量处。在叶子例程或者没有分配局部栈变量的例程中，有一个通常使用的优化就是跳过对框指针的设置。为了正确解释栈的调用轨迹并在没有框指针的例程中寻找局部变量，调试器就必须清楚这些。Codeview通过一个由没有框指针的例程组成的队列来做到这一点。

## 实际的问题

多数情况下，链接器仅仅传递调试信息而不对其进行解释，也可能在这个过程中会重

定位和段相关的地址。

链接器开始做的一件事情就是探测和去除重复调试信息。在 C 和某些特定的 C++ 中，程序通常都会有一系列定义类型和声明函数的头文件，每一个源文件会将定义了该文件可能使用的类型和函数的头文件都包括进来。

编译器会为每一个源代码文件包括的所有头文件中的所有内容都扫描生成调试信息。这意味着如果某个特定的头文件被 20 个会编译和链接到一起的源文件所包括的话，那链接器将会收到该文件的 20 份调试信息副本。虽然保留这些冗余信息调试器工作起来不会有任何麻烦，但头文件，尤其是在 C++ 中会有大量的头文件，这意味着重复的头文件信息是相当巨大的。链接器可以放心的忽略掉重复的部分，这样既可以加快链接器和调试器的速度，也可以节省空间。某些情况下，编译器会将调试信息直接放到文件或数据库中供调试器读取，而绕过了链接器。这样链接器就只需要添加和升级与分布在源文件中的各个段相对位置有关的信息即可，而诸如跳转表之类的数据会由链接器自己来创建。

当调试信息存储在目标文件中时，有时候调试信息会和链接器符号表混杂在一个大的符号表中，而有时，它们是独立的。很多年来，UNIX 系统一点一点增加了编译器中的调试信息，最后就变成了现在这个巨大的符号表。包括微软 ECOFF 在内的其它一些格式趋向于将链接器符号、调试符号和行号信息分开处理。

有时调试信息结果会存储到输出文件中，有时会输出到单独的调试文件，有时两者都会有。在构建过程中将所有调试信息都放到输出文件中的做法有一个显而易见的好处，就是调试程序所需要的信息都存放在一个地方。明显的缺点就是这将导致可执行程序体积非常庞大。同样如果调试信息被分离出去，就很容易构建最终版本的程序，然后出售没有调试信息的可执行程序。这会减小发布程序的大小并增加逆向工程的难度，但开发者还拥有在调试已发货软件错误时会用到的调试文件。UNIX 系统有一个 `strip` 命令，可以将调试符号从一个目标文件中去除而不改变任何代码。开发者可以保存未 `strip` 的文件并发布 `strip` 过的版本。即使这种情况下两个文件是不同的，但运行代码是一样的，并且调试器可以通过未进行 `strip` 的文件中符号来调试 `strip` 过版本生成的核心转储（core dump）文件。

## 练习

1. 写一个有很多函数的 C++ 程序，这些函数的修改后名称只有最后几个字母不相同。看看需要多久才能完成编译。将它们调整为修改后名称的头几个字母就有差别，再次计算编译和链接的时间。你需要一个新的链接器吗？

2. 研究一下你喜欢的链接器所使用的调试符号格式（参考书中列出了一些线上资源）。写一个程序从一个目标文件中将调试信息导出来，看看通过它你可以重建多少源代码程序。

## 项目

项目 5-1：扩展链接器来处理符号名解析。让链接器从每一个文件中读取符号表，并创建一个（链接器后继部分会用到的）全局符号表。全局符号表中的每一个符号要包括名称、

该符号是否有定义，以及哪个模块定义了它。注意要检查未定义的和重复定义的符号。

项目 5-2：为链接器添加符号值解析功能。由于多数符号都是相对于输入文件的段定义的，它们的数值需要根据每个段被重定位后的地址来进行调整。例如一个符号定义为某文件中文本段内偏移量 42 的位置，而该段被重定位到 3710，则该符号的值要调整为 3752。

项目 5-3：完成项目 4-2 的工作：处理 UNIX 风格的公共块。为每一个公共块赋予位置数值。

# 第 6 章 库

所有的现代链接器都可以处理库，即按照被链接程序的需要加入的目标文件集合。本章我们将涉及传统的静态链接库，更为复杂的动态链接库将在第 9 章和第 10 章中看到。

## 库的目的

在 40 年代和 50 年代早期，软件工作室实际上已经存在有成圈的磁带及后期成迭的卡片构成的代码库了，程序员可以查看并选择例程加入到他们自己的程序中。在加载器和链接器开始解析符号引用后，通过从库中选择例程来解析未定义符号，自动处理这个过程就成为了可能。

从本质上说，库文件就是由多个目标文件聚合而成的，通常还会加入一些有助于快速查找的目录信息。细节信息总是会比基本思想复杂的多，因此本章中我们会逐步深入。我们使用术语“文件”表示一个单独的目标文件，“模块”表示被包含到库中的一个目标文件。

## 库的格式

最简单的库格式就是仅仅将目标模块顺序排列。在诸如磁带和纸带这样的顺序访问介质上，对于增加目录要注意的是，由于链接器不得不将整个库读入，因此跳过库成员和将他们读入的速度差不多。但在磁盘上，目录可以相当显著的提高库搜索速度，现在已经成为标准组件。

## 使用操作系统

OS/360 包括 MVS 在内的后继型号提供了分区数据集（PDS，partitioned data set），它包含了多个命名成员，每个都可以被当作一个顺序文件来处理。系统具有可以为单一成员提供多个别名的特性，这是为了在程序运行期间将多个 PDS 当作一个逻辑 PDS 来处理，为了在一个逻辑 PDS 中枚举所有名称，当然也为了对成员进行读和写操作。成员名称为 8 个字符长，很可能与链接器中的外部符号长度不一致（MVS 引入了一种扩展的 PDS 或 PDSE 可以支持最长 1024 个字符的名称，这对于 C、C++ 和 Cobol 程序是有好处的）。

一个链接器库就是一个 PDS，它的每一个成员均为根据其入口点命名的目标文件。定义了多个全局符号的目标文件对每一个全局符号都有一个在构建库时手工创建的别名。链接器以寻找所含名称与未定义符号相匹配的成员库的方式搜索逻辑 PDS。这种方法的好处是不需要额外的目标库升级程序，对于 PDS 来说标准的文件操作工具就可以胜任了。

虽然我从未见过某个链接器这么做，但是在类 UNIX 操作系统上的链接器可以采用相同的方法来处理库：库可以是一个目录，成员目标文件位于目录中，成员目标文件中的每一个全局符号在目录中都有一个文件名（UNIX 允许一个文件具有多个文件名）。

## UNIX 和 Windows 的 Archive 文件

UNIX 链接器库使用一种称为“archive”的格式，它实际上可以用于任何类型文件的聚合，但实践中很少用于其它地方。库的组成，首先是一个 archive 头部，然后交替着是文件头部和目标文件。最早的 archive 没有符号目录，只有一系列的目标文件，但后续版本就出现了多种类型的目录，最后沉淀为一个类型并在 BSD 版本（文本 archive 头部和一个称为 \_\_SYMDEF 的目录）和 System V.4 的 COFF 和 ELF 库当前版本中使用了将近十年，BSD 的后期版本，Linux，Windows 的 ECOFF 库使用和 COFF 库相同的 archive 格式，但是对于目录，虽然仍然称为 /，但是格式是不同的。

### UNIX archive

所有的现代 UNIX 系统都采用大同小异的 archive 格式，如图 1 所示。该格式在 archive 头部中只使用文本字符，这意味着文本文件的 archive 文件本身就是文本的（尽管在实践中可以知道这个特性并没有太大的用处）。archive 文件都是以 8 字符的标志串 !<arch>\n 开头，其中 \n 是换行符。在每一个 archive 成员之前是一个 60 字节的头部，包含有：

该成员名称，补齐到 16 个字符（下面会讲到）。

修改时间，由从 1970 年到当时的十进制秒数表示。

十进制数字表示的用户和组 ID。

一个八进制数表示的 UNIX 文件模式。

以字节为单位的十进制数表示的文件尺寸。如果该尺寸为奇数，那么文件的内容中会补齐一个换行符使得总长度为偶数，但这个补齐的字符不会计算在文件尺寸域中。

保留的两个字节，为引号和换行符。这样就可以让头部成为一行文本，并可用来简单的验证当前头部的有效性。

每一个成员头部都会包含修改时间、用户和组 ID、文件模式，尽管链接器会将它们忽略。

---

图 6-1 Unix 档案文件格式

File header:

!<arch>\n

Member header:

```
char name[16];      /* 成员名称 */  
char modtime[12];   /* 修改时间 */  
char uid[6];        /* 用户 ID */  
char gid[6];        /* 组 ID */  
char mode[8];       /* 8 进制文件模式 */  
char size[10];      /* 成员大小 */  
char eol[2];         /* 保留空间，一对引号/换行符 */
```

---

成员名称是 15 个字符或更少，紧随其后的空格将它补齐为 16 个字符，或者在 COFF 或 E

LF 的 archive 格式中，会在斜杠后面跟随足够多的空格将总数补齐为 16 个字符（UNIX 和 Windows 都是使用斜杠来分割文件名称）。a.out 文件所使用的这种文档格式版本不支持长于 16 个字符的成员名称，这也与 BSD 之前的 UNIX 文件系统将文件路径中的各部分长度限制为 14 个字符相呼应（某些 BSD 档案文件格式确实可以支持更长的文件名称，但是由于链接器不能正确处理更长的名称，因此没有人使用它们）。COFF, ELF 和 windows 档案文件在称为//的档案成员中存储超过 16 个字符的长名称。该成员包含着（在 UNIX 下）由斜杠、换行分割，或（在 windows 下）由空字符 NUL 分割的长名称。一个具有长名称的头部成员的名称域包含一个斜杠和随后由数字表示的在//成员中对应于名称串的偏移量。在 Windows 档案文件中，//成员必须是档案文件的第三个成员。在 UNIX 系统中如果没有长名称则该成员无须存在，但如果有长名称的话，它会跟在符号目录后面。

虽然符号目录的格式多少有些不同，但功能上都是相似的，即将各名称与成员位置相映射，以便链接器可以直接移动到它所需要的成员处并进行读取。

a.out 档案文件将目录存储在一个称为\_.SYMDEF 的成员中，如图 2 所示，它必须是档案文件的首个成员。该成员起始的第一个字包含了以字节为单位表示的随后符号表的大小，因此符号表中的表项个数应当是该字数值的 1/8。紧随符号表后的一个字表示了随后的字符串表大小，然后接着是字符串表，每个字符串都以空字节结尾。每个符号表项都包含一个以 0 为起始的偏移量，它指示了该符号名称在符号表中的位置，以及定义了该符号的成员的头部在文件中的位置。符号表项的顺序通常与文件中各成员的顺序相同。

---

图 6-2 SYMDEF 目录格式

```
int tablesize;           /* 表示随后符号表的大小，以字节为单位*/
struct symtable {
    int symbol;          /* 在字符串表中的偏移量 */
    int member;           /* 成员指针 */
} symtable [];
int stringsize;          /* 表示随后字符串表的大小，以字节为单位 */
char strings[];          /* 多个以空字符结尾的字符串 */
```

---

COFF 和 ELF 档案文件格式使用了另一个不可能出现在文件名中的符号/作为符号目录的名称（而不是采用\_.SYMDEF）并使用了一种更简单的格式，如图 3 所示。最初 4 个字节的数值为符号个数。随后是由档案文件成员在文件中的偏移量构成的数组，然后是一系列由空字符结尾的字符串。第一个偏移量指向的成员定义了由字符串表中第一个字符串命名的符号，以此类推。COFF 档案文件通常会忽略当前体系结构的字节序而符号表中采用 big-endian 的字节序。

---

Figure 6-3: COFF/ELF 目录格式

```
int nsymbols;           /* 符号个数 */
int member[];            /* 成员偏移量数组 */
char strings[];          /* 多个以空字符结尾的字符串 */
```

---

微软的 ECOFF 档案文件格式增加了第二个符号目录成员，如图 4 所示。跟在第一个的后面并且莫名其妙的也称为 /。

---

Figure 6-4: ECOFF 的第二个符号目录

```
int nmembers;      /* 成员偏移量的个数 */
int members[];    /* 成员偏移量数组 */
int nsymbols;     /* 符号的个数 */
ushort symndx[]; /* 成员偏移量的指针 */
char strings[];   /* 符号名称，以字母顺序排列 */
```

---

ECOFF 目录由一个成员项个数和跟在其后的成员偏移量数组组成，数组的每个元素对应一个档案文件成员。后面依次是符号项个数、两字节的成员偏移量指针构成的数组，以及相应数量的按照字母顺序排列、以空字符结尾的符号字符串。成员偏移量指针包含从 1 开始的成员偏移量表的索引，该表属于定义了相对应符号的成员。例如，如果要定位与第 5 个符号相对应的成员，就可以去查找指针数组中的第 5 项，它包含了在成员偏移量数组中对应该符号定义的索引。理论上经过排序的符号可以进行快速查找，但在实际中速度的提升并没有（预计的）那么大，因此链接器通常会扫描整个表来查找要加载的符号。

## 扩展到 64 位

即使一个档案文件包含 64 位架构的目标文件，只要档案文件大小没有超过 4GB，就无须为 ELF 和 ECOFF 改变档案文件格式。不过有一些 64 位架构有不同的符号目录格式和成员名称（例如/SYM64/）。

## Intel OMF 库文件

---

我们最后看的库文件格式是 Intel OMF 库文件。同样，一个库是一系列的目标文件和一个符号目录。与 UNIX 库不同的是，目录位于文件的尾部，如图 5 所示。

---

图 6-5: OMF 库

LIBHED 记录

第 1 个目标模块（文件）

第 2 个目标模块（文件）

...

LIBNAM 模块名称记录

LIBLOC 模块位置记录

LIBDIC 符号目录

---

该库起始是一个 LIBHED 记录，它包含 LIBNAM 在文件中的偏移量，偏移量采用 Intel IS IS 操作系统使用的 (block, offset) 格式来表示。LIBNAM 记录只是简单的包含一个模块名称的列表，每个名称之前都有一个计数字节指明该名称串的长度。LIBLOC 记录包含了由 (file, offset) 对组成的队列，它们标识了各个模块在文件中的起始位置。LIBDIC 包含了由若干具有长度计数的字串群组构成的队列，这些字串的内容是各个模块中定义的名称，每个字串群组的末尾跟着一个空字节将当前字串群组与后续字串群组分割开来。

虽然这种格式有点晦涩，但必要的信息它都具有，而且工作的很不错。

## 建立库文件

每种档案文件格式都有它自己建立库文件的方法。根据操作系统对档案格式支持程度的不同，库的创建会涉及包括从标准系统文件管理程序到库特定工具在内的任何东西。

做为一个极端，IBM MVS 库可以通过标准的 IEBCOPY 工具来创建，该工具可以创建分区的数据集。做为中间的一种情况，UNIX 库由 ar 命令来创建，它可以将多个文件合并为档案文件。对于 a.out 格式的档案文件，有一个名为 ranlib 的独立程序来添加符号目录，从每个成员中读取符号，创建\_.SYMDEF 成员并将其放入文件中。原则上说 ranlib 也可以将符号目录创建为一个单独的文件然后调用 ar 命令将该文件加入到档案文件中，但实际上 ranlib 会直接操作档案文件。对于 COFF 和 ELF 档案文件，ranlib 创建符号目录（如果有成员是目标代码模块的时候）的功能被转移到了 ar 中，尽管 ar 也可以创建没有目标代码模块的档案文件。

做为另一个方向的极端，OMF 和 Windows ECOFF 档案文件是由专门程序库管理程序创建，因为除了目标代码库外这些档案格式不会被用在其它任何地方。

库创建中有一个小问题，是目标文件的顺序，尤其是对那些不具有符号目录的古老格式。在 ranlib 出现之前的 UNIX 有一对叫做 lorder 和 tsort 的程序来帮助创建档案文件。lorder 程序的输入是一系列的目标文件（而不是库），输出是一个依赖性列表记录了一个文件其它文件中的哪些符号（这并不难，经典的 lorder 代码实现，曾经而且现在也仍然是 shell 脚本，它使用一个符号显示工具将符号都提取出来，对这些符号进行少许的文字处理，然后使用标准的 sort 和 join 程序来创建自己的输出）。tsort 对 lorder 的输出进行拓扑排序，产生一个排序后的文件的列表，这样符号可以在所有对它的引用后面来定义，这就可以通过对该文件的一次顺序扫描来解析所有的未定义引用。lorder 的输出会被用来控制 ar。虽然现代的库中的符号目录允许链接过程在工作时可以忽略一个库中各个目标模块的顺序，但大多数库仍然会由 lorder 和 tsort 来创建以提高链接过程的速度。

## 搜索库文件

一个库文件在创建后，链接器还要能够对它进行搜索。库的搜索通常发生在链接器的第一边扫描时，在所有单独的输入文件都被读入之后。如果一个或多个库具有符号目录，那

么链接器就将目录读入，然后根据链接器的符号表依次检查每个符号。如果该符号被使用但是未定义，链接器就会将符号所属文件从库中包含进来。仅将文件标识为稍后加载是不够的，链接器必须像处理那些在显式被链接的文件中的符号那样，来处理库里各个段中的符号。段会记入段表，而符号，包括定义的和未定义的，都会记入全局符号表。一个库例程引用了另一个库中例程的符号是相当普遍的现象，譬如诸如 `printf` 这样的高级 I/O 例程会引用像 `putc` 或 `write` 这样的低级例程。

库符号解析是一个迭代的过程，在链接器对目录中的符号完成一遍扫描后，如果在这边扫描中它又从该库中包括进来了任何文件，那么就还需要在进行一次扫描来解析新包括进来的文件所需的符号，直到对整个目录彻底扫描后不再需要括入新的文件为止。并不是所有的链接器都这么做的，很多链接器只是对目录进行一次连续的扫描，并忽略在库中一个文件对另一个更早扫描的文件的向后依赖。像诸如 `tsort` 和 `lorder` 这样的程序可以尽量减少由于一遍扫描给链接器带来的困难，不过并不推荐程序员通过显式的将相同名称的库在链接器命令行中列出多次来强制进行多次扫描并解析所有符号。

UNIX 链接器和很多 Windows 链接器在命令行或者控制文件中会使用一种目标文件和库混合在一起的列表，然后依次处理，这样程序员就可以控制加载目标代码和搜索库的顺序了。虽然原则上这可以提供相当大的弹性并通过将同名私有例程列在库例程之前而在库例程中插入自己的似有同名例程，在实际中这种排序的搜索还可以提供一些额外的用处。程序员总是可以先列出所有他们自己的目标文件，然后是任何应用程序特定的库，然后是和数学、网络等相关的系统库，最后是标准系统库。

当程序员们使用多个库的时候，如果库之间存在循环依赖的时候经常需要将库列出多次。就是说，如果一个库 A 中的例程依赖一个库 B 中的例程，但是另一个库 B 中的例程又依赖了库 A 中的另一个例程，那么从 A 扫描到 B 或从 B 扫描到 A 都无法找到所有需要的例程。当这种循环依赖发生在三个或更多的库之间时情况会更加糟糕。告诉链接器去搜索 A B A 或者 B A B，甚至有时为 A B C D A B C D，这种方法看上去很丑陋，但是确实可以解决这个问题。由于在库之间几乎不会有重复的符号，如果链接器可以像 IBM 的大型主机系统链接器或者 AIX 链接器那样，简单的将它们作为一个组一起搜索，那程序员就很舒服了。

该规则的一个主要例外是应用程序有时候会对少许例程定义自己的私有版本，尤其是对 `malloc` 和 `free`，为了进行堆存储管理往往想采用自己的私有版本而不是标准的系统库版本。在这种情况下，比使用一个链接器标志会注明“不要在库中搜寻这些符号”（效果相同但）更好的方法是在搜索顺序中将私有的 `malloc` 放在公共版本之前。

## 性能问题

和库相关的主要性能问题是花费在顺序扫描上的时间。一旦符号目录成为标准之后，从一个库中读取输入文件的速度就和读取单独的输入文件没有什么明显差别了，而且只要库是拓扑排序的，那链接器在基于符号目录进行扫描时很少会超过一遍。

如果一个库有很多小尺寸成员的话，库搜索的速度也会很慢。一个典型的 UNIX 系统库有超过 600 个成员。尤其是现在很普遍的一种情况就是库的所有成员会在运行时合并为一个单一的共享库，因此如果创建一个单一的目标文件包定义库中所有的符号，而在链接时使用

这个目标文件而不进行库的搜索，那么这种方法的速度似乎可以更快一点。这一点我们将在第 9 章中详细的检验。

## 弱外部符号

符号解析和库成员选择中所采用的简单的定义引用模式对很多应用而言显得灵活有余效率不足。例如，大多数 C 程序会调用 printf 函数族中的例程来格式化输出数据。printf 可以格式化各种类型的数据，包括浮点类型。这就意味着任何使用 printf 的程序都会将浮点库链接进来，即便它根本不使用浮点数。

曾经很多年里，UNIX 程序不得不使用一些技巧来避免在只使用整数的程序中链接入浮点库。C 编译器会在任何使用的浮点代码的例程中产生一个对特殊符号 `f1tused` 的引用。C 库的布局见图 6，它利用了链接器顺序搜索库的特点。如果程序使用了浮点，那么对 `f1tuse`d 的引用将会导致链接真正的浮点例程，包括真正的 `fcvt`（浮点输出例程）。然后当 I/O 模块被链接进来以定义 printf 时，就已经有一个可以满足 I/O 模块引用的 `fcvt` 在那里了。在那些不使用浮点的程序中，由于不会有任何未解析的符号，在 I/O 模块中引用的 `fcvt` 将会解析为库中跟在 I/O 例程后面的伪<sup>2</sup>浮点例程，因此真正的浮点例程将不会被加载。

---

图 6-6：经典的 UNIX C 库

...

真正的浮点模块，定义 `f1tused` 和 `fcvt`

I/O 模块，定义调用 `fcvt` 的 `printf` 函数

伪浮点例程，定义了伪 `fcvt`

...

---

虽然这个技巧可以工作，但用它处理多于一个或两个以上的符号时就会变得很难处理，而且它的正确性严重依赖于库中模块的顺序，尤其在重新构建库之后很容易产生问题。

解决这个困境的方法就是弱外部符号，就是不会导致加载库成员的外部符号。如果该符号存在一个有效的定义，无论是从一个显式链接的文件还是普通的外部引用而被链接进来的库成员中，一个弱外部符号会被解析为一个普通的外部引用。但是如果不存在有效的定义，弱外部符号就不被定义而实际上解析为 0，这样就不会被认为是一个错误。在上面这个例子中，I/O 模块将会产生一个对 `fcvt` 的弱引用，真正的浮点模块在库中跟在 I/O 模块后面，并且不再需要伪例程。现在如果有一个对 `f1tused` 的引用，则链接浮点例程并定义 `fcvt`。否则，对 `fcvt` 的引用保持未定义。这将不再依赖于库的顺序，即使对于对库进行多次扫描解析也没有问题。

ELF 还添加了另一种弱符号，和弱引用（weak reference）等价的弱定义（weak definition）。“弱定义”定义了一个没有有效的普通定义的全局符号。如果存在有效的普通定义，那么就忽略弱定义。弱定义并不经常使用，但在定义错误伪函数<sup>3</sup>而无须将其分散在独立的

---

2 译者注：这里的伪函数，也被称为桩函数，或桩子函数，可以认为是不进行任何工作的空函数。

3 译者注：这里的错误伪例程，可以理解为在程序中被用到，在调试模式下可以输出报错信息而在发布模式

模块中的时候，是很有用的。

## 练习

如果在不同库中的两个模块定义了相同的符号，链接器会怎么处理？这种情况是一种错误吗？

库的符号目录通常只包括被定义的全局符号。如果将未定义的符号也包括进来会有好处吗？

在使用 `lorder` 和 `tsort` 排序目标文件时，很有可能 `tsort` 不能够生成一个文件的全序排序。这种情况如果发生了，会是一个问题吗？

有一些库会将符号目录放在库的开头，也有另外一些库会将符号目录放在库的末尾。在实际中这会造成什么不同呢？

描述另外一些会使用弱外部引用和弱定义的情况。

## 项目

项目的这一部分为链接器增加了库的搜索功能。我们将对两种不同的库格式进行测试。第一种是本章早先所建议的类 IBM 目录格式。一个库就是一个目录，每个成员都是该目录下的一个文件，每个文件的（多个）名称都是该目录下文件要输出的符号的名称<sup>4</sup>。如果你使用的系统不支持 UNIX 风格的多名称，就造假的。给每个文件一个名字（从要输出的符号中选择一个）。然后制作一个名为 MAP 的文件，包含如下格式的行：

```
name sym  sym  sym ...
```

其中 `name` 是文件的名称，`sym` 是剩余要输出的符号。

第二种库格式是一个单独的文件，该库起始第一行如下：

```
LIBRARY      nnnn      pppppp
```

这里 `nnnn` 是库中模块的个数，`pppppp` 是文件中库目录起始位置偏移量。跟在后面的都是库成员，一个接着一个。从偏移量 `pppppp` 处开始的是库目录，它由多行构成，每个模块对应一行，格式如下：

```
pppppp  111111  sym1  sym2  sym3  ...
```

其中 `pppppp` 是模块在文件中的起始位置，`111111` 是模块的长度，`symi` 是定义在该模块中的符号。

**项目 6-1：**写一个库管理程序，可以根据一系列的目标文件创建目录格式的库。要确定对重复符号进行合理的处理。可选的，将这个库管理程序扩展为可以对已存在的库中的模块进行添加、替换、删除操作。

**项目 6-2：**将该链接器扩展为可以处理多个目录格式的库。当链接器遇到一个输入它的

---

下为空的例程。

<sup>4</sup> 译者注：在支持 UNIX 风格的多文件名情况下，一个文件可以具有多个名称。

输入文件列表中的库时，搜索该库并将其中定义了某个未定义符号的每个模块都包含进来。要确保对那些还依赖定义在其它库模块中定义的符号的库模块能够进行正确处理。

项目 6-3：写一个库管理程序，可以根据一系列的目标文件创建文件格式的库。注意除非你知道库中所有模块的大小，否则无法正确写入文件开头的 LIBRARY 行。一个可行的办法是先写入一个伪造行，待确定所有输入文件的大小并计算出尺寸后，或将整个都缓冲在内存后，再回过头来写入正确的数值。可选的，扩展这个库管理程序使其可以升级已存在的库，要注意这比升级目录格式的库要难得多。

项目 6-4：将该链接器扩展为可以处理多个文件格式的库。当链接器遇到属于它的输入文件列表中的库时，搜索该库并将其中定义了某个未定义符号的每个模块都包含进来。你必须修改你的读取目标文件的例程，以使它们可以从库文件的中间来读取一个目标模块。

# 第 7 章 重定位

\$Revision: 2.2 \$

\$Date: 1999/06/30 01:02:35 \$

为了决定段的大小、符号定义、符号引用，并指出包含那些库模块、将这些段放置在输出地址空间的什么地方，链接器会将所有的输入文件进行扫描。扫描完成后的下一步就是链接过程的核心，重定位。由于重定位过程的两个步骤，判断程序地址计算最初的非空段，和解析外部符号的引用，是依次、共同处理的，所以我们讲重定位即同时涉及这两个过程。

链接器的第一次扫描会列出各个段的位置，并收集程序中全局符号与段相关的值。一旦链接器确定了每一个段的位置，它需要修改所有的相关存储地址以反映这个段的新位置。在大多数体系结构中，数据中的地址是绝对的，那些嵌入到指令中的地址可能是绝对或者相对的。链接器因此需要对它们进行修改，我们稍后会讨论这个问题。

第一遍扫描也会建立第五章中所讲的全局符号表。链接器还会将符号表中的地址解析为引用全局符号时所有存储的地址。

## 硬件和软件重定位

由于几乎所有的现代计算机都具有硬件重定位，可能会有人疑问为什么链接器或加载器还需要进行软件重定位（当我于 60 年代后期在 PDP-6 上编程时，这个问题就困扰着我，而从那以后情况就变得更复杂了）。答案部分在于性能的考虑，部分在于绑定时间。

硬件重定位允许操作系统为每个进程从一个固定共知的位置开始分配独立的地址空间，这就是程序容易加载，并且可以避免在一个地址空间中的程序错误破坏其它地址空间中的程序。软件链接器或加载器重定位将输入文件合并为一个大文件以加载到硬件重定位提供的地址空间中，然后就根本不需要任何加载时的地址修改了。

在诸如 286 或 386 那样有几千个段的机器上，实际上有可能做到为每一个例程或全局数据分配一个段，独立的进行软件重定位。每一个例程或数据可以从各自段的 0 位置开始，所有的全局引用通过查找系统段表中的段间引用来处理并在程序运行时绑定。不幸的是，x86 段查找非常的慢，而且如果程序对每一个段间模块调用或全局数据引用都要都要进行段查找的话那速度要比传统程序慢的多。同样重要的时，虽然运行时绑定会对此有一些帮助（这是我们将在第 10 章涉及的话题），但大多数程序都没有采用（鉴于当前的硬件性能和容量对于程序运行都颇为富余）。由于可信的理由，程序文件最好绑定在一起并且在链接时确定地址，这样它们在调试时静止不变而出货后仍能保持一致性。当一个程序运行的库超出了作者预期的版本时，库二进制兼容是程序错误的一个长期并且难以发现的来源。（MS Windows 应用程序由于使用了大量的共享库，就倾向于存在这种问题。由于某些库的不同版本会因安装各种应用程序被加载到同一个计算机上）。即使不考虑 286 风格段的限制，动态连接比起静态连接而言也要慢的多，而且没有理由为不需要的东西付钱。

## 链接时重定位和加载时重定位

很多系统即执行链接时重定位，也执行加载时重定位。链接器将一系列的输入文件合并在一个准备加载到特定地址的单一输出文件。当这个程序被加载后，所存储的那个地址是无效的，加载器必须重新定位被加载得程序以反应实际的加载地址。在包括 MS-DOS 和 MVS 在内的一些系统上，每一个程序都按照加载到地址 0 的位置而被链接。实际的地址是根据有效地存储空间而定的，这个程序在被加载时总是会被重定位的。在其它的一些系统上，尤其是 MS Windows，程序按照被加载到一个固定有效地址的方式来链接，并且一般不会进行加载时重定位，除非发生该地址已被别的程序所占用之类的异常情况（当前版本的 Windows 实际上从不对可执行程序进行加载时重定位，但是对 DLL 共享库会进行重定位。相似的，UNIX 系统从不对 ELF 程序进行重定位，虽然它们对 ELF 共享库会进行重定位）。

加载时重定位和链接时重定位比起来就颇为简单了。在链接时，不同的地址需要根据段的大小和位置重定位为不同的位置。在加载时，整个程序在重定位过程中会被认为是大的单一段，加载器只需要判断名义上的加载地址和实际加载地址的差异即可。

## 符号和段重定位

链接器的第一遍扫描将各个段的位置列出，并收集程序中所有全局符号和段相关的值。一旦链接器决定了每一个段的位置，它就需要调整存储地址。

- 数据地址和段内绝对程序地址引用需要进行调整。例如，如果一个指针指向位置 100，但是段基址被重定位为 1000，那么这个指针就需要被调整到位置 1000。
- 程序中的段间引用也需要被调整。绝对地址引用要调整为可以反映目标地址段的新位置，同样相对地址需要调整为可以同时反映目标段和引用所在段的新位置。
- 对全局符号的引用需要进行解析。如果一个指令调用了例程 detonate，并且 detonate 位于起始地址为 1000 的段的偏移地址 500，在这个指令中涉及到的地址要调整为 1500。

重定位和符号解析所要求的条件有些许不同。对于重定位，基址的数量相当小，也就是一个输入文件中的段的个数，不过目标文件格式允许对任何段中任何地址的引用进行重定位。对于符号解析，符号的数量远远大的多，但是大多数情况下链接器只需要对符号做一件事即将符号的值插入到程序的一个字大小的空间中。

很多链接器将段重定位和符号重定位统一对待，这是因为它们将段当作是一种值为段基址的“伪符号”。这使得和段相关的重定位就成了和符号相关的重定位的特例。即使在将两种重定位统一对待的链接器中，此二者仍有一个重要区别：一个符号引用包括两个加数，即符号所在段的基值和符号在段内的偏移地址（译者注：这里作者少说了半句话，即将段作为符号处理时，这个特殊符号只有段基址，没有段内偏移量）。有一些链接器在开始进入重定位阶段之前就会预先计算所有的符号地址，将段基址加到符号表中符号的值中。当每一项被重定位时会查找到段基址并相加（译者注：即对符号地址中所包含段基址进行修改）。大多数情况下，并没有强制的理由要以这种或那种方法来进行这种操作。在少数链接器，尤其是那些针对实模式 x86 代码的链接器中，一个地址可以被重定位到和若干不同段相关的多个

地址上，因此链接器只需要确定在上下文中一个特定引用的符号在特定段中的地址。

## 符号查找

目标代码格式总是将每个文件中的符号当作数组对待，并在内部使用一个小整数指代符号，即数组的索引。这对链接器带来了一些小麻烦，就像第五章所讨论的，每一个输入文件均有不同的索引，如果输出文件是可以重链接的话那它们也会有不同的索引。最直截了当的解决办法是为每个输入文件保留一个指针数组，指向全局符号表中的表项。

## 基本的重定位技术

每一个可重定位的目标文件都含有一个重定位表，其中是在文件中各个段里需要被重定位的一系列地址。链接器读入段的内容，处理重定位项，然后再解决整个段，通常就是将它写入到输出文件中。通常而不总是，重定位是一次操作，处理后的结果文件不能被重定位第二次。但一些目标文件格式，尤其是 IBM 360 的，是可以重定位的并在输出文件中包含所有重定位信息（在 360 的情况下，输出文件在加载的时候需要被重定位，因此它必须包含所有的重定位信息）。对于 UNIX 链接器，有一个选项能产生可再次链接的输出文件，在某些情况下，尤其是共享库，由于它在加载时需要被重新定位因此总是带有重定位信息。

在最简单的情况下，如图 1，一个段的重定位信息仅是段中需要被重定位的位置列表。在链接器处理段时，它将段基址加上由重定位项标识的每个位置的地址。这就处理了直接寻址和内存中指向某个段的指针数值。

---

图 7-1：简单重定位项

address | address | address | ...

---

由于支持多个段和寻址模式的原因，在现代计算机上实际的程序会比这更复杂一些。经典的 UNIX a.out 格式，如图 2，可能是解决这些问题的最简单的实例。

---

图 7-2：a.out 重定位项

```
int address          /* 文本或数据段中的偏移量 */
unsigned int r_symbolnum : 24, /* 加到符号上的序数号 */
r_pcrel : 1,           /* 如果是指令相关的则为 1 */
r_length : 2,           /* 数值宽度的以 2 为底的 log 数 */
r_extern : 1,           /* 如果需要将符号加到数值上则为 1 */
```

---

每个目标文件都有两个重定位项集合，一个是文本段的，一个是数据段的（bss 段被定义为全 0，因此没有什么需要重定位的）。每一个重定位项都有标志位 r\_extern 指明它是段相关或者符号相关的项。如果该位为空，它是段相关的并且 r\_symbolnum 实际上是段的一个代码，可能是 N\_TEXT(4)，N\_DATA(6)，或者 N\_BBS(8)。pc\_relative 位指明该引用针对当

前位置（译者注：当前位置是指程序计数器，即指令指针寄存器而言）是绝对还是相对的。

每一个重定位项的其它多余信息是和它的类型及对应的段相关的。在下面的讨论中，TR，DR 和 BR 一次分别是文本段、数据段、BSS 段的重定位后基址。

对同一个段中的指针或直接地址，链接器将地址 TR 或 DR 加到段中已经保存的数值上。

对于从一个段到另一个段的指针或直接地址，链接器将目标段的重定位基址，TR，DR 或 BR，加到存储的数值上。由于 a.out 格式的输入文件中已经带有每一个重定位到新文件的段中的目标地址，这就是所有必须的了。例如，假定在输入文件中，文本从地址 0 开始，数据从地址 2000 开始，并且在文本段中的一个指针指向数据段中偏移量为 100 的位置。在输入文件中，被存储的指针的值为 2200（**为什么是 2200，我觉得应该是 2100**）。如果最后在输出文件中数据段的重定位位置为 15000，那么 DR 将为 13000，链接器将会把 13000 加入到已存在的 2200 产生最后的数值 15200。

一些体系结构的地址具有不同的尺寸。IBM 360 和 Intel 386 都具有 16 位和 32 位的地址，链接器一般都支持对这两种尺寸的重定位。确保程序地址满足十六位的限制是程序员自己的责任，链接器不会对地址有效性进行更多的确认。

## 指令重定位

由于多种指令格式的缘由，重定位指令中的地址要比重定位数据的指针麻烦一些。上面描述的 a.out 格式只有两个重定位格式，绝对的，与程序计数器相对的，但是大多数计算机体系结构需要更长的重定位格式以处理所有的指令格式。

### X86 指令重定位

不考虑 x86 指令的复杂编码方式，从链接器的角度看这种体系结构是易于处理的，因为它只需要处理两种地址，直接地址和与程序计数器相对的地址（我们在这里像大多数 32 位链接器那样忽略段）。引用数据的指令可以带有 32 位目标地址，链接器可以像其它 32 位地址那样对其进行重定位，加上目标所在段的段基址。

call 和 jump 指令使用相对寻址，因此指令中的地址是指令当前地址和目标地址的差值。对于相同段内的 call 和 jmp 指令，由于一个段内的相对地址是永不会改变的因此不需要进行重定位。对于段间 jump 链接器加上目标段重定位地址并减去指令段的地址。例如，对于从文本段到数据段的 jump，重定位值将为 DR-TR（译者注：原文这里说得比较含糊，我们只需要明白这里需要进行一个差值的转换即可）。

### SPARC 指令重定位

很少有体系结构能像 x86 那样提供对链接器方便的指令编码。例如 SPARC，没有直接寻址，有四种不同的分支指令格式，有一些专门用于合成 32 位地址的特殊指令，还有个别只包含部分地址的指令。链接器需要处理所有这些情况。

不像 x86 架构，没有一个 SPARC 指令的格式中为自己保留了一个 32 位地址的空间。这意味着在输入文件中，一个指令在内存中引用的重定位的目标地址不能通过存储在指令中。作为替代，如图 3，SPARC 的重定位项中有一个额外的域 r\_addend 包含了 32 位的引用地址。鉴于 SPARC 的重定位不能像 x86 的那样简单描述，一系列的类型标识位被标识重定位格式的

代码域 r\_type 替代。同样，不仅仅使用一个位去区分段或者符号重定位，每一个输入文件定义了符号 .text, .data 和 .bss，用来标识各自对应段的起始位置，并且段的重定位会涉及到这些符号。

---

图 7-3: SPARC 重定位项

```
int r_address;      /* 需重定位的数据的偏移量 */
int r_index:24,    /* 符号的符号表索引 */
r_type:8;          /* 重定位类型 */
int r_addend;      /* 数据加数 */
```

---

SPARC 重定位有 3 类：数据指针的绝对地址重定位，各种尺寸的分支和调用指令的相对地址重定位，和有点黑客味道的特殊的 SETHI 绝对地址重定位。绝对地址的重定位和 x86 上几乎一样，链接器将 TR, DR 或 BR 加到存储的数值上。这种情况下，由于在被存储的值中有足够的空间保存整个地址，因此重定位项中的加数实际上并不是必须的，但是链接器为了致性会将加数加到存储的值上（译者注：不使用重定位项中的加数的加法，和使用这个加数的加法，肯定加的数是不一样的）。

对于分支指令，鉴于加数就是距离目标的偏移量（即目标地址和存储的值之间的差值），因此存储的偏移量值通常是 0。链接器通过将适当的重定位数值加到这个加数上得到重定位的相对地址。鉴于 SPARC 的相对地址不保存低 2 位，还会将这个相对地址向右移 2 位，然后检查确认移位后的数值符合有效的位数（根据不同格式可能为 16 位、19 位、22 位或 30 位），通过位掩码取出移位后地址的有效位数来并将他们加到指令中。16 位格式的有效位数的低 14 位存储在指令的低 14 位中，但第 15 位和第 16 位却存储在指令的第 20 和 21 位中（译者注：从这里看，有效地址的最低位应该是从 1 开始数的）。链接器需要进行适当的位移和位掩码操作来存储这些有效位并不修改指令中的其它位。

特殊的 SETHI 黑客方法通过 SETHI 指令合成了一个 32 位地址，它从指令中获得 22 位的地址并将其放置在某个寄存器的高 22 位，然后接着通过一个 OR 操作将地址的低 10 位赋予相同的寄存器。链接器通过两种特殊的重定位模式来处理这种情况，其一将重定位地址（加数加上相应的重定位段基址）的高 22 位放置在存储值的高 22 位，其二将重定位地址的低 10 位放置在存储值的低 10 位。不像上面的分支模式那样，这些重定位模式不检查每一个值是否都是都能满足所存储的位数，因为两种模式下存储的位数都不能表示整个地址值。

在其它体系结构上的重定位会使用和 SPARC 不同的技术，包括对每一个可对内存寻址的指令格式采用不同的重定位类型。

## ECOFF 段重定位

Microsoft 的 COFF 目标文件格式是 COFF 格式（从 a.out 格式演变而来）的扩展版本，因此 Win32 的重定位和 a.out 的重定位有颇多相似行为也是不足为奇的。COFF 目标文件的每个段都有一个和 a.out 相似的重定位项列表，如图 4。COFF 重定位项有一个乖僻就是，即使

在 32 位机器上，它们也是 10 个字节长，这意味着在那些需要数据对齐的机器上，链接器不能在一次读操作中将整个重定位表加载到内存中的数组里，而需要连续完成读取和补齐两个操作（COFF 是很老的，那时每个重定位项节省 2 个字节还是很值得的）。在每一项中，地址均是所存储数据的相对虚拟地址（Relative Virtual Address），索引是段或者符号索引，类型是机器特定的重定位类型。对于输入文件的每一个段，符号表中包含一个像 .text 这样名字的项，这就可以使用这个符号的索引对相应的目标段进行重定位了。

---

图 7-4: MS COFF 重定位项

```
int address; /* 需重定位的数据的偏移量 */
int index; /* 符号索引 */
short type; /* 重定位类型 */
```

---

在 x86 平台上，ECOFF 重定位所作的工作和 a.out 中的非常相似。符号 IMAGE\_REL\_I386\_DIR32 是一个 32 位的直接地址或存储的指针，符号 IMAGE\_REL\_I386\_DIR32NB 是和程序计数器（指令指针寄存器）的 32 位相对地址。还有一些其它重定位类型对特殊的 Windows 特性提供支持特殊，这个我们稍后会涉及到。

ECOFF 支持包括 MIPS 在内的一些 RISC 处理器和 Power PC。这些处理器都存在和 SPARC 相同的重定位问题，地址受限的分支指令，多个指令序列合成一个直接地址。除了通常的全字重定位类型外，ECOFF 还具有处理这些特殊情况的各种重定位类型。

以 MIPS 为例，有一个跳转指令，它包含的 26 位地址向左移动 2 位保存在程序计数器（译者注：即指令指针寄存器）的低 28 位中，保持高 4 位不变。重定位类型 IMAGE\_REL\_MIPS\_JMPADDR 对一个分支指令的目标地址进行重定位。由于存储的指令中已经保存了未重定位的目标地址，因此没有在重定位项中保存这个目标地址。为了进行重定位，链接器不得不提取出保存的指令中的低 26 位，将其进行位移和掩码操作，然后将其加上重定位的目标段基址，来重新构建未重定位的目标地址，再反向进行位移和位掩码操作以恢复原先的指令。这个过程中，链接器还要检查目标地址对于当前指令是否可达。MIPS 还有一个和 SETHI 类似的窍门。MIPS 指令可以保存 16 位数值。如果要加载一个强制的 32 位数，可以先使用一个 LUI（Load Upper Immediate）指令将一个立即数的高 16 位存储在某个寄存器的高 16 位，然后紧接着一个 ORI（OR immediate）指令将立即数的低 16 位放置到这个寄存器的低 16 位。s 重定位类型 IMAGE\_REL\_MIPS\_REFHI 和 IMAGE\_REL\_MIPS\_REFLO 可以支持这个技巧，告诉链接器分别对重定位的指令中目标值的高 16 位或低 16 位进行重定位。但 REFHI 存在一个问题。想象一下重定位前的目标地址为 16 进制的 00123456，所以保存的指令中将包含未重定位值的高 16 位 0012。现在想想重定位值为 1E000。那最后的值将是 123456 加上 1E000 为 141456，所以存储的值将变为 0014。但是等等，如果要做这个运算，链接器需要完整的值 00123456，但是只有 0012 存储在指令中。它从那里找到低 16 位的数值呢？ECOFF 格式的答案是 REFHI 后面的重定位项是 IMAGE\_REL\_MIPS\_PAIR，这里保存着前面 REFHI 的目标地址的低 16 位。相比较于在每一个重定位项中增加一个额外的域保存多出来的加数，这是一个可论证的更优方法，因为 PAIR 项仅仅会出现在 REFHI 之后，这比在每一个重定位项中浪费空间要好的多。不利之处在于重定位项的前后顺序现在变得很重要了，而以前不是这样。

## ELF 重定位

ELF 的重定位与 a.out 和 COFF 相近。ELF 将带加数和不带加数的重定位问题合理化了，它有两种重定位段，是不带加数的 SHT\_REL 和带加数的 SHT\_REL\_A。实际上一个单独的文件中所有的重定位段具有相同的类型，它依赖于目标体系结构。如果目标体系结构像 x86 那样在目标代码中为加数留有空间，它就使用 REL 类型，否则就使用 RELA 类型。原则上编译器可以在那些需要加数的体系结构上通过使用空的重定位加数的方法节省些许空间，例如，将过程引用放置在 SHT\_REL 类型段中，其余的放在 SHT\_REL\_A 类型段中。

ELF 同样增加了一些额外的重定位类型来处理动态连接和位置无关代码，我们将在第 8 章讨论。s

## OMF 重定位

OMF 重定位和我们前面已经看到的方法在概念上是相同的，但是细节要颇为复杂一些。由于 OMF 原本是在内存和存储空间受限的微型计算机上使用的，这种格式允许在整个段没有被加载到内存中时就可以进行重定位。OMF 通过 FIXUPP 重定位记录将 LDATA 和 LEDATA 类型的数据混合起来，每一 FIXUPP 记录对应于它前面的数据。因此，链接器可以读取和缓冲一个数据记录，并读取其后的 FIXUPP 记录，然后实施重定位，并将重定位后的数据输出。FIXUPPS 是重定位时的线索，有一个两位的代码间接的涉及到活动框（译者注：这是在 DOS 编程时可以接触到的一个概念，经常会涉及到地址的切换，在编程之道一书中，也提到过的），即 OMF 的重定位基地址。链接器必须跟踪 4 个活动框，根据 FIXUPP 记录的新定义更新它们，按照 FIXUPP 记录的指示使用它们。

## 可重链接和重定位的输出格式

有一小部分格式是可以重链接的，即输出文件带有符号表和重定位信息，这样可以作为下一次链接的输入文件来使用。很多格式是可以重定位的，这意味着输出文件保存有供加载时重定位使用的重定位信息。

对于可重链接文件，链接器需要从输入文件的重定位项中建立输出文件的重定位项。有一些重定位项被原样传递给输出了，有一些被修改了，还有一些被忽略了。对于那些不在相连段中且段相对地址固定的重定位项，通常会直接传递给输出而不需要对段索引进行修改，这是因为最终链接器还会对其进行链接。而在那些段相连格式中的重定位项，每一项的偏移量需要修改。例如，在一个被链接的 a.out 格式文件中，有一个位于某个文本段中偏移量为 400 的段相对地址重定位项，如果另一个段与它所在的段相连且重定位在地址 3500 处，那么这个重定位项就要被修改为 3900 而不是 400。

符号解析项可以不加修改的传递，或因为段重定位而被修改，或被忽略。如果一个外部符号仍未被定义，那么链接器会传递这个重定位项给输出，可能会为了反映链接的段而修改偏移量和符号索引，以及输出文件符号表中的符号顺序。若这个符号被链接器根据符号引

用的细节而解析。如果这个引用是同一个段中的程序计数器相对地址，鉴于引用的相对地址和目标不会移动，故链接器可以忽略掉它的重定位项。如果这个引用是绝对引用或段间引用，那重定位项就是相对于段的。

对于可以重定位但不能重链接的输出格式，链接器忽略掉除相对段地址固定的以外所有的重定位项。

## 其它重定位格式

虽然多数重定位项的普遍格式是数组，但也有别的可能，包括链表和位图。多数格式也具有需要被链接器特殊对待的段。

### 以链表形式组织的引用

对于外部符号引用，一种意料之外的有效格式是在目标文件自身中包含的引用链表。符号表项指向一个引用，对应位置的一个字（译者注：即 2 个字节）宽的数据指向后面的另一个引用，一直延伸下去直到遇到诸如空或者 -1 这样的截止符。这种方法在那些地址引用是完全一个字宽的体系结构上有效，或者至少引用地址宽度足以表示目标文件中段的最大尺寸（以 SPARC 的分支指令为例，它的偏移地址为 22 位宽，由于指令地址是按照四个字界对齐的，因此足够覆盖 2 的 24 次方字节长的段，这个长度限制对于文件中的一个段而言是合理的）。

但这个技巧不能解决带偏移量的符号引用，对于代码引用这个限制通常是可以接受的，但是对于数据引用就有问题了。例如在 C 语言中，可以写一个指向数组中间的被初始化的静态指针：

```
extern int a[];
static int *ap = &a[3];
```

在 32 位的机器上，ap 的内容是 a 加上 12（译者注：这里的问题作者没有明确的指出，我猜测这里的错误在于被初始化的静态变量会被放置在数据段中，而且由于它是局部静态变量因此偏移地址很可能不会使用“全尺寸”的数值，这样当它的初始化值不在当前文件中，那么该初始化值很可能就会超出这个静态变量指针所能表示的地址范围，这种情况下，问题就出现了。个人认为这种情况在保护模式的扁平线性寻址中可能不会出现，但是在非扁平的段式寻址模式中可能会发生）。和此问题差不多的还有对数据指针使用这种方法，或对无偏移量引用的普通情况使用了链表，或对带偏移量引用其它处理方式。

### 以位图形式组织的引用

对于像 PDP-11，Z8000 和一些采取绝对寻址的 DSP 这样的体系结构上，由于大量的内存引用指令包含需要被重定位的地址，因此代码段最终会被进行多次的段重定位。相比于使用一个链表去维护需要修改的地址，使用位图来存储更为有效，一个位代表一个字宽的空间，

如果这个位置需要被修改那么对应的位被置位。对于 16 位的体系结构，当一个段中多于 1/16 的字的空间需要进行重定位那么位图就可以节省空间；对于 32 位体系结构，则多于 1/32 的字的空间需要重定位时可节省空间（译者注：这个结论是和指针的宽度有关系的）。

## 特殊段

很多目标文件格式定义了一些特殊重定位过程所需要的特殊段：

- Windows 目标文件具有 TLS ( Thread Local Storage ) 段，这个特殊的段保存着一个进程中每个线程启动时需要复制的全局变量。
- IBM 360 的目标文件中具有“伪寄存器集”，它和 TLS 相似，是被那些不同输入文件中的命名子块引用的区域。
- 不少 RISC 体系结构定义了可以被收集到一个区域中的“small”段，通过程序启动时设置一个寄存器指向这个区域，允许在程序中的任何地方是进行直接寻址。

在上面这些情况中，链接器都需要一种或两种个数的重定位类型来处理特殊的段。

对于 Windows 的 TLS，重定位类型的细节依体系结构而有所不同。对于 x86，IMAGE\_REL\_I386\_SECREL fixup ( 译者注：这个 fixup 实在是不知道应该怎么样翻译，总之是一个名词 ) 保存着目标符号相对于它所在段开始位置的偏移量。这个 fixup 通常是一个带有在运行时指向当前线程 TLS 的索引寄存器的指令，所以 SECREL 可以提供 TLS 中的偏移量。对于 MIPS 和其它 RISC 处理器，SECREL fixup 存储的 32 位值，与 SECRELLO 和 SECRELHI ( 像 REFHI 那样后面跟一个 PAIR ) 都可以生成段相对地址。

对于 IBM 的伪寄存器集，目标代码格式增加了两种重定位类型。一种是 PR 伪寄存器引用，它将伪寄存器的偏移量存储在一个 load 或 store 指令的 2 个字节中。另一个是 CXD，是程序中所使用的伪寄存器的总尺寸。这个数值用来在代码启动时确定需要给伪寄存器集分配多少存储空间。

对于 small 数据段，目标文件格式为基于 Alpha 平台的 MIPS 或 LITERAL 系统定义了诸如 GPREL ( 全局指针重定位 ) 的重定位类型，以存储 small 数据段中相对于目标数据的偏移量。链接器定义了诸如 \_GP 的符号作为 small 数据段的基址，这样运行时的启动代码可以将指针加载到一个固定的寄存器指定的区域。

## 特殊情况的重定位

很多目标文件格式都有“弱”外部符号：如果输入文件碰巧定义了它的话，那么它就会被当作是普通的全局符号，否则就为空（细节请参看第 5 章）。无论是那种方式，都会像其它符号那样进行引用解析。

一些更老的目标文件格式允许比我们讨论过的格式更为复杂的重定位。例如在 IBM 360 的格式中，每一个重定位项可以加上或减去它所引用的地址，多个重定位项可以修改相同的位置，允许诸如 A-B ( A 减去 B，这里 A、B 既可某一个为外部符号，也可以同为外部符号 ) 这样的引用。一些更老的链接器允许强制的复杂重定位，通过精心预留的修正字串来表示需

要被解析并存储在程序内存中的链接时表达式。虽然这些方案都有强大的表达能力，但是过于强大到没有太多用处，因此现代链接器的重定位方案都退回到采用带有可选偏移量的引用。

## 练习

为什么 SPARC 链接器在重定位分支地址时会检查地址溢出，但是在处理 SETHI 序列中高部分和低部分时没有检查？

在 MIPS 的例子中，一个 REFHI 重定位项需要跟着一个 PAIR 项，但是 REFL0 不需要，为什么呢？

对于伪处理器集和 TLS 的符号引用被解析为相对于段开始地址的偏移量，而普通的符号引用被解析为绝对地址，为什么？

我们说过 a.out 和 COFF 重定位不能处理诸如 A-B（A 和 B 同为全局符号）的引用。你能提出一种方法仿造它吗？

## 项目

回忆如下格式的重定位格式：

loc seg ref type ...

loc 是要被重定位的位置，seg 是该位置所在的段，ref 是该位置所引用的段或符号，type 是重定位类型。我们具体定义了以下这些重定位类型：

- A4 绝对引用。loc 的四个字节是对段 ref 的绝对引用。
- R4 相对引用。loc 的四个字节是对段 ref 的相对引用。即 loc 中的字节为 loc 随后的地址 ( $loc+4$ ) 和目标地址之间的差值（这是 x86 相对跳转指令的格式）。
- AS4 绝对符号引用。loc 的四个字节是对符号 ref 的绝对引用，加数是已经存储在 loc 中的值（通常加数为 0）。
- RS4 相对符号引用。loc 的四个字节是对符号 ref 的相对引用，加数是已经存储在 loc 中的值（通常加数为 0）。
- U2 上半部引用。loc 中的两个字节是对符号 ref 的引用地址的高两个字节。
- L2 低半部引用。loc 中的两个字节是对符号 ref 的引用地址的低两个字节。

项目 7-1：让链接器处理这些重定位类型。在链接器创建了符号表并为所有的段和符号赋予地址后，处理每一个输出文件中的重定位项。别忘了重定位的定义是影响目标代码中的实际地址数值，而不是十六进制的表示。如果你用 perl 写自己的链接器，那么使用 perl 的 pack 功能将目标代码中的段转换为二进制字串，进行重定位后再使用 perl 的 unpack 功能将其转换回十六进制表示是最容易的。

项目 7-2：当你在处理项目 7-1 中的重定位时，你会采用哪一种字节序？修改你的链接器以采用另外一种字节序。

# 第8章 加载和重叠

\$Revision: 2.3 \$

\$Date: 1999/06/15 03:30:36 \$

加载是将一个程序放到主存里使其能运行的过程。这一章我们看看加载过程，并将注意力集中在加载那些已经连接好的程序。很多系统曾经都有过将连接和加载合为一体的链接加载器，但是现在除了我知道的运行 MVS 的硬件和第十章将会谈到的动态链接器外，其它的实际上已经基本消失了。链接加载器和单纯的加载器没有太大的区别，主要和最明显的区别在于前者的输出放在内存中而不是在文件中。

## 基本加载

在第三章的目标文件设计中，我们已经接触了大多数加载的基本知识。依赖于程序是通过虚拟内存系统被映射到进程地址空间，还是通过普通的 I/O 调用读入，加载会有一点小小的差别。

在多数现代系统中，每一个程序被加载到一个新的地址空间，这就意味着所有的程序都被加载到一个已知的固定地址，并可以从这个地址被链接。这种情况下，加载是颇为简单的：

- 从目标文件中读取足够的头部信息，找出需要多少地址空间。
- 分配地址空间，如果目标代码的格式有独立的段，那么就将地址空间按独立的段划分。
- 将程序读入地址空间的段中。
- 将程序末尾的 bss 段空间填充为 0，如果虚拟内存系统不自动这么做的话。
- 如果体系结构需要的话，创建一个堆栈段 (stack segment)。
- 设置诸如程序参数和环境变量的其他运行时信息。
- 开始运行程序。

如果程序不是通过虚拟内存系统映射的，读取目标文件就意味着通过普通的 read 系统调用读取文件。在支持共享只读代码段的系统上，系统检查是否在内存中已经加载了该代码段的一个拷贝，而不是生成另外一份拷贝。

在进行内存映射的系统上，这个过程会稍稍复杂一些。系统加载器需要创建段，然后以页对齐的方式将文件页映射到段中，并赋予适当的权限，只读 (R0) 或写时复制 (COW)。在某些情况下，相同的页会被映射两次，一个在一个段的末尾，另一个在下一个段的开头，分别被赋予 R0 和 COW 权限，格式上类似于紧凑的 UNIX a.out。由于数据段通常是和 bss 段是紧挨着的，所以加载器会将数据段所占最后一页中数据段结尾以后的部分填充为 0 (鉴于磁盘版本通常会有一些符号之类的东西在那里)，然后在数据分配足够的空页面覆盖 bss 段。

## 带重定位的基本加载

仅有一小部分系统还仍然为执行程序在加载时进行重定位，大多数都是为共享库在加载时进行重定位。诸如 MS-DOS 的系统，很少使用硬件的重定位；另外一些如 MVS 的系统，具有硬件重定位（却是从一个没有硬件重定位的系统继承来的）；还有一些系统，具有硬件重定位，但是却可以将多个可执行程序和共享库加载到相同的地址空间。所以链接器不能指望某些特定地址是有效的。

如第七章讨论的，加载时重定位要比链接时重定位简单的多，因为整个程序作为一个单元进行重定位。例如，如果一个程序被链接为从位置 0 开始，但是实际上被加载到位置 15 000，那么需要所有程序中的空间都要被修正为“加上 15000”。在将程序读入主存后，加载器根据目标文件中的重定位项，并将重定位项指向的内存位置进行修改。

加载时重定位会表现出性能的问题，由于在每一个地址空间内的修正值均不同，所以被加载到不同虚拟地址的代码通常不能在地址空间之间共享。MVS 使用的，并被 Windows 和 AIX 扩展的一种方法，使创建一个出现在多个地址空间的共享内存区域，并将常用的程序加载到其中（MVS 将其称为 link pack 区域）。这仍然存在普通进程不能获取可写数据的单独副本的问题，所以应用程序必须在编写时明确地为它可写区域分配空间。

## 位置无关代码

对于将相同程序加载到普通地址的问题的一个常用的解决方案就是位置无关代码 (position independent code, PIC)。他的思想很简单，就是将数据和普通代码中那些不会因为被加载的地址改变而变化的代码分离出来。这种方法中代码可以在所有进程间共享，只有数据页为各进程自己私有。

这是一个令人吃惊的老想法。TSS/360 在 1966 年就使用它了，并且我相信它也不是最早采用该方法的（TSS 有很多臭名昭著的 bug，但是从我个人经验而言，他的 PIC 特性的确可以工作）。

在现代体系结构中，生成 PIC 可执行代码并不困难。跳转和分支代码通常是位置相关的，或者与某一个运行时设置的基址寄存器相关，所以需要对他们进行非运行时的重定位。问题在于数据的寻址，代码无法获取任何的直接数据地址。由于代码是可重定位的，而数据不是位置无关的。普通的解决方案是在数据页中建立一个数据地址的表格，并在一个寄存器中保存这个表的地址，这样代码可以使用相对于寄存器中地址的被索引地址来获取数据。这种方式的成本在于对每一个数据引用需要进行一次额外的重定位，但是还存在一个问题就是如何获取保存到寄存器中去的初始地址。

## TSS/360 位置无关代码

TSS 采用了一种非常残暴的方法。每一个例程具有两个地址，称为 V-con (V style address constant 的缩写) 和数据的地址，称为 R-con。标准的 OS/360 调用序列要求调用者提供由寄存器 13 指向的一个 18 字节大小的寄存器保存区域。TSS 将这个保存区域扩展为 19 个字，并要求调用者在进行调用前需将它的 R-con 放置到第 19 个字中，如图 1。每一个例程在

自己的数据段中有所有他要调用的历程的 V-con 和 R-con，并在调用前将对应的 R-con 放置在的即将使用的保存区域。提供初始 R-con 的程序主例程从操作系统那里获取一个保存区域。

图 8-1: TSS 风格的不同地址空间的 2 过程调用

R-con 在保存区域的 TSS 风格

调用者:

将 R-con 复制到保存区域

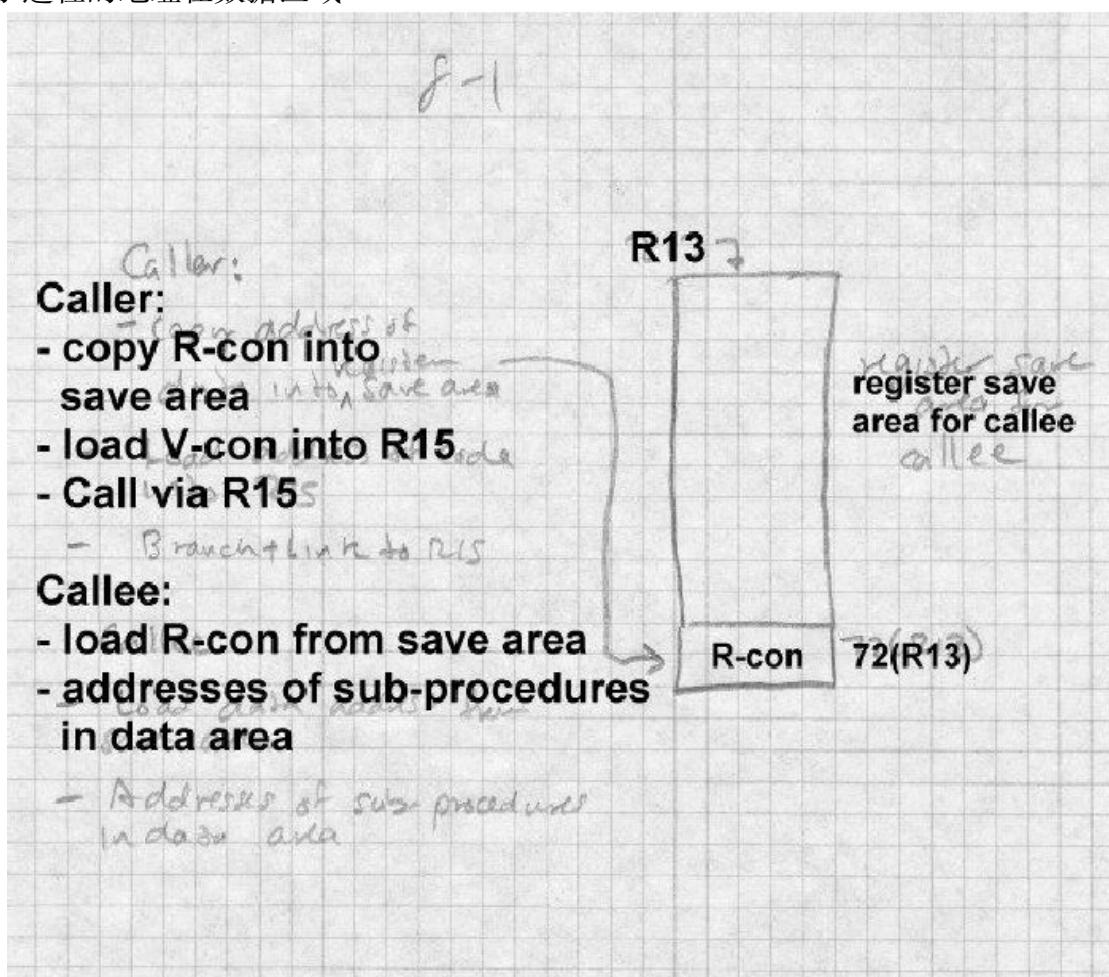
将 V-con 加载到寄存器 R15

通过寄存器 R15 进行调用

被调用者:

从保存区域加载 R-con

子过程的地址在数据区域



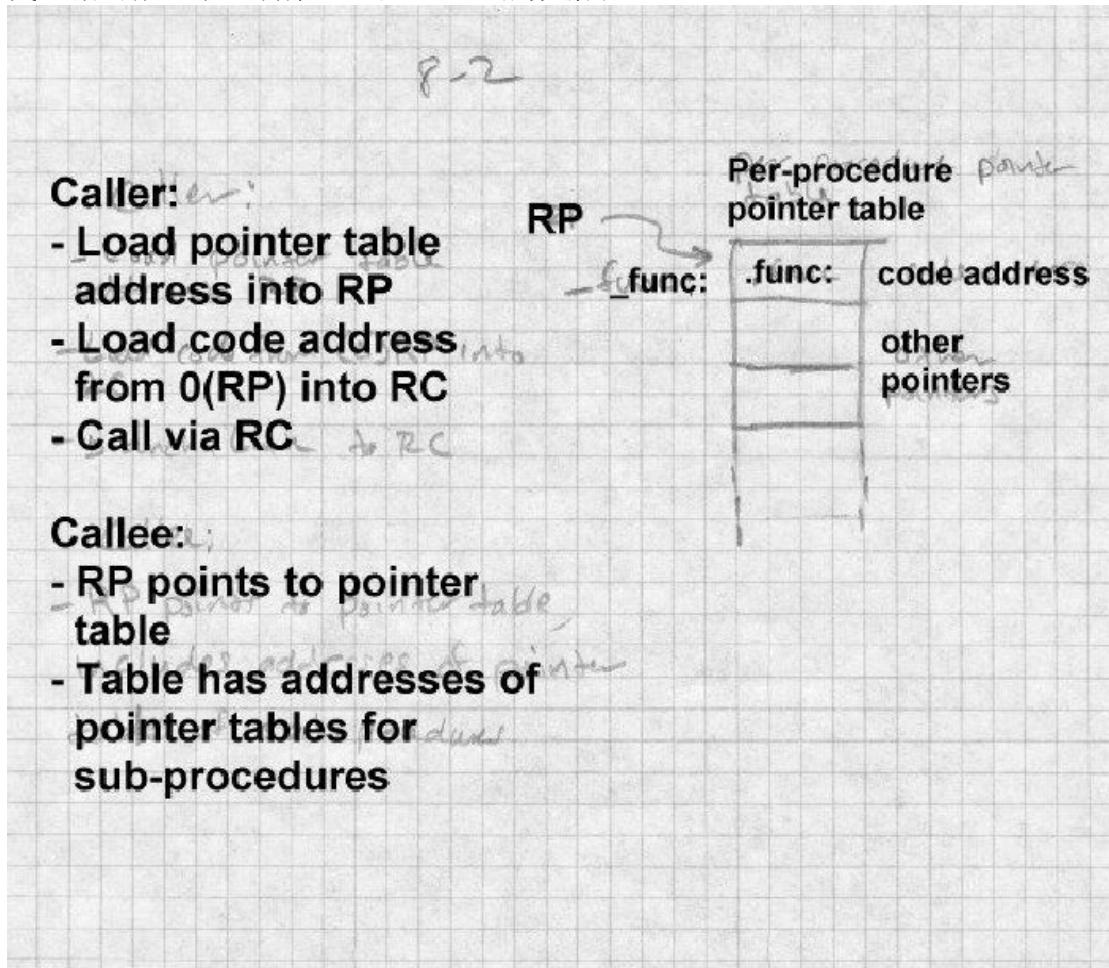
这种方案是可行的，但是对于现代系统却不是很合适。一个原因，复制 R-con 使得调用过程较为庞大，另一个原因，它使得过程的指针为 2 个字长，这在 1960 年代没有关系，但是现在大多数程序都是使用 C 写的，要求所有的指针需要相同的尺寸，这就有问题了（C 标准并没有强制要求所有指针长度一致，但是目前非常多的程序都是基于这种假设的）。

## 例程指针表

在许多 UNIX 系统中采用的一种简单修改是将一个过程的数据地址假当作这个过程的地址，并在这个地址上放置一个指向该过程代码的指针，如图 2。如要调用一个过程，调用者就将该例程的数据地址加载到约定好的数据指针寄存器，然后从数据指针指向的位置中加载代码地址到一个寄存器，然后调用这个历程。这很容易实现，而且性能还算不错。

图 8-2：通过数据指针调用的代码

[代码指针放置在开始位置的 ROMP 风格数据表]



## 目录表

IBM AIX 使用了这种方案的改良版本。AIX 程序将多个例程组成模块，模块就是使用单独的或一组相关的 C/C++ 源代码文件生成的目标代码。每个模块的数据段保存着一个目录表 (Table Of Content, TOC)，该表是由模块中所有例程和这些例程的小的静态数据的指针组成的。寄存器 2 通常用来保存当前模块的 TOC 地址，在 TOC 中允许直接访问静态数据，并可通过 TOC 中保存的指针间接访问代码和数据。由于调用者和被调用者共享相同的 TOC，因此

在一个模块内的调用就是一个简单的 call 指令。模块之间的调用必须在调用之前切换 TOC，调用后再切换回去。

编译器将所有的调用都生成为 call 指令，其后还紧跟一个占位操作指令 no-op，对于模块内调用这是正确的。当链接器遇到一个模块间调用时，他会在模块文本段的末尾生成一个称为 global linkage 或 glink 的例程。Glink 将调用者的 TOC 保存在栈中，然后从调用者的 TOC 中指针中加载被调用者的 TOC 和各种地址，然后跳转到要调用的例程。链接器将每一个模块间调用都重定向为针对被调用历程的 glink，并将其后的空操作指令修改为从栈中恢复 TOC 的加载指令。过程的指针都变为 TOC/代码配对 (TOC/code pair) 的指针，所有通过指针的 call 都会借助一个使用了该指针指向的 TOC 和代码地址的普通 glink 例程。

这种方案使得模块内调用尽可能的快。模块间调用由于借助了 glink 所以会稍微慢一些，但是比起我们接下来要看到的其它替代方案来，这种速度的降低是很小的。

## ELF 位置无关代码

UNIX SVR4 为它的 ELF 共享库引入了一个类似于 TOC 的位置无关代码 (PIC) 方案。SVR4 方案现在被使用 ELF 可执行程序的系统广泛支持，如图 3。它的优势在于将过程调用恢复为普通方式，即一个过程的地址就是这个过程的代码地址，不管它是存在于 ELF 库中的 PIC 代码，或存在于普通 ELF 可执行文件中的非 PIC 代码，付出的代价就是这种方案比 TOC 的开销稍多一些。

ELF 的设计者注意到一个 ELF 可执行程序中的代码页组跟在数据页组后面，不论程序被加载到地址空间的什么位置，代码到数据的偏移量是不变的。所以如果代码可以将他自己的地址加载到一个寄存器中，数据将位于相对于代码地址确定的位置，并且程序可以通过相对于某一个固定偏移量的基址寻址方式有效的引用自己数据段的数据。

链接器将可执行文件中寻址的所有全局变量的指针保存在它创建的全局偏移量表 (Global Offset Table, GOT) 中（每一个共享库拥有自己的 GOT，如果主程序和 PIC 代码一起编译，它也会有一个 GOT，虽然通常不这么做）。鉴于链接器创建了 GOT，所以对于每个 ELF 可执行程序的数据只有一个地址，而不论在该可执行程序中有多少个例程引用了它。

如果一个过程需要引用全局或静态数据，那就需要过程自己加载 GOT 的地址。虽然具体细节随体系结构不同而有所变化，但 386 的代码是比较典型的：

```
call .L2 ;; push PC in on the stack
.L2:
popl %ebx ;; PC into register EBX
addl $_GLOBAL_OFFSET_TABLE_+[.-.L2], %ebx;; adjust ebx to GOT address
```

它存在一个对后面紧接着位置的 call 指令，这可以将 PC(译者注：程序计数器，即指令指针寄存器)压入栈中而不用跳转，然后用 pop 指令将保存的 PC 加载到一个寄存器中并立刻加上 call 的目标地址和 GOT 地址之间的差。在一个由编译器生成的目标文件中，专门有一个针对 addl 指令操作数的 R\_386\_GOTPC 重定位项。它告诉链接器替换从当前指令到 GOT 基地址的偏移量，同时也是告诉链接器在输出文件中建立 GOT 的一个标记。在输出文件中，由

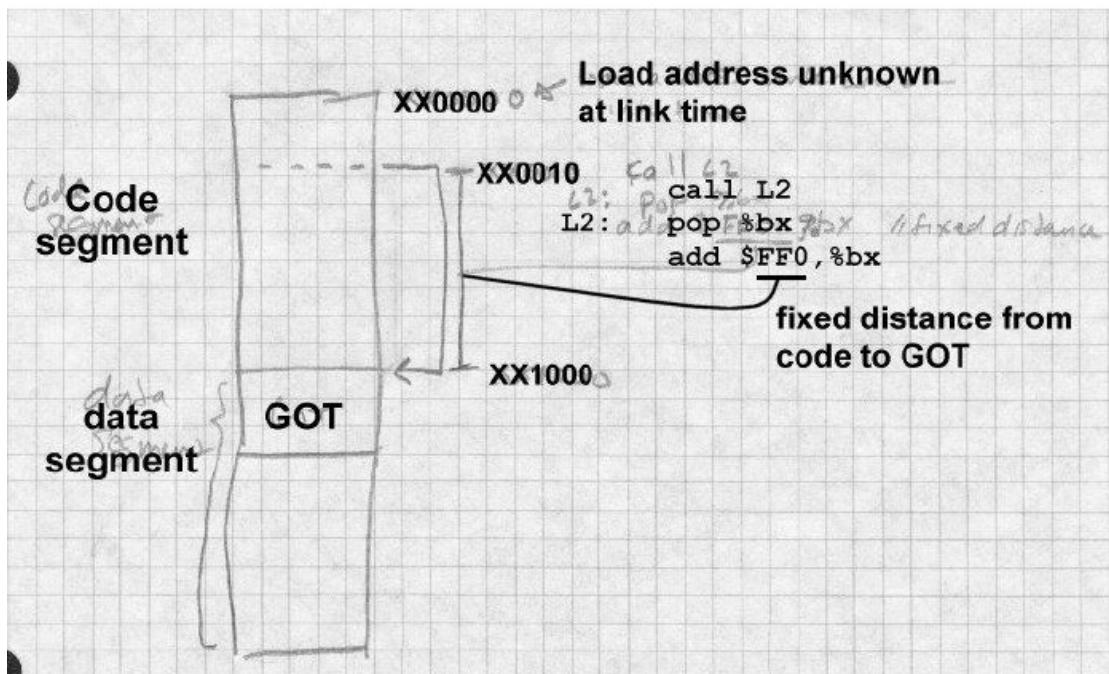
于 addl 到 GOT 之间的距离是固定的，所以就不再需要重定位了。

(译者注：上面这段代码是比较典型的，主要目的是获取 GOT 的地址，保存在 ebx 中，为以后访问程序的全局/局部变量作准备。\_GLOBAL\_OFFSET\_TABLE 是链接器可以理解的一个量，在链接的时候链接器会将它替换为当前指令地址到 GOT 基地址之间的距离差值。由于在引用这个量的时候，ebx 中的地址是 call 指令行的地址，不是 addl 指令行的地址，所以 ebx 在加上\_GLOBAL\_OFFSET\_TABLE 之后，还要加上 addl 指令行到 call 指令行的距离[. - L2]，才能够调整为 GOT 的基地址。这个细节原文中没有提到)

图 8-3：具有固定偏移量的位置无关代码/数据(PIC code and data)

Figure 8-3: PIC code and data with fixed offsets

下图展示了即使程序被加载到普通地址空间的不同地址，代码页对与数据仍具有不变的偏移量。



GOT 寄存器被加载之后，程序数据段中的静态数据与 GOT 直接的距离在链接时被固定了，所以代码就可以将 GOT 寄存器作为一个基址寄存器来引用局部静态数据。全局数据的地址只有在程序被加载后才被确定(参看第 10 章)，所以为了引用全局数据，代码必须从 GOT 中加载数据的指针，然后引用这个指针。这个多余的内存引用使得程序稍微慢了一些，尽管大多数程序员为了方面的使用动态链接库愿意付出这个代价。对速度要求较高的代码可以使用静态共享库(参看第 9 章)或者根本不使用共享库。

为了支持位置无关代码(PIC)，ELF 还定义了 R\_386\_GOTPC(或与之等价的标识)之外的一些特殊重定位类型代码。这些类型是体系结构相关的，但是 x86 下的是比较典型的：

- R\_386\_GOT32: GOT 中槽位(slot)的相对位置，链接器在这里存放了对于给定符号的指针。用来标识被引用的全局变量。
- R\_386\_GOTOFF: 给定符号或地址相对于 GOT 基地址的距离。用来相对于 GOT 对静态数据进行寻址。

- R\_386\_RELATIVE: 用来标记那些在 PIC 共享库中并在加载时需要重定位的数据地址。例如，参看下列 C 代码片断：

```
static int a; /* static variable */  
extern int b; /* global variable */  
  
...  
a = 1; b= 2;
```

变量 a 被分配在目标文件的 bss 段，这意味着它与 GOT 之间的距离是固定可知的。目标代码可以用 ebx 作为基址寄存器并结合一个与 GOT 的相对偏移量直接引用这个变量：

```
movl $1, a@GOTOFF(%ebx);; R_386_GOTOFF reference to variable "a"
```

变量 b 是全局的，如果他在不同的 ELF 库或可执行文件中，那么它的位置只有在运行时才能知道。这种情况下，目标代码引用一个链接器在 GOT 中创建的指向 b 的指针：

```
movl b@GOT(%ebx), %eax;; R_386_GOT32 ref to address of variable "b"  
movl $2, (%eax)
```

注意编译器仅创建一个 R\_386\_GOT32 引用，需要链接器收集所有类似的引用并为他们在 GOT 中创建槽位(slot)。

最终，ELF 共享库保存了若干供运行时加载器（我们在第 10 章将要讨论的动态链接器的一部分）进行运行时重定位的 R\_386\_RELATIVE 重定位项。由于共享库中的文本总是位置无关代码，所以对于代码没有重定位项，但数据不是位置无关的，所以对于数据段的每一个指针都有一个重定位项（实际上你也可以创建没有位置无关代码的共享库，这种情况下同样会存在文本的重定位项，但由于这样将使文本无法共享所以几乎没有人这么做）。

## 位置无关代码的开销和得益

PIC 的得益是明显的：它使得不需加载时重定位即可加载代码成为可能；可以在进程间共享代码的内存页面，即使它们没有被分配到相同的地址空间中。可能的不利之处就是在加载时、在过程调用中以及在函数开始和结束时会降低速度，并使全部代码变得更慢。

在加载时，虽然一个位置无关代码文件的代码段不需要被重定位，但是数据段需要。在一个大的库中，TOC 或 GOT 可能会非常大以至于要花费很长的时间去解析其中的所有项。这同样是一个我们将在第 10 章动态链接中要讨论的一个问题。处理同一个可执行文件中的 R\_386\_RELATIVE (或等价符号) 来重定位 GOT 中的数据指针是相当快的，但是问题是很多 GOT 项中的指针指向别的可执行文件并需要查找符号表来解析。

在 ELF 可执行文件中的调用通常都是动态链接的，甚至于在相同库内部的调用，这就增加了明显的开销。我们将在第 10 章再次看到这个问题。

在 ELF 文件中函数的开始和结束是相当慢的。他们必须保存和恢复 GOT 寄存器，在 x86 中就是 ebx，并且通过 call 和 pop 将程序计数器保存到一个寄存器中也是很慢的。从性能的观点来看，AIX 使用的 TOC 方法更好，因为每一个过程可以假定它的 TOC 寄存器已经在过程项中设置了。

最后，PIC 代码要比非 PIC 代码更大、更慢。到底会有多慢很大程度上依赖于体系结构。

对于拥有大量寄存器且无法直接寻址的 RISC 系统来说，少一个用作 TOC 或 GOT 指针的寄存器影响并不明显，并且缺少直接寻址而需要的一些排序时间是不变的。最坏的情况是在 x86 下。它只有 6 个寄存器，所以用一个寄存器当作 GOT 指针对代码的影响非常大。由于 x86 可以直接寻址，一个对外部数据的引用在非 PIC 代码下可以是一个简单的 MOV 或 ADD，但在 PIC 代码下就要变成加载紧跟在 MOV 或 ADD 后面的地址，这既增加了额外的内存引用又占用了宝贵的寄存器作为临时指针。

特别在 x86 系统上，对于速度要求严格的任务，PIC 代码的性能降低是明显的，以至于某些系统对于共享库退而采用一种类似 PIC 的方法。

## 自举加载

在这里讨论加载都有一个前提就是计算机系统中已经存在一个操作系统或至少有一个程序加载器在运行并负责程序的加载。这些一个被另一个加载的程序链总得有一个开始的地方吧，所以就有一个显而易见的问题即最初的程序时如何被加载到计算机中去的。

在现代计算机中，计算机在硬件复位后运行的第一个程序总是存储在称为 bootstrap ROM 的随机只读存储器中。就像自己启动自己一样。当处理器上电或者复位后，它将寄存器复位为一致的状态。例如在 x86 系统中，复位序列跳转到系统地址空间顶部下面的 16 字节处。Bootstrap ROM 占用了地址空间顶端的 64K，然后这里的 ROM 代码就来启动计算机。在 IBM 兼容的 x86 系统上，引导 ROM 代码读取软盘上的第一个块，如果失败的话就读取硬盘上的第一个块，将它放置在内存位置 0，然后再跳转到位置 0。在第 0 块上的程序然后从磁盘上一个已知位置上加载另一个稍微大一些的操作系统引导程序到内存中，然后在跳转到这个程序，加载并运行操作系统（可能存在更多的步骤，例如引导管理器需要决定从那个分区上读取操作系统的引导程序，但加载器的主要功能是不变的）。

为什么不直接加载操作系统？因为你无法将一个操作系统的引导程序放置在 512 个字节内。第一级引导程序只能从被引导磁盘的顶级目录中加载一个名字固定且大小不超过一个段的程序。操作系统引导程序具有更多的复杂代码如读取和解释配置文件，解压缩一个压缩的操作系统内核，寻址大量内存（在 x86 系统上的引导程序通常运行在实模式下，这意味着寻址 1MB 以上地址是比较复杂的）。完全的操作系统还要运行在虚拟内存系统上，可以加载需要的驱动程序，并运行用户级程序。

很多 UNIX 系统使用一个近似的自举进程来运行用户态程序。内核创建一个进程，在其中装填一个只有几十个字节长度的小程序。然后这个小程序调用一个系统调用运行 /etc/init 程序，这个用户模式的初始化程序然后依次运行系统所需要的各种配置文件，启动服务进程和登录程序。

这些对于应用级程序员没有什么影响，但是如果你想编写运行在机器裸设备上的程序时就变得有趣多了，因为你需要截取自举过程并运行自己的程序，而不是像通常那样依靠操作系统。一些系统很容易实现这一点（例如只需要在 AUTOEXEC.BAT 中写入你要运行的程序名字，再重新启动 Windows 95），另外一些系统则几乎是不可能的。它同样也给定制系统提供了机会。例如可以通过将应用程序的名字改为 /etc/init 基于 UNIX 内核构建单应用程序系统。

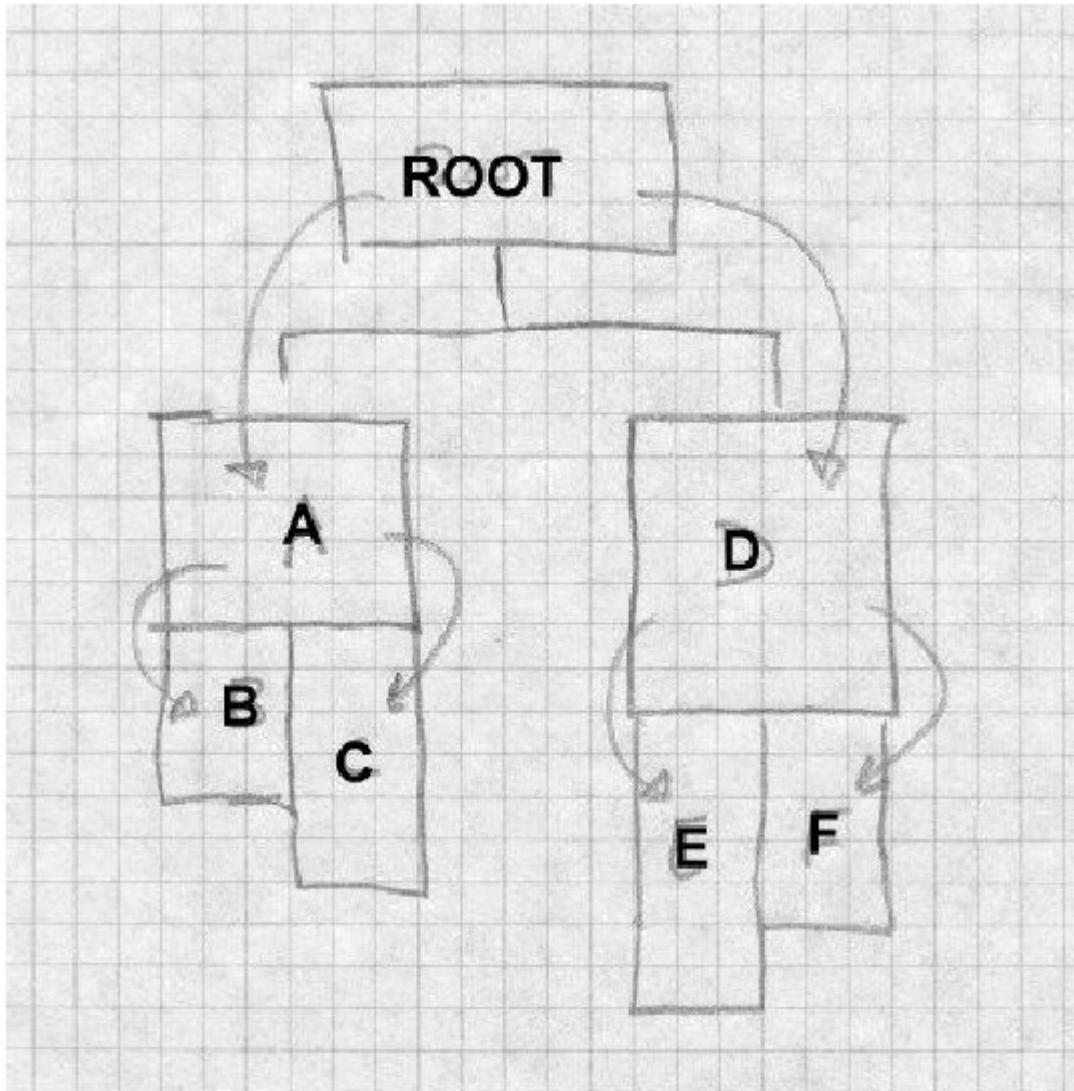
## 树状结构的覆盖

我们通过一个对树状结构的覆盖来结束这一章，这是在虚拟内存出现之前让程序运行在比自己小的内存中而广泛使用的一种方案。覆盖是另一个使用在 1960 年代以前的技术，并且在一些内存限制严格的系统中仍然使用。1980 年时就有一些 MS-DOS 链接器像 25 年前就采用此技术的大型主机计算机那样支持树状结构覆盖。虽然现在常用的系统上已经很少使用覆盖技术了，这种链接器用以创建和管理覆盖的技术仍然很有趣。同样，为覆盖开发的段间调用技巧也为动态链接库提供了思路。对于类似仅有很小程序地址空间的 DSP 的环境，覆盖技术仍然是加载程序的一个好办法，尤其是覆盖管理器很小。OS/360 的覆盖管理器仅有 500 字节大小，有一次我为一个具有 512 个字大小地址空间的图形处理器编写的覆盖管理器仅用了几十个字的空间。

采用覆盖技术的程序将代码分为若干个段组成的树，如图 4 所示的这个。

图 8-4：一个典型的覆盖树结构

根(ROOT)调用 A 和 D。A 调用 B 和 C，D 调用 E 和 F。



程序员手工的将目标文件或单个的目标代码分配个多个覆盖段。覆盖树中的兄弟段分

享相同的内存。例如，段 A 和 D 分享相同的内存，B 和 C 共享相同的内存，E 和 F 分享相同的内存。到达一个特定段所经过的各个段的序列称为一个路径（path），所以 E 的路径包括根（root），D 和 E。

当程序启动后，系统加载包含程序入口点的根（root）段。每次当一个例程产生一次“向下（downward）”的段间调用时，覆盖管理器要确保被调用目标的路径都被加载。例如，如果根（root）调用了段 A 中的一个例程，如果段 A 没有在内存中，那么覆盖管理器就要加载段 A。如果 A 中的一个例程调用了 B 中的一个例程，管理器就要确保 A 和 B 的路径都被加载了。向上的调用由于从根（root）到当前点的路径都已被加载到内存中了，所以不需要链接器的任何帮助。

跨越树的调用被称为“独占调用（exclusive call）”，由于它不可能返回所以通常被认为是一种错误。覆盖链接器允许程序员在自己很清楚调用例程不需要返回的情况下强制使用独占调用。

## 定义覆盖

覆盖链接器从普通的输入目标文件创建支持覆盖的可执行程序。目标代码不包含任何的覆盖指令，而是由程序员使用一种由链接器读取和解释的命令语言来指明覆盖结构。图 5 展示了与前面相同的覆盖结构，包括加载到每一个段中的例程名字。

---

图 8-5：一个典型的覆盖树结构

根（ROOT）中的 rob、rick 调用 A（aaron、andy）和 D

A 调用 B（bill、betty）和 C（chris），D（dick、dot）调用 E（edgar）和 F（fran）

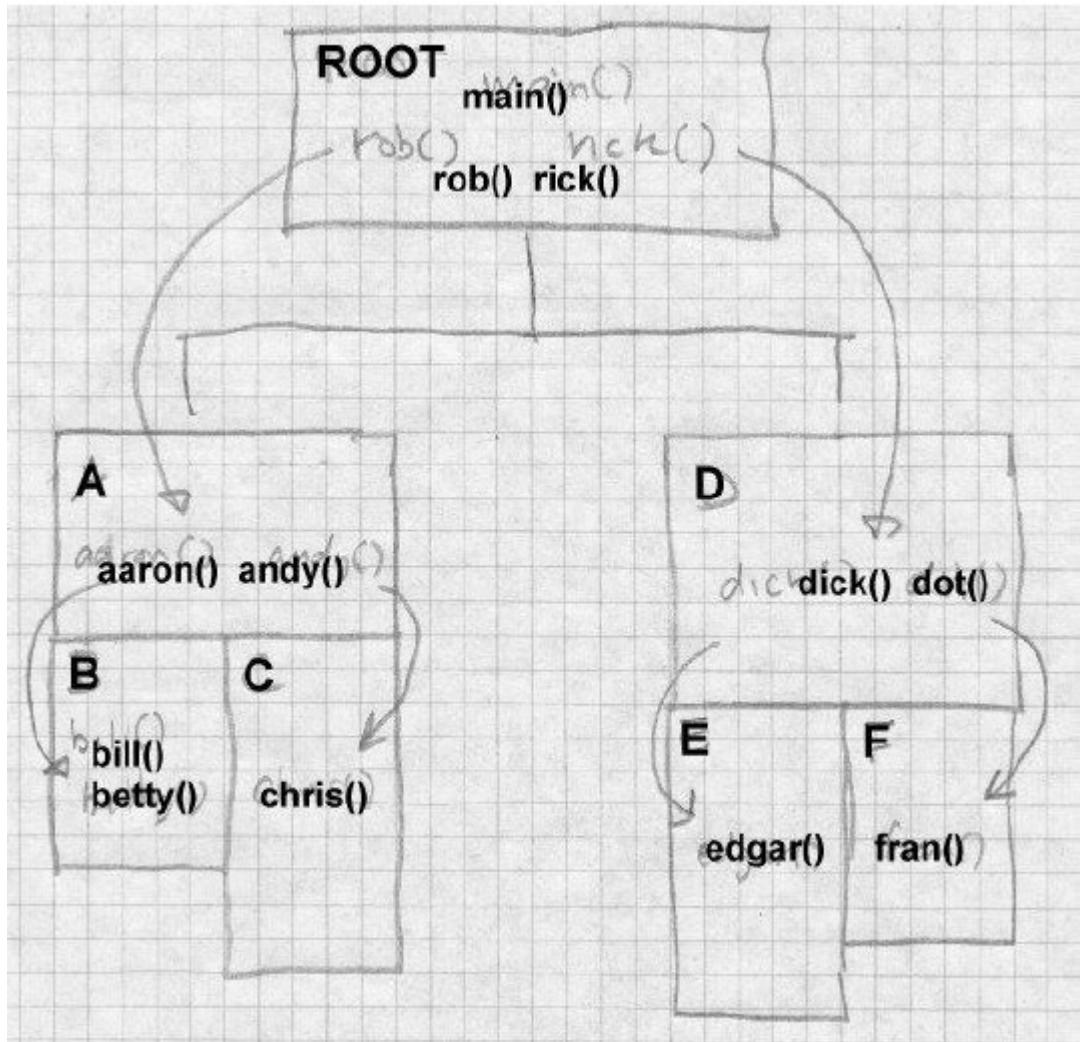


图 6 展示了可以告诉 IBM 360 链接器以创建这种结构的链接器命令。空格并不碍事，所以我们对命令进行了缩进以对应这种树状结构。OVERLAY 命令定义了每一个段的开头；相同覆盖名的段定义了覆盖彼此的段。所以第一个 OVERLAY AD 定义了段 A，第二个 OVERLAY AD 定义了段 D。覆盖段按照从左到右的深度优先顺序在树中进行定义。INCLUDE 命令为链接器要读取的逻辑文件命名。

(译者注：这里定义覆盖的命令顺序实际上就是从左到右以深度优先顺序遍历覆盖树的顺序)

图 8—6：链接器命令

```

INCLUDE ROB
INCLUDE RICK
OVERLAY AD
INCLUDE AARON, ANDY
OVERLAY BC
INCLUDE BILL, BETTY
OVERLAY BC
INCLUDE CHRIS

```

```
OVERLAY AD
INCLUDE DICK,  DOT
OVERLAY EF
INCLUDE EDGAR
OVERLAY EF
INCLUDE FRAN
```

---

通过覆盖有效的利用空间是程序员的责任。为每一个段分配的空间是分享这个空间的任意段中最长的那个。例如，假设下面这些以十进制表示长度的文件们：

名字	长度
rob	500
rick	1500
aaron	3000
andy	1000
bill	1000
betty	1000
chris	3000
dick	3000
dot	4000
edgar	2000
fran	3000

存储空间的分配，如图 7 所示。每一个段的开始位置紧跟着路径中的前一个段，程序总的大小由最长的路径决定。这个程序的覆盖结构是比较平衡的，其中最长的路径是 11500  
(译者注：图中实际是 12000)，最短的是 8000。在保持有效（没有独占调用）的前提下将覆盖结构调整尽可能紧凑和高效，是一个需要大量检验、调错工作的“魔法”。由于覆盖是完全在链接器中处理定义的，每次调试所进行的重新链接并不需要重新编译。

---

图 8-7：覆盖存储布局

```
0 rob
500 rick

2000 aaron          2000 dick
5000 andy           5000 dot

6000 bill      6000 chris
7000 betty     9000 ----   9000 edgar   9000 fran
8000 ----           11000 ----   12000 ----
```

---

## 覆盖的实现

覆盖的实现是惊人的简单。由于链接器决定了各个段的布局，每一个段中代码的重定位适当地基于段在内存中分配的位置。链接器需要在根（root）段中创建一个段表，然后在每一个段中，将作为向下调用目标的例程和代码胶合起来。

段表如图 8 所示，其中列出了每一个段、标识该段是否被加载的标志、段的路径，以及从磁盘加载段时所需的信息。

---

图 8-8：理想的段表

```
struct segtab {  
    struct segtab *path; // 路径中的前一个段  
    boolean ispresent; // 如果该段已被加载则为真  
    int memoffset; // 相对的加载地址  
    int diskoffset; // 在可执行程序中的位置  
    int size; // 段大小  
} segtab[];
```

---

链接器在每一个向下调用前提取胶合代码，这样使得覆盖加载器能够确保加载需要的段。段可以在较高级别的例程中使用胶合代码，但是不能在较低级别的例程中使用胶合代码。例如，如果根（root）中的例程调用 arron, dick 和 betty，根（root）就需要这三个符号的胶合代码。如果段 A 包含对 bill、betty 和 chris 的调用，则 A 需要 bill 和 chris 的胶合代码，但是可以直接使用已经存在与根（root）中的 betty 的胶合代码。所有的向下调用（针对全局符号的）都需要通过胶合代码解析，如图 9 所示，而不是直接调用真正的例程。由于胶合代码对于调用者和被调用者都是透明的，所以它需要保存任何会被它修改的寄存器，然后跳转到覆盖管理器，提供真正例程的地址并指明该例程所在的段。这里我们使用了一个指针，但是使用段表 segtab 中的一个索引也是可行的。

---

图 8-9：x86 下理想的胶合代码

```
glue' betty: call load_overlay  
.long betty // 实际例程的地址  
.long segtab+N // 段 B 在段表 segtab 中的地址
```

---

在运行时，系统加载根（root）段并运行它。对于每一个向下调用，胶合代码都会调用覆盖管理器。管理器查看目标段的状态。如果段当前在内存中，则管理器跳转到实际的例程。如果段当前不在内存中，则管理器加载目标段和其它在路径上的未加载段，将任何冲突的段都标记为当前不在内存中，并将刚加载的段标记为当前在内存中，然后再调转到对应地址。

## 覆盖的其它细节

细节总使得优雅的覆盖树结构比他们应该具有的样子要杂乱一些。

### 数据

我们已经讨论了结构化代码的覆盖，而没有讨论任何关于数据的问题。每一个例程可能都具有需要一同加载到段中去的自己的私有数据，但是任何从一个调用到另一个调用过程中要被记住的数据都需要被推动到树的足够高的层次，以确保数据不会被错误的卸载或重复加载，否则将会丢失掉执行中所进行的修改。在实际中，这意味着多数全局变量都记录在根（root）中。当在 Fortran 程序中使用覆盖时，覆盖链接器可以正确的为那些用作通讯区域的普通块定位。例如，如果 dick 调用 edgar 和 fran，而后两个例程都引用了一个普通块，那么这个块可以作为一个通讯区域保存在段 D 中。

### 复制的代码

一个采用覆盖技术程序的覆盖结构经常可以通过复制代码予以改善。可以想象在我们的例子中，chris 和 edgar 都调用一个长度为 500 字节称为 greg 的例程。那么在根（root）中就必须存在一个 greg 的副本，因为在树的其它任何地方放置 greg 都会在 chris 或 edgar 调用它的时候导致独占调用（exclusive call），这就增加了被加载程序的总尺寸。另一方面来看，如果段 C 和 E 都包含 greg 的副本，那么被加载程序的总尺寸将不变，因为段 C 的结尾会从 9000 增长到 9500，段 E 的结尾会从 11000 增长到 11500，这都小于段 F 需要的 12000。

### 多区域

一个程序的调用结构经常不能很好的映射到单一的树。所以覆盖系统通过对每一个区域建立一个树来处理多区域代码。区域之间的调用都是通过胶合代码进行。IBM 链接器总共支持 4 个区域，但在我的经验中还没有发现需要用到 2 个以上区域的情况。

## 覆盖技术小结

虽然由于虚拟内存系统，覆盖技术已经几乎被淘汰了，他们仍然具有重要的历史意义，因为它第一次使用了链接时的代码生成和代码修改。它需要程序员大量的手工作业以设计和指定覆盖结构，通常都伴随着大量对“digital origami”错误的检查和调错工作。但是它是一种非常有效的将大程序加载到空间受限内存中去的方法。

覆盖促使了链接器中非常重要的“包装”call 指令的技术的产生，这可以让一个简单的过程调用做更多的工作，例如加载需要的覆盖段。链接器在多种方法中使用了包裹技术。最重要的是我们将在第 10 章涉及到的动态连接，链接一个尚未被加载的库中的被调用例程。在测试和调试中包裹也是非常有用的，例如在可疑例程前面插入检测/验证代码，这就不需要改变或重新编译源文件了。

## 练习

使用位置无关代码（PIC）和非位置无关代码（non-PIC）编译一些小的C例程。位置无关代码要比非位置无关代码慢多少？是否速度足够慢而值得我们在程序中使用非位置相关版本的库？

在覆盖的例子中，假设 dick 和 dot 各自调用 edgar 和 fran，但是 dick 和 dot 彼此不掉用。重新分配覆盖的结构，使得 dick 和 dot 分享相同的空间，并调整结构分布使得调用树仍然可以工作。现在这个使用覆盖的程序需要多大的空间？

在覆盖段表中，没有针对冲突段的明确的标记。当覆盖管理器加载一个段或者段路径时，管理器如何决定将哪些段标记为当前不在内存中呢？

在一个没有独占调用的采用覆盖技术的程序中，是否有可能因为经过一系列的调用而跳转到没有被加载的代码处而导致程序结束呢？在上面的例子中，如果 rob 调用 bill，bill 调用 aaron，aaron 调用 chris，然后所有的例程都返回，会发生什么？要让链接器或者覆盖管理器探测或避免这个问题会有多困难？（译者：对于这个问题的翻译，我不能很确定，目前是尽我自己的理解来翻译了）

## 项目

项目 8-1：为链接器增加一个“包裹”例程的特性。建立一个链接器选项

`-w name`

对给定的例程进行包裹。将所有程序中引用到的给定名字的例程都改变为引用 wrap\_name（确认不要漏掉定义了这个名字的段内部的引用）。将原先例程的名字改为 real\_name。这可以使程序员编写一个调用 wrap\_name 的包裹例程，该例程也可以将 real\_name 作为原先的例程来调用。

项目 8-2：开始第 3 章里的链接器框架，编写一个可以修改目标文件以包裹某个名字的工具。即，对改名子的引用都转为引用外部符号 wrap\_name，而现存的例程被重命名为 real\_name。为什么有人想要使用这样一种工具而不是将这个特性加入到链接器中去呢（提示：要考虑你不是链接器的作者或者维护者的情况）。

项目 8-3：使链接器支持可以使用位置无关代码（PIC）生成可执行程序的功能。我们增加了一些 4 字节的重定位类型：

```
loc seg ref GA4  
loc seg ref GP4  
loc seg ref GR4  
loc seg ref ER4
```

这些类型是：

- GA4：（GOT address）位于位置 loc，保存着到 GOT 的距离。
- GP4：（GOT pointer）放置一个指向 GOT 中符号 reg 的指针，位于位置 loc，保存 GOT 相对于这个指针偏移量。
- GR4：（GOT relative）位置 loc 保存着段 reg 中的一个地址。使用从 GOT 开始位置到该地址的偏移量来替换这个地址。
- ER4：（Executable relative）位置 loc 保存着一个相对于该可执行程序起始位置

的相对地址。忽略 **reg** 域。

在你的链接器的第一遍扫描中，查找 GP4 重定位项，建立一个包含所有被请求的指针的 GOT 段，并将 GOT 段分配在数据段和 BSS 段之前。第二遍扫描时，处理 GA4，GP4 和 GR4 项。在输出文件中，如果输出文件会被加载到它名义地址以外的位置则还要为任何需要重定位的数据建立 ER4 重定位项。这将包含输入中任何被 A4 或者 AS4 标记的重定位项（提示：不要忘记 GOT）。

# 第 9 章 共享库

\$Revision: 2.3 \$

\$Date: 1999/06/15 03:30:36 \$

程序库的产生可以追溯到计算技术的最早期，因为程序员很快就意识到通过重用程序的代码片段可以节省大量的时间和精力。随着如 Fortran 和 COBOL 等语言编译器的发展，程序库成为编程的一部分。当程序调用一个标准过程时，如 `sqrt()`，编译过的语言显式地使用库，而且它们也隐式地使用用于 I/O、转换、排序及很多其它复杂得不能用内联代码解释的函数库。随着语言变得更为复杂，库也相应地变复杂了。当我在 20 年前写一个 Fortran 77 编译器时，运行库就已经比编译器本身的工作要多了，而一个 Fortran 77 库远比一个 C++ 库要来得简单。

语言库的增加意味着：不但所有的程序包含库代码，而且大部分程序包含许多相同的库代码。例如，每个 C 程序都要使用系统调用库，几乎所有的 C 程序都使用标准 I/O 库例程，如 `printf`，而且很多使用了别的通用库，如 `math`, `networking`, 及其它通用函数。这就意味着在一个有一千个编译过的程序的 UNIX 系统中，就有将近一千份 `printf` 的拷贝。如果所有那些程序能共享一份它们用到的库例程的拷贝，对磁盘空间的节省是可观的。（在一个没有共享库的 UNIX 系统上，单 `printf` 的拷贝就有 5 到 10M。）更重要的是，运行中的程序如能共享单个在内存中的库的拷贝，这对主存的节省是相当可观的，不但节省内存，也提高页交换。

所有共享库基本上以相同的方式工作。在链接时，链接器搜索整个库以找到用于解决那些未定义的外部符号的模块。但链接器不把模块内容拷贝到输出文件中，而是标记模块来自的库名，同时在可执行文件中放一个库的列表。当程序被装载时，启动代码找到那些库，并在程序开始前把它们映射到程序的地址空间，如图 1。标准操作系统的文件映射机制自动共享那些以只读或写时拷贝的映射页。负责映射的启动代码可能是在操作系统中，或在可执行体，或在已经映射到进程地址空间的特定动态链接器中，或是这三者的某种并集。

---

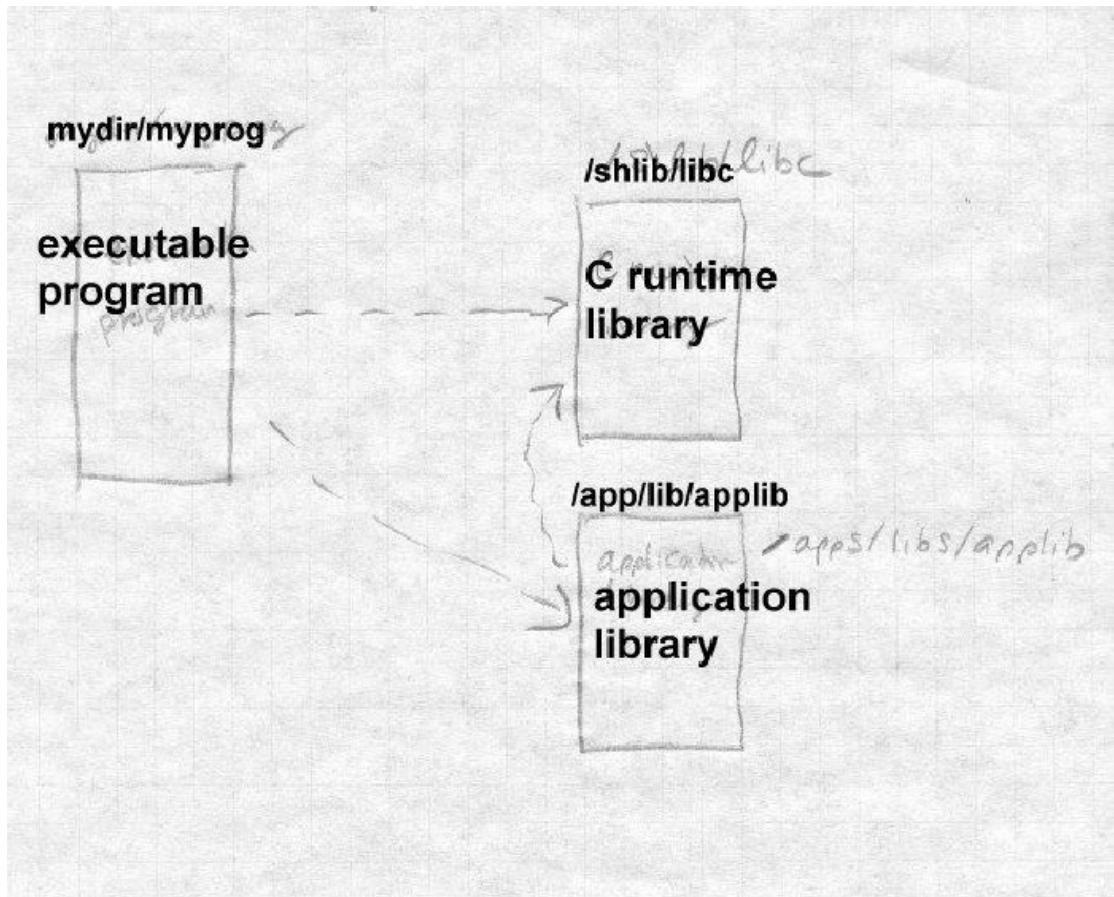
图 9-1：带有共享库的程序

可执行程序，共享库的图例

可执行程序 `main`, `app` 库, C 库

不同位置来的文件

箭头展示了从 `main` 到 `app`, `main` 到 C, `app` 到 C 的引用



在本章，我们着眼于静态链接库，也就是库中的程序和数据地址在链接时绑定到可执行体中。在下一章我们着眼于更复杂的动态链接库。尽管动态链接更灵活更“现代”，但也比静态链接要慢很多，因为在链接时要做的大量工作在每次启动动态链接的程序时要重新做。同时，动态链接的程序通常使用额外的“胶合(glue)”代码来调用共享库中的例程。胶合代码通常包含若干个跳转，这会明显地减慢调用速度。在同时支持静态和动态共享库的系统上，除非程序需要动态链接的额外扩展性，不然使用静态链接库能使它们更快更小巧。

## 绑定时间

共享库提出的绑定时间问题，是常规链接的程序不会遇到的。一个用到了共享库的程序在运行时依赖于这些库的有效性。当所需的库不存在时，就会发生错误。在这情况下，除了打印出一个晦涩的错误信息并退出外，不会有更多的事情要做。

当库已经存在，但是自从程序链接以来库已经改变了时，一个更有趣的问题就会发生。在一个常规链接的程序中，在链接时符号就被绑定到地址上而库代码就已经绑定到可执行体中了，所以程序所链接的库是那个忽略了随后变更的库。对于静态共享库，符号在链接时被绑定到地址上，而库代码要直到运行时才被绑定到可执行体上。（对于动态共享库而言，它们都推迟到运行时。）

一个静态链接共享库不能改变太多，以防破坏它所绑定到的程序。因为例程的地址和库中的数据都已经绑定到程序中了，任何对这些地址的改变都将导致灾难。

如果不改变程序所依赖的静态库中的任何地址，那么有时一个共享库就可以在不影响

程序对它调用的前提下进行升级。这就是通常用于小 bug 修复的“小更新版”。更大的改变不可避免地要改变程序地址，这就意味着一个系统要么需要多个版本的库，要么迫使程序员在每次改变库时都重新链接它们所有的程序。实际中，永远不变的解决办法就是多版本，因为磁盘空间便宜，而要找到每个会用到共享库可执行体几乎是不可能的。

## 实际的共享库

本章余下的部分将关注于 UNIX System V Release 3.2 (COFF 格式)，较早的 Linux 系统 (a.out 格式)，和 4.4BSD 的派生系统 (a.out 和 ELF 格式) 这三者提供的静态共享库。这三者以几近相同的方式工作，但有些不同点具有启发意义。SVR3.2 的实现要求改变链接器以支持共享库搜索，并需要操作系统的强力支持以满足例程在运行时的启动需求。Linux 的实现需要对链接器进行一点小的调整并增加一个系统调用以辅助库映射。BSD/OS 的实现不对链接器或操作系统作任何改变，它使用一个脚本为链接器提供必要的参数和一个修改过的标准 C 库启动例程以映射到库中。

## 地址空间管理

共享库中最困难的就是地址空间管理。每一个共享库在使用它的程序里都占用一段固定的地址空间。不同的库，如果能够被使用在同一个程序中，它们还必须使用互不重叠的地址空间。虽然机械的检查库的地址空间是否重叠是可能的，但是给不同的库赋予相应的地址空间仍然是一种“魔法”。一方面，你还想在它们之间留一些余地，这样当其中某个新版本的库增长了一些时，它不会延伸到下一个库的空间而发生冲突。另一方面，你还想将你最常用的库尽可能紧密的放在一起以节省需要的页表数量（要知道在 x86 上，进程地址空间的每一个 4MB 的块都有一个对应的二级表）。

每个系统的共享库地址空间都必然有一个主表，库从离应用程序很远的地址空间开始。Linux 从十六进制的 60000000 开始，BSD/OS 从 A0000000 开始。商业厂家将会为厂家提供的库、用户和第三方库进一步细分地址空间，比如对 BSD/OS，用户和第三方库开始于地址 A0800000。

通常库的代码和数据地址都会被明确的定义，其中数据区域从代码区域结束地址后的一个或两个页对齐的地方开始。由于一般都不会更新数据区域的布局，而只是增加或者更改代码区域，所以这样就使小更新版本成为可能。

每一个共享库都会输出符号，包括代码和数据，而且如果这个库依赖于别的库，那么通常也会引入符号。虽然以某种偶然的顺序将例程连接为一个共享库也能使用，但是真正的库使用一些分配地址的原则而使得更容易，或者至少使在更新库的时候不必修改输出符号的地址成为可能。对于代码地址，库中有一个可以跳转到所有例程的跳转指令表，并将这些跳转的地址作为相应例程的地址输出，而不是输出这些例程的实际地址。所有跳转指令的大小都是相同的，所以跳转表的地址很容易计算，并且只要表中不在库更新时加入或删除表项，那么这些地址将不会随版本而改变。每一个例程多出一条跳转指令不会明显的降低速度，由

于实际的例程地址是不可见的，所以即使新版本与旧版本的例程大小和地址都不一样，库的新旧版本仍然是可兼容的。

对于输出数据，情况就要复杂一些，因为没有一种像对代码地址那样的简单方法来增加一个间接层。实际中的输出数据一般是很少变动的、尺寸已知的表，例如 C 标准 I/O 库中的 FILE 结构，或者像 errno 那样的单字数值（最近一次系统调用返回的错误代码），或者是 tzname（指向当前时区名称的两个字符串的指针）。建立共享库的程序员可以收集到这些输出数据并放置在数据段的开头，使它们位于每个例程中所使用的匿名数据的前面，这样使得这些输出地址在库更新时不太可能不会有变化。

## 共享库的结构

共享库是一个包含所有准备被映射的库代码和数据的可执行格式文件，见图9-2。

---

图 9-2：典型共享库的结构

文件头, a.out, COFF 或 ELF 头

(初始化例程, 不总存在)

跳转表

代码

全局数据

私有数据

---

一些共享库从一个小的自举例程开始，来映射库的剩余部分。之后是跳转表，如果它不是库的第一个内容，那么就把它对齐到下一个页的位置。库中每一个输出的公共例程的地址就是跳转表的表项；跟在跳转表后面的是文本段的剩余部分（由于跳转表是可执行代码，所以它被认为是文本），然后是输出数据和私有数据。在逻辑上 bss 段应跟在数据的后面，但是就像在任何别的可执行文件中那样，它并不在于这个文件中。

## 创建共享库

一个 UNIX 共享库实际上包含两个相关文件，即共享库本身和给链接器用的空占位库(stub library)。库创建工具将一个档案格式的普通库和一些包含控制信息的文件作为输入生成了这两个文件。空占位库根本不包含任何的代码和数据（可能会包含一个自举例程），但是它包含程序链接该库时需要使用的符号定义。

创建一个共享库需要以下几步，我们将在后面更多的讨论它们：

- 确定库的代码和数据将被定位到什么地址。
- 彻底扫描输入的库寻找所有输出的代码符号（如果某些符号是用来在库内通信的，那么就会有一个控制文件是这些不对外输出的符号的列表）。
- 创建一个跳转表，表中的每一项分别对应每个输出的代码符号。
- 如果在库的开头有一个初始化或加载例程，那么就编译或者汇编它。

- 创建共享库。运行链接器把所有内容都链接为一个大的可执行格式文件。
- 创建空占位库：从刚刚建立的共享库中提取出需要的符号，针对输入库的符号调整这些符号。为每一个库例程创建一个空占位例程。在 COFF 库中，也会有一个小的初始化代码放在占位库里并被连接到每一个可执行体中。

## 创建跳转表

最简单的创建一个跳转表的方法就是编写一个全是跳转指令的汇编源代码文件，如图 3，并汇编它。这些跳转指令需要使用一种系统的方法来标记，这样以后空占位库就能够把这些地址提出取来。

对于像 x86 这样具有多种长度的跳转指令的平台，可能稍微复杂一点。对于含有小于 64K 代码的库，3 个字节的短跳转指令就足够了。对于较大的库，需要使用更长的 5 字节的跳转指令。将不同长度的跳转指令混在一起是不能让人满意的，因为它使得表地址的计算更加困难，同时也更难在以后重建库时确保兼容性。最简单的解决方法就是都采用最长的跳转指令；或者全部都使用短跳转，对于那些使用短跳转太远的例程，则用一个短跳转指令跳转到放在表尾的匿名长跳转指令。（通常由此带来的麻烦比它的好处更多，因为第一跳转表很少会有好几百项。）

---

Figure 9-3: Jump table

```
... start on a page boundary
.align 8; align on 8-byte boundary for variable length insns
JUMP_read: jmp _read
.align 8
JUMP_write: jmp _write
...
_read: ... code for read()
...
_write: ... code for write()
```

---

## 创建共享库

一旦跳转表和加载例程（如果需要的话）建立好之后，创建共享库就很容易了。只需要使用合适的参数运行链接器，让代码和数据从正确的地址空间开始，并将自引导例程、跳转表和输入库中的所有例程都链接在一起。它同时完成了给库中每项分配地址和创建共享库文件两件事。

库之间的引用会稍微复杂一些。如果你正在创建，例如一个使用标准 C 库例程的共享数学库，那就要确保引用的正确。假定当链接器建立新库时需要用到的共享库中的例程已经建

好，那么它只需要搜索该共享库的空占位库，就像普通的可执行程序引用共享库那样。这将让所有的引用都正确。只留下一个问题，就是需要有某种方法确保任何使用新库的程序也能够连接到旧库上。对新库的空占位库的适当设计可以确保这一点。

## 创建空占位库

创建空占位库是创建共享库过程中诡秘的部分之一。对于库中的每一个例程，空占位库中都要包含一个同时定义了输出和输入的全局符号的对应项。

数据全局符号会被链接器放在共享库中任何地方，获取它们的数值的最合理的办法就是创建一个带有符号表的共享库，并从符号表中提取符号。对代码全局符号，入口指针都在跳转表中，所以同样很简单，只需要从共享库中提取符号表或者根据跳转表的基址和每一个符号在表中的位置来计算符号地址。

不同于普通库模块，空占位库模块既不包含代码也不包含数据，只包含符号定义。这些符号必须定义成绝对数而不是相对，因为共享库已经完成了所有的重定位。库创建程序从输入库中提取出每一个例程，并从这些例程中得到定义和未定义的全局变量，以及每一个全局变量的类型(文本或数据)。然后它创建空占位例程，通常都是一个很小的汇编程序，以跳转表中每一项的地址的形式定义每个文本全局变量，以共享库中实际地址的形式定义每个数据或 bss 全局变量，并以“未定义”的形式定义没有定义的全局变量。当它完成所有空占位后，就对其进行汇编并将它们合并到一个普通的库档案文件中。

COFF 空占位库使用了一种不同的、更简单的设计。它们是具有两个命名段的单一目标文件。“.lib”段包含了指向共享库的所有重定位信息，“.init”段包含了将会链接到每一个客户程序去的初始化代码，一般是来初始化库中的变量。Linux 共享库更简单，a.out 文件中包含了带有设置向量(“set vector”)的符号定义，我们将在后面的连接部分详细讨论设置向量。.

共享库的名称一般是原先的库名加上版本号。如果原先的库称为/lib/libc.a，这通常是 C 库的名字，当前的库版本是 4.0，空占位库可能是/lib/libc\_s.4.0.0.a，共享库就是/s hlib/libc\_s.4.0.0 (多出来的 0 可以允许小版本的升级)。一旦库被放置到合适的目录下面，它们就可以被使用了。

## 版本命名

任何共享库系统都需要有一种办法处理库的多个版本。当一个库被更新后，新版本相对于之前版本而言在地址和调用上都有可能兼容或不兼容。UNIX 系统使用前面提到的版本命名序号来解决这个问题。

第一个数字在每次发布一个不兼容的全新的库的时候才被改变。一个和 4.x.x 的库链接的程序不能使用 3.x.x 或 5.x.x 的库。第二个数是小版本。在 Sun 系统上，每一个可执行程序所链接的库都至少需要一个尽可能大的小版本号。例如，如果它链接的是 4.2.x，那么它就可以和 4.3.x 一起运行而 4.1.x 则不行(译者注：就是说使用尽可能大的小版本号，确

保可执行程序可以运行）。另一些系统将第二个数字当作第一个数字的扩展，这样的话使用一个 4.2.x 的库链接的程序就只能和 4.2.x 的库一起运行。第三个数字通常都被当作补丁级别。虽然任何的补丁级别都是可用的，可执行程序最好还是使用最高的有效补丁级别。

不同的系统在运行时查找对应库的方法会略有不同。Sun 系统有一个相当复杂的运行时加载器，在库目录中查看所有的文件名并挑选出最好的那个。Linux 系统使用符号连接而避免了搜索过程。如果库 libc.so 的最新版本是 4.2.2，库的名字是 libc\_s.4.2.2，但是这个库也已经被链接到 libc\_s.4.2，那么加载器将仅需打开名字较短的文件，就选好了正确的版本。

多数系统都允许共享库存于多个目录中。类似于 LD\_LIBRARY\_PATH 的环境变量可以覆盖可执行程序中的路径，以允许开发者使用它们自己的库替代原先的库进行调试或性能测试（使用“set user ID”特性替代当前用户运行的程序将忽略 LD\_LIBRARY\_PATH 以避免使用恶意用户添加了安全漏洞的“特洛伊木马”库）。

## 使用共享库链接

使用静态共享库来链接，比创建库要简单得多，因为几乎所有的确保链接器正确解析库中程序地址的困难工作，都在创建空占位库时完成了。唯一困难的部分就是在程序开始运行时将需要的共享库映射进来。

每一种格式都会提供一个小窍门让链接器创建一个库的列表，以便启动代码把库映射进来。COFF 库使用一种残忍的强制方法；链接器中的特殊代码在 COFF 文件中创建了一个以库名命名的段。Linux 链接器使用一种不那么残忍的方法，即创建一个称为设置向量的特殊符号类型。设置向量象普通的全局符号一样，但如果它有多个定义，这些定义会被放进一个以该符号命名的数组中。每个共享库定义一个设置向量符号 \_\_SHARED\_LIBRARIES\_\_，它是由库名、版本、加载地址等构成的一个数据结构的地址。链接器创建一个指向每个这种数据结构的指针的数组，并称之为 \_\_SHARED\_LIBRARIES\_\_，好让启动代码可以使用它。BSD/OS 共享库没有使用任何的此类链接器窍门。它使用 shell 脚本建立一个共享的可执行程序，用来搜索作为参数或隐式传入的库列表，提取出这些文件的名字并根据系统文件中的列表来加载这些库的地址，然后编写一个小汇编源文件创建一个带有库名字和加载地址的结构数组，并汇编这个文件，把得到的目标文件加入到链接器的参数列表中。

在每一种情况中，从程序代码到库地址的引用都是通过空占位库中的地址自动解析的。

## 使用共享库运行

启动一个使用共享库的程序需要三步：加载可执行程序，映射库，进行库特定的初始化操作。在每一种情况下，可执行程序都被系统按照通常的方法加载到内存中。之后，处理方法会有差别。系统 V.3 内核具有了处理链接 COFF 共享库的可执行程序的扩展性能，其内核会查看库列表并在程序运行之前将它们映射进来。这种方法的不利之处在于“内核肿胀”，会给不可分页的内核增加更多的代码；并且由于这种方法不允许在未来版本中有灵活性和可升级性，所以它是不灵活的（系统 V.4 整个抛弃了这种策略，转而采用了我们下章会涉

及到的 ELF 动态共享库)。

Linux 增加了一个单独的 `uselib()` 系统调用，以获取一个库的文件名字和地址，并将它映射到程序的地址空间中。绑定到可执行体中的启动例程搜索库列表，并对每一项执行 `use lib()`。

BSD/OS 的方法是使用标准的 `mmap()` 系统调用将一个文件的多个页映射进地址空间，该方法还使用一个链接到每个共享库起始处的自举例程。可执行程序中的启动例程遍历共享库表，打开每个对应的文件，将文件的第一页映射到加载地址中，然后调用各自的自举例程，该例程位于可执行文件头之后的起始页附近的某个固定位置。然后自举例程再映射余下的文本段、数据段，然后为 `bss` 段映射新的地址空间，然后自举例程就返回了。

所有的段被映射了之后，通常还有一些库特定的初始化工作要做，例如，将一个指针指向 C 标准库中指定的系统环境全局变量 `environ`。COFF 的实现是从程序文件的“`.init`”段收集初始化代码，然后在程序启动代码中运行它。根据库的不同，它有时会调用共享库中的例程，有时不会。Linux 的实现中没有进行任何的库初始化，并且指出了在程序和库中定义相同的变量将不能很好工作的问题。

在 BSD/OS 实现中，C 库的自举例程会接收到一个指向共享库表的指针，并将所有其它的库都映射进来，减小了需要连接到单独的可执行体中的代码量。最近版本的 BSD 使用 ELF 格式的可执行体。ELF 头有一个 `interp` 段，其中包含一个运行该文件时需要使用的解释器程序的名字。BSD 使用共享的 C 库作为解释器，这意味着在程序启动之前内核会将共享 C 库先映射进来，这就节省了一些系统调用的开销。库自举例程进行的是相同的初始化工作，将库的剩余部分映射进来，并且，通过一个指针，调用程序的 `main` 例程。

## malloc hack 和其它共享库问题

虽然静态共享库具有很好的性能，但是它们的长期维护是困难和容易出错的，下面给出一些轶事为例。

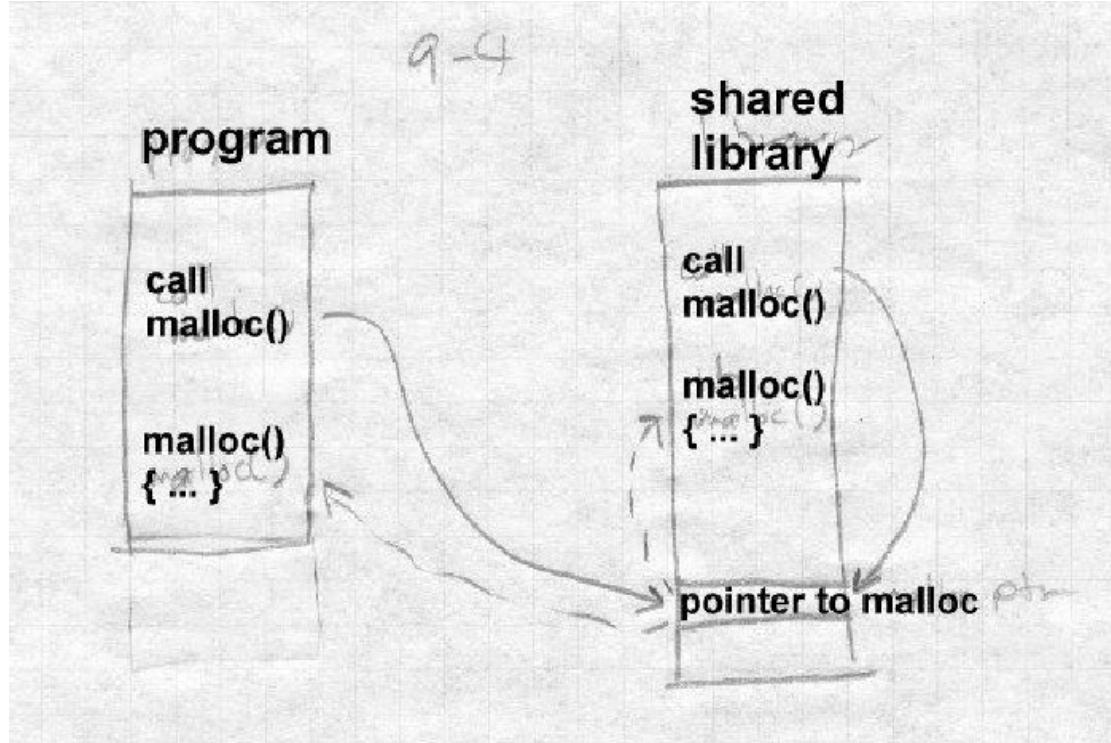
在一个静态库中，所有的库内调用都被永久绑定了，所以不可能将某个程序中所使用的库例程通过重新定义替换为私有版本的例程。多数情况下，由于很少有程序会对标准库中例如 `read()`、`strcmp()` 等例程进行重新定义，所以永久绑定不是什么大问题；并且如果它们自己的程序使用私有版本的 `strcmp()`，但库例程仍调用库中标准版本，那么也没有什么大问题。

但是很多程序定义了它们自己的 `malloc()` 和 `free()` 版本，这是分配堆存储的例程；如果在一个程序中存在这些例程的多个版本，那么程序将不能正常工作。例如，标准 `strdup()` 例程，返回一个指向用 `malloc` 分配的字符串指针，当程序不再使用它时可以释放它。如果库使用 `malloc` 的某个版本来分配字符串的空间，但是应用程序使用另一个版本的 `free` 来释放这个字符串的空间，那么就会发生混乱。

为了能够允许应用程序提供它们自己版本的 `malloc` 和 `free`，System V.3 的共享 C 库使用了一种“丑陋”的技术，如图 4 所示。系统的维护者将 `malloc` 和 `free` 重新定义为间接调用，这是通过绑定到共享库的数据部分的函数指针实现的，我们将称它们为 `malloc_ptr` 和 `free_ptr`。

```
extern void *(*malloc_ptr)(size_t);  
extern void (*free_ptr)(void *);  
#define malloc(s) (*malloc_ptr)(s)  
#define free(s) (*free_ptr)(s)
```

图 9-4: malloc hack  
程序, 共享 C 库的图例  
malloc 指针和初始化代码  
从库代码中的间接调用



然后它们重新编译了整个 C 库，并将下面的几行内容(或汇编同类内容)加入到占位库的 init 段，这样它们就被加入到每个使用该共享库的程序中了。

```
#undef malloc  
#undef free  
malloc_ptr = &malloc;  
free_ptr = &free;
```

由于占位库将被绑定到应用程序中的，而不是共享库，所以它对 malloc 和 free 的引用是在链接时解析的。如果存在一个私有版本的 malloc 和 free，它将指向私有版本函数的指针（译者注：指 malloc\_ptr 和 free\_ptr），否则它将使用标准库的版本。不管哪种方法，库和应用程序使用的都是相同版本的 malloc 和 free。

虽然这种实现方法让库的维护工作更加困难了，而且只能用于少数手工选定的名字，但只要它可以自动进行而不需要手工编写脆弱的源代码，这种在程序运行时通过指针进行解析进行库内例程调用的方法就是一个很好的主意，我们将会在下一章看到自动版本是如何工作的。

全局数据中的名字冲突仍然是遗留在共享库中的一个问题。考虑一下图 5 所示的小程序。如果你用任何一个我们本章描述过的共享库编译和链接它，他将打印一个值为 0 的状态代码而不是正确的错误代码。这是由于

```
int errno
```

创建了一个没有绑定到共享库中去的新的 errno 实例。如果你不将 extern 注释掉，这个程序就可以正常运行，因为它现在引用了一个未定义的全局变量，这将使链接器将其绑定到共享库中的 errno。就像我们将要看到的，动态连接很好地解决了这个问题，但是会付出一些性能的代价。

---

Figure 9-5: Address conflict example

```
#include <stdio.h>
/* extern */
int errno;
main()
{
    unlink("/non-existent-file");
    printf("Status was %d\n", errno);
}
```

---

最后，即使 UNIX 共享库中的跳转表也会引起兼容性的问题。在共享库外的例程看来，库中输出的每个例程的地址就是一个跳转表表项的地址。但是在库内部的例程看来，例程的地址可能是跳转表表项，也可能是跳转表要跳转到的实际入口点。有时为了处理某些特殊情况，一个库例程会比较作为参数传递给它的地址，看它是否是某个库例程的地址。

一种显而易见但是不完全有效的解决方案就是在建立共享库的过程中将例程的地址绑定到跳转表表项，因为这样可以确保库中所有例程的符号引用都被解析到对应的表项。但是如果两个例程在同一个目标文件中，那么在这个目标文件中的引用通常是对例程文本段地址的相对引用。（由于是同一个目标文件，该例程地址已知；除了这种特殊情况，没有什么别的理由需要返回到同一目标文件中去引用一个符号例程）。虽然通过扫描可重定位的文本段引用来找到相应的输出符号的地址是可能的，但是实际中最常用的解决方法是“别那么做”，不要编写依赖于需要识别库例程地址的代码。

Windows 的 DLL 库也存在相似的问题，因为在每一个 EXE 或者 DLL 内部，输入例程的地址被认为是可以间接跳转到例程实际地址的占位例程地址（译者注：由于这里的地址并不是实际地址，所以才会被认为是一个问题）。同样，对这个问题最常采用的解决方法是“别那么做”。

## 练习

如果你在一个带有共享库的 UNIX 系统上查看/shlib 目录，你会发现每个库都会有 3 到 4 个版本，诸如 libc\_s.2.0.1、libc\_s.3.0.0。为什么不使用最新的一个呢？

在一个空占位库中，为什么将每一个例程中的未定义全局符号都包含进来是非常重要的，即使在一个未定义的全局符号引用了该库中的另一个例程？

一个空占位库是包含了诸如在 COFF 或 Linux 中的所有库符号的单一可执行体，另一个是具有多个单独的模块的实际库，两者什么不同呢？

## 项目

我们要扩展链接器以支持静态共享库。这包括很多子项目，第一个就是建立共享库，然后就是使用共享库来链接可执行体。

在我们的系统中，共享库只是一个被链接了给定地址的目标文件。虽然它可以引用其它的共享库，但不会有重定位和未解析的全局符号引用。占位库是普通的目录格式或者文件格式的库，库中的每一项包含针对对应库成员的输出（绝对的）和输入符号，但是没有文本段或数据段。每一个占位库必须告诉链接器对应的共享库的名字。如果你使用目录格式的占位库，那么一个名为“LIBRARY NAME”的文件将包含一行一行的文本。第一行是对应共享库的名称，剩下的行是该共享库依赖的其它共享库名称（空格避免了符号的名字冲突）。如果你使用文件格式的库，那么库的初始行要有些额外的域：

```
LIBRARY nnnn pppppp ffffff ggggg hhhhh ...
```

这里 fffff 是共享库的名字，剩下的是它所依赖的其它共享库的名称。

项目 9-1：让链接器可以从规则的目录格式或文件格式中生成静态共享库和占位库。如果你还没有那么做，你将需要给链接器增加一个标识（译者注：参数格式）来设置链接器分配的段基地址。输入是一个规则的库，和这个库所依赖的其它任何共享库的占位库。输出是一个包含所有输入库成员的段的可执行格式的共享库，和一个对每个包含输入库的成员都有对应占位成员的占位库。

项目 9-2：扩展链接器以使用静态共享库生成可执行体。鉴于在一个执行体中引用共享库中的符号与在一个共享库中引用另一个共享库的符号的方法是相同的，所以项目 9-1 已经完成了搜索占位库符号解析的大多数工作。链接器需要将必要的库名放到输出文件中，以便运行时加载器知道需要加载什么库。让链接器建立一个名为.lib 的段，保存需要的共享库名称，这些名称之间以 null 字节标识间隔，以 2 个 null 字节标识结尾。建立一个名为.\_SHARED\_LIBRARIES 的符号，它引用.lib 段的开始地址，可以让库的初始化例程使用。

# 第 10 章 动态链接和加载

\$Revision: 2.3 \$

\$Date: 1999/06/15 03:30:36 \$

动态链接将很多链接过程推迟到了程序启动的时候。它提供了一系列其它方法无法获得的优点：

动态链接的共享库要比静态链接的共享库更容易创建。

动态链接的共享库要比静态链接的共享库更容易升级。

动态链接的共享库的语义更接近于那些非共享库。

动态链接允许程序在运行时加载和卸载例程，这是其它途径所难以提供的功能。

当然这也有少许不利。由于每次程序启动的时候都要进行大量的链接过程，动态链接的运行时性能要比静态链接的低不少，这是付出的代价。程序中所使用的每一个动态链接的符号都必须在符号表中进行查找和解析（Windows 的 DLL 某种程度上有所改善，下面将会讲到）。由于动态链接库还要包括符号表，所以它比静态库要大。

在调用的兼容性问题之上，一个顽固的麻烦根源是库语义的变化。和非共享或静态共享库而言，变更动态链接库要容易很多。所以很容易就可以改变已存在程序正在使用的动态链接库。这意味着即使程序没有任何改变，程序的行为也会改变。在微软 Windows 系统下这是一个常见的问题所在，程序会使用大量的共享库，而这些库有不同的版本，库之间的版本控制非常的复杂。多数程序在出货时都带有它们所需库的副本，而安装程序经常会不假思索的将安装包中的旧版本共享库覆盖已存在的新版本库，这就破坏了那些依赖新版本库特性的程序。考虑周全的安装程序会在使用旧版本库覆盖新版本库的时候弹出告警框提示，但这样的话，依赖新版本库特性的那些应用程序又会发生旧版本库替换新版本库时发生的类似问题。

## ELF 动态链接

在八十年代晚期，是 SUN 微系统公司首次在 UNIX 系统中引入了动态共享库技术。SUN 合作开发的 UNIX 系统 V 版本 4，引入了 ELF 目标格式，并采用了 SUN 的 ELF 方案。很明显 ELF 是对之前目标格式的改进，在九十年代末它成为 UNIX、诸如 Linux 的类 UNIX 系统和 BSD 这样的衍生版本的标准。

## ELF 文件内容

如在第三章中提到的那样，一个 ELF 文件可以看成是由链接器解释的一系列区段（section），或由加载器解释的一系列段（segment）。ELF 程序和共享库的通用结构相同，但具体的段（segment）或者区段（section）有所区别。

ELF 共享库可被加载到任何地址，因此它们总是使用位置无关代码（PIC）的形式，这

样文件的代码页无须重定位即可在多个进程之间共享。像在第八章描述的那样，ELF 链接器通过全局偏移量表（GOT）支持 PIC 代码，每个共享库中都有 GOT，包含着程序所引用的静态数据的指针，如图 1 所示。动态链接器会解析和重定位 GOT 中的所有指针。这会引起性能的问题，但是在实际中除了非常巨大的库之外，GOT 都不大。通常使用的标准 C 库中超过 350K 的代码的 GOT 也只有 180 个表项。

由于 GOT 位于代码所引用的可加载 ELF 文件中，因此无论被加载到何处，位于文件中的相对地址都不会发生变化。代码可以通过相对地址来定位 GOT，将 GOT 的地址加载到一个寄存器中，然后在需要寻址静态数据的时候从 GOT 中加载相应的指针。如果一个库没有引用任何的静态数据那么它可以不需 GOT，但实际中所有的库都有 GOT。

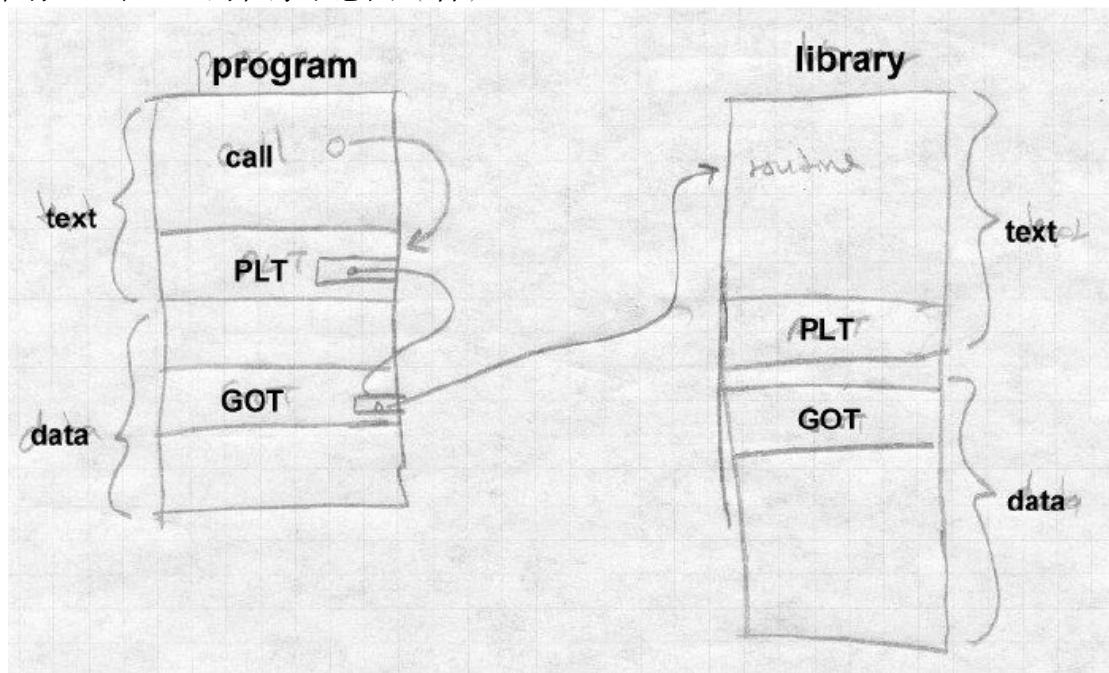
为了支持动态链接，每个 ELF 共享库和每个使用了共享库的可执行程序都有一个过程链接表（Procedure Linkage Table，PLT）。PLT 就像 GOT 对数据引用那样，对函数调用增添了一层间接途径。PLT 还允许进行“懒惰计算法”，即只有在第一次被调用时，才解析过程的地址。由于 PLT 表项要比 GOT 多很多（在上面提到的 C 库中会有超过 600 项），并且大多数例程在任何给定的程序中都不会被调用，因此“懒惰计算法”既可以提高程序启动的速度，也可以整体上节省相当可观的时间。

---

图 10-1：过程链接表 PLT 和全局偏移量表 GOT

带有 GOT 的程序示意图（左）

带有 PLT 和 GOT 的程序示意图（右）



---

下面我们讨论 PLT 的细节。

一个被动态链接的 ELF 文件包含了运行时链接器在重定位文件和解析任意未定义符号时所需的所有链接器信息。动态符号表，即 .dynsym 区段，包含了文件中所有的输入和输出符号。而 .dynstr 和 .hash 区段包含了符号的名称字串，以及有助于加快运行时链接器查找速度的散列表。

最后一个 ELF 动态链接文件的额外部分是 DYNAMIC 段（也被标识为. dynamic 区段），动态链接器使用它来寻找和该文件相关的信息。它作为数据段的一部分被加载，但由 ELF 文件头部的指针指向它，这样运行时动态链接器就可以找到它了。DYNAMIC 区段是一个由被标记的数值和指针组成的列表。一些表项类型只会出现在程序中，一些表项类型只会在库中，还有一些类型在两者中都会出现。

NEEDED：该文件所需的库的名称。（通常在程序中，如果一个库依赖其它库时有时也会在这个库中，这种情况可以嵌套发生）

SONAME：共享的对象名称。链接器所需要的文件的名称。（在库中）

SYMTAB、STRTAB、HASH、SYMENT、STRSZ：指向符号表，相关联的字串表和散列表，符号表项大小，字串表大小。（程序和库中都有）

PLTGOT：指向 GOT，或者在某些架构下指向 PLT。（程序和库中都有）

REL、RELSZ 和 RELENT，或者 RELA、RELASZ 和 RELAENT：重定位表的指针、大小和表项大小。重定位表中不包含加数，加数表中才包含它们。根据名字也能猜到，RELA、RELASIZE 和 RELAENT 是加数表指针、加数表大小和加数表项的大小<sup>5</sup>。（程序和库中都有）

JMPREL、PLTRELSZ 和 PLTREL：由 PLT 引用的数据的重定位表的指针、大小和格式（REL 或 RELA）。（程序和库中都有）

INIT 和 FINI：初始化和终止例程的指针，在程序启动和终止的时候调用。（可选的，但是通常在库和程序中都有）

还有其它少量晦涩而且少用的类型，在这里就忽略不提了。

一个完整的 ELF 共享库看起来会像图 1 那样。首先是只读部分，包括符号表、PLT、文本和只读数据，然后是可读写部分，包括常规数据、GOT 和动态区段。BSS 在逻辑上跟在最后一个可读写区段后面，但通常都不会出现在文件中。

---

图 10-2：ELF 共享库  
(很多表示指针的箭头)

只读页：

.hash  
.dynsym  
.dynstr  
.plt  
.text  
.rodata

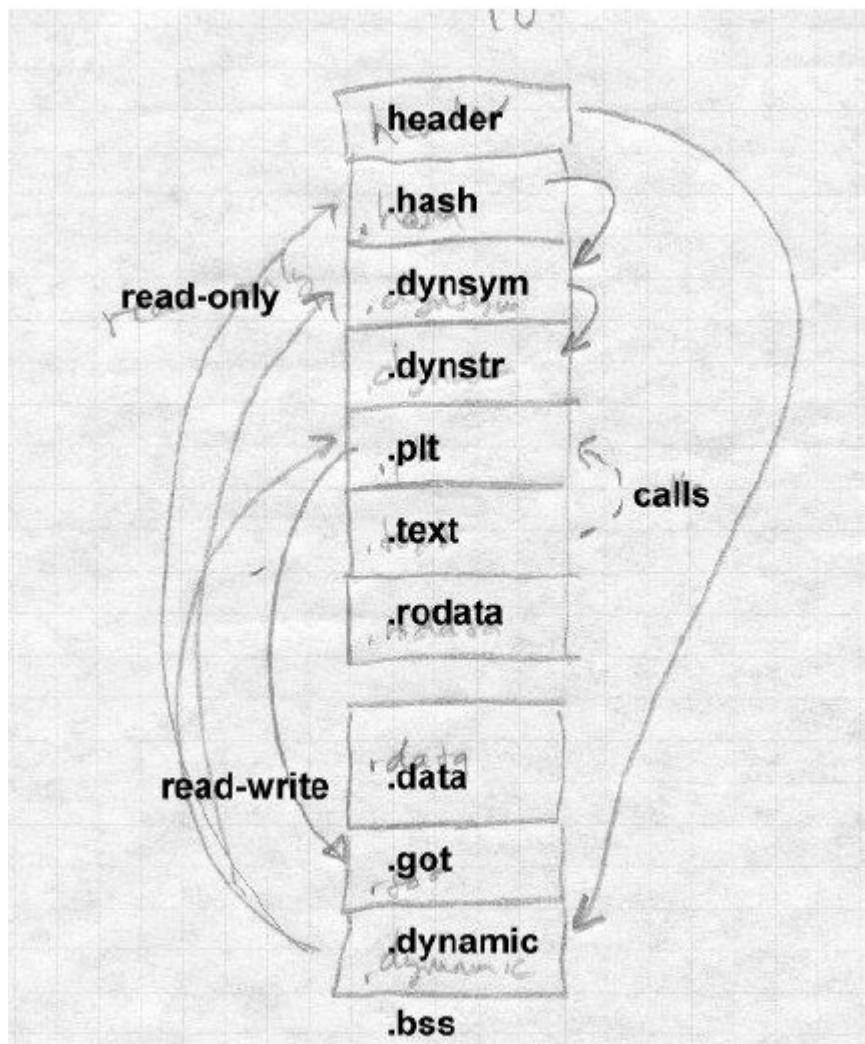
可读写页：

.data  
.got  
.dynamic

---

<sup>5</sup> 这句话是译者增加的，原作者没有对 RELA, RELASZ 和 RELAENT 进行说明。

.bss



一个 ELF 程序看起来和上图很相似，但是在只读段还有初始化和终止例程，以及靠近文件前部用来指示动态链接器（通常是 ld.so）的 INTERP 区段。由于程序文件不需要在运行时被重定位，因此数据段没有 GOT。

## 加载一个动态链接的程序

加载一个动态链接的程序，这个过程冗长但简单。

## 启动动态链接器

在操作系统运行程序时，它会像通常那样将文件的页映射进来，但注意在可执行程序中存在一个 INTERPRETER 区段。这里特定的解释器是动态链接器，即 ld.so，它自己也是 ELF 共享库的格式。操作系统并非直接启动程序，而是将动态链接器映射到地址空间的一个合适的位置，然后从 ld.so 处开始，并在栈中放入链接器所需要的辅助向量 (auxiliary vector) 信息。向量包括：

`AT_PHDR`, `AT_PHENT`, 和 `AT_PHNUM`: 程序头部在程序文件中的地址, 头部中每个表项的大小, 和表项的个数。头部结构描述了被加载文件中的各个段。如果系统没有将程序映射到内存中, 就会有一个 `AT_EXECFD` 项作为替换, 它包含被打开程序文件的文件描述符。

`AT_ENTRY`: 程序的起始地址, 当动态链接器完成了初始化工作之后, 就会跳转到这个地址去。

`AT_BASE`: 动态链接器被加载到的地址。

此时, 位于 `ld.so` 起始处的自举代码找到它自己的 GOT, 其中的第一项指向了 `ld.so` 文件中的 DYNAMIC 段。通过 dynamic 段, 链接器在它自己的数据段中找到自己的重定位项表和重定位指针, 然后解析例程需要加载的其它东西的代码引用 (Linux `ld.so` 将所有的基础例程都命名为由字串 `_dt_` 起头, 并使用专门代码在符号表中搜索以此字串开头的符号并解析它们)。

链接器然后通过指向程序符号表和链接器自己的符号表的若干指针来初始化一个符号表链。从概念上讲, 程序文件和所有加载到进程中的库会共享一个符号表。但实际上链接器并不是在运行时创建一个合并后的符号表, 而是将各个文件中的符号表组成一个符号表链。每个文件中都有一个散列表 (一系列的散列头部, 每个头部引领一个散列队列) 以加速符号查找的速度。链接器可以通过计算符号的散列值, 然后访问相应的散列队列进行查找以加速符号搜索的速度。

## 库的查找

链接器自身的初始化完成之后, 它就会去寻找程序所需要的各个库。程序的程序头部有一个指针, 指向 dynamic 段 (包含有动态链接相关信息) 在文件中的位置。在这个段中包含一个指针 `DT_STRTAB`, 指向文件的字串表, 和一个偏移量表 `DT_NEEDED`, 其中每一个表项包含了一个所需库的名称在字串表中的偏移量。

对于每一个库, 链接器会查找对应的 ELF 共享库文件, 这本身也是一个颇为复杂的过程。在 `DT_NEEDED` 表项中的库名称看起来与 `libXt.so.6` (`Xt` 工具包, 版本 6) 类似。库文件可能会在若干库目录的任意一个之中, 甚至可能文件的名称都不相同。在我的系统上, 这个库的实际名称是 `/usr/X11R6/lib/libXt.so.6`。末尾的 “`.0`” 是次版本号。

链接器在以下位置搜索库:

- 是否 `dynamic` 段有一个称为 `DT_RPATH` 的表项, 它是由分号分隔开的可以搜索库的目录列表。它可以通过一个命令行参数或者在程序链接时常规 (非动态) 链接器的环境变量来添加。它经常会被诸如数据库类这样需要加载一系列程序并可将库放在单一目录的子系统使用,
- 是否有一个环境符号 `LD_LIBRARY_PATH`, 它可以是由分号分隔开的可供链接器搜索库的目录列表。这就可以让开发者创建一个新版本的库并将它放置在 `LD_LIBRARY_PATH` 的路径中, 这样既可以通过已存在的程序来测试新的库, 或用来监测程序的行为。(因为安全原因, 如果程

序设置了 set-uid，那么这一步会被跳过）

- 链接器查看库缓冲文件/etc/ld.so.conf，其中包含了库文件名和路径的列表。如果要查找的库名称存在于其中，则采用文件中相应的路径。大多数库都通过这种方法被找到（路径末尾的文件名称并不需要和所搜索的库名称精确匹配，详细请参看下面的库版本章节）。
- 如果所有的都失败了，就查找缺省目录/usr/lib，如果在这个目录中仍没有找到，就打印错误信息，并退出执行。

一旦找到包含该库的文件，动态链接器会打开该文件，读取 ELF 头部寻找程序头部，它指向包括 dynamic 段在内的众多段。链接器为库的文本和数据段分配空间，并将它们映射进来，对于 BSS 分配初始化为 0 的页。从库的 dynamic 段中，它将库的符号表加入到符号表链中，如果该库还进一步需要其它尚未加载的库，则将那些新库置入将要加载的库链表中。

在该过程结束时，所有的库都被映射进来了，加载器拥有了一个由程序和所有映射进来的库的符号表联合而成的逻辑上的全局符号表。

## 共享库的初始化

现在加载器再次查看每个库并处理库的重定位项，填充库的 GOT，并进行库的数据段所需任何重定位。

在 x86 平台上，加载时的重定位包括：

R\_386\_GLOB\_DAT：初始化一个 GOT 项，该项是在另一个库中定义的符号的地址。

R\_386\_32：对在另一个库中定义的符号的非 GOT 引用，通常是静态数据区中的指针。

R\_386\_RELATIVE：对可重定位数据的引用，典型的是指向字串（或其它局部定义静态数据）的指针。

R\_386 JMP\_SLOT：用来初始化 PLT 的 GOT 项，稍后描述。

如果一个库具有 .init 区段，加载器会调用它来进行库特定的初始化工作，诸如 C++ 的静态构造函数。库中的 .fini 区段会在程序退出的时候被执行。它不会对主程序进行初始化，因为主程序的初始化是有自己的启动代码完成的。当这个过程完成后，所有的库就都被完全加载并可以被执行了，此时加载器调用程序的入口点开始执行程序。

## 使用 PLT 的惰性过程链接 (lazy procedure linkage)

使用共享库的程序通常都会有对大量函数的调用。在这个程序的一次运行中，其中（在错误处理和程序的其它部分的）绝大多数函数永远都不会被调用到。进一步的，每一个共享库中也存在了对其它库中大量函数的调用，由于它们大多都在几乎不会被程序直接或间接调用到的例程中，因此在一次程序的执行中它们被执行到几率的更少得多。

为了加快程序启动的速度，动态链接的 ELF 程序使用了对过程地址的懒惰绑定 (lazy binding)。即一个过程的地址直到第一次被调用时才会被绑定。

ELF 通过过程链接表（Procedure Linkage Table, PLT）支持懒惰绑定。每一个动态绑定的程序和共享库都有一个 PLT，PLT 中的每一个表项对应一个程序或库中被调用的非本地例程，如图 3 所示。注意在位置无关代码中的 PLT 本身也是位置无关代码，因此它可以成为只读文本段的一部分。

---

图 10-3：x86 代码中的 PLT 结构

特殊的第一表项

PLT0: pushl GOT+4

jmp \*GOT+8

常规表项，非 PIC 代码

PLTn: jmp \*GOT+m

push #reloc\_offset

jmp PLT0

常规表项，PIC 代码

PLTn: jmp \*GOT+m(%ebx)

push #reloc\_offset

jmp PLT0

---

在程序或者库以调用例程在 PLT 中的表项的形式创建时，程序或库中对某个特定例程的所有调用都会被进行相应的调整。在程序和库第一次调用某个例程时，PLT 项会调用运行时链接器来解析该例程的实际地址。然后，PLT 项会直接跳转到例程的实际地址，因此在第一次调用之后，使用 PLT 的代价就是在例程调用时有一个额外的间接跳转，在调用返回时没有额外的代价。

PLT 中的第一项，我们称之为 PLT0，是一段会调用动态链接器的特殊代码。在加载时，动态链接器会自动的将两个数值放置在 GOT 中。在 GOT+4 (GOT 的第二个字) 处放置了用来标识特定库的代码，在 GOT+8 处放置了动态链接器的符号解析例程的地址。

PLT 中的其余表项，我们称之为 PLTn，每一个都是由 GOT 项中的间接跳转开始。每一个 PLT 项都有一个对应的 GOT 项，该 GOT 项被初始的设置为指向 PLT 项中跟在 jmp 指令后面的 push 指令处（在 PIC 文件中这需要加载时的重定位，但这里的符号查找开销并不大）。跟在跳转指令 jmp 后面的压栈指令 push 将一个重定位偏移量压入栈中，这个偏移量是类型为 R\_386\_JMP\_SLT 的特殊重定位项在文件的重定位表中的偏移量。该重定位项的符号引用指向文件符号表中的符号，它的地址指向相应的 GOT 项。

这种紧凑但相当新颖的方法意味着程序或者库在第一次调用 PLT 项时，PLT 项中的第一个跳转指令实际上没做什么，因为它所跳转到的 GOT 又会指回到这个 PLT 项。然后 push 指令将间接标识了需要解析的符号和解析符号所需的 GOT 项的偏移量压入栈中，然后跳转到 PLT0。在 PLT0 中的指令将另一个标明当前是哪个程序或库的代码压入栈中，然后跳入到动态链接器的桩子代码 (stub code) 中（此时两个标识代码位于栈的顶部）。注意到这里是跳转指令 jmp，而不是调用指令 call，栈中位于两个标识字之上的是返回到调用该 PLT 的例程

的地址。

现在桩子函数保存所有的寄存器并调用动态链接器内部的例程来进行解析。栈中的两个标识字对于寻找库的符号表和例程在该符号表中对应的表项，已经足够了。动态链接器使用串联的运行时符号表来查找符号值，并将例程的地址存储在 GOT 项中。然后桩子代码恢复寄存器，将 PLT 压栈的两个标识码推出栈，然后跳转到这个例程中去。这时 GOT 项已经被更新了，后续对该 PLT 项的调用，就无须通过动态链接器而直接跳转到例程自己了。

## 动态链接的其它特性

ELF 链接器和动态链接器存在大量晦涩的代码以处理特殊情况，并尝试和保持运行时语义尽可能的与非共享库相似。

## 静态的初始化

如果一个程序存在对定义在一个库中的全局变量的引用，由于程序的数据地址必须在链接时被绑定，因此链接器不得不在程序中创建一个该变量的副本，如图 4 所示。这种方法对于共享库中的代码没有问题，因为代码可以通过 GOT 中的指针（链接器会调整它）来引用变量。但如果库初始化这个变量就会产生问题。为了解决问题，链接器在程序的重定位表（仅仅包含类型为 R\_386 JMP\_SLOT、R\_386\_GLOB\_DAT、R\_386\_32 和 R\_386\_RELATIVE 的表项）中放入一个类型为 R\_386\_COPY 类型的表项，指向该变量在程序中的副本被定义的位置，并告诉动态链接器从共享库中将该变量被初始化的数值复制过来。

---

图 10-4：全局数据初始化

### 主程序中：

```
extern int token;
```

### 共享库中的例程：

```
int token = 42;
```

---

虽然这个特性对于特定类型的代码是关键的，但在实际中很少发生。这是一种橡皮膏（译者注：权宜之计的意思），因为它只能用于单字的数据。好在初始化程序通常的对象是指向过程或其它数据的指针，所以这个橡皮膏够用了。

## 库的版本

动态链接库通常都会结合主版本和次版本号来命名，例如 `libc.so.1.1`。但是应用程序只会和主版本号绑定，例如 `libc.so.1`，次版本号是用于升级的兼容性的。

为了保持加载程序合理的速度，系统会设法维护一个缓冲文件，保存最近用过的每一个库的全路径文件名，该文件会在一个新库被安装时有一个配置管理程序来更新。

为了支持这个设计，每一个动态链接的库都有一个在库创建时赋予的称为 SONAME 的“真名”。例如，被称为 `libc.so.1.1` 的库的 SONAME 为 `libc.so.1`（缺省的 SONAME 是库的名称）。当链接器创建一个使用共享库的程序时，它会列出程序所使用库的 SONAME 而不是库的真实名称。缓冲文件创建程序扫描包含共享库的所有目录，查找所有的共享库，提取每一个的 SONAME，对于具有相同 SONAME 的多个库，除版本最高的外其余的忽略。然后它将 SONAME 和全路径名称写入缓冲文件，这样在运行时动态链接器可以很快的找到每一个库的当前版本。

## 运行时的动态链接

虽然动态链接器会在程序启动或访问 PLT 的时候隐含的被调用，但程序仍然可以通过使用 `dlopen()` 函数显式的加载一个共享库，并通过 `dlsym()` 函数来查找一个符号（通常是一个会被调用的函数）的地址。这两个例程通常会被简单的封装为在动态链接器中的回调函数。当动态链接器通过 `dlopen()` 加载一个库的时候，它会进行对其他库一样的重定位和符号解析，这样被动态加载的程序可以引用正在运行程序中的全局变量，而无须任何特殊的对已加载例程的回调（callback）。

这就允许用户为程序增加额外的功能，而无须访问程序的源代码，甚至不需要停止和重新起动程序（在程序是数据库或 web 服务器的时候，是相当有用的）。大型主机操作系统早在 60 年代早期就已经提供与此类似的“退出例程（exit routines）”的访问，尽管还没有如此方便的接口，但它长期以来都是为打包的应用程序提供强大弹性的一种方法。这也提供了一种让程序扩展自己的方法。没有理由不能让一个程序编写一个 C 或者 C++ 的程序然后运行编译器和链接器将其创建为一个共享库，然后动态的加载并运行它。（大型主机上的程序中为各类作业定制的被链接和加载的内部回环代码已经存在了几十年了）

## Microsoft 动态链接库

微软 Windows 系统也提供共享库，称为动态链接库或 DLL，形式上与 ELF 共享库相似但某种程度上要更简单一些。16 位的 Windows 3.1 和 32 位的 Windows NT 和 Windows 95 上的 DLL 的设计是有本质区别的。这里只讨论更现代的 Win32 库。DLL 通过与 PLT 相似的策略来导入过程的地址。虽然 DLL 的设计可以实现通过与 GOT 相似的策略来导入数据地址，但实际上它们使用了一种更简单的方法，即要求明确的程序代码对共享数据的导入指针进行解析。

在 Windows 下，程序和库都是 PE（Portable Executable）格式文件，可以被内存映射到

一个进程中。与 Windows 3.1 中所有的应用程序共享单一的地址空间不同，Win32 为每个应用程序提供了自己独立的地址空间，当可执行程序和库被使用时可以映射到各自的地址空间中。对于只读代码，这没有任何特殊的区别，但对于数据而言，就意味着每个使用 DLL 的应用程序对 DLL 的数据都有属于自己的副本（这有点过分简单了，因为可以通过标识 PE 文件的某些区段为共享数据而在使用这个文件的多个应用程序之间共享一份数据副本，但是大部分数据都是非共享的）。

加载一个 Windows 可执行程序或者 DLL 与加载一个动态链接的 ELF 程序相似，尽管在 Windows 下动态链接器是操作系统内核的一部分。首先内核根据可执行程序文件的 PE 头部中的区段表，将可执行程序映射进来。然后它在根据可执行程序所使用到的 DLL 文件的 PE 头部的信息，将所有这些 DLL 映射进来。

PE 文件可以包含有重定位项。通常一个可执行程序不会包含可重定位项，因此必须将它们映射到在被链接确定的地址上。DLL 都包含有重定位项，并且在它们被链接进来的地址空间无效的时候都会被重定位（微软将运行时重定位称为 rebasing）。

所有的 PE 文件，包括可执行程序和 DLL，都有一个入口点，在 DLL 被加载、被卸载，以及每一次进程的线程 attach 或 deattach 这个 DLL 的时候，加载器都会调用 DLL 的入口点（每一次加载器都会传递一个参数说明调用原因）。这就可以提供类似 ELF 的 .init 和 .fini 区段的钩子代码来实现初始化和终结操作。

## PE 文件中的输入/输出符号 (imported and exported symbols)

Pe 文件通过文件中的两个特殊的区段来支持共享库，即被输出数据的.edata，是从文件中被输出的符号列表，以及.idata，是输入到文件中的符号列表。程序文件通常只有一个.idata 区段，而 DLL 文件总是具有.edata 区段，如果该 DLL 还使用了其它 DLL 的话那也会具有.idata 区段。符号既可以通过符号名称被输出，也可以通过“序号”（一个小整数，是被输出地址表中对应符号的索引号）被输出。通过序号进行链接效率会稍高一点，因为它避免了符号查找，但是如果创建库的作者不能保证不同版本的库之间相同的序号必须指向相同的符号，那么这种方法就很容易产生错误。实际中，序号被用来调用系统服务（因为它们甚少改变），而名称用在其它的场合。

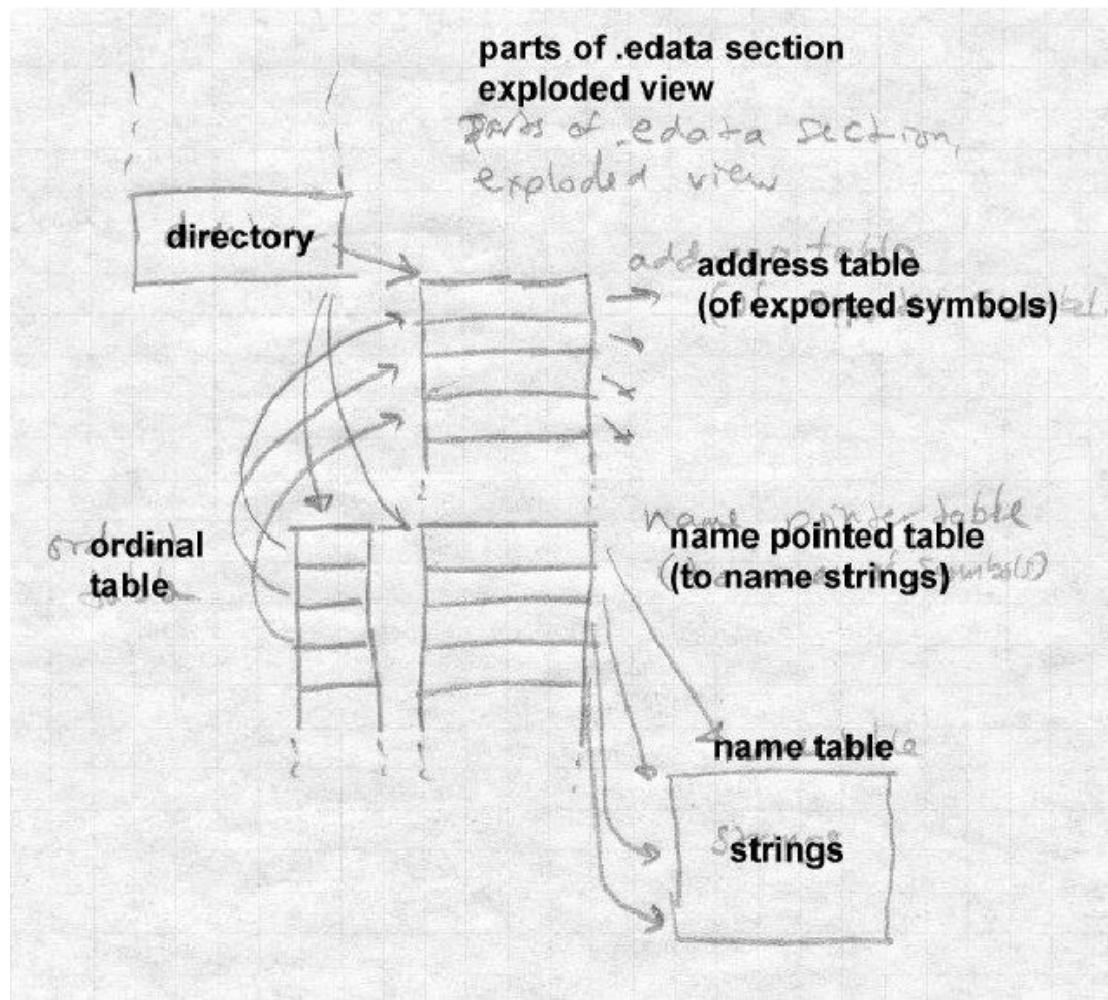
.edata 区段中包含一个描述了其余区段的输出目录表，后面跟着的是定义了被输出符号的各个表，如图 5 所示。

---

图 10-5：.edata 区段的结构

输出目录指向：

export address table	输出地址表
ordinal table	序号表
name pointer table	名称指针表
name strings	名称字符串



输出地址表包含符号的 RVA (Relative Virtual Address, 相对虚拟地址, 相对于 PE 文件的基地址)。如果 RVA 指回到. edata 区段, 则它是一个“转发”引用 (“forwarder” reference), 指向的数值是一个用以满足该引用而命名符号的字串, 很有可能定义在另外一个 DLL 中。序号和名称指针是平行存在的, 即名称指针表中的每一项是该名称字串的 RVA, 而对应的序号是位于输出地址表中的索引 (序号并不需要从 0 起始, 用来与序号值相减而得到在输出地址表索引的序号基数, 存储在输出目录中, 并且通常为 1)。虽然实际中每一个输出符号都有名称, 但这并不是必须的。在名称指针表中的符号按照字母顺序排列, 可以允许加载器使用折半查找。

.idata 区段做的事情与. edata 恰恰相反, 它将符号或序号映射到虚拟地址中。该区段包含空字符结尾输入目录表数组, 每一个需要导入符号的 DLL 对应一个输入目录表, 后面跟着输入查找表 (每个 DLL 一个), 然后是名称表。

图 10-6: .idata 区段的结构

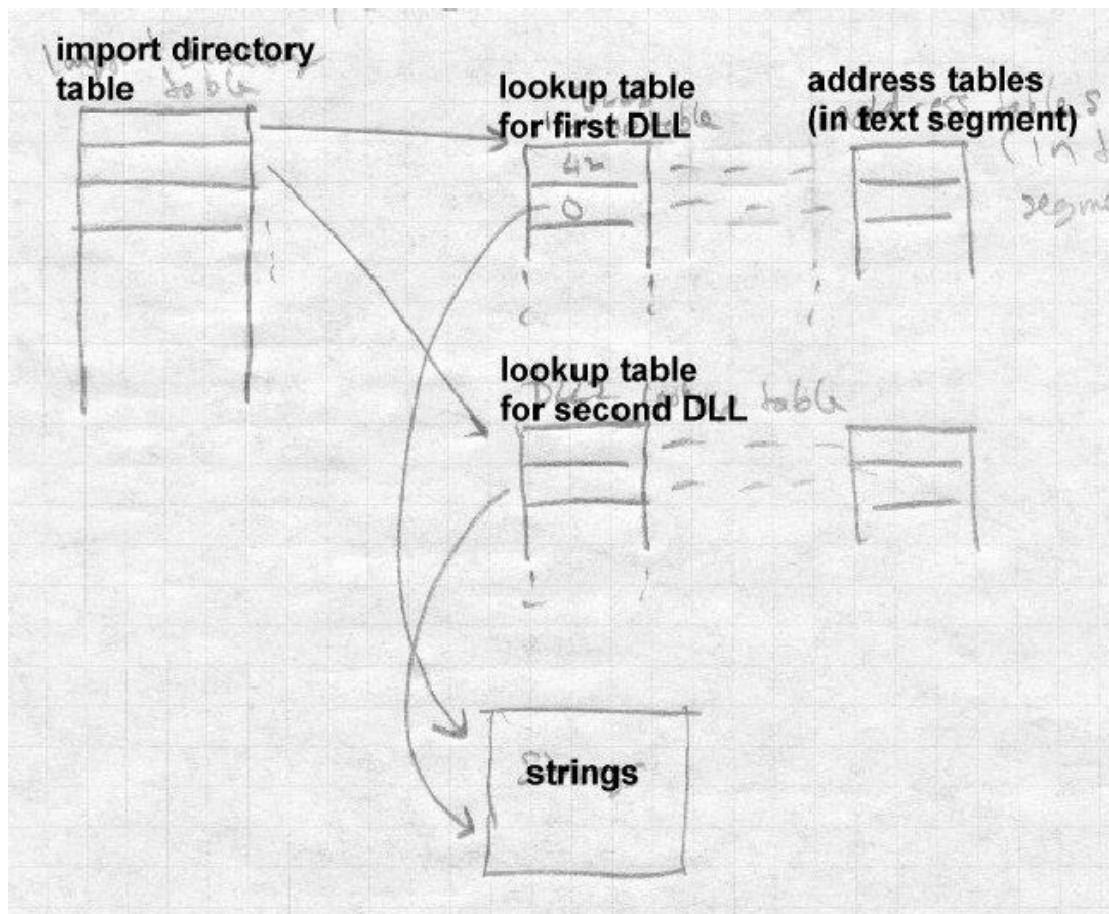
输入目录表, 带有大量箭头

每一个都具有输入查找表 (RVA), 时间/日期戳, 转发引用链 (没有用到?), DLL 名称, 输入地址 RVA 表。

空结尾

输入表, 表项具有高位标志位 (每一个 DLL 一个表)

## 名称表



对于每一个被输入的 DLL，都有一个输入地址的数组，一般都位于程序的文本段中，程序的加载器会将解析后的地址放入其中。输入查找表标识了被输入的符号，输入查找表中的表项是与输入地址表中的表项平行对应的。查找表包含有 32 位的表项，如果某一个表项的高位被置位，则低 31 位是要被输入的符号对应的序号，否则该表项就是名称表中对应表项的 RVA。每一个名称表（也称为提示/名称表，hint/name table）表项还有一个 4 字节的提示，可以用来推测符号在 DLL 书出名成指针表中的索引，后面跟着一个空字节结尾的符号名称。程序加载器使用这个提示数值来探测输出表，如果符号名称能匹配上，它就使用这个符号，否则就在整个输出表中对该名称进行折半查找（由于使用某个 DLL 的程序是被链接的，因此如果该 DLL 没有发生改变，或至少它的输出符号列表没有发生改变，这个推测就是正确的）。

与 ELF 输入符号不同，通过 .idata 输入的符号值只会放置在输入地址表中，而不会在输入文件的其它位置被修改。对于代码地址，则稍有不同。在链接器创建程序或 DLL 时，它会在文本段中创建一个名为 thunks（这个名字起的不好）的表，通过输入地址表项进行间接跳转，然后将 thunks 的地址做为输入例程的地址来使用，这个地址对于程序员是透明的（ thunks 和 .idata 区段中的多数数据实际上来自于和 DLL 一起创建的一个桩子库）。在微软最近版本的 C/C++ 编译器中，如果程序员知道某个例程将会被 DLL 所调用，该例程可被声明为“dllimport”，然后编译器将会生成一个地址表项的间接调用，从而避免了额外的间接跳

转。对于数据地址，情况稍微麻烦点，因为隐藏对位于其它可执行程序中的符号进行寻址所需要的额外一级重定向是非常困难的。传统上，程序员只是隐含的将输入值声明为真实值的指针，并隐含的解析它。最近的微软 C/C++ 编译器也可以让程序员将全局数据声明为“`dllimport`”，并且编译器可以为之生成额外的指针引用，这与 ELF 代码通过 GOT 中的指针间接引用数据的方法很相似。

## 惰性绑定

Windows 编译器的近期版本增加了延迟加载输入（`delay loaded import`），这就允许对过程进行懒惰符号绑定了，某种程度上与 ELF 的 PLT 很相似。延迟加载的 DLL 具有一个和 `.idata` 输入目录表相似的一个数据结构，但是不在 `.idata` 区段中，因此程序不能自动处理它。输入地址表中的所有表项都被初始化为指向可以寻找和加载 DLL 并使用实际地址替换地址表中内容的辅助函数。延迟加载的目录表有一个位置用来存储原先输入地址表的内容，这样当 DLL 稍后被卸载的时候，可以把这些值再恢复回去。微软提供了一个标准的辅助例程，不过它的接口都有文档说明，如果需要程序员可以编写自己的版本。

Windows 也允许使用 `LoadLibrary` 和 `FreeLibrary` 函数来明确的加载和卸载 DLL，并使用 `GetProcAddress` 来查找符号的地址。

## DLL 库和线程

Windows 的 DLL 模式不能很好工作的特例之一是线程本地存储。Windows 程序可以在同一个进程中启用多个线程，它们共享进程的地址空间。每一个线程都有一小块线程本地存储（Thread Local Storage， TLS）区域来保存和特定线程相关的数据，例如指向当前线程正在使用的数据结构或资源的指针。TLS 对于程序或每个使用 TLS 的 DLL 中的数据，需要使用槽位<sup>6</sup>（slot）。Windows 链接器可以在 PE 可执行程序中创建一个 `.tls` 区段，它定义了可执行程序及其直接引用的 DLL 中的例程所需的 TLS 的布局。每次在进程创建一个线程时，新的线程会获得自己的 TLS，该 TLS 是以 `.tls` 区段为模板创建的。

问题是，多数 DLL 既可以从可执行程序中被隐含的链接，也可以通过 `LoadLibrary` 来显式的加载。由于 DLL 的作者无法预测该库是隐式还是显式激活的，因此显式加载的 DLL 不能自动获得 `.tls` 存储区域，它不能够倚赖 `.tls` 区段。

Windows 定义了运行时系统调用，可以在 TLS 的末尾分配槽位。除非 DLL 知道自己只会被隐含的激活，否则就会使用那些系统调用，而不是 `.tls` 区段。

## OSF/1 伪静态共享库

OSF/1 是源自开放软件基金会（Open Software Foundation， OSF）的一种不幸的变种，

---

<sup>6</sup> 译者注：每个 TLS 就是一块存储区，此存储区被分割为若干个基本的存储单元，每个存储单元就是一个 slot， slot 的数目一般有上千，每个 slot 可以存放一个数据。

使用了一种介于静态和动态链接之间的共享库策略。它的作者注意到，由于静态库的重定位要比动态库少，所以前者要比后者快得多。由于这些库的升级并不是那么频繁，而且并不需要将系统中所有的可执行程序都链接一遍，所以系统管理员还能够忍受升级共享库所带来的痛苦。

因此 OSF/1 采用的方式就是维护一个所有进程都可见的全局符号表，并在系统启动时将所有的共享库都加载到一个共享的地址空间。这样在系统运行时所有库的地址都不会变化。每次程序启动时，如果它需要使用共享库，它将会所有这些库和符号表映射进来并使用全局符号表来解析可执行程序中的未定义引用。由于所有被链接的程序需要加载的地址空间部分能够确保是对所有进程都有效的，因此就不需要加载时的重定位了，并且库的重定位在系统引导时就已经完成了。

当某一个共享库发生变化时，通常系统只需要重新引导一遍，再次引导时系统就可以加载新的库并为可执行程序创建新的全局符号表。

这种策略是聪明的，但并不是非常令人满意。例如，处理符号查找要比处理重定位项慢得多，因此避免重定位并不能带来非常多的性能提升。另一个例子，动态链接可以允许在运行时加载和运行一个库，但 OSF/1 的策略就无法提供这个功能了。

## 让共享库快一些

共享库，尤其是 ELF 共享库，有时会非常缓慢。造成这个情况的原因有多种，有一些我们在第八章中曾提到过：

- 加载时库的重定位
- 加载时库和可执行程序中的符号解析
- 位置无关代码（PIC）函数初始代码带来的开销
- PIC 间接数据引用带来的开销

更慢的代码是由于 PIC 保留的寻址寄存器。前两个问题可以通过缓冲 cache 来改善，后两个问题就需要对纯 PIC 代码进一步优化。

在地址空间巨大的现代计算机中，为共享库选择一个对所有进程或至少使用这些库的进程都有效的地址空间，是可能的。有一个非常有效的技术与 Windows 的方法很相似。无论是库第一次被链接或它第一次被加载，实验性的将它的地址绑定到一块地址空间中。然后每次程序链接这个库的时候，尽可能的使用相同的地址，这意味着就不需要进行重定位了。如果这个地址空间在新的进程中不再有效了，这个库就像先前那样再次被重定位。

SGI 系统使用术语 QUICKSTART 来描述链接时的对象预重定位过程，或对共享库的一次单独的扫描。BeOS 会将重定位的库在它第一次被加载到进程中的时候缓冲起来。如果多个库之间存在依赖关系，在原则上是可以对这些库一起预先进行重定位和解析符号引用的，虽然我并不知道有哪些链接器是这么做的。

如果一个系统使用了预先重定位的库，PIC 就变得没有那么重要了。所有从预重定位地址加载库的进程都可以共享库的代码，而无论库是否是 PIC 代码，因此一个放置在适当位置上的非 PIC 库实际中也可以像 PIC 代码那样被共享，而没有 PIC 代码的性能损失。这是第 9 章中的静态链接库的基本方法，除非地址空间发生冲突。但如果这种情况发生，程序不会失

败，而是由动态链接器将库的位置移动，当然这会带来一些性能上的损失。Windows 就是使用这样的方法。

BeOS 十分彻底的实现了缓冲的重定位库，包括在库变化时保持正确的语义。当一个新的库被安装时，BeOS 会注意到这个现实并创建新版本的缓冲，这样在程序引用库的时候就不再使用旧版本了。库的变更还会有一些连带作用。当库 A 引用了库 B 中的符号，而库 B 更新了，如果任何 A 中引用的 B 中的符号发生了变化，那么也必须创建 A 的缓冲。这的确可以让程序员的好过一些，但我不明白的是，实际中库的升级还是比较经常的，似乎应该有相当数量的系统代码来跟踪库的升级才对。

## 几种动态链接方法的比较

在某些有趣的方面，UNIX/ELF 和 Windows/PE 动态链接是存在差异的。

ELF 的策略为每个程序使用单一的名字空间，而 PE 策略为每一个库使用一个名字空间。一个 ELF 可执行程序会列出它所需要的符号的列表和库的列表，但它不记录哪个符号在哪个库中。一个 PE 文件，会列出从每一个库中输入的所有符号。PE 的策略虽然稍微不那么灵活，但对无意欺诈的抵抗性更好一点。想象一下，一个可执行程序调用库 A 中的例程 AFUNC，和库 B 中的例程 BFUNC。如果新版本的库 A 中恰好也有一个自己的 BFUNC 例程，那么 ELF 程序就会使用库 A 中的 BFUNC 代替库 B 中的旧 BFUNC，而这种情况在 PE 程序中就不会发生。在很多大的库中这会是一个问题。一个不完全的解决方案就是使用很少有文档说明的 DT\_FILTER 和 DT\_AUXILIARY 域来告诉动态链接器这个输入符号是从哪个库里来的，这样链接器就会在搜索可执行程序和其余库之前先在那些指定库中搜索输入符号。DT\_SYMBOLIC 域告诉动态链接器首先搜索该库自己的符号表，这样别的库就无法屏蔽这种库内引用了（其实这并不总是人们想要的，想一想在上一章中提到的 malloc 修改技巧）。这些特定的方法降低了不相关库在无意中屏蔽了正确符号的可能性，但是它们不是（在第十一章中我们将看到 Java 具有的）分层次链接时名字空间的替代品。

ELF 策略比 PE 策略更努力的尝试去维护静态链接程序的语义。在一个 ELF 程序中，对于从领一个库中输入数据的引用，会自动被解析，而 PE 程序需要对输入数据进行专门处理。PE 策略在比较指向函数的指针值时会有麻烦，因为一个输入函数的地址实际上是调用它的 t hunk 的地址，并不是在另一个库中的函数的实际地址。而 ELF 以相同的方式处理所有指针。

在运行时，几乎所有的 Windows 动态链接器都在操作系统内核中，而 ELF 的动态链接器则完全作为应用程序的一部分运行，而内核只是将初始文件映射进来。Windows 的策略更快一些（也有待商榷），因为它在开始链接前不需要将动态链接器映射进来并重定位。ELF 的策略则肯定是灵活的多。因为每一个可执行程序都命名了要使用的“解释器”（现在总是名为 ld.so 的动态链接器）程序，不同的可执行程序可以使用不同的解释器而无须要求操作系统进行任何变更。在实际中，这就更容易让可执行程序支持多种版本的 UNIX，尤其在 Linux 和 BSD 上，可以通过一个链接到兼容库上的动态链接器来支持非本地的可执行程序。

## 练习

在 ELF 共享库中，库经常被链接为即使同一个库内部的例程间调用也要通过 PLT 进行并且在运行时才绑定它们的地址，这样做有用吗？为什么？

想象一个程序调用了一个共享库中的例程 plug()，然后程序员生成了一个使用那个库的动态连接而成的程序。稍后系统管理员发现，plug 作为例程名称有点愚蠢，于是安装了一个新版本的库并将名称更换为 xsazq。在下一次程序员运行这个程序的时候，会发生什么事情？

如果运行时环境变量 LD\_BIND\_NOW 被设置，ELF 动态加载器会在加载时绑定程序所有的 PLT 项。如果在前一个问题中 LD\_BIND\_NOW 被设置的话，情况又会如何呢？

微软通过在链接器中增加了一些额外功能，并使用现存操作系统的功能，实现了无须操作系统协助的懒惰过程绑定（Lazy Procedure Binding）。如果避免采用当前策略所使用的额外一级指针，那么要它提供对共享数据的透明访问，有多困难？

## 项目

为我们项目的链接器创建一个完全动态链接的环境是不现实的，因为动态链接的大部分工作是在运行时发生的，而不是链接时。创建一个共享库的大多数工作都已经在项目 8-3 中完成了（即创建 PIC 可执行程序）。一个动态链接的共享库，就是具有良好定义的输入输出符号列表和库依赖列表的 PIC 可执行代码。为了将文件标识为一个共享库，或者一个使用共享库的可执行程序，文件的第一行是：

LINKLIB lib1 lib2 ...

或者

LINK lib1 lib2 ...

libN 就是当前文件依赖的其它共享库。

项目 10-1：从项目 8-3 中的链接器版本开始，扩展该链接器以生成共享库或使用共享库的可执行程序。该链接器将输入文件的列表作为输入，并将它们合并为输出的可执行程序或库，或其它供搜索的共享库。输出文件包含一个带有已定义（输出的）和未定义（输入的）符号的符号表。重定位类型是对引用了输入符号的 PIC 文件和 AS4、RS4 的那些类型。

项目 10-2：写一个运行时绑定器，即解析一个使用共享库的应用程序的引用。它应当读取应用程序，然后读取必须的库，将它们重定位到不重叠的有效地址，然后创建一个逻辑上合并的符号表（你也许真的想创建一个这样的表，或者像 ELF 那样将每个文件一个的表组织成一个链表）。然后解析所有的重定位和外部引用。当你完成了这些，所有的代码和数据还应当被赋予内存地址，并且代码和数据中的所有地址都应当被重定位到分配的地址上。

# 第 11 章 高级技术

\$Revision: 2.1 \$

\$Date: 1999/06/04 20:30:28 \$

这一章描述了一些并不是在任何地方都适用的链接器技术。

## C++的技术

C++对链接器来说存在三个明显的挑战。一个是它复杂的命名规则，主要在于如果多个函数具有不同的参数类型则可以拥有相同的名称。name mangling 可以对他们进行很好的地址分配，所有的链接器都使用这种技术的不同形式。

第二个是全局的构造和析构代码，他们需要在 main 例程运行前运行和 main 例程退出后运行。这需要链接器将构造代码和析构代码片段（或者至少是指向它们的指针）都收集起来放在一个地方，一边在启动和退出时将他们一并执行。

第三，也是目前最复杂的问题即模板和“extern inline”过程。一个 C++模板定义了一个无穷的过程的家族，每一个家族成员都是由某个类型特定的模板。例如，一个模板可能定义了一个通用的 hash 表，则就有整数类型的 hash 表家族成员，浮点数类型的 hash 表家族成员，字符串类型的，或指向各种数据结构的指针的类型的。由于计算机的存储器容量是无穷的，被编译好的程序需要包含程序中用到的这个家族中实际用到的所有成员，并且不能包含其它的。如果 C++编译器采用传统方法单独处理每一个源代码文件，他不能确定是否所编译的源代码文件中用到的模板是否在其它源代码文件中还存在被使用的其它家族成员。如果编译器采用保守的方法为每一个文件中使用到的每一个家族成员都产生相应的代码，那么最后将可能对某些家族成员产生了多分代码，这就浪费了空间。如果它不产生那些代码，它就有漏掉某一个需要的家族成员的可能性存在。

inline 函数存在一个相似的问题。通常，inline 函数被像宏那样扩展开，但是在某些情况下编译器会产生该函数相反的 out of line 版本。如果若干个不同的文件使用某个包含一个 inline 函数的单一头文件，并且某些文件需要一个 out of line 的版本，就会产生代码重复的相同问题。

一些编译器采用改变源代码语言的方法以帮助产生可以被“哑”链接器 (dump linkers) 链接的目标代码。很多最近的 C++系统都把这个问题放到了首位，或者让链接器更聪明些，或者将程序开发系统的其它部分和链接器整合在一起，以解决这个问题。下面我们概要的看看后一种途径。

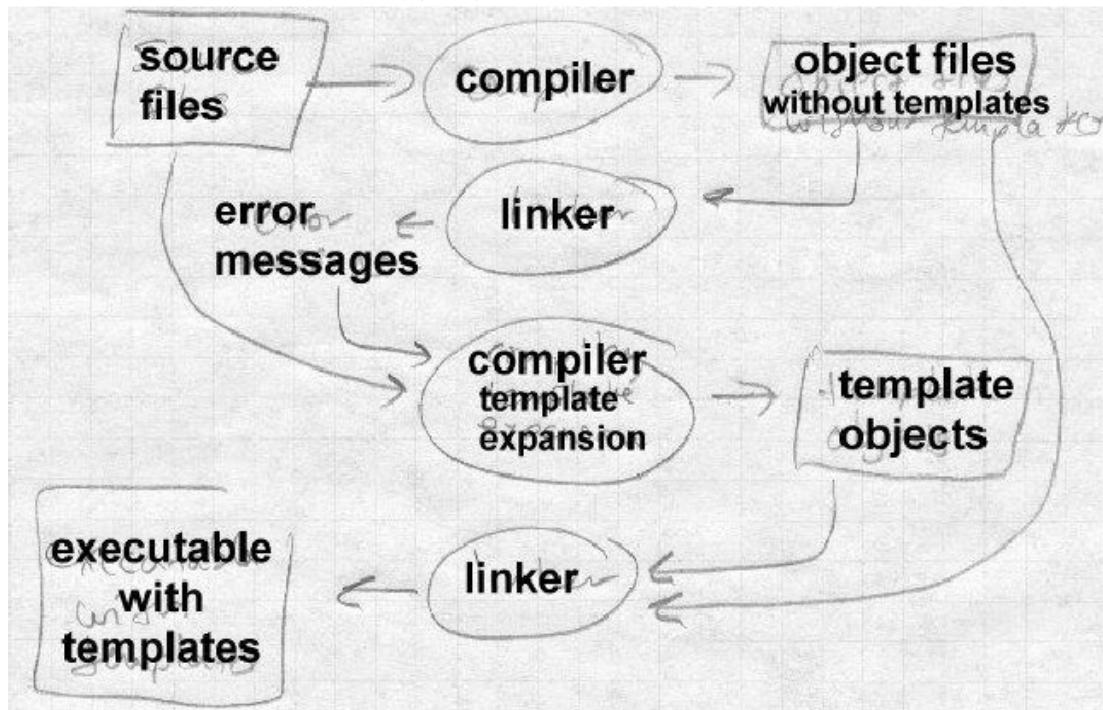
## 试验链接

对于使用“头脑简单”的链接器构建起来的系统，C++系统使用了多种技巧来使得 C++ 程序得以被链接。一种方法是先用传统的 C 前端实现来进行通常都会失败的试验链接，然后

让编译器驱动（运行各种编译器、汇编器、链接器代码片段的程序）从链接结果中提取信息，再重新编译和连接以完成任务。图 1

图 11-1：试验链接

输入文件传递给链接器以产生试验输出和错误信息，然后将输入文件和错误信息，可能还有更多产生的目标文件一起再传递给链接器以产生最终的目标文件。



在 UNIX 系统上，如果 linker 在一次链接任务中不能够解析所有的未定义符号引用，他可以选择仍然输出一个作为后续链接任务的输入文件的输出文件。在连接过程中链接器使用普通的库查找规则，使得输出文件包含所需的库，这也是再次作为输入文件所包含的信息。试验链接解决了上面所有的 C++问题，虽然很慢，但却是有效的方法。

对于全局的构造和析构代码，C++编译器在每一个输入文件中建立了完成构造和析构功能的例程。这些例程在逻辑上是匿名的，但是编译器给他们分配了可识别的名称。例如，GNU C++编译器会对名为 `junk` 的类中的变量创建名为 `_GLOBAL_.I._4junk` 和 `_GLOBAL_.D._4junk` 的构造例程及析构例程。在试验链接结束后，链接器驱动程序会检测输出文件的符号表并为全局构造和析构例程建立一个链表，这是通过编写一个由数组构成的队列的源代码文件来实现的（通过 C 或者汇编语言）。然后在再次链接中，C++的启动和退出代码使用这个数组中的内容去调用所有对应的例程。这和那些针对 C++的链接器的功能基本相同，区别仅仅是它是在链接器之外实现的。

对于模板和 `extern inline` 来说，编译器最初不会为他们生成任何代码。试验链接会获得程序中实际使用到的所有模板和 `extern inline` 的未定义符号，编译器驱动程序会利用这些符号重新运行编译器并为之生成代码，然后再次进行链接。

这里会有一个小问题是为模板寻找对应的源代码，因为所要找寻的目标可能潜伏在非常大量的源代码文件中。C 前端程序使用了一种简单而特别的技术：扫描头文件，然后猜测

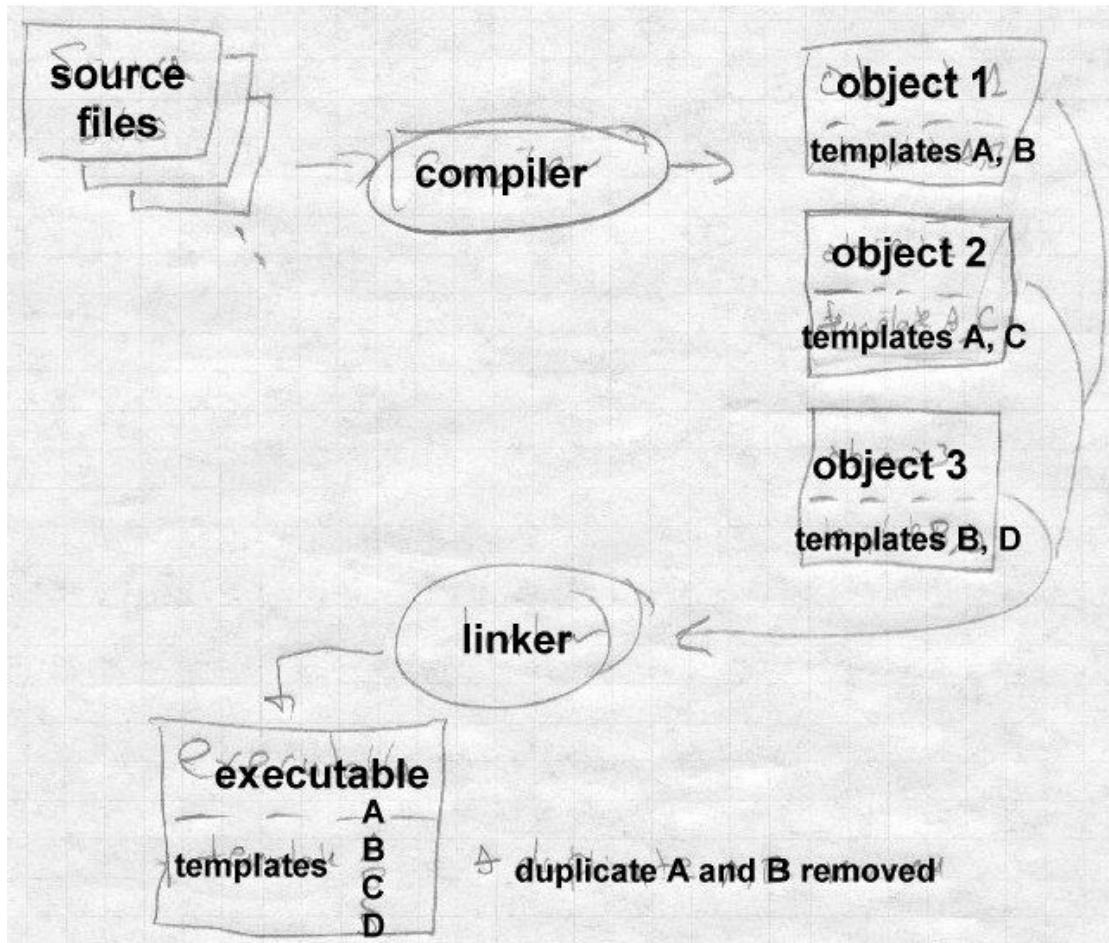
一个在 foo.h 中声明的模板会定义在 foo.cc 中。新近版本的 GCC 会使用一种在编译过程中生成，以注明模板定义代码的位置的小文件，称之为“仓库”（repository）。在试验链接后，编译器驱动程序仅需要扫描这些小文件就可以找到模板对应的源代码。

## 消除重复代码

试验链接的方法会产生尽可能小的代码，在试验链接之后会再为第一次处理遗留下的任何源代码继续产生代码。之所以采用这种前后颠倒的方法是为了生成所有可能的代码，然后让链接器将那些重复的丢掉。图 2，编译器为每一个源文件都生成了他们各自所需的每一个扩展模板和 extern line 代码。每一个可能冗余的代码块都被放到他们各自的段中并用唯一的名字来标识它是什么。例如，GCC 将每一个代码块放置在一个命名为 .gnu.linkonce.d.mangledname 的 ELF 或 COFF 段中，这里“缺损名称”（mangled name）是指增加了类型信息的函数名称。有一些格式可以仅仅通过名字就识别出可能的冗余段，如微软的 COFF 格式使用带有精确类型标志的 COMDAT 段来表示可能的冗余代码段。如果存在同一个名字的段的多个副本，那么链接器就会在链接时将多余的副本忽略掉。

图 11-2：消除重复

传递给链接器的文件中的重复段，经链接器处理后形成单一的段。



这种方法非常好的做到了为每一个例程在可执行程序中仅仅生成一个副本，作为代价，会产生非常大的包含一个模板的多个副本的目标文件。但这种方法至少提供了可以产生比其它方法更小的最终代码的可能性。在很多情况下，当一个模板扩展为多个类型时所产生的代码是一样的。例如，鉴于 C++ 的指针都具有相同的表示方法，因此一个实现了类型为<TYPE> 可进行边界检查的数组的模板，通常对所有指针类型所扩展的代码都是一样的。所以，那个已经删除了冗余段的链接器还可以检查内容一样的段，并将多个内容一样的段消除为只剩一个。一些 windows 的链接器就是这么做的。

## 借助于数据库的方法

GCC 所用的“仓库”实际上就是一个小的数据库。最终，工具开发者都会转而使用数据库来存储源代码和目标代码，就像 IBM 的 Visual Age C++ 的 Montana 开发环境一样。数据库跟踪每一个声明和定义的位置，这样就可以在源代码改变后精确的指出哪些例程会对此修改具有依赖关系，并仅仅重新编译和链接那些修改了的地方。

## 增量链接和重新连接

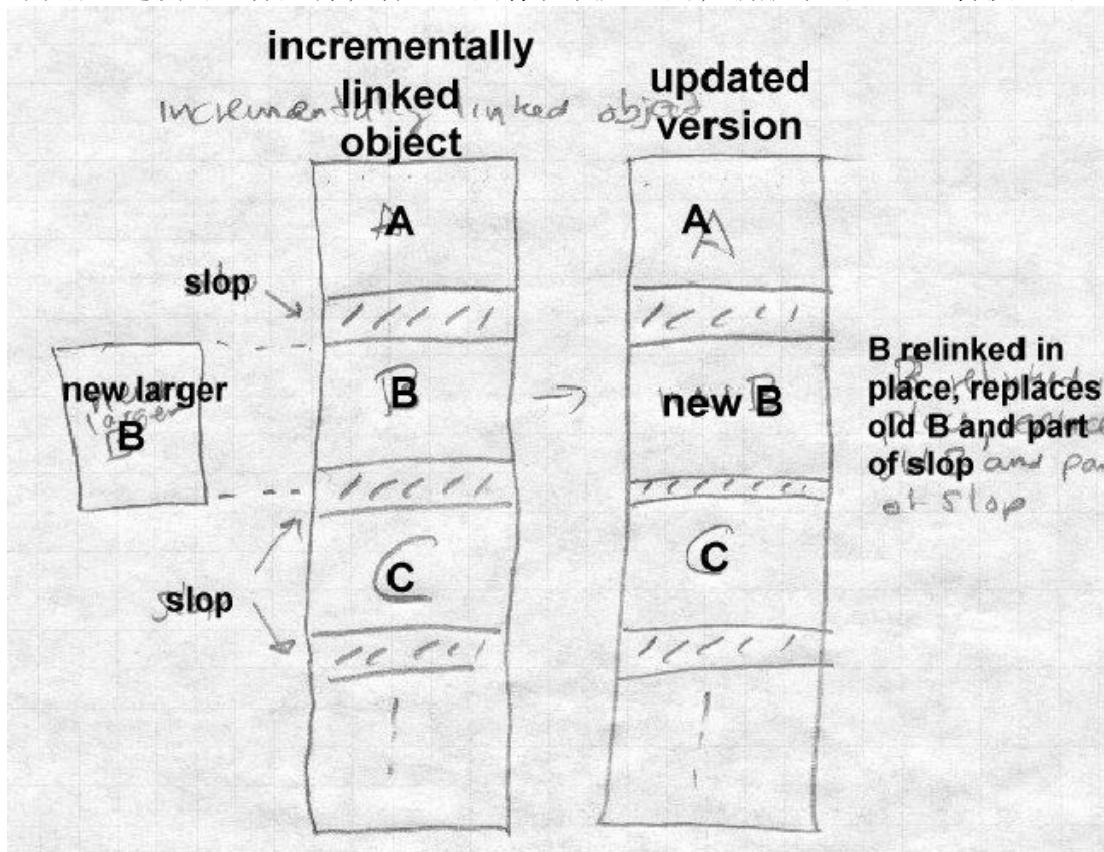
在很长一段时间里，有一些链接器都允许增量的链接和重新连接。UNIX 链接器提供了一个-r 参数告诉链接器在输出文件中保存符号和重定位信息，这样输出文件就可以作为下一次链接的输入文件了。在 IBM 的目标代码格式中，每个输入文件中的段（IBM 称这些段为控制段或 CSECT）在输出文件中都保存着他们各自的标识符。一个人可以重新编辑一个被链接的程序，并替换或删除这些控制段。这个特性在 60 年代和 70 年代早期被非常广泛的应用，因为那时用手工方式来仅仅替换被重新编译过的 CSECT 段，并重新连接程序的努力相比于非常慢的编译和链接速度而言还是很值得的。被替换的 CSECT 段并不一定要和原先的大小一样，链接器会在输出文件中调整用来计算那些被移动了的 CSECT 段不同位置的重定位信息。

在 80 年代的中后期，Stanford 的 Quong 和 Linton 在一个 UNIX 链接器上试验了增量链接技术，以尝试加快编译—链接—调试这个循环的速度。他们的链接器第一次运行时，它链接了一个传统的静态连接的可执行程序，然后在内存中保存着程序的符号表，并作为一个激活的守护程序运行在后台。在下一次的链接中，他只处理那些被改变的输入文件，在输出文件中替换掉相应的代码，并且保持其它部分不变，而不会对已被移动的引用符号的地方进行修改。由于两次链接之间，被重新编译过的文件中的段大小变化不会太大，他们在建立输出文件的最初版本时会生成比输入文件应生成的段稍微多一点空间的段，如图 3。每一次后继的链接，只要被改变的输入文件的段不会增长到超过那个多出的空间，被改变的文件的段只需要替换掉输出文件中的前一个版本即可。如果它增长到超过了那个多出来的空间时，链接器会减少这个段在输出文件中后面紧跟着的若干个段中那部分多余空间，而将那些段向后移动，以让出空间来放置这个增大的段。如果需要被移动的段超过一个很小的数量限制后，那么链接器将放弃移动索性从头重新链接。

---

图 11-3：增量链接

图示为增量链接的目标文件，各段之间存在填充空间，新版本的段可以替换旧的段



作者对典型的开发活动中两次链接之间所需编译的文件个数和段大小的增量收集了相当多的数据。他们发现一般只有 1 到 2 个文件被改变，而段的大小紧紧增长几个字节。通过在每一个段之间多增加 100 字节的空间，他们几乎避免掉了所有的重新连接。他们还发现创建对于调试工作所需的符号表的工作量，和创建这些段的工作量几乎相当，因此就使用简单的技术来对符号表进行增量更新。

他们的性能测试结果是颇具戏剧性的，相对于传统链接所需的 20 到 30 秒，增量链接只需半秒即可完成。该方案的主要不足在于链接器需要大约 8MB 的内存空间保留输出文件的符号表和其它信息，在那个时候这可是相当多的内存啊（工作站也很少有超过 16MB 内存的）。

一些现代操作系统也采用与 Quong 和 Linton 相同的方法进行增量连接。微软 Visual Studio 的链接器在缺省情况下就采用增量连接。他会在模块之间多留出一些空间来，并可以在某些情况下，将一个被升级的模块从可执行程序的一个部位移动到另一个部位，并在老的地址上放置一些“粘合”代码（译者注：就是能够执行被移动到新位置的代码的代码，拗口啊）。

## 链接时的垃圾收集

List 和其它可以自动分配存储空间的语言提供垃圾收集的功能已经有几十年了，这是一种可以自动标识和释放不在被程序其它部分所引用的存储空间的服务。有一些链接器也提供类似的功能，从目标文件中去除无用的代码。

大多数程序的源代码文件和目标文件都包含有多于一个的例程。如果编译器在每个例程之间划分边界，那么链接器就能确定每一个例程都定义了哪些符号，哪些例程都引用了哪些符号。根本没有被引用的任何例程都可以被安全的忽略掉。每次当一个例程被忽略掉时，由于这个例程可能还引用了一些唯一被该历程引用的其它例程，而那些例程也会随后被忽略掉，因此链接器需要重新计算“定义/引用”表。

较早进行链接时垃圾收集的系统之一是 IBM 的 AIX（译者注：高级交互式可执行体，IBM 的 UNIX 操作系统）。XCOFF 格式的目标文件将每一个例程放入一个单独的段中。链接器就可以通过符号表的符号项知道每个段中定义了哪些符号，通过表中的重定位项知道哪些符号被引用了。

缺省情况下，所有的未引用例程都会被忽略掉，但是程序员可以通过链接器的开关参数告诉它不要进行任何的垃圾收集，或对特定的文件或段不进行垃圾收集。

有一些链接器，包括 Codewarrior，Watcom 和微软 Visual C++ 的最近版本，都可以进行垃圾收集。有一个可选的编译器开关可以创建使每一个例程都单独在一个段中的目标文件。链接器查找那些没有被引用的段，并删除它们。在大多数情况下，链接器会同时查找相同内容的多个例程（通常从我们上面提到的模板的扩展而来）并将多于的副本清除。

对可收集垃圾的链接器的一个替代方案就是更广泛的使用库。程序员可以将被连接到程序中的库转换为每个库成员只有一个例程的库，然后从这些库中进行链接，这样链接器可以挑选需要的例程而跳过那些没有被引用的例程。这种方法中最难的部分是重新处理源代码以将含有多个例程的源代码文件分割为很多只有单一例程的小文件，并为每一个都替换掉相应数据声明及从头文件中包含过来的代码，并在内部重新对各个例程命名以防止名字冲突（译者注：原先属于多个源代码文件中的本地例程，在划分为每个库成员一个例程的库的时候，这些本地例程名字在对外公开后很有可能存在名字相同的若干个例程，因此需要为避免名字冲突进行一些处理）。这样的结果是可以产生尺寸最小的可执行程序，相应的代价是编译和链接的速度非常之慢。这是一个很古老的方法，在 60 年代后期 DEC TOPS-10 的汇编器就可以直接产生被链接器当作一个可查询的库的具有多个独立段的目标文件。

## 链接时优化

在大多数系统上，链接器是在软件建立过程中唯一会同时检查程序所有部分的程序。这就意味着他可以做一些别的部件无法进行的全局优化，特别是当程序由多个使用不同语言和编译器编写的多个模块组成的时候。例如，在一个带有类继承的语言中，一个类的方法可能会在子类中被覆盖，因此对它的调用通常都是间接的。但是如果没有任何的子类，或者存在子类但是没有一个覆盖了这个方法，那就可以直接调用这个方法。链接器可以对这种情况进行特殊优化以避免面向对象语言在继承时的低效率。Princeton 的 Fernandez 曾经写过一个针对 Modula-3 的优化链接器，可以将 79% 的间接方法调用转换为直接调用，同时减少了 10% 的执行指令。

一种更激进的方法是对整个程序在链接时进行标准的全局优化。Srivastava 和 Wall 编写过一个优化链接器，可以将 RISC 体系结构的目标代码反编译为一种中间格式的数据，并对之实施诸如 inline 这样的高层次优化或诸如将一个更快但限制更多的指令替换为一个稍

慢但常用的指令的低层次优化，然后再重新生成目标代码。特别是在 64 位体系结构的 Alpha 体系结构中，对静态或者全局数据，以及任意例程的寻址方法，是将指向地址池中某一项的地址指针从内存中加载到寄存器里，然后把这个寄存器作为基址寄存器使用（地址通过一个全局的指针寄存器来寻址）。他们的 OM 优化链接器会寻找多个连续指令引用一系列地址足够紧接的全局变量或静态变量的情况（这些全局变量和静态变量的彼此位置接近到足够可以通过同一个指针即可对他们寻址），然后重写目标代码以去除多余的从全局地址池中加载地址的指针。它也寻找那些通过分支跳转指令在 32 位地址范围内的过程调用，并将他们替换为需加载一个地址的间接调用。它也可以重新排列普通块的位置，使得较小的块排列在一起，这样以增加同一个指针被引用的次数。通过这些及其它的标准优化技术，OM 在可执行程序上实现了显著的提高，在一些 SPEC 寄存测试中总指令数减少了 11%。

Tera 计算机编译器工具采用了非常激进的链接时优化以支持 Tera 的高性能高并行度体系结构。C 编译器就是一个可以产生含有带标记的源代码的“目标文件”的解析器。链接器来解决各个模块之间的引用并生成所有的目标文件。鉴于代码生成器会同时处理整个程序，因此包括在单个模块中和多个模块之间，对很多过程调用它都是激进的以 inline 的方式来处理的。为了获得合理的编译性能，这个系统采用了增量编译和增量链接。在一次重新编译中，链接器会从上次得到的可执行程序的版本开始工作，重写那些对应源代码文件而发生了改变的代码（由于采用了优化和 inline 处理，可能由这些文件生成的代码会没有变化），并生成新的更新的可执行程序。Tera 系统中所有的编译和链接技术几乎没有什么是新的，但是对于生成数据的很多激进的优化技术它是独一无二的。

其它链接器都会对进行一些别的体系结构相关的优化。如多流的 VLIW 机器具有大量的寄存器，并且寄存器内容的保存和回复是一个主要的瓶颈。有一个测试工具会使用统计数据指出哪些例程会频繁的调用其它哪里例程。它修改了代码中所使用的寄存器以尽量减少例程调用者和被调用者之间重叠使用的寄存器数量，进而尽量减少了保存和恢复的次数。

## 链接时代码生成

很多链接器会生成少量的输出目标代码，例如 UNIX ELF 文件的 PLT（译者注：procedure linkage table）中的跳转项。但是一些实验链接器会产生比那更多的代码。

Srivastava 和 Wall 的优化链接器首先将目标文件反编译为一种中间格式的代码。多数情况下，如果链接器想要中间格式代码的话，他可以很容易的告诉编译器跳过代码生成，而创建中间格式的目标文件，让链接器去完成代码生成工作。上面这些确实是 Fernandez 优化器所描述的。链接器可以使用所有的中间格式代码，对其进行大量的优化工作，然后再为输出文件产生目标代码。

对于商业链接器有很多理由说明为什么它们根据中间格式代码进行代码生成。理由之一是中间格式代码的语言趋向于和编译器的目标语言相关。设计一种中间格式代码的语言以处理包括 C 和 C++ 在内的类 Fortran 语言并不是很难的事情，但是要设计既能处理那些语言又能处理诸如 Cobol 和 Lisp 这样鲜有共性的语言，那是一件相当难的事情。链接器通常都是链接从任何编译器和汇编器生成的目标代码，因此使其和特定语言关联起来是会有问题的。

## 链接时统计和工具

有一些小组曾编写过链接时统计和优化的工具。华盛顿大学的Romer等人编写了一个在windows x86下运行的工具Etch。它分析ECOFF格式的可执行程序，查找主程序及其所调用的动态链接库中所有的可执行代码（一般都是和数据混合在一起的）。它被用来建议一个图形化的调用统计描述和指令调度描述。ECOFF可执行体的格式和x86指令编码的复杂度是创建Etch的主要挑战。

DEC的Cohn等人曾写过一个名为Spike的软件，这是针对基于Alpha处理器Windows NT执行程序的优化工具。它既向可执行程序和动态链接库中增加统计功能代码，又可以使用这些统计数据提高寄存器分配并重新组织可执行程序以提高缓冲区访问，这样达到优化的目的。

## 链接时汇编

在链接传统二进制目标代码和链接中间格式语言之间有一个有趣的妥协就是将汇编语言的源程序作为中间格式的目标语言。链接器同时将整个程序汇编以生成输出文件。作为Linux灵感来源的MINIX（一种类UNIX的小操作系统）就是这么做的。

汇编语言足够接近于机器语言因此任何编译器都可以生成它，并且它也足够高级到可以进行一些有用的优化，包括无用代码消除、代码重组、一些有力的代码缩减，以及诸如对某一操作在确保足够操作位数的前提下选择最小版本指令的标准汇编优化。

由于汇编的执行速度很快，因此这样的系统可以很快的执行，尤其是当目标语言是一种被进行了标识的汇编语言而不是完全的汇编源代码时（这是因为像在其它编译器中一样，在汇编中最初添加标识的过程是整个处理过程中最慢的部分）。

## 加载时代码生成

有一些系统将代码生成从程序链接时推迟到了程序加载时。Franz和Kistler曾经创建过“Slim Binaries”，最初对应于Macintosh的可以同时包含老式68000 Mac和更新的Power PC Mac两种目标代码的“fat binaries”。slim binary实际上就是程序模块抽象分析的压缩编码。程序加载器读取和展开slim binary并为内存中的模块生成稍后可执行的目标代码。slim binary的发明者似是而非的声称现代CPU的速度非常之快，因此程序加载的时间主要取决于磁盘I/O，对于代码生成阶段甚至也是如此，由于slim binary的磁盘文件通常很小，因此它的加载过程要比标准二进制文件快一些。

Slim binary最初是为支持Oberon而创建的，这是一种类似Pascal的强类型语言，运行在Macintosh和稍后基于x86平台的Windows上，并且显然在这些平台上它们工作的很好。它的作者也希望slim binary可以针对其它程序语言和体系结构上同样出色的工作。这种说法是不可信的；Oberon程序是因为它的强类型和一致的运行环境才获得了很好的可移植性，而已支持的三种目标机器（译者注：指的是M68K，Power PC和x86）出了x86上的字节顺序

外，在数据和指针格式方面都非常接近。追溯到 50 年代 UNCOL 计划的“通用中间格式语言（universal intermediate language）”项目中一系列针仅针对少数源代码和目标语言的失败的尝试结果，我们没有理由认为 slim binary 不会重蹈覆辙。但是作为一些相似运行环境的程序发行格式，例如基于 68K 或 PPC 的 Macs，或基于 x86、Alpha 或 MIPS 的 Windows，它会工作的很好。

IBM 的 S/38 和 AS/400 为了在不同硬件体系结构的机器之间提供二进制软件兼容性，他们已经使用类似的技术很多年了。为 S/38 和 AS/400 所定义的机器语言实际上基于带有大量单级地址空间，不会在硬件上真正实现的虚拟体系结构。当一个 S/38 或 AS/400 二进制程序被加载时，加载器将虚拟代码翻译为机器上实际运行的处理器所针对的机器代码。被翻译后的代码被缓冲起来以备下次该程序运行时加快加载速度。这使得 IBM 可以在升级了 S/38 的软件后，从带有多个 CPU 板的中等规模系统到运行单个 Power PC CPU 的桌面系统的 AS/400 产品线都可以同样确保软件的二进制兼容性。如果没有一个可以完全控制这种虚拟体系结构和它所运行的所有计算机模型的统一厂商的话，那么这种策略很可能无法实现。但是这是一种从定价适当的硬件获得不菲性能的有效方法。

## Java 链接模式

Java 编程语言的加载和链接模式颇有趣，也很老道。Java 源程序是一种语法上和 C++ 很相似的强类型面向对象语言。使它变得有趣的是 Java 也定义了一个可以之的二进制目标代码格式，通过虚拟机可以运行这种二进制格式的可执行程序，它的加载系统允许一个 Java 程序在运行中向它自己增加代码。Java 通过类来组织程序，程序中的每一个类都编译到一个单独的逻辑（而通常也是实际上的）二进制目标代码文件中。每一个类定义其类成员所在的域，通常可能是一些静态变量，或一系列对类成员进行操作的例程（方法）。Java 采用单一继承，因此每一个类都是别的某个类的子类，所有的类都是通用基类对象的子孙。一个类从它的父类继承所有的域和方法，并且可以增加新的域和方法，也可以覆盖在父类中已经存在的方法。

Java 每次加载一个类。一个 Java 程序最开始会以一种依赖于具体实现的方法加载一个最初的基类。如果这个类引用了别的类，则那些类也会在需要的时候被加载。一个 Java 应用程序可以使用内建的自举类加载器从磁盘上的文件中加载类，也可以使用自己的类加载器以任何自己想采用的方法来创建和恢复类。多数情况下一个定制的类加载器通过网络链接获得类文件，同样也可以在运行时生成代码，以及从压缩或加密的文件中提取代码。当一个类由于其它类的引用而被加载时，系统会使用与加载引用它的类相同的加载器。每一个类加载器各自具有独立的名字空间，因此即使一个从磁盘上运行的应用程序和另外一个从网络上运行的应用程序具有相同的类或类成员名称，也不会发生名字冲突。

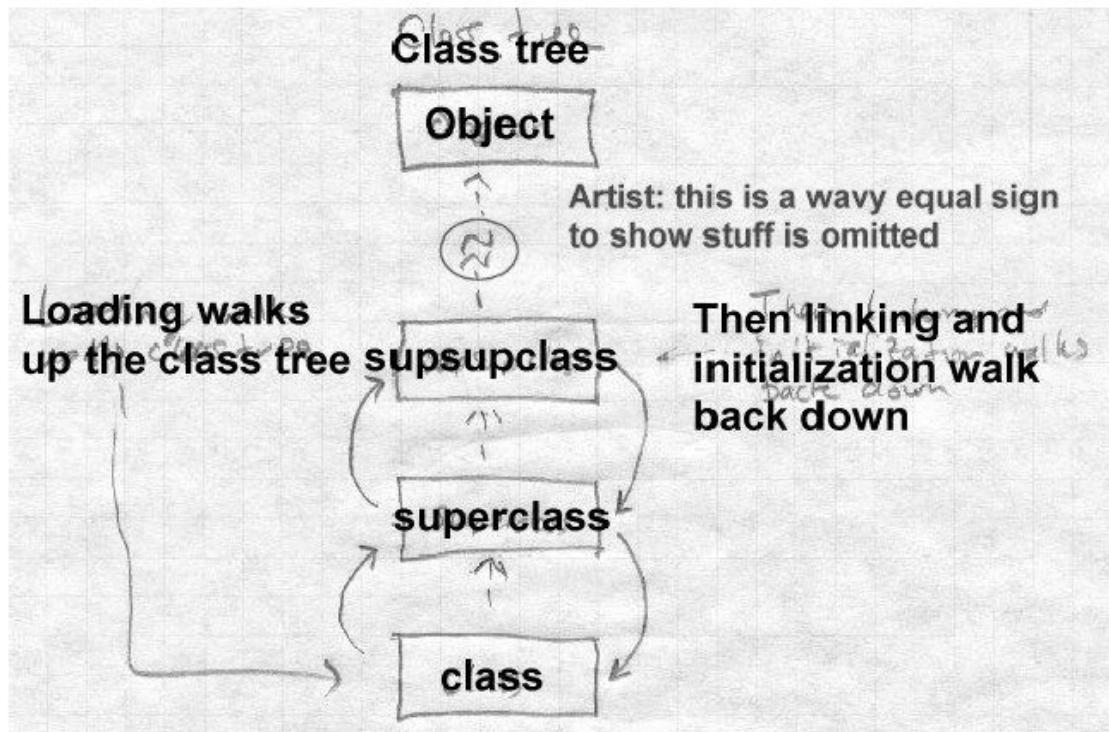
Java 相当详细的定义了加载和连接过程的规范。当虚拟机需要使用一个类时，首先它通过调用类加载器加载这个类。当类被加载后，就是链接阶段，包括验证二进制代码是否有效的验证过程和分配类的静态域的准备阶段。最后一步是初始化，当一个类的实例第一次创建时或该类的静态函数第一次运行时，会运行所有初始化静态域的例程。

## 加载 Java 类

加载和链接是两个独立的过程，因为任何类在开始链接前都需要确认是否它的所有父类都已被加载并链接好了。这意味着整个过程在概念上会遍历一次类的继承树，如图 4。加载过程从根据类的名字调用 classLoader 过程开始。类加载器处理类的数据，然后调用 defineClass 将数据传递给虚拟机。defineClass 分析类文件并检查一系列的格式错误，如果发现任何错误就生成一个例外消息。它也会查看这个类的父类，如果它的父类没有被加载，还要调用 classLoader 递归的加载该类的父类们。当调用返回时，父类就被加载和链接好了，这时 Java 系统继续从这里开始链接当前的类。

图 11-4：加载和链接 Java 类文件

向上和向下遍历树



下一步就是确认，进行一系列的静态正确性检查，例如确保每一个虚拟指令都有一个正确的操作码，每一个分支的目标都是有效的指令，每一个指令都能正确处理所引用数值的类型。这些检查在程序运行后可以不必再进行因此可以提高程序执行的速度。如果确认时发现了错误，它会产生一个例外消息。然后准备阶段会为类中所有的静态成员分配存储空间，然后将它们初始化为标准的缺省值，一般都是 0。大多数 Java 实现会在这时创建一个方法表，它包含着指向该类及其从父类继承来的所有方法的指针。

Java 链接的最后一步就是解析，相当于其它语言的动态链接过程。每一个类包含一个常量池 (constant pool)，其中既有诸如数字和字符串这样的常规常量，也有对其它的类的引用。所有在编译好的类中的引用，甚至是针对其父类的，都是符号连接，并在这个类被加载后进行解析（它的父类可能会在它被加载后被修改并重新编译，只要它可通过某种方法

保持引用的域和方法仍有定义，那他们就仍是有效的）。Java 规范允许具体实现从确认后到指令实际使用某个引用前的任何时候对引用进行解析，如调用父类或其它类中定义的函数。如果不考虑实际解析引用的时间的话，那么一个失败的引用只有在它被使用时才会导致例外发生，因此程序的行为就好像 Java 使用了 Just-In-Time 的“懒惰”解析策略。这种在解析时的灵活性允许多种可能的实现方案。这样在将类翻译为本地机器码时就可以立即解析所有的引用了，包括当一个引用不能解析时所要跳转到的例外处理历程。一个纯解释器会像解释代码时那样解析这个引用而不是束手无策。

加载和链接的设计所带来的影响是类可以按需加载和解析。Java 的垃圾收集策略像对其他数据那样应用于类，只要一个类的所有引用都被删除了那么这个类就会被卸载。

Java 的加载和链接模式是我们在这本书里见到的最复杂的。但 Java 尝试去满足一些对立的目标，可移植的类型安全的代码和合理快速的可执行程序。这里的加载和链接模式支持增量加载，支持多数类型安全标准的静态确认，允许那些想让程序运行的更快的系统将类翻译为机器码。

## 练习

你所使用的链接器将一个大程序完全链接好需要多长时间？使用工具检查一下你的链接器是怎么花费掉这些时间的（甚至在没有链接器源代码时，你仍可以通过跟踪系统调用来获得一些不错的思路）。

看看一个 C++ 或其它面向对象的编译器生成的代码。链接时优化可以对其改进多少呢？为了让链接器做一些有趣的优化，编译器在目标模块中都放入了什么信息？共享库让这个计划变得有多糟糕？

概括出一个你喜欢的 CPU 的被标识的汇编语言作为一个目标语言。处理程序中的符号有什么好办法吗？

AS/400 使用二进制翻译提供了在不同模式机器之间的二进制代码兼容性。包括 IBM 360 /370/390，DEC VAX 和 Intel x86 使用微码（microcode）来实现不同硬件上的相同指令集。AS/400 所采用策略的优势在哪里？采用微码的优势在哪里？如果你定义了一个今天的计算机体系结构，你会采用哪种方案呢？

## 项目

项目 11-1：为链接器增加一个垃圾收集器。假定每一个输入文件有多个命名为 .text1, .text2 等的文本段。利用符号表和重定位项建立一个全局的“定义/引用”数据结构，并标明那些没有被引用的段。你必须增加一个命令行参数去将启动例程设置为被引用的（如果不这么作会发生什么呢？）。在垃圾收集器运行后，更新这些段的位置好将被删除的段的位置挤压掉。改进这个垃圾收集器是他可以反复运行。每一次运行后，更新“定义/引用”数据结构去除那些逻辑上已经被删除的段，然后再次运行垃圾收集器，直到不再有任何东西被删除掉为止。