

Linkers & Loaders

翻译校对版 Beta1

第 0 章 引子 5

献给.....	5
介绍.....	5
本书的目标读者是哪些人？	5
章节摘要.....	6
工程项目.....	7
致谢.....	7
联系我们.....	8
第 1 章 链接和加载.....	9
连接器和载入器是做什么的呢?.....	9
地址绑定：从一个历史的角度.....	9
连接和载入.....	11
编译器驱动.....	15
连接：一个真实的例子.....	16
练习.....	19
第 2 章 体系结构的问题.....	20
应用二进制界面.....	20
内存地址.....	20
地址信息.....	22
指令格式.....	22
过程调用和可寻址性.....	23
数据和指令引用.....	25
SPARC	27
分页和虚拟内存.....	30
Intel 386 分段	35
嵌入式体系机构.....	36
练习.....	37
第 3 章 目标文件.....	39
目标文件中都有什么?.....	39
空目标格式: MS-DOS .COM 文件	40
代码区: Unix a.out 文件.....	40
重载: MS-DOS EXE 文件.....	46
符号和重载.....	47
可重载 a.out.....	47
IBM 360 目标格式	58
微软可移植执行体格式.....	61
Intel/Microsoft 的 OMF 文件格式.....	67
不同目标格式的比较.....	71
项目.....	71
练习.....	72
第 4 章 存储空间分配.....	73
段和地址.....	73
简单存储布置.....	73
多种段类型.....	74

段和页面对齐.....	76
公共块和其他特殊的段.....	76
连接器控制脚本.....	83
嵌入式系统存储分配.....	84
实际的存储分配.....	85
练习.....	89
项目.....	90
第 5 章 符号管理.....	91
绑定和符号解析.....	91
符号表格式.....	91
名字调整(搅拌).....	97
弱外部和其他种类的符号.....	100
维护调试信息.....	100
练习.....	102
项目.....	102
第 6 章 链接库.....	104
连接库的目的.....	104
库文件格式.....	104
建立库文件.....	108
搜索库文件.....	109
性能问题.....	110
弱(隐式)外部符号(Weak external symbols).....	110
练习.....	111
项目.....	112
第 7 章 重定位.....	114
硬件和软件重定位.....	114
链接时重定位和加载时重定位.....	115
符号和段重定位.....	115
基本的重定位技术.....	116
可重链接和重定位的输出格式.....	120
其它重定位格式.....	121
特殊情况的重定位.....	122
练习.....	123
项目.....	123
第 8 章 加载和重叠.....	125
基本加载.....	125
带重定位的基本加载.....	126
位置无关代码.....	126
自举加载.....	133
树状结构的覆盖.....	133
练习.....	139
项目.....	139
第 9 章 共享库.....	141
绑定时间 (Binding time)	142

共享库的结构.....	144
创建共享库.....	144
使用共享库链接.....	147
使用共享库运行.....	147
malloc hack 和其它共享库问题	148
练习.....	150
项目.....	151
第 10 章 动态连接和装载.....	152
ELF 动态连接.....	152
ELF 文件内容.....	152
装载一个动态连接程序.....	155
运行时的动态连接.....	160
Microsoft 动态连接库.....	160
OSF/1 准共享库	164
让共享库加速.....	165
几种动态连接方法的比较.....	166
练习.....	166
工程.....	167
第 11 章 高级技术.....	168
C++的技术	168
增量链接和重新连接.....	172
链接时的垃圾收集.....	173
链接时优化.....	174
链接时代码生成.....	175
Java 链接模式.....	177
练习.....	179
项目.....	179

第 0 章 引子

\$Revision: 2.2 \$
\$Date: 1999/06/09 00:48:48 \$

献给

我的家人， 托尼娅和莎拉。

介绍

只要有计算机，就会有连接器和加载器， 因为它们是非常重要的工具。它们不但可以处理单一源文件，更可以处理多个输入模块，从而产生目标代码。

早在 1947 年，程序员就开始使用加载器处理存放在分离磁带上上的代码，并且最终把它们整合，重定位在一个目标文件里。到了上世纪 60 年代早期，这些加载器就发展的比较完善了。因为那时内存仍旧很贵、很有限并且计算机的速度也比较慢(以现代计算机的速度比较)，连接器为了让大程序加载到较小的内存空间中，就建立了复杂的内存布局，为了让以前编译过的文件节省重构时间，它也做了一些优化。

上世纪 70 到 80 年代，连接技术几乎没有什么大的改进。连接器趋向于变得更加简单，虚拟存储技术的出现，把很多存储管理方面的工作从应用程序层面移到了操作系统里面，随着计算机速度越来越快，磁盘容量越来越大，通过替换少量模块的方式重新建立被链接程序要比仅仅链接程序中被修改的部分更加容易（译者注：重新建立这里提到了两种方式：1，仅仅将修改过的部分编译后重新连接到程序中；2，将需要替换的模块重新编译后替换原先的模块。就这两种方法而言，后者更加容易一些）。到了 90 年代，连接器又变得复杂起来，增加了一些对当时流行特性的支持，如动态连接、共享库以及 C++。一些拥有扩展指令字长和编译器显式调度（译者注：在 IA64 中确实存在此类方法，即在编写程序时向编译器显式的指定某些调度策略，从而优化程序运行，例如可以在程序中调用编译器的功能指定更高层次的流水线，我想这里可能就指的此类意思。）内存访问方式特性的新处理器架构，诸如 IA-64，将对连接器提出新需求，以确保目标文件中出现所需的代码。

本书的目标读者是哪些人？

本书预计供下述几种读者使用。

- 学生：正在学习编译器构造课程，而且粗略地学习过操作系统但缺乏对连接和装载过程的认识，往往认为连接过程琐碎或是浅显。虽然说过去 Fortran, Pascal, 和 C 语言吃香的时候，操作系统并不使用内存交

换技术或共享库，但现在这个论调越来越偏离实际情况了。 C++, Java 以及其他面向对象等语言需要一个更加完善的连接环境。对可执行程序的内存交换，共享库，动态连接等技术影响操作系统的各个方面，操作系统设计人员忽视连接问题将承担巨大的风险。

- 有经验的程序员同样应该了解连接器如何运作，特别是在新兴语言中。 C++在连接器中允许安插特殊请求，在大型 C++程序连接过程中发生意外事件时，（最常见的情况是静态构造函数没有按照程序员预计的次序执行）就更容易进行漏洞的硬性诊断。连接器中象共享库和动态连接这类功能如果使用得当的话，能够提供巨大的弹性和能量。
- 语言设计和开发人员应该了解在编译各种语言和编译器时，连接器如何运作以及连接器的作用。由手工编写程序历时 30 年之久，现在 C++依靠连接器处理细节可以自动生成代码。（想象一下程序员在 C 语言中获得类似 C++模板所需工作量，或者确保初始化例程在上百个 C 源文件中的程序体开始前被调用所需工作量）未来程序语言借助功能强大的连接器，自动化程度会更高，不只局限在程序范围内的任务收集。连接器在处理全局程序优化时会更棘手，因为在编译过程中唯有连接器把整个程序代码集中处理，也只有连接器能够做出可以影响程序整体的改动。

（编写连接器的人员当然都需要本书。但是全球所有的连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，连接和装载，对连接的过程进行了一个简短的历史回顾，并论述了连接过程中的各个阶段，本章用一个连接器运行实例来结束，简短而完整，由输入目标文件开始，直到可执行的“Hello, world”程序。

第 2 章，架构讨论，以连接器设计的角度来回顾计算机架构。例举了 SPARC，一个有代表性的精简指令集架构，IBM 360/370，一个古老但仍旧十分有生机的寄存器-内存架构。还有自成体系的 Intel x86。重要的架构特征包括内存结构，程序寻址方式和单个指令内地址域的布局。

第 3 章，目标文件，分析研究目标文件和可执行文件的内部结构。本章由最简单的文件，MS-DOS 的 COM 文件开始，进一步分析更复杂的文件：DOS 下的 EXE，Windows 下的 COFF 和 PE (EXE 和 DLL)，Unix 下的 a.out 和 ELF，还有 Intel/Microsoft 的 OMF。

第 4 章，存贮器分配，涵盖了连接过程的首阶段，使用实际的连接器操作实例，为进行连接的程序分配多个内存段。

第 5 章，符号管理，涵盖符号的装订和解析，在某文件中对某名字的符号访问在其他文件中被解析为某个机器地址的过程。

第 6 章，库，涵盖目标代码连接库的创建和使用，以及库的结构和性能问题。

第 7 章，重定位，涵盖地址重定位，将程序内的目标代码影射到运行时的实际地址的过程。本章同时涵盖位置无关代码 (PIC)，这种代码使用避免重定位的方法建立，同时说明了位置无关代码的代价和优势。

第 8 章，加载和内存布局，包括加载过程，如何把程序文件装入计算机内存并且运行。本章同时说明了树-结构的重复占位(程序)段，一种古老但十分有

效的地址空间保存技术。

第 9 章，共享库，着重介绍在不同程序中如何共享一个共享库代码拷贝的方法。本章的焦点集中在静态连接共享库上。

第 10 章，动态连接和装载，将第 9 章的讨论延伸至动态连接共享库。本章使用了两个详细的实例进行论述，Windows32 的动态连接库（DLLs），和 Unix/Linux 的 ELF 共享库。

第 11 章，高级技巧，着眼于成熟的现代连接器各种变化。包括 C++需要的新功能，如“name mangling”，全局构造函数和析构函数，模板扩充，和重复代码的消除。其他技巧，如增量连接，连接时碎片收集，连接时代码生成和优化，装载时代码生成，整形工作和工具。本章由 Java 连接器模型的总结结束，这个模型比其他连接器所包含的语意复杂性要高的多。

第 12 章，参考，附说明的参考书目。

工程项目

从第 3 章到第 11 章有一个持续的工程项目，开发一个小型使用的 perl 语言连接器。尽管 perl 不象是真正的连接器所实现的语言，这仍旧是一个极好的学间项目。Perl 处理许多低层设计细节，在设计类似 C 或 C++的时候这些细节会出现并阻滞项目顺利进行，这样学生可以把精力集中在手头项目的算法和数据结构上。Perl 在当今大多数计算机上可以免费使用，包括 Windows 95/98/NT，Unix/Linux，而且还有很多适用于初学者学习 perl 语言的优秀书籍。（参考第 12 章的书目）

在第 3 章里，项目初期建立了一个连接器的框架，它可以读写一个简单而完整的目标格式的文件，在后续章节中陆续为连接器增加功能，最终成为支持共享库并产生可动态连接的目标文件的完备连接器。

perl 完全有能力处理任意的二进制文件和数据结构，如果愿意的话，这个项目的连接器能够适应处理主流的目标格式。

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 Davi

d 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摩根卡夫曼的两位编辑, 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 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 \$

连接器和载入器是做什么的呢？

任何连接器和载入器的基本任务都非常简单：将相对抽象的名字和相对精确的名字绑定起来，好让程序员能用更抽象的名字书写程序。也就是把程序员写的诸如 `getline` 等的名字绑定成模块 `iosys` 中的可执行代码的第 612 个字节的位置。或把一个较抽象的地址，如这个模块的静态数据部分的第 450 个字节的地址转换成一个数字地址。

地址绑定：从一个历史的角度

要知道连接器和载入器的内部是如何工作的，一个有用的方法就是看一看在他们在计算机编程系统中起的作用。

最早的计算机完全是用机器语言进行编程的。程序员需要在纸上写下程序的符号，手工翻译成机器代码，然后用开关输入到计算机中，或者通过在纸带或卡片上打孔输入（真正的高手都是直接用开关遍程序）。如果程序员完全使用符号地址，需要手工将符号翻译成地址。如果一条指令需要加入程序，或需要删除一条指令，整个程序都需要重新手工检查任何受到影响的地址。

问题在于名字和地址的绑定过早。汇编语言让程序员用符号写程序，将名字和机器地址绑定在一起，解决了这个问题。如果程序发生了变化，程序员需要重新编译，但工作已经由程序员转换到计算机来完成。

代码库增加了地址转换问题的复杂性。计算机能够完成的基本操作非常简单，真正有用的程序都是由完成高级复杂功能的子程序组成。安装的计算机都包括一系列预先编写调试的子程序，程序员可以在他们的基础上写出自己的程序，而不需要自己写出所有的子程序。程序员将子程序装载到主程序中来完成一个复杂的工作。

程序员们甚至在使用汇编语言之前就使用了子程序。在 1947 年，领导 ENIAC 项目的 John Mauchly，写文章描述了如何把磁带中的子程序和程序一起装载到计算机中，同时需要将子程序重载以反映他们装载的地址。令人惊奇的是，尽管 Mauchly 认为程序和子程序都是由机器语言编写，重载和库搜索这两个连接器的基本功能甚至比汇编器出现的还要早。重载载入器让子程序的作者和用户可以各自写程序，并假定他们都是从地址零开始的，只有当一个主程序和子程序连接在一起的时候再完成地址的翻译。

随着操作系统的诞生，和连接器分离的重载载入器以及程序库成为必需。在操作系统出现之前，每个程序都可以支配整个机器的所有内存，所以程序可以在固定的内存地址汇编和连接，并认为计算机的所有地址都可以使用。但在操作

系统之下，程序需要和操作系统共享计算机内存，甚至还要和其他程序分享。这意味着程序的真实运行地址只有到程序装载到内存的时候才能知道，将最后的地址绑定从连接过程推迟到载入过程。至此，连接器和载入器将工作分解开来，连接器完成地址绑定，完成每个程序中的相对地址解析，载入器完成最终的实际地址翻译。

随着计算机系统变得越来越愈复杂，越来越多的复杂的名称管理和地址绑定需要由连接器完成。Fortran 程序使用多个子程序和公共块技术，就是一段数据由多个子程序共享，而这些都需要连接器为子程序和公共块分配存储空间和地址。连接器越来越需要和目标代码库打交道，包括由 Fortran 和其他语言写的库，编译器也支持在编译的代码中直接调用库函数来完成 I/O 和其他高级操作。

程序很快就变得无法在可用内存中装下，所以连接器提供了覆盖技术，允许程序员让不同部分共享同一块内存，覆盖的每个部分可以动态地按需要装入内存。覆盖技术在 60 年代随着磁盘的使用在主机系统中广泛应用，直到 70 年代出现了虚拟内存技术，然后重新在 80 年代的微型计算机中出现，直到 90 年代虚拟内存 PC 中的应用而消失。它们还使用在内存受限的嵌入系统中，而且也许还会由于需要精细的程序员或编译器通过提高内存的利用率以提高性能而出现。

随着硬件重载和虚拟内存技术的出现，连接器和载入器实际上变得不那么复杂了，因为每个程序又可以使用整个地址空间了。程序可以连接装载到固定的地址，并由硬件而不是软件完成装载时的重载。但是带有硬件重载功能的计算机都是运行多个程序，经常保留同一个程序的多个拷贝。当一台计算机运行一个程序的多个实例的时候，程序的有些部分在所有实例中都是一样的（尤其是可执行代码），其他的一些部分是每个实例所独有的。如果那些保持不变的部分可以和那些变化的分别开来，那么操作系统只需保留一份保持不变的部分的拷贝，节省可观的存储空间。编译器和汇编器就被修改成生成多段目标代码，其中一段就是只读的代码，其他的段是可写的数据部分，连接器需要能够把每种类型的段所有部分合并起来，使得连接的程序可以将所有的代码保留在一起，所有的数据在另一个地方。这个过程并没有让地址绑定发生的更晚，因为所有的地址仍然在连接过程中绑定，但更多的工作被推迟到载入器为每个部分分配地址的时候。

即使计算机中有很多不同的程序在运行，它们仍然需要共享很多代码。例如，几乎所有用 C 语言写的程序都要使用诸如 fopen 和 printf 这样的例程，数据库应用都需要使用一个大存储库来连接数据，那些运行在某种图形界面下的程序，如 X Windows，MS Windows 或 Macintosh 都使用某种图形库。现代系统都为程序提供“共享库”（Shared libraries），好让所有的程序使用的库在计算机中只需保留一份拷贝。这不仅提高运行时的性能，还能节省大量的磁盘空间，在很多小程序中，公共子例程往往比主程序占用更多的空间。

在比较简单的静态库中，当建立库的时候，每个库都绑定到特定的地址，连接过程中连接器把程序中指向这些库的地方绑定到特定的地址。静态库非常不灵活，因为当库发生任何变化后，所有程序都需要重新连接，建立静态库的过程也非常乏味。又出现的可以动态连接的库，库的段和符号直到使用库的程序运行的时候才绑定实际的地址。有时候，绑定甚至更加靠后，使用完全动态联编，只有当被调用过程第一次调用的时候再被绑定地址。更进一步，程序可以在运行的时候绑定库，在程序执行之中将库调入。这种技术提供了一种强大和高效的扩展程序功能方法。微软的 Windows 利用这种共享库的运行时载入技术（即 DLL，动

态连接库)来构建和扩展程序。

连接和载入

连接器和载入器完成几个相关但概念上分离的工作。

- **程序载入:**将程序从次存储设备(自 1968 年后就特指磁盘)拷贝到主内存中等待运行。在某些情况下,载入仅仅是将数据从磁盘拷入内存;在其他情况下,还包括分配存储资源,设置保护位或把虚拟地址和磁盘对应安排虚拟内存。
- **重载:**编译器和汇编器通常把程序的每个目标文件的起始地址设定为零,但很少有计算机允许程序装载在地址零的位置。如果一个程序由不同的子程序组成,所有的子程序必需装载到非覆盖地址中。重载就是为程序的不同部分分配装载地址的过程,修改程序中的代码和数据来反映被分配的地址。在很多系统中,重载需要发生多次。连接器经常要从多个子程序中建立一个程序,建立一个从地址零开始的连接过的程序,并把子程序重载到大程序的不同地方。当载入程序时,系统选出实际装载地址,连接的程序整个重载到装载地址。
- **符号解析:**当一个程序是由多个子程序构建起来的时候,子程序之间的引用是用符号表示的;一个主程序也许会调用一个叫做 sqrt 的平方根子程序,而数学库定义了 sqrt。连接器需要解析出库中 sqrt 的地址,并将目标代码中的调用部分修改过来,以便调用指令指向该地址。

尽管在连接器和载入器中存在着相当大的功能重叠,将一个完成载入功能的程序定义为载入器,而将完成符号解析功能的程序定义为连接器仍然是合情合理的。两者都可以完成重载,实际上也有完成这三个功能的连接载入器。

重载和符号解析的区别有时候很模糊。由于连接器已经将引用解析成符号,一种处理代码重载的办法是为其分配一个基于程序的每个部分的基地址的符号,并将可重载的地址看做相对基地址的引用。

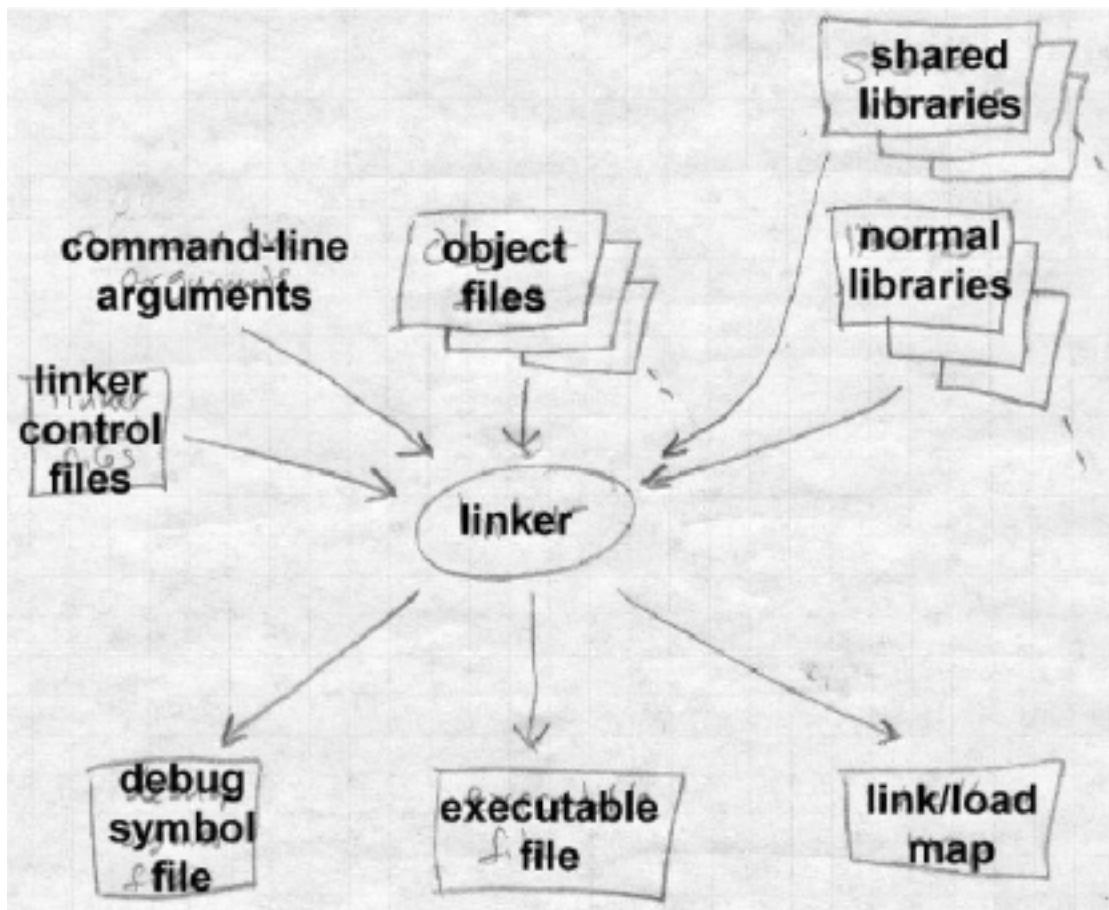
连接器和载入器共有的一个重要功能就是修改目标代码,也许是比调试器更广法使用并有此功能的程序。这是一个独特并强大的功能,虽然需要和特定机器的细节密切相关,如果出现错误会导致难以琢磨的 bug。

两遍连接

现在我们来看看连接器的基本结构。就象编译器和汇编器一样,连接基本上也是一个两遍过程。连接器将目标文件、库、有时候还有命令文件作为输入,然后将结果作为输入目标文件,有时候还包括补充信息,如载入表或一个包含调试符号的文件。图1。

图 1.1 连接过程

连接器接受输入文件,产生输出文件,有时候还有其他的文件。



每个输入文件都包括一系列段，将代码或数据放入输出文件的相邻的块中。每个输入文件至少包含一个符号表。有些符号在文件中定义为导出的，为在其他的文件中使用，一般都是可以在外部调用的例程的名字。其他的符号为导入的，需要在这个文件中使用但没有在这个文件里定义，一般都是要调用的但文件中没有定义的例程名字。

当连接器运行时，它首先会扫描输入文件，找出每个段的大小，收集所有符号的定义和引用。连接器会建立一个段表记录输入文件中所有的段，一个符号表记录所有导入或导入的符号。

利用第一遍扫面的得到的数据，连接器为每个符号分配数字地址，确定输出文件中段的大小和位置，计算出各个部分需要出现在输出文件的什么位置。

第二遍扫描利用在第一遍扫描中收集的信息来控制实际的连接过程。连接器读入并重载目标代码，把符号引用换成数字地址，调整代码和数据中的内存地址以反映重载后的段地址，将重载后的代码写入输出文件中。最后将输出文件连同头部信息，重载的段和符号表信息写入磁盘。如果程序使用动态连接技术，符号表中保留着运行时连接器在解析动态符号时需要的信息。在很多情况下，连接器会在输出文件中生成很少的代码或数据，如用来调用覆盖的或动态库中子例程的“胶水代码”，或在程序启动的时候用来初始化例程的指针数组。

不论程序中是否使用了动态连接技术，虽然输出文件本身没有使用，输出文件中仍然需要保存符号表，以另其他程序重新连接或调试。

有些目标代码是可重连接的，也即，一个连接器的输出文件可以作为下一个连接器的输入文件。这就需要输出文件包含一个和输入文件中类似的符号

表，还要包含其它输入文件具有的辅助信息。

几乎所有的目标格式都可以包含调试符号，当程序在一个调试器的控制下运行的时候，调试器可以让程序员用这些符号控制程序的运行，得到诸如行号，源程序用的变量名等信息。根据目标文件格式的不同，调试符号可以和连接器需要的符号放在同一个表中，也可以单独放在一个与连接器需要的符号表不同的调试符号表中。

只有很少的连接器使用一次扫描方法。在扫描过程中，它们会将输入文件的一些或所有信息缓冲到内存或磁盘中，过后在读入缓冲的信息。这些不过都是写实现的小技巧，并没有改变连接的两遍扫描的基础，我们以后不再描述。

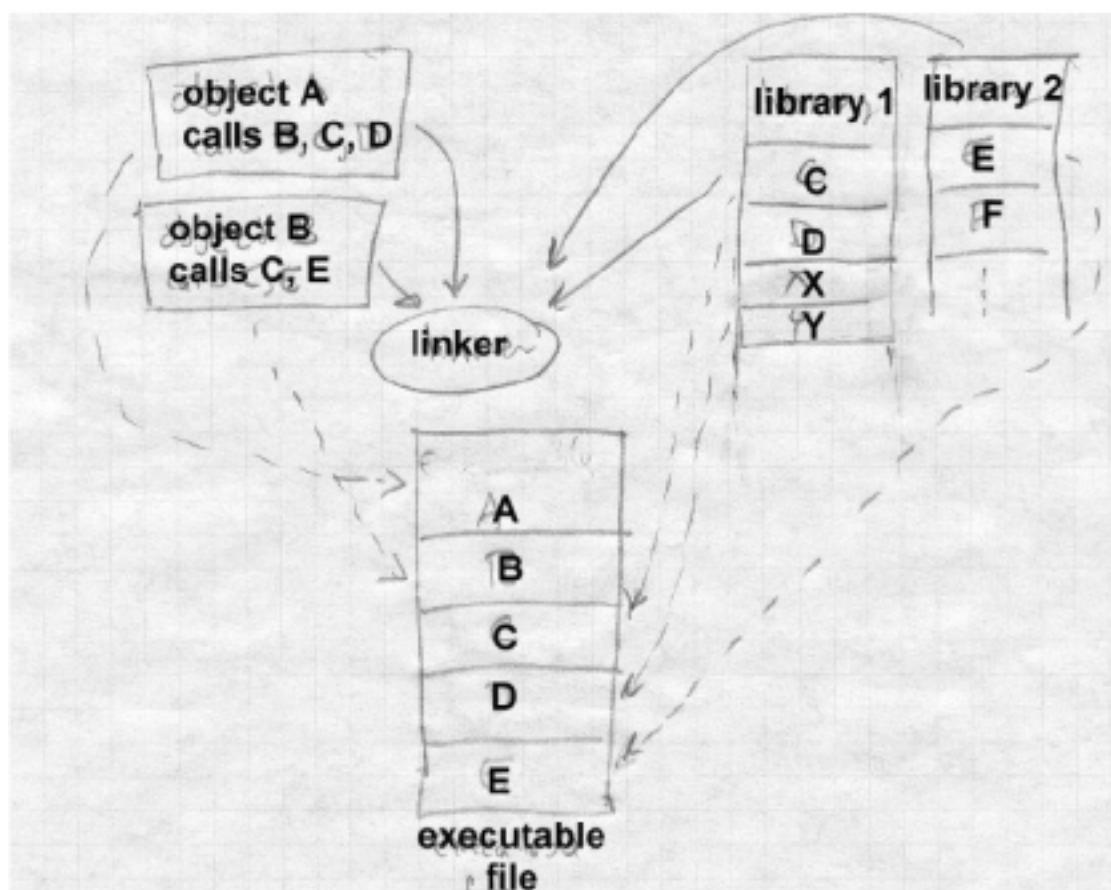
目标代码库

所有的连接器都支持某种形式的目标代码库，多数还支持不同形式的共享库。

目标代码库的基本原则非常简单，如图2所示。一个库基本就是一组目标代码文件。(实际上，在一些系统中，你可以将很多文件连续保存起来，并将结果作为一个连接库)。当连接器处理完所有的输入文件后，如果仍然有一些符号没有定义，它会扫描库，将保存一个或多个符号的文件连接到输出文件。

图1-2 目标代码库

连接器读入目标文件，再跟上很多库文件。



共享库把一些工作从连接过程推迟到载入过程，使得任务变得更复杂一点。连接器在连接过程中确定那些解析了未定义名字的动态库，但并不将他们连接到程序中，连接器在输出文件中记录了这些符号是在哪些库中定义的。这样，当程序载入的时候可以和动态库绑定在一起。第9和10章对此有详细说明。

重载和代码修改

连接器和载入器的核心动作是重载和修改代码。当编译器或汇编器生成一个目标文件的时候，文件中已经定义的代码和数据利用非重载地址来进行编码，那些其他地方定义的代码和数据通常用地址零定义。作为连接过程的一部分，连接器修改目标代码来反映实际分配的地址。例如，考虑下面这个x86代码的片断，这段代码通过eax寄存器将变量a的内容移到变量b。

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

如果a已经在同一个文件中定义到1234X地址，而b是从其他的地方导入的，生成的代码就成为，

```
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单字节指令，后面跟着一个4字节的地址。第一个指令使用指向了1234的地址(字节翻转，因为x86使用了一个从右到左的字节顺序)。第二个指令分配了一个零地址，因为b的地址仍然是未知的。

现在假定连接器把这段代码连接起来，保存a的段的地址重载到10000X地址上，而b被解析成9A12X。连接器把代码修改成：

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

也就是说，连接器把10000加到地址上，所以现在a的重载地址就是11234X，同时修改了b的地址。这些调整对指令产生影响，但目标文件的数据部分中的任何指针也需要修改。

在旧式的只有很小的地址空间和直接地址的计算机系统中，修改过程相当简单，因为连接器只需要处理一种或两种地址格式。现代计算机系统，包括所有的RISC系统，需要相当复杂的代码修改方法。单指令没有足够的位数来保存一个直接地址，所以连接器和载入器需要一个复杂的编址技巧来处理出现在任意地址的数据。在有些情况下，有可能需要2到3条指令来合成一个地址，每个指令保存地址的一部分，并用位修改方法来得到整个地址。在这种情况下，连接器需要能够正确地修改每个指令，在每个指令的地址位中插入一些位。在另外一些情况下，一个例程或一组例程使用的的所有地址都被放在了一个作为“地址池”的数组中，初始化代码将机器的寄存器指向这个数组，代码在需要的时候把这个寄存器作为一个基础寄存器从数组中取出指针。连接器可能需要从程序中使用的所有地址中建立这个矩阵，然后修改指令使它指向合适的地址池入口。我们在第7章中说明这个问题。

有些系统需要与位置无关的代码，这些代码不论载入到任何地址空间中都能正确运行。连接器需要更多的技巧来支持这个特性，就是将不能做到地址无关的地址的程序部分分离开来，然后再安排两个部分互相通信。(详见第8章)

编译器驱动

很多情况下，连接器的运作对于程序员是不可见的，或几乎如此，因为它作为编译过程的一部分是自动被执行的。多数编译系统有一个编译驱动，可以自动地按照需要启动编译过程。例如，如果程序员写了两个C语言源程序，在Unix系统中，编译驱动会象这样按顺序运行一系列程序：

- C预处理器处理A，生成预处理的A
- C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预处理器处理B，生成预处理的B
- C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
- 连接器连接目标文件A和B，以及系统C库

也就是说，系统将每个源文件编译成汇编文件，然后是目标文件，然后将目标文件和系统C库中需要的例程连接在一起。

编译驱动通常要比这个聪明的多。他们会比较源文件和目标文件的生成日期，仅仅重新编译那些修改过的源程序。(Unix下的make程序是一个典型的例子。)尤其当编译C++和其他面向对象语言的时候，编译驱动可以通过各种技巧来抵消连接器或目标文件中的限制。例如，C++模板定义了一系列几乎无限多的相关例程，为了找到一个程序实际使用的例程，一个编译驱动可以把程序的目标文件连接起来，而不连接模板代码，然后从连接器的错误信息中找到那些没有定义的符号，然后调用C++编译器生成包含必需的模板例程的代码，再重新连接。我们会再11章中谈到这个问题。

连接器控制语言

每个连接器都有某种形式的命令语言来控制连接过程。连接器至少需要目标文件和库文件来完成连接，一般说来，连接都有很多选项可选：是否保留调试符号，是否使用共享或非共享库，使用可能的输出格式中的哪一种。多数连接器允许通过某种方式规定连接后的代码绑定到什么地址上，当编译系统核心或其他不在操作系统控制下运行的程序是非常有用。在支持多个代码和数据段的连接器中，一个连接器控制语言可以定义段的连接顺序，对某些段的特殊处理以及其他与特定应用有关的选项。

有四种常见的方法向一个连接器传送指令：

- **命令行:**多数系统都使用命令行或相当的界面，可以用来传送各种文件名和连接开关。Unix和Windows下的连接器多使用这种方法。在那些对命令长度有限制的系统中，一般都有一种方法从一个文件中读入命令，就象在命令行界面中输入的一样。
- **与目标文件混合:**有些连接器，如IBM主机系统的连接器，从一个文件中接受目标文件和连接器命令。这种方式来源于卡片输入时代，在那个时代，程序员需要把目标卡片摞起来和手工打制的命令卡片一起送到读卡器中。
- **嵌入到目标文件中:**一些目标格式，特别是微软的，允许连接器命令嵌

入到目标文件中。这种方法允许编译器将任何连接需要的选项嵌到文件之中。如，C编译器将搜索命令嵌入用来搜索标准的C库。

- **独立的配置语言:**很少有连接器支持完全的配置语言来控制连接。GNU 连接器，可以处理非常多的目标文件格式，硬件体系结构和地址空间惯例。它使用一种复杂的控制语言，允许程序员确定段的连接顺序和其他很多选项。其他的连接器具有比较简单的语言来处理特定的特性，如程序员定义的覆盖。

连接：一个真实的例子

至此，我们已经通过一些很小单真实的连接实例完成了对连接的介绍。图3 显示了一对C源程序，m. c有一个主程序调用一个叫做a的例程，以及a. c，包含了一个对库例程strlen和printf的调用。

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));
}
```

主程序m. c在我的Pentium机器上用gcc编译成一个165字节的典型的a. out 目标格式的目标文件，如图4所示。这个目标文件包含一个固定长度的头，16字节的”文本”段，包含了只读的程序代码；16字节的”数据”段，包含了字符串。后面是2个重载入口，一个标志在pushl指令上，这个指令将串的地址压到堆栈中准备对a的调用，另一个标志在对a的调用指令上，这个指令将控制权交给a。符号表导出_main，导入_a，并包含几个用于调试的其他符号。(每个全局符号都由一个下划线开始，原因在第5章中说明)注意pushl指令指向地址10X，串的假定地址，因为他们都在同一个目标文件中，而调用指令指向地址零，因为_a还是未知的。

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
...
-----
```

子程序文件a.c编译成一个160字节的目标文件，如图5所示。包括头部，一个28字节的文本段，没有数据段。2个重载入口标志到对strlen和write的调用上，符号表导出_a，导入_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	00000001c	00000001c	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
```

要得到一个可执行程序，连接器将这两个目标文件连接和一个标准的C程序的初始化例程以及必要的C库中的例程连接在一起，生成一个如图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>:
...
-----  
连接器将每个输入文件的相关段合并在一起，所以只有一个合并的文本段，一个合并的数据段和一个bss段(用0初始化的数据，2个输入文件并没有用到)。每个段填充到一个4K的边界以适应x86的页面尺寸，所以文本段是4K(减去一个出现在文件中20字节的a.out头，但逻辑上它并不属于这个段)，数据和bss段分别也是4K字节大小。
```

合并后的文本段包含了库启动代码，叫做start-c，然后是m.o中重载到10a4的文本，a.o重载到10b4，还有从C库中连接的例程被重载到文本段中的高端地址。数据段并没有在这里显示，包含和文本段合并顺序一样的合并数据。因为_main的代码已经被重载到10a4X，这个地址已经修改到start-c中。在主例程中，串的引用被重载到2024X，_a的最终地址。在_a中，对_strlen和_write的调用被修订到这两个例程的最终地址中。

可执行文件还有C库中差不多一打的其他例程并没有在这里显示，它们是有启动代码或_write中直接或间接调用的(在后一种情况下就是错误处理例程)。因为这个文件格式不是可重载的，可执行文件没有可重载数据，操作系统将会把它装载到一个固定的已知地址中。它还包含一个用于调用的的符号表，尽管执行文件并不使用这些符号，而且符号表可以剥去以节省空间。

在这个例子中，从库中连接的代码要远远大于程序本身的代码。这种情况非常常见，尤其是那些使用了大型的图形或窗口库，这也推动了共享库的应用，详见第9和10章。连接后的程序是8K，但同样的程序使用共享库仅仅为264字节。当然，这仅仅是一个象玩具一样的例子，但真实程序一般也会节省这样多的空间。

练习

将连接器和载入器分成不同的程序有什么好处？在哪些情况下合并的连接载入器比较有用？

在过去50年中，几乎每个编程系统都包含一个连接器，为什么？

这一章中，我们讨论了连接和载入汇编后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，连接器和载入器是否依然有用？如果一个解释系统直接将源程序编程一种中间代码，象P-code或者Java虚拟机呢？

第 2 章 体系结构的问题

\$Revision: 2.3 \$

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

连接器和载入器，以及编译器和汇编器，与体系结构的细节密切相关，体系结构既包括计算机的硬件体系结构也包括要开发的目标操作系统的体系结构系统。在这一章，我们将对计算机系统的体系结构做足够的研究，保证我们能够理解连接器需要完成什么样的任务。这一章对计算机体系结构的描述故意不完整，去掉了对我们理解连接器没有帮助的部分，如浮点运算和I/O。

硬件体系结构的两个方面影响到连接器：程序地址和指令格式。连接器所要做的一件事情是修改数据段和指令段的地址和偏移量。在这两种情况下，连接器必须保证所做的修改符合计算机使用的地址格式，当修改指令时，必须保证修改不会生成非法指令。

在这章的最后，我们还要探讨地址空间结构，也就是说，一个程序需要哪些地址来运行。

应用二进制界面

每个操作系统都有一个让程序在这个系统下运行的**应用二进制界面(ABI)**。ABI包括在这个操作系统下运行的应用要遵循的编程协定。ABI总是包含一些系统调用，以及调用这些系统调用的方法，还有程序可以什么样内存地址的规则，通常还有如何使用机器寄存器的规则。从一个应用的观点来看，ABI和底层的硬件结构都是非常重要的，因为如果违反了这两者之一的规定，程序都会出现非常严重的错误。

在很多情况下，连接器需要做很多工作来保证满足ABI的要求。例如，如果ABI要求每个程序都有一个表来保存所有的被例程使用的静态数据的地址，连接器通常需要收集连接到这个程序中的所有模块中的地址信息来建立这个表。ABI经常影响连接器的部分是标准进程调用的定义，我们后面会谈到这个问题。

内存地址

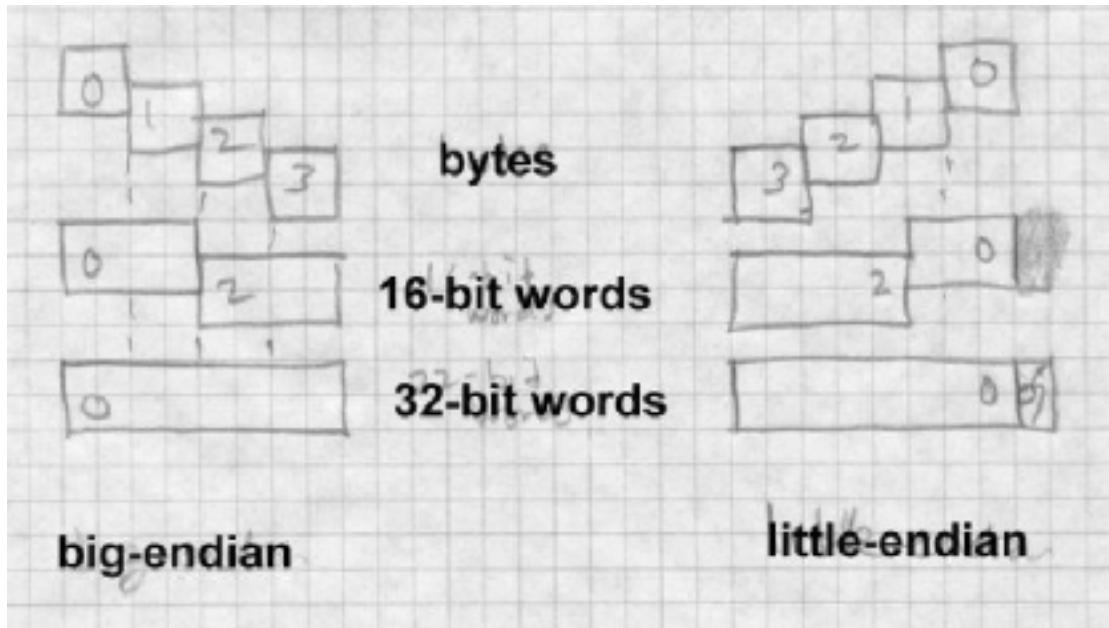
每一台计算机都有一个主存系统。主存系统总是表示为一个存储空间的数据组，每个位置都有个数字地址。地址从零开始一直到某个大的整数，这取决于地址的位数。

字节顺序和对齐

每个存储单元都是一个固定位数的数字。在过去的50年中，这个单元被设计成多达64位或只有1位，但现在几乎所有的计算机都设计成一个8位字节。因为计算机处理的数据，尤其是程序地址都比8位大，计算机仍然可以处理16、32位，

通常是64和128位，由多个字节连接起来。在某些计算机系统中，主要是IBM和Motorola的系统中，多字节前面的(数字小的地址)字节是最高的字节，而其他的计算机系统，主要是DEC和Intel，是最低位字节，如图1所示。沿用“格列佛游记”中的方法，IBM/Motorola的方法叫做big-endian，而DEC/Intel的方法叫做little-endian。

图2-1 内存地址定位
内存地址的图示



很多年来，为两种机制的优缺点引起了强烈的争论。在实际应用中，对两种字节顺序的选择主要基于对旧的系统的兼容性的考虑，因为在使用相同字节顺序的机器之间移植程序要比在不同字节顺序的机器之间容易的多。最近很多芯片都设计成可以支持两种字节顺序，可以很容易地通过电路板的走线或这在引导的时候通过程序选择，有些情况下，甚至可以由应用程序选择。(在这些即时切换的芯片中，由提取和存贮的指令的数据的字节顺序发生变化，但作为常量编入指令的数据的字节不变化。这些细节使连接器的编写变得非常有趣)

多字节数据必须对齐到一个特定得边界。就是说，一个4字节的数据必须对齐到一个4字节的边界，2字节到2字节的边界，依此类推。另一种理解就是任何N字节的数据的地址必须有至少 $\log_2(N)$ 个低位的零。没有对齐的数据会造成性能的降低，在一些系统（多数RISC芯片系统）中还会造成程序错误。即使在那些非对齐数据不会造成程序错误的系统中，由于性能的下降造成的影响也值得在任何可能的地方保持数据对齐。

很多处理器对程序指令都有对齐的要求，多数RISC芯片要求指令必须对齐到4字节边界。

每种体系结构都定义了寄存器，就是可以由程序指令直接存取的一组固定长度的高速内存。每种体系结构的寄存器数目都不相同，从Intel结构的8个到一些RISC系统的32个。寄存器几乎总是和程序地址具有相同的长度，就是说，具有32位地址长度的系统，寄存器有32位长；具有64位地址长度的系统，寄存器有64位长。

4位长。

地址信息

当计算机程序运行时，按照程序的指令从内存中提取和存储数据。指令本身也保存在内存中，通常保存在和数据不同的地方。

指令逻辑上是按照它们的存储顺序执行的。(有些系统使用分支来称呼一些或所有的转跳，但我们都把它们称为转跳。)每个要引用数据地址的指令和每个转跳指令都要指定数据的地址或转调的地址。所有的计算机系统都有不同的指令和地址格式，连接器在重载每个指令的地址的时候都要处理。

尽管计算机设计者们在这些年中都设计出了无数的不同的复杂的地址机制，现在生产的计算机系统都有相对简单的地址机制。(设计者们发现很难把一个复杂系统设计的很快，而且连接器很难用到这些复杂的地址特性)我们会用3个系统作为例子：

- IBM 360/370/390(我们把它们通称为370)。尽管这是仍在使用最古老的体系结构之一，在过去的35年中增加很多特性，但其相对清晰的设计仍然工作地很好，并且发展成可以和现代RISC芯片性能相当的芯片。
- SPARC V8和V9。一个非常流行的RISC体系结构，具有相对简单的地址格式。V8使用32位寄存器，V9增加了64位的大寄存器和地址格式。SPARC的设计和其他的RISC体系结构类似，如MIPS和Alpha。
- Intel 386/486/Pentium(以后称位x86)。仍在使用的最无规律和无法理解的体系结构之一，但不可否认它是最流行的。

指令格式

每种体系结构都有不同的指令格式。我们仅仅探讨和程序和数据有关的格式细节，因为这些是影响连接器的只要部分。370使用和数据引用地址相同的转跳地址格式，而SPARC和Intel对数据的引用地址和转跳地址有些的相同，有些的不同。

每条指令都包括确定指令做什么的操作码和操作数。一个操作数可以编入指令本身(一个**立即**操作数)，或存在内存中。每个保存在内存中的操作数都要经过某种计算。有时候，地址直接保存在指令中(直接寻址)。更多的时候地址是保存在某个寄存器中(寄存器间接寻址)，或将某个寄存器中的内容加上一个常数。如果寄存器存储的是一个数据的地址，指令中的常数是存储空间中要处理的数据的偏移量，这种机制称为**基**寻址。如果把两者交换，寄存器保存偏移量，这种机制称为**索引**寻址。基寻址和索引寻址并没有严格的区分，很多体系结构将两者混合在一起，如370有一种寻址模式是将两个寄存器和一个常量相加到指令中，将任意一个寄存器定义为基寄存器，另一个为索引寄存器，尽管它们的处理方式完全一样。

还有其他更加复杂的地址计算方法仍在使用，但连接器基本无需关心这些机制，因为他们保存的内容连接器无需修改。

有些体系结构使用固定长度的指令，有些使用变长的。所有的SPARC指令都是4个字节长，对齐到4字节的边界。IBM 370的指令可以是2、4或6字节长，指令

的第一个字节的前两位标志着指令的格式和长度。Intel x86的指令可以是从1个字节到14个字节的任意长度。编码方式非常复杂，部分因为x86最初设计成在内存有限的空间下使用密集的指令编码格式，另一部分原因是在286、386中增加的新指令，而最新的芯片必须使用已经存在的指令集中没有使用的位定义。幸运的是，从一个连接器的角度来看，连接器需要调整的地址和偏移量域都对齐到字节边界，所以连接器无需考虑指令编码。

过程调用和可寻址性

在早期的计算机中，内存非常小，每条指令有足够的地址域来寻址计算机内存中的任何位置，就是现在成为直接寻址的机制。直到60年代，可寻址的内存变大，如果使用可以保存整个地址空间得指令集会占掉过多的仍然十分宝贵的内存空间。为了解决这个问题，计算机设计师在一些或全部指令中取消了直接寻址方法，而使用索引和基寄存器来提供寻址中使用的位数中的部分或全部。指令可以变得更短，但造成了更加复杂的编程。

在没有直接寻址的体系结构中，包括IBM 370和SPARC，程序在为数据寻址的时候有一种”自举”问题。一个例程使用寄存器中存储的地址来计算数据的地址，但要把地址存入寄存器的标准方法是从一个内存空间中调入，但这又需要另一个保存在寄存器中的地址。自举问题是在程序开始的时候如何把第一个地址放入一个寄存器，并且保证后面的例程在寄存器中保存需要使用的数据的地址。

过程调用

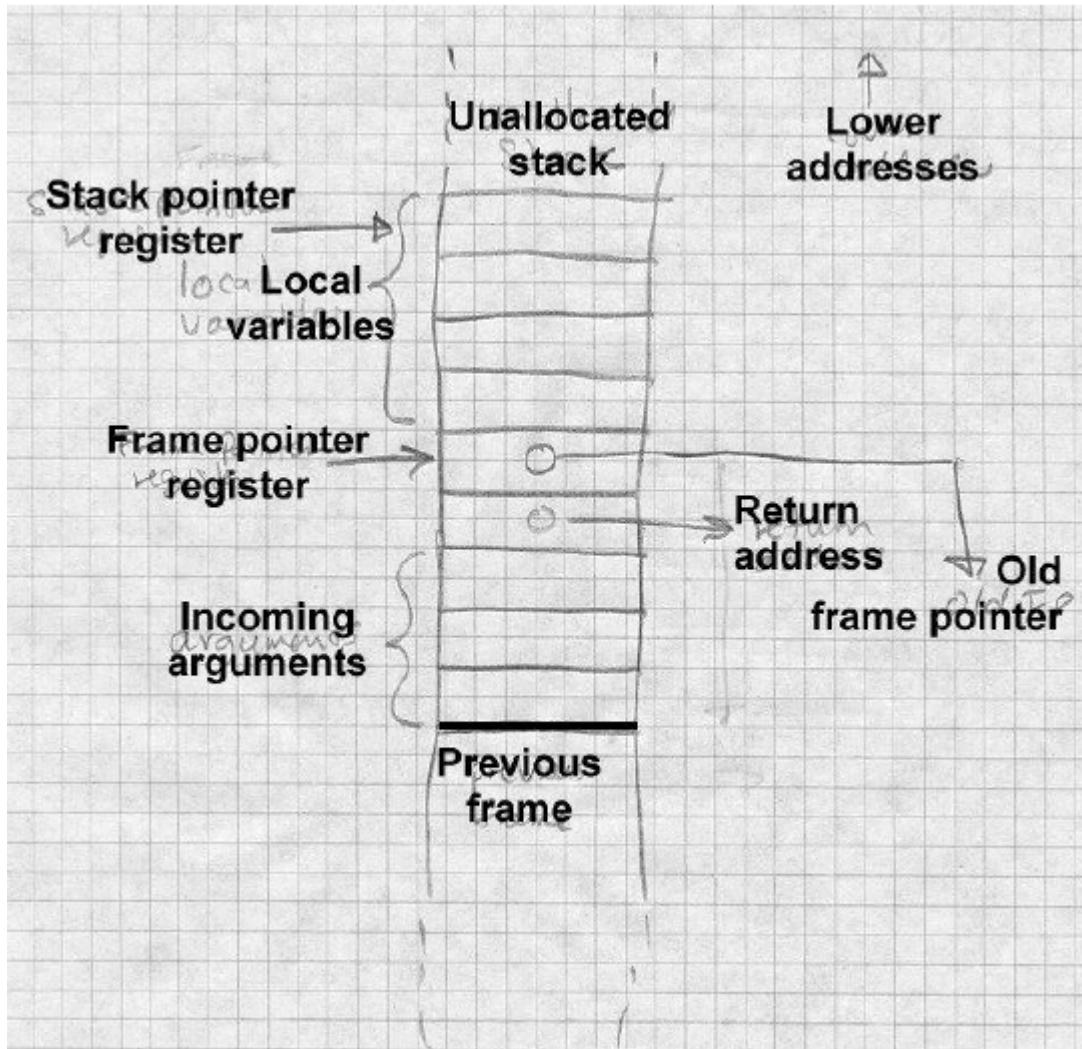
每个ABI都定义了把硬件定义的调用指令和寄存器及内存使用的规定组合在一起的过程调用序列。硬件调用指令保存返回地址(调用指令下一条指令的地址)，然后转跳到过程中。在象x86这样具有硬件堆栈的体系结构中，返回地址被压入堆栈，而其他的体系结构则把返回地址保存到寄存器中，软件需要负责在需要的时候把寄存器的内容保存到内存中。具有硬件堆栈的体系结构通常都有一个把返回地址弹出堆栈并转跳到该地址的硬件返回指令，其他的体系结构使用一个”分支到寄存器中的地址”的指令来返回。

在过程中间，数据寻址有四种类型：

- 调用者可以向过程中传递**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**保存在内存中的一个固定地址中，仅对过程有效。
- **全局静态数据**保存在内存中的一个固定地址中，且可以被很多过程引用。

为一个单一过程分配的一串堆栈内存称为堆栈帧。图2显示了一个典型的堆栈帧。

图2-2 堆栈内存布局
一个堆栈帧的图示。



参数和本地变量通常都保存在堆栈中。寄存器之一可以作为基寄存器当作堆栈指针。在SPARC和x86中使用这种机制的一种常见的变化，当一个过程开始的时候，一个单独的帧指针和基指针从堆栈指针寄存器中提取出来。这样，就可以把变长的对象压入堆栈，把堆栈指针指向一个预先定死的值，而帧指针在过程运行中保持不变，过程仍然可以通过一个固定的偏移量提取参数和本地变量。假设堆栈从高端地址向低端地址生长，帧指针指向保存在内存中的返回地址，参数都保存在相对帧指针的一个小的正的偏移量的地方，本地参数保存在相对帧指针的一个小的负的偏移量的地方。操作系统通常在程序之前设置初始堆栈指针，所以程序只需在压入和弹出数据的时候修改寄存器。

对于本地和全局静态数据，编译器可以生成一个例程所需要的所有静态对象的指针的表。如果有一个保存着指向这个表的指针的寄存器，例程可以通过表指针寄存器把指向任一个对象的指针存入到另一个寄存器作为基指针，然后再通过第2个寄存器寻址这个对象。这样，就需要一个技巧把表的地址保存到第1个寄存器中。在SPARC中，例程可以通过一系列带有直接操作数的指令把表地址放入第1个寄存器，在SPARC或370中例程可以用一个子程序调用指令的变形把程序寄存器(保存目前指令地址的寄存器)保存到一个基寄存器，由于一个我们后面会谈到的原因，这些技术在目标编码中会产生问题。一个更好的办法是把提取表指针

的工作交给例程的调用者，因为调用者已经拥有了自己的表指针，并在其中取出被调用的例程的表。

图3显示了一个典型的例程调用过程。Rf是帧指针，Rt是表指针，Rx是临时寄存器。调用者把自己的表指针保存到自己的堆栈帧中，然后把要调用的例程的地址和它的指针表存入寄存器中，再进行调用。被调用的例程可以用存在Rt中的表指针找到它所需要的所有数据，包括它依次要调用的任何例程的地址和表指针。

图2-3 理想化的调用过程

~~... 将参数压入堆栈 ...~~

```

store Rt → xxx(Rf) ; save caller's table pointer in c
aller's stack frame
load Rx ← MMM(Rt) ; load address of called routine in
to temp register
load Rt ← NNN(Rt) ; load called routine's table point
er
call (Rx) ; call routine at address in Rx
load Rt ← xxx(Rf) ; restore caller's table pointer

```

有很多可能的优化方法。很多情况下，一个模块中的所有例程都共享一个单一的指针表，因此模块内的调用就无需改变表指针。SPARC的规定是由连接器生成的一个库共享一个表，这样模块内的调用就无需改变表指针寄存器。同一个模块内的调用通常可以通过一种嵌入了到被调用例程的偏移量的”call”指令完成。通过这两种优化，一个模块内的例程的调用过程被简化到一个调用指令。

我们回到地址自举的问题，那么这个表指针链是如何开始的呢？如果每个例程都使用它的调用者已经提取的表指针，最初的例程如何得到这个指针？有很多答案，但都需要特殊的代码。主例程的表可以保存再一个固定的地址，或初始指针可以标记在可执行文件中，操作系统可以在程序运行之前把它装载到系统中。无论使用什么技术，都需要连接器的帮助。

数据和指令引用

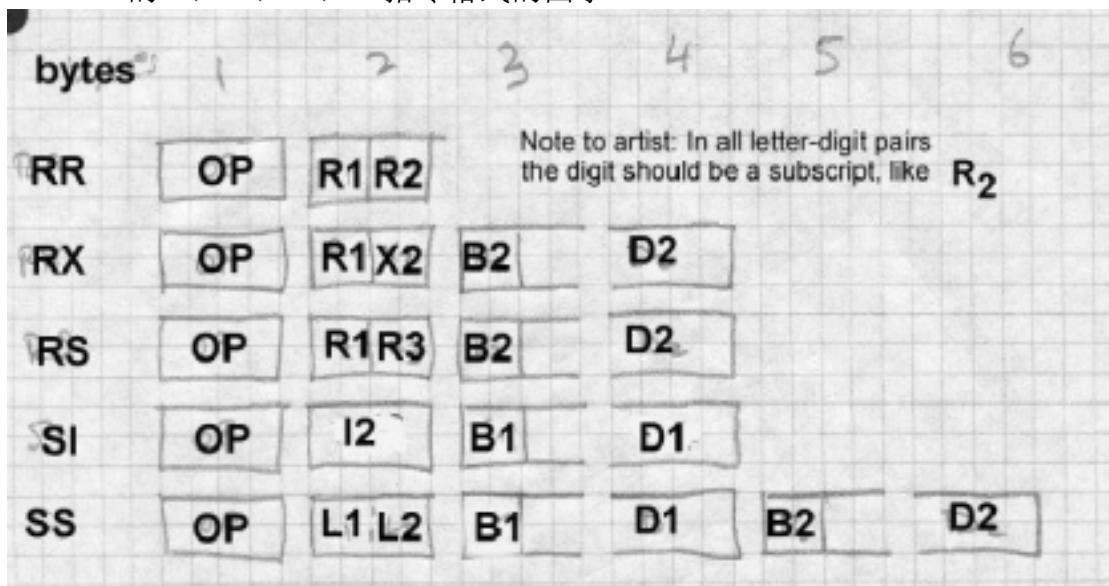
我们再更具体地看看程序是如何在我们的3种体系结构中寻址数据的。

IBM 370

60年代制造的System/360使用一种直接寻址的数据寻址机制，随着360演进到370和390，寻址机制变得越来越复杂。每个引用内存数据的指令都用一个基寄存器或索引寄存器加上一个指令中的无符号12位偏移量计算地址。共有16个32位的通用寄存器，编号从0到15，除了一个都可以作为索引寄存器。如果在地址计算中使用了寄存器0，实际并不使用寄存器0的内容而用零值计算。（寄存器0确实存在，但不用作寻址，而用于计算）如果在使用寄存器中保存目标地址的跳转指令中，如果使用寄存器0表示不跳转。

图4显示了主要的指令格式。一条RX指令包含了寄存器操作数和一个单一内存操作数，地址是把指令中保存的偏移量和一个基寄存器相加得到。很多时候索引寄存器是0，所以地址就是基地址加上偏移量。在RS、SI和SS格式中，12位的偏移量加到一个基寄存器中。一条RS指令有一个内存操作数，一个或两个保存在寄存器中的操作数。一条SI指令有一个内存操作数，另一个操作数是指令中保存的8位立即量。一条SS指令有两个内存操作数，是存储到存储操作。RR格式有两个寄存器操作数，没有内存操作数，尽管这其中一个寄存器或两者都保存着指令要操作的内存的指针。370和390在这些格式中增加了很少的变化，但没有改变数据寻址格式。

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来直接存取最低的4096个内存单元。这个特性对底层系统编程非常有用，但从不用于应用程序中，应用程序都使用基寄存器寻址方式。

注意在所有的指令中，12位地址偏移量永远保存在一个16位对齐的半字的低12位。这样就有可能通过修改目标文件的地址偏移量而不用修改任何对指令的引用，因为偏移量的格式都是一样的。

最初的360系统有24位地址，在内存或寄存器中保存的地址都存到一个32位的字的低24位中，高8位忽略不计。370系统将寻址扩展到31位。不幸的是，很多程序，包括非常流行的OS/360操作系统在内存的32位的高位字节保存各种标志或其他数据，所以不可能直接将地址扩展到32位而不影响已经存在的目标文件。作为替代，系统具有24位和31位模式，在任何时刻，一个CPU会处理24位寻址或31位寻址。一个由软硬件组合出来的规定了，如果一个地址字的高位置1，则该地址字保存的是31位地址；如果高位置0，则该地址保存24位地址。因此，连接器必须有能力处理24位地址和31位地址，因为程序能够也确实根据一个特定例程的长度而切换模式。由于历史的原因，370的连接器也能处理16位地址，因为早期的360产品线经常只有64K甚至更少的主存，而程序利用半字指令来存储地址值。

后期的370和390增加了象x86系列方式的分段地址空间。操作系统利用这些

特性可以定义多个31位地址空间供程序使用，存取控制和地址空间切换的规则极其复杂。据我所知，没有任何编译器和连接器支持这些特性，它们主要用于高性能数据库系统，所以我们不再讨论它们。

370的指令寻址也是相对直接了当。在最初的360系统中，转跳指令(总被称作分支指令)都是RR或RX格式。在RR转跳中，第2个寄存器操作数保存着转跳目标，寄存器0意味着不转跳。在RX转跳中，内存操作数是转跳目标。过程调用是分支和连接(后来被31位寻址的分支和储存代替)，通过将返回地址保存到特定的寄存器然后转跳到RR格式中的第2个寄存器保存的地址，或者RX格式中的第2个地址操作数。

当在同一个例程内的转跳时，例程需要具有”可寻址性”，就是说，一个基寄存器指向(或至少接近)例程的开始，以让RX指令使用。传统上来讲，寄存器15保存着一个例程的进入点，并可以用作一个基寄存器。一个RR方式的分支及连接转跳或分支及储存转跳指令，如果第2个寄存器使用保存着后续指令地址的寄存器0，并不转跳，但可以用来把第1个寄存器设为基地址，如果第1个寄存器的内容未知的话。因为RX指令有一个12位的偏移量域，一个独立的基寄存器可以”覆盖”4K大小的代码。如果例程要比4K大，则必须使用多个段寄存器来覆盖所有的例程的代码。

390系统增加了一些相关的转跳指令格式。这些新的格式包含了一个16位的偏移量，加到指令地址上得到转跳的目标地址，偏移量逻辑地向左移了1位，因为所有的指令都对齐到偶数字节。这些格式不再使用寄存器来计算地址，允许在 $+/-64K$ 的范围内转跳，除了最大的例程外，可以完成几乎所有的例程内的转跳。

SPARC

就像SPARC的名字一样，SPARC是一种精简指令集处理器，虽然随着体系结构演进到第九个版本，最初简单的设计已经变得比较复杂了。SPARC V8以前的版本都是32位的，SPARC V9扩展到64位的体系结构。

SPARC V8

SPARC有四种主指令格式和31种次指令格式。图5显示了4种转跳格式和2种数据寻址模型。在SPARC V8中有31个通用寄存器，每个为32位，编号从1到31。寄存器0是一个永远保存数值0的伪寄存器。

通过一种少见的寄存器窗口机制来减少所有过程调用和返回所需的寄存器保存和恢复的次数。窗口机制对连接器没什么影响，所以我们不再讨论。(寄存器窗口最初是出现在Berkeley的RISC设计中，后来衍生出SPARC)

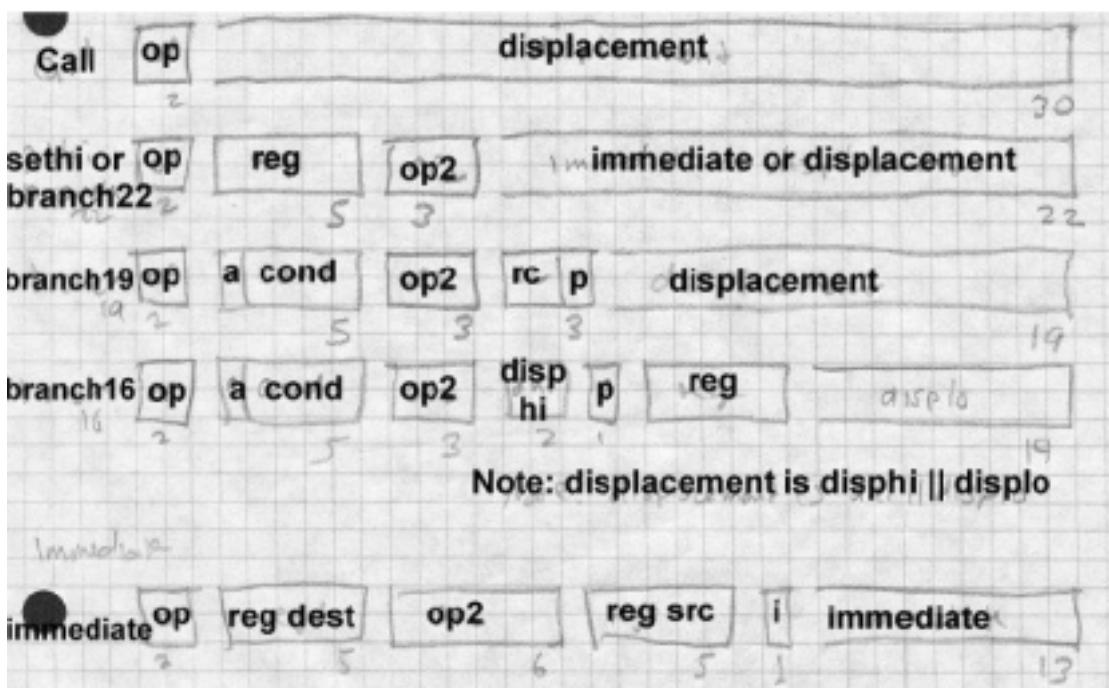
数据引用使用一种或两种模型。一种模型是将两个寄存器的值加在一起得到地址。(如果一个寄存器已经保存着要转跳的地址，可以用寄存器0来计算)另一种模式是在一个基寄存器上加上一个无符号的13位的偏移量。

SPARC的汇编器和连接器通过一个双指令序列支持一种伪直接寻址机制。这两个指令是SETHI，将一个22位的立即数装入一个寄存器的高22位，将寄存器的低10位置零，第2个指令与一个13位的立即数求或，存入寄存器的低10位。汇编器和连接器负责根据需要的32位地址将需要的高位和低位部分写入两个指令中。

图2-5 SPARC

30位调用22位分支和SETHI19位分支16位分支(仅用于V9)，R+R操作和R+I1

3操作。



过程调用指令和多数有条件转跳指令(按照SPARC的叫法为分支)使用相对寻址方法, 分支偏移量从16到30位之间。无论采用多长的偏移量, 转跳指令都将偏移量左移2位, 因为所有的指令必须在一个四字节字的地址上, 符号将结果扩展到32或64位, 并将结果加到转跳或调用指令的地址上来得到目标地址。调用指令使用一个30位的偏移量, 可以对V8的32位地址空间的任何地方进行寻址。不同的转跳使用16、19或22位偏移量, 足够在所有尺寸的例程中完成任意的转跳。16位偏移量把偏移量分成2位的高位和14位的低位, 保存到指令字的不同部分中, 但对连接器不会造成多大的麻烦。

SPARC还有一种”转跳和连接”指令, 使用和数据寻址的一样的办法来计算目标地址, 即把两个寄存器相加, 或把一个寄存器和一个常数偏移量相加。还可以把返回地址保存到一个寄存器中。

过程调用使用调用或转跳及连接指令, 将返回地址保存在寄存器15, 然后转跳到目标地址中。过程返回使用JMP8[r15], 调用后返回两个指令。(SPARC调用和转跳都是”延迟”的, 在转跳之前允许先执行转跳或调用指令后面的指令。)

SPARC V9

SPARC V9将所有的寄存器扩展为64位, 使用低32位来兼容旧的32位程序。所有已有的指令可以继续象以前那样工作, 只是现在的寄存器操作数为64位而不是32位。新的提取和保存指令处理64位的数据, 新的分支指令可以测试前面的指令结果是32位还是64位的。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。合成全64位地址可以通过SETHI和OR指令序列的将地址的两个32位半地址合并得出, 将高32位左移, 再将两个部分求或得出。在实际使用中, 64位地址是从一个指针表中提取出来, 模块内调用将目标例程的地址从表中提取出来保存到一个寄存器中, 再用转跳和连接完成调用。

Intel x86

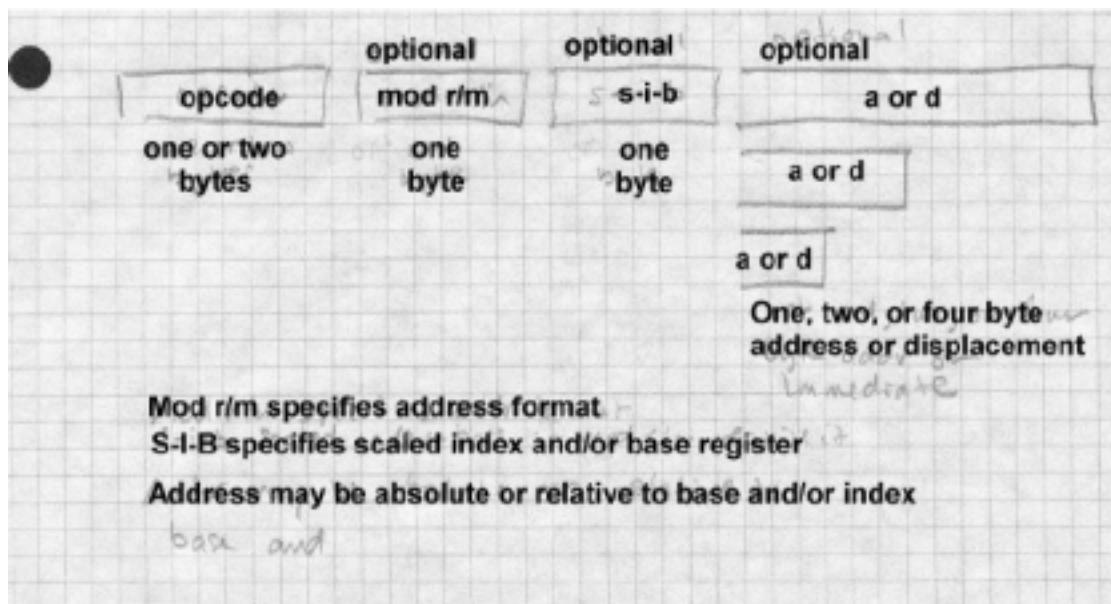
目前Intel x86是我们讨论的这三种体系结构中最复杂的。它包括一个非对称指令集和分段地址。共有6个32位的通用寄存器，为EAX、EBX、ECX、EDX、ESI和EDI，还有两个主要用于寻址的寄存器，EBP和ESP，还有6个专用的段寄存器CS、DS、ES、FS、GS和SS。每个32位的寄存器的低16位可以作为16位寄存器，称为AX、BX、CX、DX、SI、DI、BP和SP。AX到DX寄存器的低8位和高8位又可以作为8位寄存器，称为AL、AH、BL、BH、CL、CH、DL和DH。在8086、186和286中，很多指令需要特定的寄存器作为操作数，但在386之后的芯片中，很多但不是全部原来需要特定寄存器的指令已经可以使用任何寄存器了。ESP是硬件堆栈指针，永远保存着目前堆栈的地址。EBP一般用作指向当前堆栈帧的基地址的帧寄存器。(指令集建议但不强制如此使用。)

任何时刻一个x86都会运行在3种模式中的一种：仿真原来的16位的8086的实模式，在286中增加的16位保护模式，或者在386中增加的32位保护模式。这里我们主要讨论32位保护模式。保护模式需要使用x86的声名狼藉的分段，但我们现在暂不讨论。

多数需要对数据进行寻址的指令都用一个通用指令模式，如图6显示的。(那些不使用特定的体系机构规定的寄存器的指令，如PUSH和POP指令都使用ESP来寻址堆栈。)地址是将指令中1、2或4个有符号字节替代置相加得到的，任何一个32位寄存器都可以作为基寄存器，除了ESP的任何32位寄存器都可以作为一个可选的索引寄存器。索引可以逻辑左移0、1、2或3位，使定位多字节数值更加容易。

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字节替代



尽管可以把所有的替代字节、基地址和所以地址放到一条指令中，多数指令只使用32位替代自己，来完成直接寻址；或一个基地址带一个或两个替代字节，

来提供堆栈寻址和指针解析。从一个连接器的角度来看，直接寻址允许一条指令或数据地址嵌入到程序的任何地方，不论对齐到多少字节。

条件转跳和无条件转跳还有子程序调用都使用相对寻址。任何转跳指令都有1个、2个或4个字节的偏移量，加到当前指令的下一条指令地址上来得出目标地址。调用指令要么有一个4字节的绝对地址，或者使用一个保存着目标地址的内存地址的普通寻址模式。这样就可以转跳和调用到可以在当前的32位地址空间中的任何地方。无条件转跳和调用也利用上面说的计算数据全地址的方法来计算目标地址，更多的时候用来转跳或调用一个保存在寄存器中的地址。调用指令将返回地址压到ESP指向的堆栈。

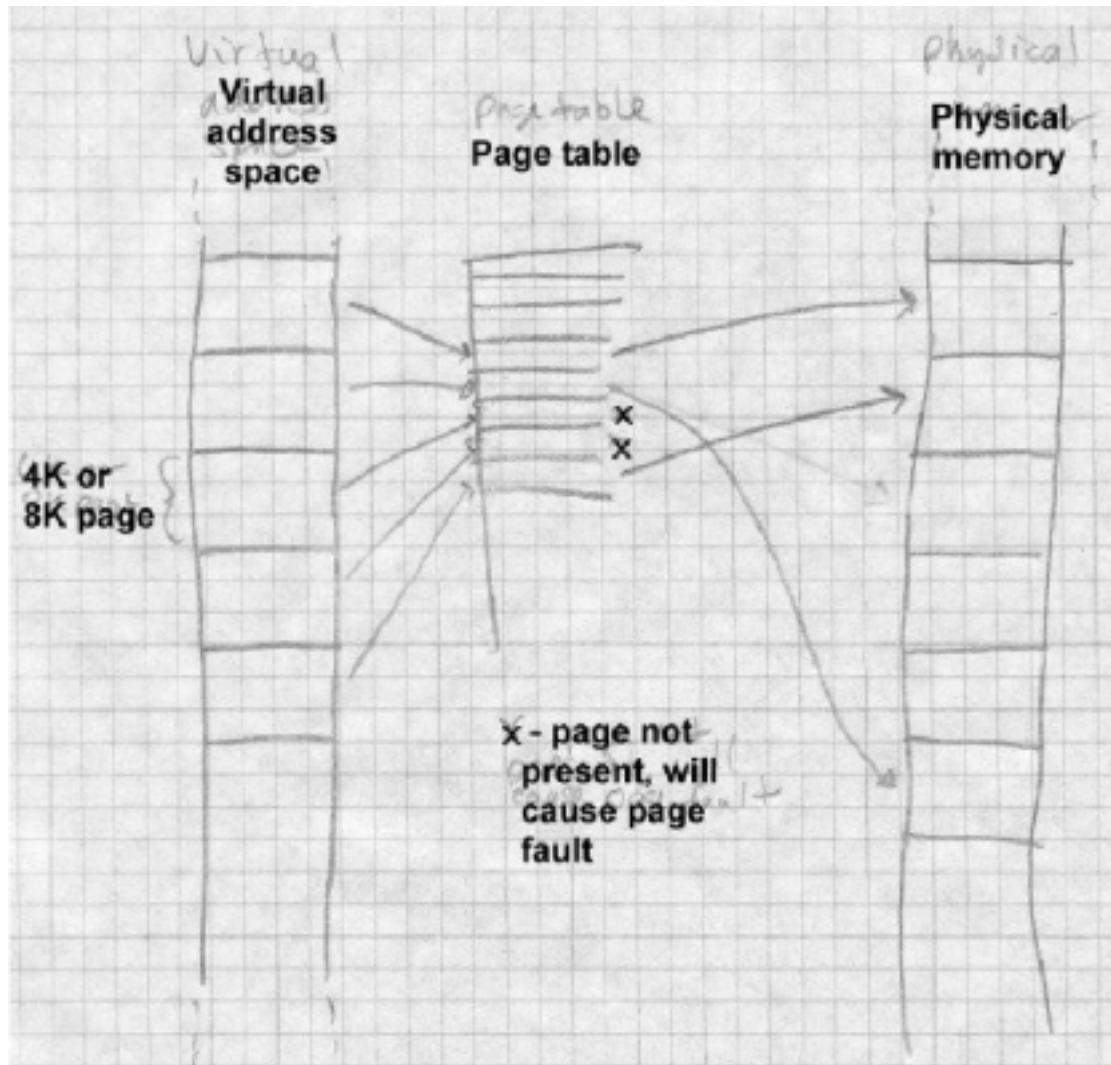
无条件转跳和调用也可以用指令中的完全6字节段/偏移量地址，或者计算保存段/偏移量目标地址的地址。这些调用指令把返回地址和调用者的段数都压到堆栈中，可以完成跨段的调用和返回。

分页和虚拟内存

在现代多数计算机系统中，每个程序可以寻址一个巨大的地址，在32位系统中是4G。只有很少的计算机有那么多内存，即使有这么多内存也需要在多个程序中共享。分页硬件把一个程序的地址空间分成固定大小的页面，通常是2K或4K大小，并把计算机的物理内存分成同样大小的页面帧。硬件保存着在内存中每个页面入口的页面表，如图7所示。

图2-7 页面映射

将页面和实际页面帧通过一个大页面表对应起来



一个页面表保存着每个页面的实际页面帧，或者标志着页面“不可用”。当应用程序试图使用一个不存在的页面的时候，硬件生成一个“**页面错误**”中断，并由操作系统处理。操作系统会从磁盘中调入页面内容的拷贝放到一个可用的页面帧中，再让应用程序继续运行。在需要的时候通过在主存和磁盘中来回移动页面，操作系统为应用程序提供了比实际使用的内存大的**虚拟内存**。

然而使用虚拟内存也要付出代价。一个指令的执行时间是一毫秒的一小部分，但是页面错误和接下来的页面调入调入(从磁盘送到内存，或者反过来)因为需要磁盘处理需要几个毫秒。一个程序造成的页面错误越多，它运行的越慢，最坏情况下会造成“**系统失效**”，所有的页面错误让计算机无法完成有用的工作。一个程序需要的页面越少，它所造成的页面错误越少。如果连接器可以把多个例程压缩到一个页面或一小组页面中，可以提高分页的效率。

如果页面可以被标为只读的，性能也会提高。只读页面无需换出，因为它们可以从它们调入的地方重新调入。如果同一个页面逻辑地出现在多个地址空间中，可以用同一个物理页面为所有的地址空间服务。

使用32位寻址空间和4K页面的x86系统需要一个 2^{20} 个入口的页面表来映射所有的地址空间。因为每个页面表入口通常为四个字节，这样页面表就是不太现实的4M大小。所以，使用分页的体系机构对页面表进行分页，高层的页面表指

向低层的页面表，低层的页面表再指向与虚拟内存地址的实际页面帧。在370系统中，每个高层页面表(称为段表)的入口对应1M的地址空间，所以在31位地址空间中的段表可以保存2048个入口。段表中的入口可以为空，在这种情况下，整个段并不存在，或指向一个映射这个段的底层页面表。每个低层页面表有256个入口，每个对应内存空间的4K连续空间。X86系统也将页面表作类似的分割，虽然边界不一样。每个高层页面表(成为页面目录)对应4M空间，因此高层页面表包含1024个入口。每个低层页面表也有1024个入口来映射在那个页面表内的4M地址空间的1024个4K页面。SPARC体系结构将页面定义为4K，而且有三层而不是两层页面表。

这种两侧或三层页面表结构对应用是透明的，除了一个重要的例外：操作系统可以通过更改高层页面表的一个入口，可以将对应的页面增大(在370系统上是1M，在x86上是4M，在SPARC上是256K或16M)，所以因为效率的关系，在进程切换的时候，地址空间一般是通过改变第2层页面表的那个入口的页面尺寸而不是替换整个页面表而管理的。

程序地址空间

每个程序都运行在一个由计算机系统硬件和操作系统一起定义的内存空间中。连接器和载入器需要生成一个符合这个内存空间的可运行的程序。

这种内存空间的最简单的形式是由Unix的PDP-11提供的。地址空间是由地址0开始的64K字节。程序的只读代码装载到地址0，然后是可读写的数据部分。PDP-11使用8K的页面，所以数据从代码的8K字节边界开始。堆栈向下生长，从64K-1开始，随着堆栈和数据的增长，各自的空间也在增加，如果它们相遇的话，则程序没有内存空间可供使用。PDP-11的后续机型，VAX上的Unix也使用类似的机制。每个VAX的Unix程序的头2个字节都为0(是一个寄存器的保存掩码，说明什么也不保存。)所以，一个全为0的指针总是合法的，如果一个C程序使用一个null字符串指针，地址0的0字节作为一个空串来处理。80年代的Unix程序都存在着很难发现的有关空指针的bug，很多年来，移植到其他体系结构的Unix都在地址0处保留一个0字节，以为这比找到并修改所有的空指针的错误要容易。

Unix系统将每个程序都放到分离的地址空间中，操作系统运行在一个与应用逻辑分离的空间中。其他的系统将多个程序放到同一个地址空间中，使得连接器尤其是载入器的工作更加复杂，因为程序的实际装载位置知道程序运行的时候才能知道。

x86上的MS-DOS系统不使用硬件保护，所以系统和应用共享同一个地址空间。当系统运行一个程序的时候，它首先寻找可能出现在系统中任何地方的最大的连续可用空间，把程序调入那里运行。IBM的主机操作系统基本上也是如此，将程序载入一个连续的可用空间。在这两个系统下，程序载入器或程序本身必须把程序调整到它所装载的地方。

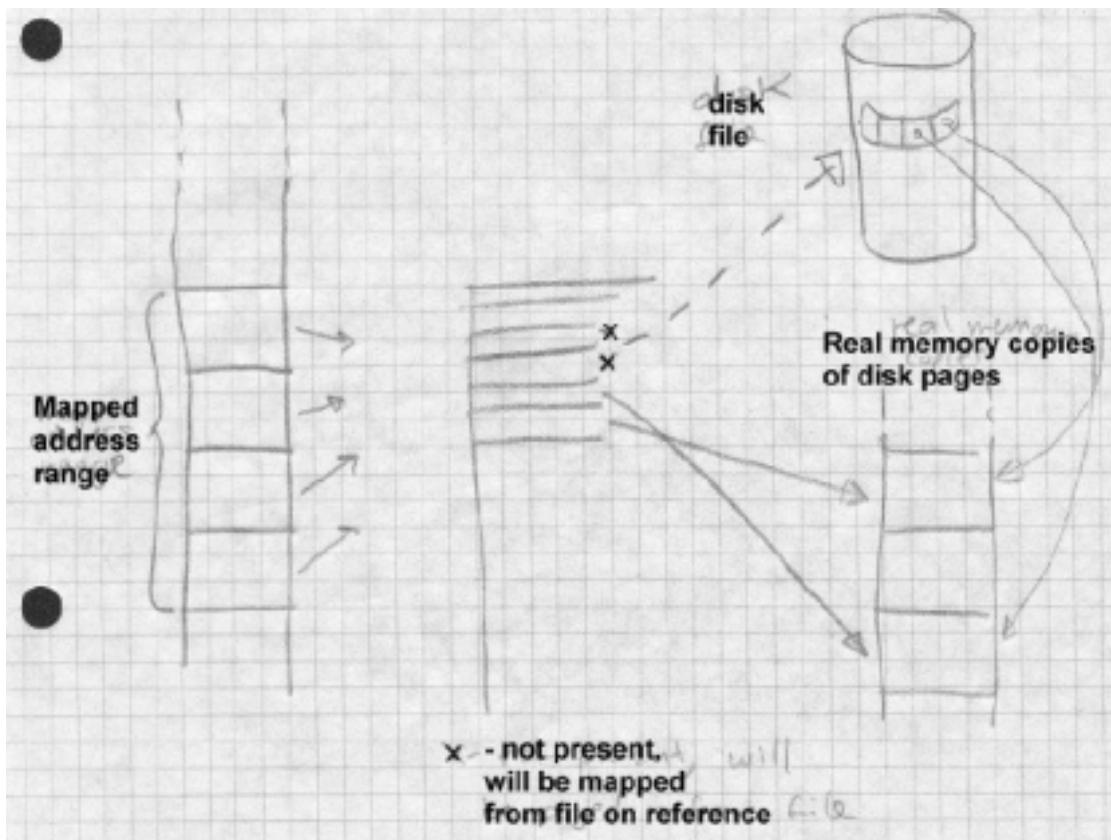
MS Windows使用一种不寻常的装载机制。每个程序都连接到一个标准的开始地址进行装载，但可执行程序文件保存着重载信息。当Windows调入程序的时候，如果开始地址可用，它就把程序放到那个地址，但如果那个地址不可用，也会把程序装载到其他的某个地方。

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，如果内存不够将数据换页到磁盘。早期的系统中，所有的页面都换到文件系统中的一个分离的”匿名”的磁盘空间。换页发明不久后，设计者们发现可以将换页系统和文件系统综合到一起，使用文件系统中的命名的磁盘文件进行换页。当程序将一个文件映射到程序的地址空间的一部分时，操作系统将地址空间中的那些页面标为不存在，然后使用这个文件作为地址空间的一部分，如图8所示。程序可以通过对地址空间的引用来读取该文件，由换页系统来完成必要的磁盘调用。

图2-8 映射文件

程序指向一些对应磁盘文件或系统内存的一组页面帧



处理映射文件的写操作有三种办法完成。最简单的办法是映射到只读文件，任何向映射的写操作都会失败，通常造成程序终止。第二种方法是映射到读写文件，对映射到这个文件的内存的修改会在文件取消映射的时候写入文件。第三种办法是将文件映射为写入时拷贝(COW，不是最恰当的缩写)。就是在程序试图写入页面之前将页面映射到只读文件，当程序要写入页面的时候，操作系统为页面制作一个拷贝，并把它按照一个没有映射到文件的私有页面处理。从程序的角度来看，映射一个COW文件非常类似于从文件中读取内容，并将其拷贝到一个没有分配的匿名自由内存空间中，程序对该空间的修改对程序本身是可见的，但其他映射该文件的程序却不会看到。

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表，使每个程序都有独立的地址空间。这让系统非常可靠，因为有错误或恶意程序不会互相影响和监视，但可能造成潜在的性能问题。如果一个运行的单一的程序或单一的程序库在系统中使用多个地址空间，如果所有的地址空间共享程序或库在物理内存中的一个拷贝，系统可以节省很多的内存资源。操作系统实现这个功能相对直接了当，只要把可执行文件映射到每个程序的地址空间中。非重载代码和只读数据标为R0，可写数据映射到COW。(如果程序在载入的时候必须重载，重载进行需要修改代码页，然后必须按照COW方式而不使R0方式处理)

完成这种共享工作需要很多连接器的支持。在可执行程序中，连接器需要将所有可以标为R0的执行代码组合起来，再将数据组合到另一个映射为COW的部分。每个部分必须逻辑地在地址空间页边界开始，物理地在文件页边界开始。当不同的程序使用一个共享库的时候，连接器需要在每个程序中标记，当该程序开始的时候，库被映射到该程序的地址空间。

位置无关的代码

当一个程序在多个不同的地址空间运行的时候，操作系统通常在程序出现的每个地址空间中将其装载在相同的位置。这样，连接器的工作就容易多了，它可以将程序中所有地址绑定到一个固定位置，程序载入的时候也无需重载。

共享库的使用让这种情况变得非常复杂。在一些简单得系统设计中，在系统启动的时候或共享库建立的时候，每个库都被分配了一个全局唯一的内存空间。这样，每个库都有一个固定的地址，代价就是共享库的管理会产生严重的瓶颈，因为系统管理员要维护一个包含库内存地址的全局表，如果一个库的新的版本比旧的版本大，而且无法放入分配的空间，整个共享库甚至所有引用那个库的程序都需要重新连接。

变通的办法就是允许不同的程序将一个库映射到地址空间的不同地方。这样降低了库管理的难度，但编译器、连接器还有载入器需要合作，让库无论出现在地址空间的什么地方都可以工作。

一个简单的办法就是将库的标准重载信息放到库中，当库映射到每个地址空间的时候，载入器可以修改程序中的重载地址来反映库的实际装载地址。不幸的是，修改的过程需要对库的代码和数据的写操作，就是说如果页面标成COW，页面无法再被共享；如果标成R0，程序会崩溃。

为了避免这种情况，共享库使用位置无关代码(PIC)，就是无论装载到内存的任何地方都能工作的代码。共享库中的所有代码通常都是PIC，所以这些代码可以映射到R0页面中。数据页通常包含需要重载的指针，因为数据页反正都要映射到COW页面中，也不会有什么共享的损失。

对于大部分计算机系统，PIC非常容易建立。我们这章讨论的三种体系机构使用相对转跳指令，所以例程内部的转跳指令无需重载。对堆栈中的本地数据的引用使用相对于基地址的基寻址，也无需任何重载。唯一的挑战是对不在共享库内的例程的调用和对全局数据的引用。直接数据寻址和SPARC的高/低寄存器技巧无法工作，因为它们需要运行时的重载。幸运的是，我们还有很多技巧可以写出

PIC代码来处理库内调用和全局数据。当我们在第9章和第10章讨论共享库的时候再讨论它们。

Intel 386 分段

这章最后的话题是声名狼藉的Intel体系机构中的分段系统。X86系统是唯一仍然使用的分段系统，除去一些传统的优利主机系统，但因为它太流行了，所以我们需要连接它。尽管我们很快要讨论的32位系统不怎么使用分段，旧式的系统和x86系统中非常流行的嵌入式版本用了很多分段。

最初的8086是Intel非常流行的8位8080和8085微处理器的后续版本。8080有16位地址空间，8086的设计者要在两个方案中进行痛苦的选择，一种是继续使用16位地址空间，好让8085到8086的代码翻译更加容易，而且可以使用更加精简的代码；另一种是提供更大的地址空间来为未来的更大的应用程序提供更多的“头部空间”。他们最后妥协了，选择了多个16位地址空间。每个16位地址空间被成为一个段。

一个运行的x86程序有四个用四个段寄存器定义的活动的段。CS寄存器定义提取指令的代码段。DS寄存器定义了多数数据存取的数据段。SS寄存器定义用来压入弹出指令操作符的堆栈段，用于压入和弹出的指令的操作数，调用和返回指令的程序地址值的压入和弹出，并用作利用EBP和ESP作为基地址的数据引用。ES寄存器定义了另外的段，用于一些字符串操作指令。386和后续的芯片增加了两个段，FS和GS。通过段超越指令，对数据的引用可以直接使用某个特定的段。如，指令MOV EAX, CS:TEMP可以从代码段而不是数据段的TEMP地址取一个数据。FS和GS段只用在段超越的指令中。

段的数据不一定总是不同。多程序将DS和SS的值设成一样，这样指向段变量和全局变量的指针可以交换地使用。有些小程序将四个段设成一样地值，提供一种称为微小模式的地址空间。

在8086和186的平台上，体系结构通过将段值左移4个位来提供一种将段和内存地址固定映射的方法。例如，段值0x123会从内存地址0x1230开始。这种简单的寻址方式也称为实模式。程序员通常将这称为段落，一个段值可以寻址的16个字节内存。

286增加了一种保护模式，操作系统可以将段和实际内存中的任何一个地址映射，并将段掩为不可见，提供基于段的虚拟内存。每个段可以标为可执行，可读，读写的，提供段一级的保护。386将保护模式扩展到32位，每个段可以到4G而不是16K。

使用16位的寻址，除了极小的程序外都需要处理分段寻址。改变一个段寄存器的值是一个需要9个时钟周期的非常慢的操作，相对于在486上只需要1个时钟周期更改通用寄存器。基于这个原因，程序和程序员都尽可能地将数据和代码压缩在最少的段中，避免更改段寄存器。连接器通过将相关的数据或代码放入一个单一的段中的“分组”来帮助实现这个过程。指向代码和数据的指针可以是近，只带有一个偏移量而不带有段；或远方式，带有段和偏移量。

编译器缺省能够根据数据和地址是近还是远来生成使用不同内存模式的代码。小模式使所有的指针为近，并只有一个代码段和一个数据段。中模式有多个代码段（每个源程序文件一个），并使用远转跳，但仅有一个缺省的数据段。大模式有多个代码段和多个数据段，所有的指针缺省为远。编写高效的分段代码需要

很多技巧，并在其他的地方有很好的描述。

分段寻址需要连接器做很多的工作。程序中的每个地址都有一个段和一个偏移量。目标文件有很多块代码需要连接器打包进段中。运行在实模式的可执行文件需要标出所有出现在程序中的段值，当程序装载到系统中的时候可以重载到实际的段中。运行在保护模式的程序更需要标出数据需要被装载到哪个段，还有每个段的保护模式(代码，只读数据，读写数据)。

尽管386支持所有286支持的16位分段特性，也支持所有的分段特性的32位版本，多数32位程序根本不使用分段方式。增加到386中的分页特性提供了分段所能提供大多数实际的有点，而无需付出性能的代价和编写分段代码所带来的难度。

多数386的操作系统将程序运行在微小模式下，更多的时候称为平坦模式，因为一个386中的段不再是微小的。系统建立一个代码段和一个数据段，每个段有4G长，并将它们映射到完全的32为分页的地址空间中。即使程序只使用一个段，这个段仍然能够覆盖整个地址空间。

386使再同一程序中同时使用16位和32位分段成为可能，只有很少的操作系统，主要是Windows 95和Windows 98使用了这个特点。Windows 95和98在一个共享地址空间中的16位段中运行很多传统的Windows 3.1的代码，而每个新的32位程序运行在其自己的微小模式下的地址空间中，并将16位程序的地址空间中映射过来，允许相互调用。

嵌入式体系机构

嵌入式系统中的连接有很多很少在其他环境下出现的问题。嵌入式芯片只有有限的内存和有限的性能，但一个嵌入式程序可以放到成千上万的设备的芯片中，在仅可能少的内存中让程序运行得尽可能快会带来很多回报。因为嵌入式系统使用低成本的通用芯片，如80186，其他的使用专用的处理器，如Motorola 56000系列作为数字信号处理器(DSP)。

怪异的地址空间

嵌入系统具有很小的地址空间，而且分布怪异。一个64K的地址空间可以包含快速的芯片上的ROM和RAM，慢速的芯片外的ROM和RAM，芯片上的外设和芯片外的外设。系统会有多个非连续的ROM和RAM区域。56000系统有64K个24位的字的地址空间，每个都是RAM、ROM和外设的组合。

嵌入式芯片系统的研制是在搭载有处理器和其他逻辑电路和芯片的主板上进行的。一般说来，同一个处理器的不同的研发主板的都有不同的内存布局。不同类型的芯片有不同容量的RAM和ROM，所以程序员需要在将程序挤在一个小的内存中和换用一种更昂贵的带有更多内存的新型号的芯片中选择。

一个嵌入式系统的连接器需要一种方法来详细说明连接的程序的分布，为特定代码和数据分配的内存地址，甚至具体例程和变量的内存地址。

非统一内存

对芯片上的内存的引用比芯片外的内存的引用要快，所以在一个有两种内存的系统中，最时间敏感的例程需要放在快速内存中。有的时候可以将程序的所有的时间敏感的程序代码在连接的时候放到快速内存中。其他的时候可以将存在慢速内存中的代码和数据在需要的时候拷贝到快速的内存中，所以不同的例程可以在不同的时候分享同一块快速内存。对于这种技巧，能够告诉连接器诸如”将这个代码放到XXXX，但连接成好象它在YYYY中”是非常有用的，所以在运行的时候从慢速的XXXX拷贝到快速内存的YYYY的时候的代码正确的。

内存对齐

DSP对某些数据结构有更严格的对齐的要求。例如，56000有一种处理线路缓存非常有效的寻址模式，只要缓存的基地址是对齐在至少和缓存一样大的2的指数次方的边界(例如，一个50字的缓存需要对齐在64个字的边界上)。快速傅立叶变换(FFT)，一种对信号处理极其重要的计算，需要地址位的运算，也需要FFT处理的数据对齐在2的指数次方的边界。和传统的体系机构不一样，对齐要求取决于数据矩阵的大小，所以将数据高效地封装到内存中就需要很多技巧和复杂的处理。

练习

1. 一个SPARC程序包含这些指令。(这些并不是实用的程序，仅是一些指令的举例。)

```

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的调用完成了什么功能？

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再于指令的低13位求或放入寄存器，寄存器1中会有什么地址？

1d. 如果连接器将X移到0x2504处但不改变例子中的代码的位置，需要将10

00中的指令如何修改使其仍然指向X?

2. 一个奔腾程序包含这些指令。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. 例程和数据字节的位置是什么? (提示:在x86系统上,相对地址是根据指令下个字节的地址计算的)

2b. 如果例程B是在地址0x0F00,数据Q是在地址0x3456,例程中的?字节是什么?

3. 连接器库代码是否需要理解目标文件结构的每个指令集。如果一个目标系统新型寻址方式添加新的指令,是否需要修改连接器来支持。如果象86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,那个时候程序员需要在深夜工作因为只有那时候他们才能使用机时,而不是因为他们在深夜睡着,很多计算机系统使用位而不是字节地址。例如,PDP-10使用6位字和8位寻址方式,每行指者是一个字,操作数也在字的下部。(程序也可以在数据段的上半部分存储地址,尽管没有直接支持连接一个内存寻址的系统)一个对字寻址的系统和位寻址有多少不同?

5. 编写一个可修改目标系统的连接器有多复杂,就是说,可以编写只修改连接器代码的一小部分吗?即使一个连接器来处理不同的目标系统如何可以处理不同操作数的目标连接器呢?尽管不是同一个连接器的工作?

第 3 章 目标文件

```
$Revision: 2.6 $  
$Date: 1999/06/29 04:21:48 $
```

编译器和汇编器生成包含从源程序生成的二进制代码和数据的目标文件。连接器将多个目标文件合并成一个，载入器将目标文件调入内存。(在任何一个集成编程环境下，当程序员需要建立一个程序的时候，虽然编译器、汇编器还有连接器会自动被隐含调用，但它们确实在运行)在这章中，我们将详细探讨目标文件的格式和内容。

目标文件中都有什么？

一个目标文件包含五种信息。

- **头信息**: 关于文件的整体信息，如代码的大小，它所来自的源文件的名字，建立日期等。
- **目标代码**: 编译器或汇编器生成的二进制指令和数据。
- **重载**: 一个目标文件需要被连接器修改的目标的位置的表。
- **符号**: 这个模块定义的全局符号，需要从其他模块导入的符号和连接器定义的符号。
- **调试信息**: 和连接过程无关但但在调试的时候需要的其他信息。包括源文件和行数信息，本地符号，目标文件使用的数据结构的描述，如C结构描述。

(一些目标文件的内容比这些还要多，但在这章中，我们仅讨论这些。)

并不是所有的目标文件都包含这些信息，非常可能仅仅有有用的形式，但除了目标代码外只有很少或没有其他的信息。

设计一个目标格式

对一个目标格式的设计是把各种对目标文件的使用需求综合在一起得出的。一个文件可以是**可连接的**，可以作为一个连接编辑器或连接器的输入文件；可以是**可执行的**，可以被调入内存并作为一个程序运行；可以是**可载入的**，可以被调入内存并作为和程序一起使用的库；或者也可以都是这三者。有些格式仅支持一两种应用，其他的则都支持。

在一个可连接的文件中，除了目标代码外，还包含很多连接器需要的符号和重载信息。目标代码通常分成很多小的逻辑段，并被连接器按不同的方式处理。一个可执行文件会包含目标代码，通常对齐到页面，这样文件可以被映射到地址空间中，但无需包含任何符号(除非它们使用动态连接)，只需要很少或不需要重载信息。目标代码一个一个单一的大的段，或一组反映硬件执行环境的段，通常是只读与可读写页面。根据系统运行时环境的细节的不同，一个可载入文件可以只包含目标代码，也可以包含全部符号和重载信息，可以进行运行时符号连接。

这些不同的应用存在一些冲突。面向逻辑的可连接段的组合很少能和面向硬件的可执行段的组合相吻合。尤其在小型的计算机系统中，可连接文件是有连接器一次一个地读写的，而可执行文件是一次完整地调入系统主存中。这种区别在MS-DOS的可连接OMF格式和可执行的EXE格式之间最明显。

我们会对各种流行的格式做一个浏览，从最简单的开始，最好到最复杂的格式。

空目标格式: MS-DOS .COM 文件

你可以非常可能遇到一个除了运行代码外没有任何其他信息的可用的目标文件。MS-DOS的.COM格式是最著名的例子。一个.COM文件除了二进制代码外没有任何其他东西。当操作系统运行一个.COM文件的时候，仅仅是将文件的内容载入一块从0x100开始的可用的内存空间，(0-FF是PSP，程序段前缀，包含程序行参数和其他参数)，将x86所有的段寄存器都指向PSP，SP(堆栈指针)指向段的稳步，因为堆栈是向下生长的，然后转跳到载入程序的开头。

x86的分段的体系结构允许这种方式的运行。因为所有的x86程序的地址都解析成与当前段的相对地址，所有的段寄存器都指向段的开头，程序总会装载到相对与段的0x100位置。因此，对于一个可以装到一个单一的程序，无需对代码和数据的修改，因为段相关的地址可以在连接的时候确定。

那些不能装载在一个段中的程序，对代码和数据的修改是程序员的任务，而且确实存在一些程序通过在开始的时候提取段寄存器的值，在程序的其他地方再在保存的段值上增加数值。当然，这种工作非常令人厌烦，因为连接器和载入器应该自动地完成这些工作，而MS-DOS在.EXE方式下确实如此，在这章的后面会讨论。

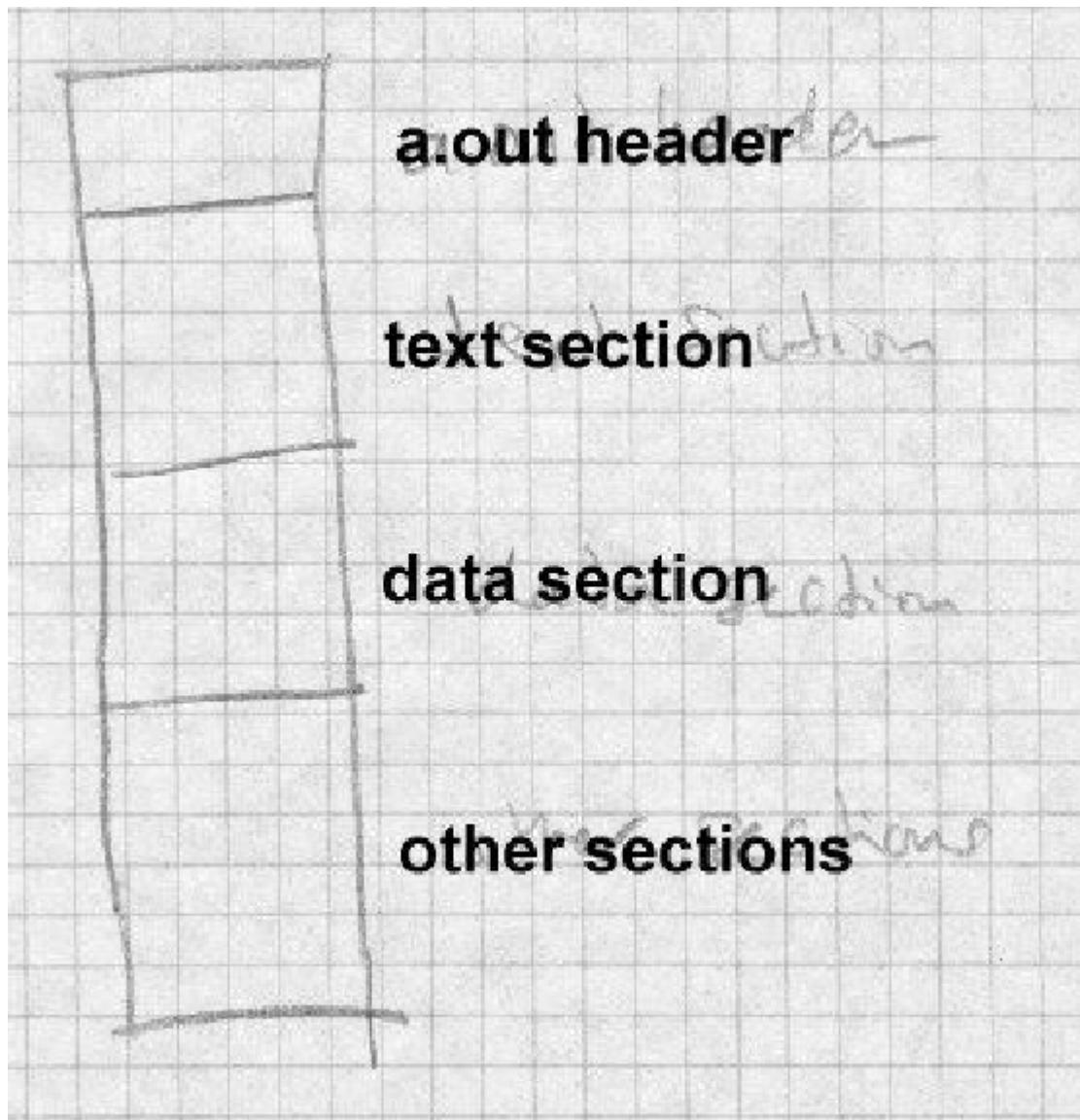
代码区: Unix a.out 文件

带有硬件内存重载的计算机系统（当前几乎所有的系统都有）通常在为每个新运行的程序建立一个新的带有空的地址空间的进程，程序可以在固定的地址开始，载入的时候无需重载。Unix的a.out就是这种情况。

在最简单的情况下，一个a.out文件包含一个小的文件头，后面接着可执行代码（由于历史的愿意称为文本区），然后是静态数据的初试值，如图1所示。PDP-11只有16位地址空间，将程序限制在64K大小。这个限制很快就变得太小，所以PDP-11得后续型号提供了分离的代码空间(I代表指令空间)和数据空间(D空间)，所以一个单独的程序可以分别有64的代码和64K的数据。为了支持这个特性，编译器、汇编器和连接器被修改成建立两区的目标文件，代码区在第一个区，数据区在第二个区，程序载入器将第一个区装载到进程的I空间，第二个到D空间。

图3-1：简化的a.out

a.out头
文本区
数据区
其他区



分离的I和D区还能带来另一个性能方面的好处：因为一个程序不能更改它自己的I区，一个程序的多份拷贝可以共享程序代码的同一个拷贝，而有各自的程序数据区的拷贝。在一个象Unix的分时系统中，shell(命令行界面)和网络进程有多个拷贝在同时运行，共享程序代码可以带来可观内存的节省。

目前唯一还比较常见的使用代码和数据分离的寻址的系统是286(或运行在16位保护模式中的386)。即使在带有更大地址空间的更加现代计算机系统中，操作系统处理在虚拟内存中的只读代码页中共享的代码比读/写页面的效率更高，所以所有现代的载入器都支持它们。这意味着连接格式必须至少能够标出只读和可读写区。在实际应用中，多数连接格式有多个区，如只读数据，为后续连接的符号和重载信息，调试符号和共享库信息。(Unix在传统上将文件区称为段，所以我们用这个术语来讨论Unix文件格式。)

a.out 文件头

文件头的格式随着不同的Unix版本而不同，但如图2所示的BSD Unix版本是

比较典型的。(在这章的例子中, int的值是32位的, short的是16位。)

图3-2: a.out文件头

```
int a_magic; // magic number
int a_text; // text segment size
int a_data; // initialized data size
int a_bss; // uninitialized data size
int a_syms; // symbol table size
int a_entry; // entry point
int a_trsize; // text relocation size
int a_drsiz; // data relocation size
```

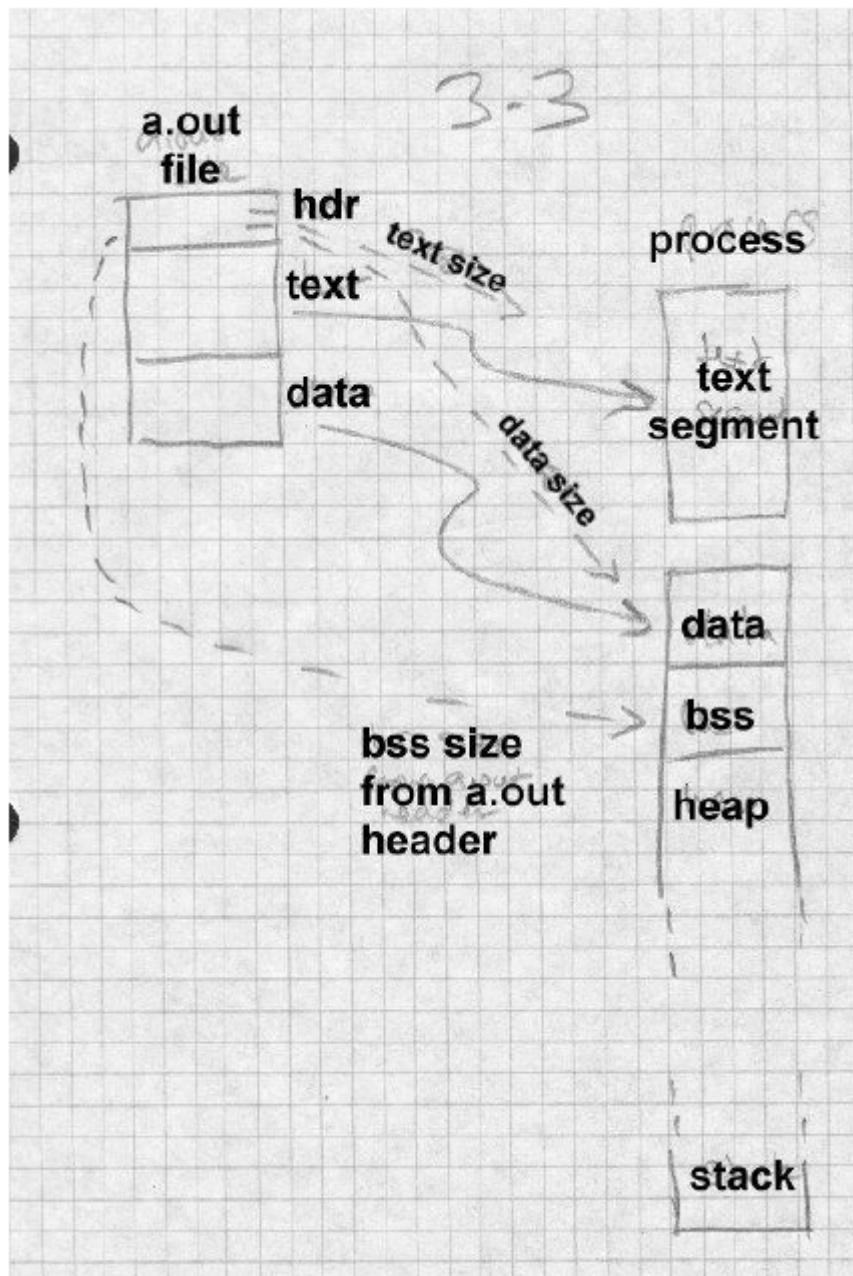
magic number是一个说明这个可执行文件是什么类型的一个**神奇数**。(脚注: 历史上, 最初的PDP-11的神奇数是八进制407, 是一个转跳到文本段开始的头部的7个字节后的位置的转跳指令。也就是一种位置无关代码的原始格式。引导程序可以将包含文件头的整个执行文件调入内存, 通常是地址0, 然后跳到载入的文件启动程序。只有很少的独立程序使用过这个特性, 但神奇数407却伴随了我们25年。)不同的神奇数告诉操作系统的程序载入器用不同的方法将文件调入内存, 我们下面来讨论一下这个问题。文本段和数据段的长度a_text和a_data是文件头后面的只读代码和可读写数据的长度。因为Unix系统自动将新分配的内存初试化为0, 任何初试值为0或初试值无所谓的数据无需出现在a.out文件中。非初试化大小a_bss是说a.out文件中逻辑地跟在数据段后的非初试化(实际是初试化为0)数据有多大。

a_entry域给出了程序地开始地址, 而a_syms, a_trsize和a_drsiz是说文件中数据段后有多少符号表和重载信息。经过连接可以运行的程序无需符号或重载, 所以在可执行文件中这些域都为0, 除非文件中还有给调试器使用的符号信息。

与虚拟内存交互

操作系统载入并开始一个两段的文件的过程非常简单, 如图3所示。

图3-3: 载入一个a.out到一个进程
文件和段的图, 带有指向数据的箭头



从*a.out*文件头中得到段的尺寸。

- 检查是否已经存在这个文件的一个可共享的代码。如果存在，将那个段映射到进程的地址空间。如果没有，建立一个，将其映射到地址空间，然后将文件的文本段读入到新建立的内存段。
- 在内存中建立一个对合并的数据和BSS足够大的私有的数据段，将其映射到进程中，从文件中将数据段读入到数据段，并将BSS段清零。
- 建立并映射一个堆栈段(通常和数据段分离，因为数据堆和堆栈分别生长。)将命令行和调用程序的参数存如堆栈。
- 设置寄存器并转跳到开始地址。

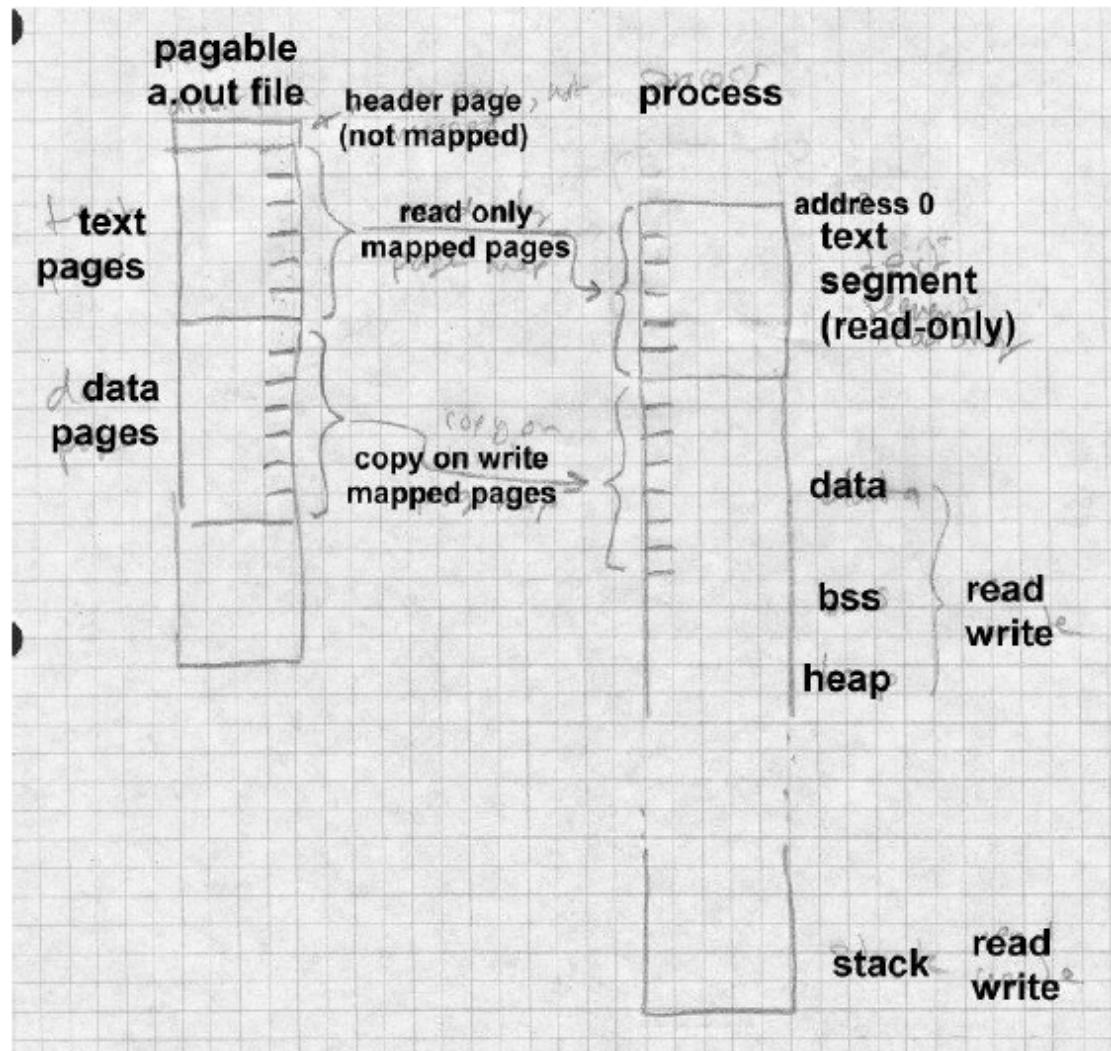
这种机制(被称为NMAGIC，N代表新，相对于1975年的技术)工作得非常好，PDP-11和早期的VAX Unix对所有的目标文件使用这种机制有很多年，而可连接的文件在*a.out*格式的生命周期内一直使用这种格式，直到90年代。当Unix系统使

用了虚拟内存技术后，对这种简单的机制进行了一些改进，加快了程序的载入速度，并节省了可观的实际内存。

在一个分页系统中，这种简单机制为每个文本段和数据段都分配新的虚拟内存。因为a.out文件已经保存在磁盘中，目标文件本身可以映射到进程的地址空间。这就可以节省磁盘空间，因为只需要为程序需要写入的页面分配虚拟内存需要的新的磁盘空间，并能加快程序的启动，因为虚拟内存系统只需要将程序确实在使用的页面从磁盘中载入，而不是整个文件。

对a.out文件进行很少的改动就可以完成这个功能，如图4所示，就成为被称作ZMAGIC的格式。这些改动就是将目标文件的段按照页边界对齐。在有4K页面的系统中，a.out文件头被扩展到4K，文本段的大小被对齐到下一个4K的边界。无需对数据段的大小进行对齐，因为BSS段逻辑地跟在数据段后面，并且被程序载入器清零。

图3-4：将a.out映射到进程中
文件和段的图片，并将页面帧映射到段中。

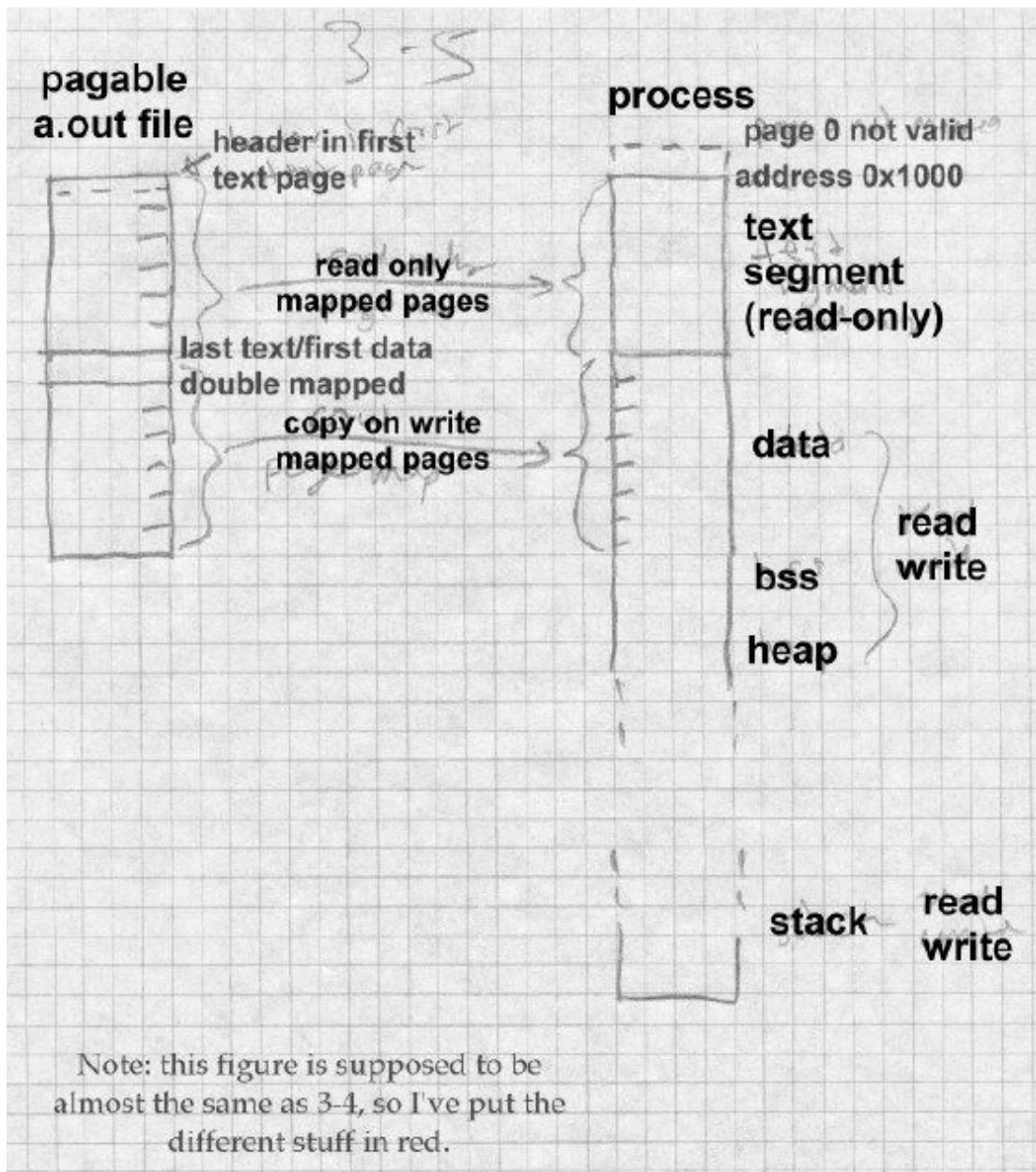


ZMAGIC文件减少了不必要的换页，但代价是浪费了很多磁盘空间。a.out文件头仅有32字节长，但需要分配整个4K空间。文本段和数据段中的沟平均也浪费

了2K，一个4K页面的一半。QMAGIC格式改进为压缩的可换页格式。

压缩的可换页文件将a.out的文件头视为文本段的一部分，因为无需将文本段中的代码从地址零开始。实际上地址零是一个非常不好的程序载入点，因为没有初始化的指针变量通常保存着零。代码通常紧接着文件头，整个页面映射到进程的第二个页面，并不映射第一个页面，这样对地址零的指针引用将会出错，如图5所示。这会产生将文件头映射也被映射到进程中的无害的副作用。

图3-5：将一个压缩的a.out文件映射到一个进程
文件和段的图，将页面帧映射到段。



QMAGIC格式的可执行文件中的文本和数据段分别都对齐到一个完整的页面，因此系统可以非常容易地将文件页面映射到地址空间页面中。数据段的最后一页用0填满用作BSS数据，如果BSS数据比填充的部分大，a.out文件头中包含着

需要分配的剩下的BSS区域的大小。

尽管BSD Unix将程序装载到地址0处(或者QMAGIC格式的0x1000)，其他版本的Unix将程序装载到其他的地方。例如，运行在Motorola 68K系列上的System V(System V) Unix将程序装载到0x80000000，在386系列上装载到0x80480000。只要页面是对齐的，连接器和操作系统都统一起来，将程序装载到哪个页面无关紧要。

重载：MS-DOS EXE 文件

a. out格式对于为每个进程分配新的地址空间的系统是足够的，每个进程可以装载到相同的逻辑地址。但很多系统没有这么幸运。有些系统将所有的程序调入到同一个地址空间中，有些则给每个程序独有的地址空间，但并不总是将程序调入到相同的地址。(32位的Windows系统属于后面的情况。)

在这些系统中，可执行文件包含着**重载入口**，通常成为**修正**，标出程序中哪里的地址在程序装载的时候需要修改。

我们前面讨论了.COM格式，DOS总是将程序装载到一块连续的可用物理内存中。如果程序不能装到一个64K段，程序需要使用明确的段值来寻址程序和数据，在载入的时候，段值需要修改来适配程序的实际的装载地址。文件中保存的段值假定程序被装载到地址0，所以修正工作就是将程序实际装载的基础段落值加到文件中保存的每个段值上。也就是说，如果程序被加载到0x5000，段落就是0x500，一个对段12的引用就被重载为一个对段512的引用。段内的偏移量并不改变，因为程序是作为一个整体装载的，所以载入器仅需要修改段值。

每个.EXE文件都是以图6所示的文件头开始的。文件头后面跟着一些变长的额外信息(用于覆盖方式的载入器，自解压档案和其他应用有关的信息(hacker?))和一个含有32位段:偏移量格式的修正地址的表。修正地址是相对于程序的基地址，所以修正本身需要重载来找处程序中需要修改的地址。跟着修正表后面就是程序代码。在代码后面也可能会有被程序载入器忽略的更多的信息。(在下面的例子中，远指针是32位的，有一个16位的段和16位的偏移量。)

图3-6: .EXE文件头的格式

```
char signature[2] = "MZ"; // magic number
short lastsize; // # bytes used in last block
short nblocks; // number of 512 byte blocks
short nreloc; // number of relocation entries
short hdrsize; // size of file header in 16 byte paragraphs
short minalloc; // minimum extra memory to allocate
short maxalloc; // maximum extra memory to allocate
void far *sp; // initial stack pointer
short checksum; // ones complement of file sum
void far *ip; // initial instruction pointer
short relocpos; // location of relocation fixup table
short noverlay; // Overlay number, 0 for program
char extra[]; // extra material for overlays, etc.
void far *relocs[]; // relocation entries, starts at
```

relocpos

装载一个. EXE文件仅比装载一个. COM文件复杂一点。

- 读入文件头，检查神奇数来验证合法性。
- 找到一个适合的内存区域。minalloc和maxalloc域标明了在载入的程序后面需要分配的额外内存段落的最小值和最大值。(连接器总是缺省地将最小值设为程序中类似BSS的非初始化数据的大小，最大值为0xFFFFF。)
- 建立一个PSP，程序头部的控制区域。
- 紧跟着PSP读入程序代码。nblocks和lastsize域定义了代码的长度
- 开始在relocpos处读取nreloc个修正。对于每个修正，把程序代码的基本地址加到修正中的段值中，然后将重载后的修正作为一个指向程序地址的指针，将指针处的地址加上程序的基址。
- 将堆栈指针设到重载后sp，然后转跳到重载后的ip来开始程序。

除了和分段寻址相关的怪异的特性外，这是一个典型的程序载入的设置。在极少的系统中，程序的不同部分用不同的方法重载。在EXE文件不支持的286的保护模式下，可执行文件中的每个代码或数据段都被系统中的一个分离的段，但由于体系结构的原因，段值无法是连续的。每个保护模式的可执行文件在靠近开始的地方都有一个列出所有程序需要的段的表。系统建立一个与可执行文件中每个段相对应的实际段值的表。当进行修正处理时，系统在表中查到逻辑段值，然后用实际段值来替换，这个过程更多的象一个符号绑定而不是重载。

有些系统也支持在载入时的符号解析，但我们把这个题目保留到第10章。

符号和重载

到目前为止，我们考虑的目标文件都是可载入的，就是说，它们可以直接调入内存并运行。很多目标文件并不是可载入的，而是由一个编译器或汇编器生成的向一个连接器或库管理器传送的中间文件。这些可连接文件比可运行文件要复杂很多。可运行文件需要足够简单以在计算机的”裸芯片”上运行，而可连接文件是在软件层处理的文件，可以进行非常复杂的处理。理论上，一个连接载入器可以在程序载入的时候完成所有的连接器的功能，但为了效率的原因，载入器通常是尽可能地简单来加快程序的启动。(在第10章会讨论的动态连接，将很多连接器的功能移到了载入器中，虽然带来了一些性能的损失，但现代的计算机系统足够快，从动态连接中得到的好处足以抵消性能的损失。)

我们现在看看五种逐渐复杂的格式：在BSD Unix系统中可重载的a.out，系统V中的ELF，IBM 360目标文件，用于32位Windows系统的扩展的可连接COFF和可执行PE格式，用于pre-COFF Windows系统中的可连接OMF格式。

可重载 a.out

Unix对于可运行和可连接文件总是使用同一种目标格式，可运行文件把只用于连接器的部分去除。我们在图2中看到的a.out格式包含几个用于连接器的域。文本段和数据段的重载表的大小是a_trsize和a_drsiz，符号表的大小是a

_syms。这三个部分跟在文本和数据段的后面，如图7所示。

图3-7：简化的a.out

a.out头

文本段

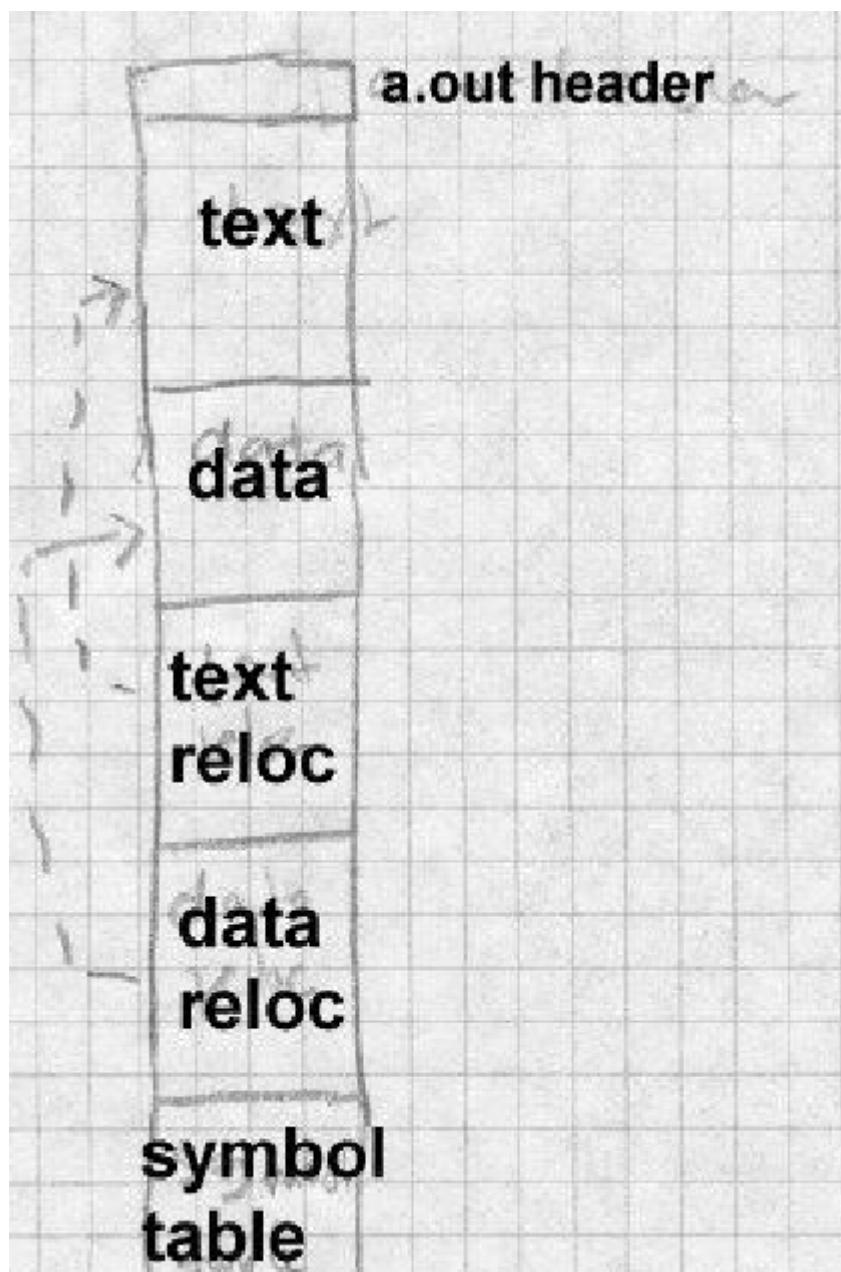
数据段

文本重载

数据重载

符号表

字符串表

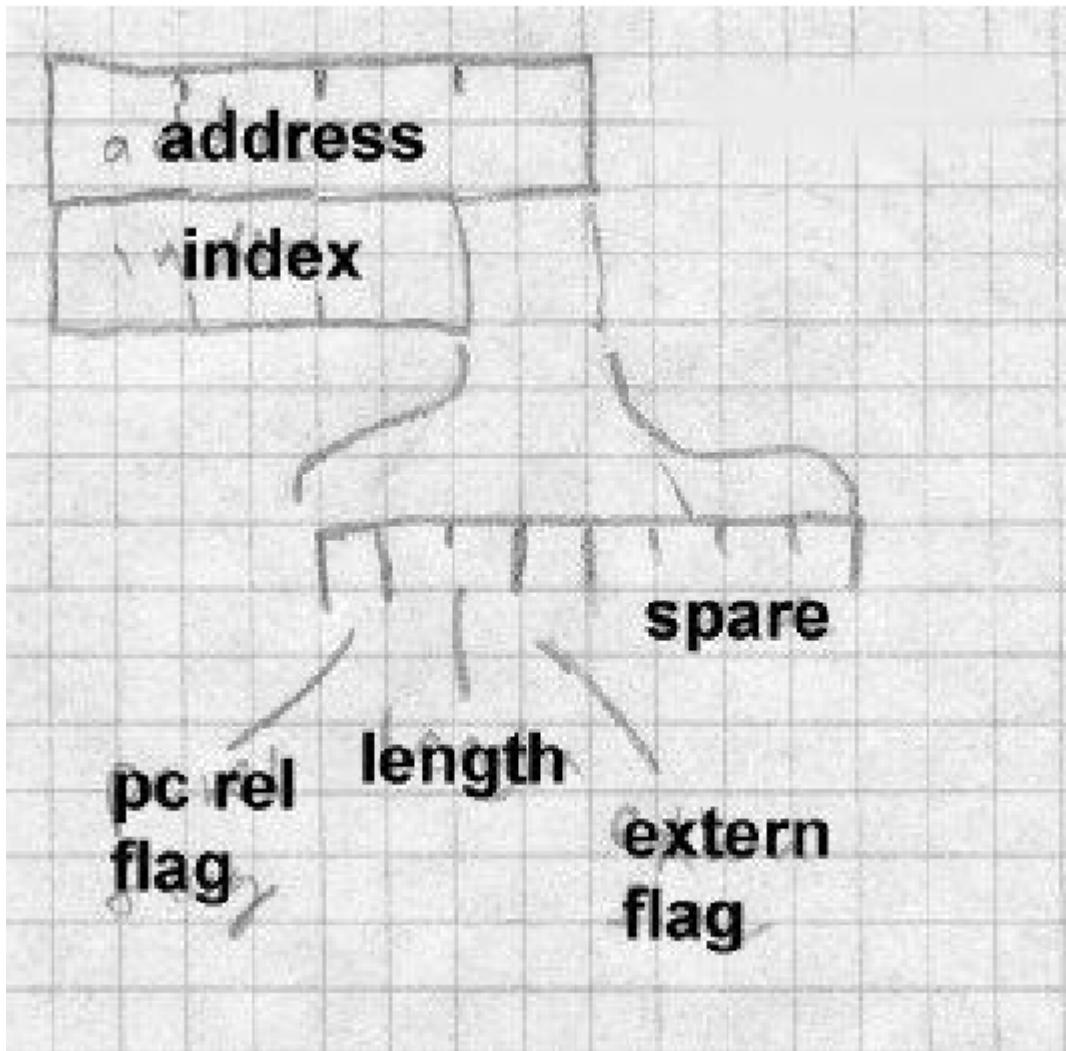


重载入口

重载入口有两个功能。当一个段被重载到一个不同的基地址的时候，重载入口标记出代码中需要修改的位置。在一个可连接的文件中，还有用来标记对未定义符号的引用的重载入口，这样，连接器就知道当符号最终被定义的时候去哪里修改符号值。

图8展示了一个重载入口的格式。每个入口包含了文本段或数据段中需要重载的地址，还有如何处理的信息。地址是一个重载项目从文本段或数据段开始的偏移量。长度域是说项目的长度，值是从0到3，也就是1, 2, 4或(在某些体系结构中)8个字节。pcrel标志是说这是一个“相对PC”的项目，就是说在指令中作为相对地址。

图3-8：重载入口格式



外部标志用来控制对索引域的翻译，来确定重载指的是哪个段或符号。如果外部标志是关，说明这是一个直接的重载项目，索引指出对哪个段(文本、数据或BSS)寻址。如果外部标志是开，说明这是一个对外部符号的引用，索引是文

件的符号表中的符号值。

在多数体系结构中这种重载格式是足够的，但有些比较复杂的结构需要额外的标志位来标志，如三字节的370地址常数或SPARC系统中高和低半地址常量。

符号和字符串

一个a.out文件的最后一个部分是符号表。每个入口有12个字节，描述一个符号，如图9所示。

图3-9：符号格式

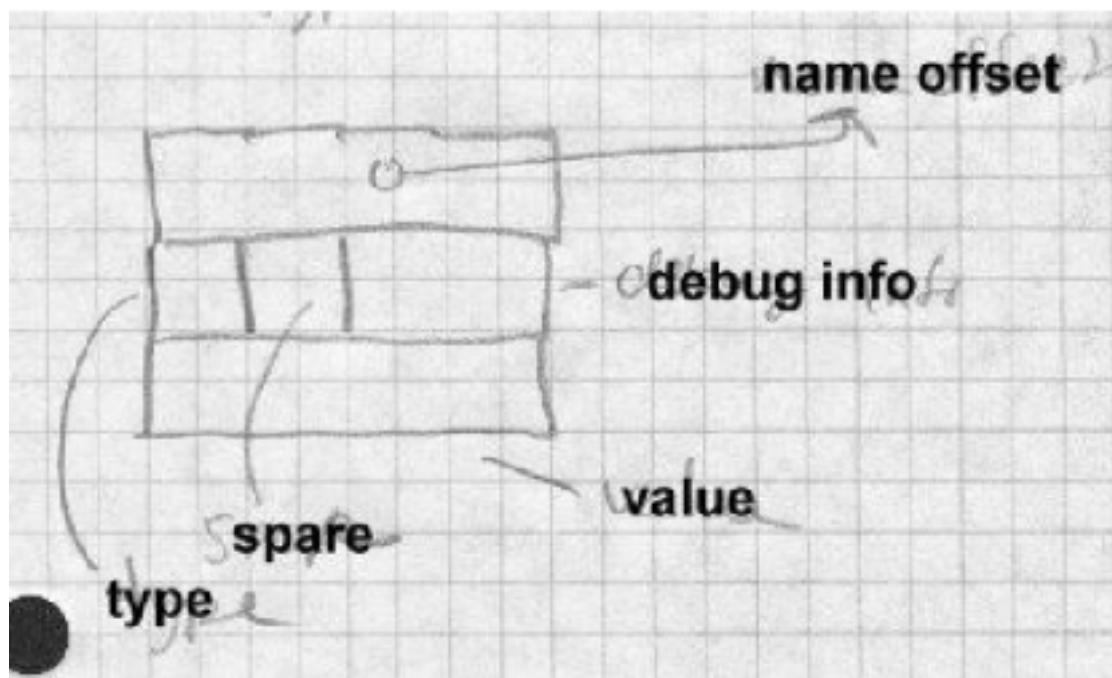
4字节名字偏移量

1字节类型

1空闲字节

2字节调试信息

4字节数值



Unix编译器允许任意长的标示，所以有名字符串都在一个跟在符号表后的字符串表。符号表入口的第一个内容是符号的用null终结的名字在字符串表中的偏移量。在类型字节中，如果低位设置为1，则符号是外部(实际这是一个错误的叫法，应该成为全局，对其他模块可见)的。非外部符号无需连接，但可以用于调试器中。剩下的位是符号类型。最重要的类型包括：

- **文本、数据或 bss:** 在这个模块中定义的符号。外部标志位可以设为开也可以设为关。值是这个符号在模块中相关的重载地址。
- **Abs:** 一个绝对非重载符号。(很少出现在调试器信息外。)外部标志可以为开也可以为关。值是符号的绝对地址。
- **未定义:** 一个在这个模块中未定义的符号。外部标志必须设置为开。值通常是0，但见下面讨论的”公共块技巧(hack)”。

- 对于老式的编程语言如C和Fortran来说，这些符号是足够的，对C++则刚刚够用。

作为一个特殊的情况，编译器可以使用未定义的符号来要求连接器来解析那个符号名字下的一个存储块。如果一个未定义的外部符号的值非零，这个值对于连接器来说就是程序要求这个符号要寻址多大的存贮空间的提示。在连接的时候，如果没有符号的定义，连接器就在BSS段中为这个符号建立一个存储空间，空间的大小就是在所有连接的模块中最大的提示值。如果符号在一个模块中定义了，则连接器使用这个定义并忽略提示大小。这种”公共块技巧”支持Fortran语言中的公共块和C语言中的未初试化外部数据的典型使用(虽然不是标准的构造)。

a.out 小结

a.out格式是使用分页的相对简单的系统中简单有效的格式。现在已经不再流行，因为它不能很容易地支持动态连接。同时，a.out还不支持C++，因为C++需要对初试化和最终化代码的特殊处理。

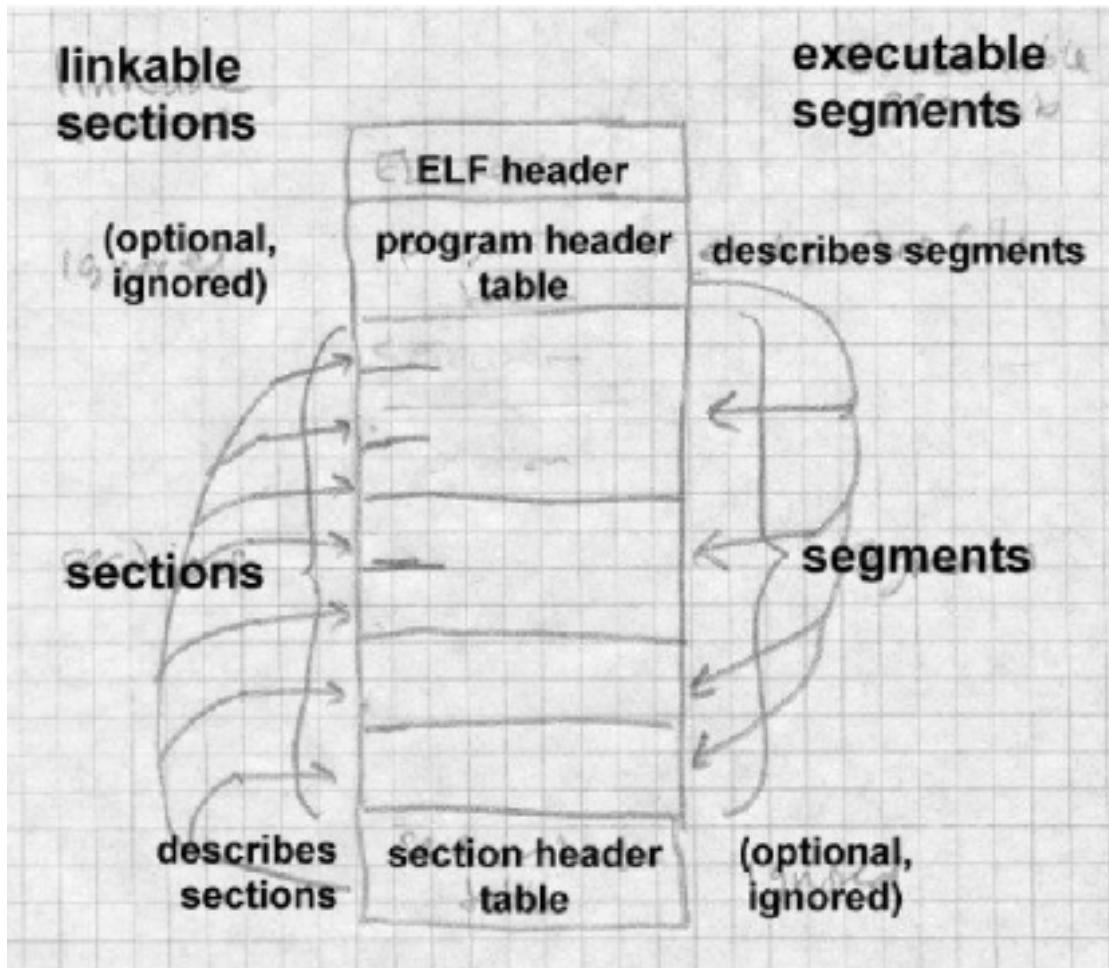
Unix ELF

Unix社区使用了10多年的a.out格式，但随着Unix System V的出现，AT&T决定需要引入更多的跨平台编译、动态连接和其他的系统特性。System V的早期版本使用COFF，通用目标格式文件格式，最初是为跨平台编译的嵌入式系统开发的，并不能在分时系统中很好地使用，因为它不能在不扩展的情况下支持C++或动态连接。System V的后期版本，COFF被ELF格式替换，即可执行和连接格式。ELF被流行的自由软件Linux和Unix的BSD变种广泛使用。ELF还有一个相关的成为DWARF的调试格式，我们在第5章讨论。这里我们讨论ELF的32位版本。还有一种直接将大小和地址扩展到64位的ELF的64位变种。

ELF文件有三种略微不同的变种：可重载的，可执行的和共享目标。可重载的文件是由编译器和汇编器生成的，但在运行前需要连接器的处理。可执行文件已经完成了所有的重载和符号的解析，除了需要在运行时解析的动态连接库符号。共享目标时共享库，包括用于连接器的符号信息和运行用的直接可运行代码。

ELF文件有一种不寻常的双面性，如图10所示。编译器、汇编器和连接器将文件看成一组由区头部表描述的逻辑区，而系统载入器将文件看成一组由程序头部表描述的段。一个单独的段通常包含多个区。例如，一个”可载入只读”段可以包含可执行代码区，只读数据区和用于动态连接的符号区。可重载文件由分区表，可执行文件有程序头部表，而共享库两者都有。区是连接器用来进行进一步处理的，而段是用来映射到内存中的。

图3-10：一个ELF文件的两种看法



所有的ELF文件都由ELF文件头开始，如图11所示。文件头设计成即使在文件面向的具有不同的字节顺序的计算机系统中也可以被正确解码的。头四个字节是标示ELF文件的神奇数，接下来的三个字节描述文件头的剩下的部分的格式。一旦程序读入class和byteorder标志，它就知道文件的字节顺序和字的大小，并可以做必要的字节交换和大小转化。如果需要的话，其他的域可以提供区头部和程序头部的大小和位置。

图3-11: ELF文件头

```

char magic[4] = "\177ELF"; // magic number
char class; // address size, 1 = 32 bit, 2 = 64 bit
char byteorder; // 1 = little-endian, 2 = big-endian
char hversion; // header version, always 1
char pad[9];
short filetype; // file type: 1 = relocatable, 2 = executable,
// 3 = shared object, 4 = core image
short archtype; // 2 = SPARC, 3 = x86, 4 = 68K, etc.
int fversion; // file version, always 1
int entry; // entry point if executable

```

```

int phdrpos; // file position of program header or 0
int shdrpos; // file position of section header or 0
int flags; // architecture specific flags, usually 0
short hdrsize; // size of this ELF header
short phdrent; // size of an entry in program header
short phdrcnt; // number of entries in program header
or 0
short shdrent; // size of an entry in section header
short phdrcnt; // number of entries in section header
or 0
short strsec; // section number that contains section
name strings
-----
```

可重载文件

一个可重载或共享目标文件可以理解成一组由区头部定义的区，如图12所示。每个区都包含一种单一类型的信息，如程序代码，只读或可读写数据，重载入口，或者是符号。模块中定义的每个符号都定义成相对一个区，所以一个过程入口就是相对于包含这个过程代码的程序代码区的。还有两种精灵区，分别是逻辑上包含非可重载符号的SHN_ABS(值为0xffff1)和非初始化数据块的SHN_COMMON(值为0xffff2)，是a.out格式中公共区技巧的发展。零区永远是一个空区，带有有一个全零的表入口。

图3-12: 区头部

```

int sh_name; // name, index into the string table
int sh_type; // section type
int sh_flags; // flag bits, below
int sh_addr; // base memory address, if loadable, or
zero
int sh_offset; // file position of beginning of secti
on
int sh_size; // size in bytes
int sh_link; // section number with related info or z
ero
int sh_info; // more section-specific info
int sh_align; // alignment granularity if section is
moved
int sh_entsize; // size of entries if section is an a
rray
-----
```

区类型包括：

- PROGBITS: 程序内容，包括代码，数据和调试器信息。
- NOBITS: 类似于PROGBITS，但文件本身中没有分配空间。用于在程序载

入时分配的BSS数据。

- SYMTAB和DYNsym: 符号表, 后面详细描述。SYMTAB表包含所有的符号, 主要用于普通的连接器, 而DYNsym只是用于动态连接的符号。(后面的这个表需要在运行时调入内存, 所以应该尽可能的小。)
- STRTAB: 一个字符串表, 和a.out文件中的字符串表类似。不象a.out文件, ELF文件能够而且通常为不同的目的保留不同的字符串表, 如区名字, 普通符号名字和动态连接名字。
- REL和RELA: 重载信息。REL入口将代码或数据中保存的基址加上重载值, 而RELA本身则在重载入口保存着需要重载的基址。(由于历史的原因, x86目标使用REL重载而68K使用RELA重载。)每种体系结构都有很多重载类型, 与a.out重载类型类似(实际是演化而来的)。
- DYNAMIC和HASH: 动态连接类型和运行时符号哈希表。
- 每个区有三个标志位: ALLOC, 在程序转载的时候这个区要占用内存; WRITE, 当该区被载入的时候是可写的; EXECINSTR, 该区包含了可执行机器代码。

一个典型的可重载可执行格式有大约12种区。其中很多区的名字对连接器是有意义的, 连接器根据其知道的名字找到该区进行特定的处理, 对于连接器不知道的类型(根据标志位)要么丢弃要么不做任何处理。

区包括:

- .text 是带有ALLOC+EXECINSTR属性的PROGBITS类型的区。对应于a.out的文本段。
- .data 是带有ALLOC+WRITE属性的PROGBITS类型的区。对应于a.out的数据段。
- .rodata 是带有ALLOC属性的PROGBITS类型的区。是只读数据, 所以没有WRITE属性。
- .bss 是带有ALLOC+WRITE属性的NOBITS类型的区。BSS区在文件中不占空间, 所以是NOBITS类型, 但在运行时分配, 所以有ALLOC属性。
- .rel.txt, .rel.data, 和 .rel.rodata 每个都是REL或RELA类型的区。保存着对应的文本或数据区的重载信息。
- .init和.fini, 每个都是带有ALLOC+EXECINSTR属性的PROGBITS类型的区, 但是分别包含了在程序启动或终止的时候运行的代码。C和Fortran语言不需要这些区, 但这些区对C++非常重要, 因为C++使用了带可执行的初始化和终止代码的全局数据。
- .symtab和.dynsym 分别是SYMTAB和DYNsym类型的区, 普通和动态连接器符号表。动态连接器符号表带有ALLOC标志, 因为需要在运行时装载。
- .strtab和.dynstr 都是STRTAB类型的区, 一个命名字串的表, 用于符号表或区表的区名字。.dynstr区保存的是动态连接器符号表的字符串, 因为需要在运行时载入系统所以所有的ALLOC标志都设置为1。
- 还有一些特殊的区, 如.got和.plt, 是用于动态连接的全局偏移量表和进程连接表, .debug包含了调试器用的符号表, .line为调试器保存着从目标代码到源程序行数的对应, .comment保存着文档字符串, 通常是在版本控制中的版本号。

还有一个不同寻常的.interp区, 包含了一个用作解释器的程序的名字。如

果这个区存在，那么不直接运行这个程序，而是运行解释器并将ELF文件作为参数传递给该程序。Unix很多年来一直使用自运行文本文件，即使用

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

作为文件的第一行。ELF将这个解释器功能扩展到非文本文件。实际使用中，这个功能用来调用运行时动态连接器来装载程序并连接任何需要的共享库。

ELF符号表类似于a.out的符号表。包含一个入口的数组，如图13所示。

图3-13: ELF符号表

```
int name; // position of name string in string table
int value; // symbol value, section relative in reloc,
// absolute in executable
int size; // object or function size
char type:4; // data object, function, section, or special case file
char bind:4; // local, global, or weak
char other; // spare
short sect; // section number, ABS, COMMON or UNDEF
```

a.out符号入口是用多一些的域充满的。尺寸域说明一个数据对象有多大(尤其是对于未定义的BSS数据，又使用了公共块技巧)。一个符号的绑定可以是本地的，仅在这个模块中可见；或者全局的，所有地方都可见；或弱的。一个弱的符号是半心的全局符号：如果存在一个未定义的弱符号，连接器就使用该定义，但如果没没有定义，则将其值设为缺省的零值。

符号的类型可以是普通的数据或功能。每个区都定义了一个区符号，通常和这个区的名字相同，用于重载入口中。(ELF重载入口都是相对于符号的，所以需要一个区符号来指定一个项目是和文件中的一个区相对的。)文件入口是一个包含了源文件名字的精灵符号。

区号是相对于定义符号的区，如，功能入口点定义为相对于.text区。还有三个精灵区，UNDEF用于未定义符号，ABS用于非重载绝对符号，COMMON用于尚未分配的公共块。(COMMON符号的值给出了对齐的粒度，而大小给出了最小的尺寸。一旦被连接器分配后，COMMON符号就移到.bss区中。)

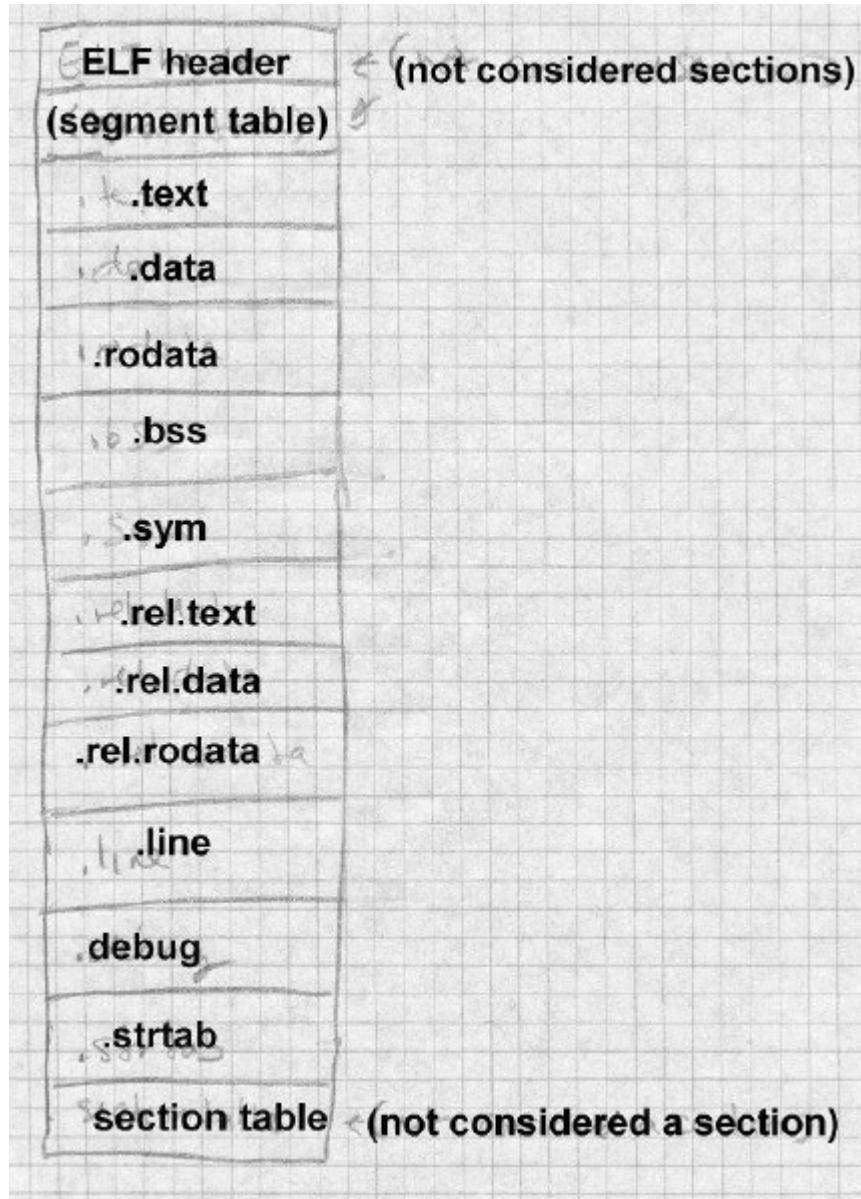
一个典型的完整的ELF文件如图14所示，包含很多区，用于代码、数据、重载信息、连接器符号和调试器符号。如果文件是一个C++程序，还可能包括.init、.fini、.rel.init和.rel.fini区。

图3-14: 一个可重载文件示例

ELF文件头

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

.rel.text
.rel.data
.rel.rodata
.line
.debug
.strtab
(段表，不能当成一个段)



ELF 可执行文件

一个可执行的ELF格式文件和一个可重载的ELF文件有相同的格式，但数据区安排成可以映射到内存中并运行。文件在ELF头部后面还有一个程序头部。程序头部定义需要被映射到内存的区。程序头部如图15所示，是一个区描述的数组。

图3-15: ELF程序头部

```

int type; // loadable code or data , dynamic linking i
nfo , etc.
int offset; // file offset of segment
int virtaddr; // virtual address to map segment
int physaddr; // physical address , not used
int filesize; // size of segment in file
int memsize; // size of segment in memory (bigger if
contains BSS)
int flags; // Read , Write , Execute bits
int align; // required alignment , invariably hardware
page size
-----
```

一个可执行格式通常只包含很少的段，一个代码的只读段，只读数据段和用于可读写数据的可读写段。所有的可载入的段都封装到合适的段中，可以通过一个或两个操作将文件映射到内存中。

ELF文件将在QMAGIC的a.out文件中的”地址空间中的头部”的技巧进行了扩展，使可执行文件尽可能的小，带来的代价就是地址空间中的一些溢出。一个段可以在任何文件偏移量处开始和结束，但段的虚拟起始地址在文件中的起始地址必须都有相对于对齐位的相同的低位模，如，都在一个页面的相同偏移量处开始。系统将段开始的页面到段结束的页面整个映射到内存中，即使段仅占用了起始和结束的页面的一部分。图16显示了一个典型的段分布。

图3-16: ELF可载入段

	File offset	Load address	Type
ELF header	0 0x8000000		
Program header	0x40 0x8000040		
Read only text (size 0x4500)	0x100 0x8000100 LOAD, Read/Execute		
Read/write data (file size 0x2200, memory size 0x3500)	0x4600 0x8005600 LOAD, Read/Write/ Execute		

映射的文本段包括ELF头部、程序头部和只读数据，因为ELF和程序头部都在文本段开始的同一个页面中。文件中除了数据外的可读写段紧跟在文本段的后面。文件中的页面被映射到文本段的最后一个只读页面和数据段的第一个写入时拷贝(copy-on-write)页面。在这个例子中，如果计算有4K的页面，可执行文件中的文本结束在0x80045ff，那么数据在0x8005600处开始。文件被页面映射到在

0x8004000开始的文本段的第一个页面，头0x600字节包含着从0x8004000到0x80045ff的文本，和从0x8005000开始的数据段，剩下的页面包含着从0x8005600到0x80056ff的数据的开始部分。

BBS区仍然是逻辑连续的，并终止在数据段的可读写区，在这个例子中，文件大小和内存大小的区别是0x1300字节。数据段的最后一个页面从文件中映射到内存中，但一旦操作系统开始将BBS段清零，写入时拷贝(copy-on-write)系统为这个页面建立一个私有拷贝。

如果文件包含.`init`或.`fini`区，这些区也是只读文本段的一部分，连接器在调用主程序之前在入口点插入代码来调用.`init`区代码，在主程序返回的时候调用.`fini`区代码。

ELF共享目标文件包含可重载和可执行文件的所有的内容。在文件的开始部分有程序头部，后面是可载入段的各个区，包括动态连接信息。接下来的由可载入段组成的区是可重载符号表和其他信息，这些信息是连接器在建立指向共享目标的可执行程序的时候所需要的，最后是区表。

ELF 格式小结

ELF是相对复杂的格式，但很好的完成了它所要完成的目标。它的可重载格式足够灵活来支持C++，对于带有动态连接的虚拟内存系统也是一个高效的可执行格式，可以很容易地把可执行页面直接映射到程序地址空间中。它也支持从一个平台到另一个平台的跨平台编译和跨平台连接，每个ELF文件都带有足够的信息来制定目标的体系结构和字节顺序。

IBM 360 目标格式

IBM 360的目标格式是在60年代早期设计的，但一直沿用至今。最初它是为80列打孔卡设计的，但已经改进成适应现代计算机的磁盘系统。每个目标文件都包含一组控制区(csect)，可以是分别命名的可重载代码和/或数据块。通常每个源程序都编译成一个csect，或一个代码csect和一个数据csect。如果一个csect有名字，可以用作指向这个csect的开始地址的符号；其他符号的类型包括一个csect中定义的符号，未定义的外部符号，公共块和一些其他符号。在目标文件中定义或使用的符号都被分配一个小的整数，即外部符号ID(ESID)。一个目标文件是一个有通用格式的80字节记录的序列，如图17所示。每个记录的第一个字节是0x02，一个标记这个记录是文件一部分的值。(一个由空开始的记录被连接器作为命令处理。)第2到4字节是记录类型，TXT代表程序代码或“文本”，ESD代表一个定义符号和ESID的外部符号目录，RLD代表重载目录，END代表最后一个记录还定义了起始点。记录中余下的直到第72个字节根据记录类型为特定的内容。第73到80字节被忽略。在一个实际的打孔中，它们通常是序列号。

目标文件由一些定义csect的ESD记录开始，然后是TXT记录，RLD记录，最后是END记录。记录的顺序有很大的灵活性。几个TXT记录都可以重新定义同一个地方的内容，其中最后一个起作用。这就可以(而且并不少见)在一摞目标卡的最后打一些“补丁”卡，而不用重新汇编或编译。

图3-17: IBM目标记录格式

```
char flag = 0x2;
char rtype[3]; // three letter record type
char data[68]; // format specific data
char seq[8]; // ignored, usually sequence numbers
```

ESD 记录

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ESID。

图3-18: ESD记录格式

```
char flag = 0x2; // 1
char rtype[3] = "ESD"; // 2-4 three letter type
char pad1[6];
short nbytes; // 11-12 number of bytes of info: 16, 3
2, or 48
char pad2[2];
short esid; // 15-16 ESID of first symbol
{ // 17-72, up to 3 symbols
    char name[8]; // blank padded symbol name
    char type; // symbol type
    char base[3]; // csect origin or label offset
    char bits; // attribute bits
    char len[3]; // length of object or csect ESID
}
```

每条ESD记录可以定义至多三个符号, 以及连续的ESID。符号最多可有8个E BDDIC字符。符号类型有:

- SD和PC: 定义一个csect为区定义或私有代码。这个(SD)csect最初是来源于csect的开始的逻辑地址, 通常是零, 长度就是csect的长度。属性字节保存着标志这个程序寻址是24位还是31位的标志, 还有是否需要载入到一个24位或31位地址空间的标志。PC是一个带有空名字的csect, 在一个程序中所有的csect名字必须唯一, 但可以有多个非命名的PC区。
- LD: 标签定义。Base域就是标签在这个csect中的偏移量, len域就是这个csect的ESID, 没有属性标志。
- CM: 公共块。Len域就是公共块的长度, 其他域忽略。
- ER和WX: 外部引用和弱外部符号。符号在其他地方定义。如果在程序的其他地方没有符号的定义, 连接器会对一个ER符号报错, 但不对WX符号报错。
- PR: 精灵寄存器, 一个在连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

接下来就是文本记录，如图19所以，包含了程序代码和数据。每个文本记录定义了在一个csect中至多56个连续字节。

图3-19: TXT记录格式

```
char flag = 0x2; // 1
char rtype[3] = "TXT"; // 2-4 three letter type
char pad;
char loc[3]; // 6-8 csect relative origin of the text
char pad[2];
short nbytes; // 11-12 number of bytes of info
char pad[2];
short esid; // 15-16 ESID of this csect
char text[56]; // 17-72 data
```

RLD 记录

再下来是RLD记录，如图20所示，每个记录包含了一个重载入口序列。

图3-20: RLD格式

```
char flag = 0x2; // 1
char rtype[3] = "TXT"; // 2-4 three letter type
char pad[6];
short nbytes; // 11-12 number of bytes of info
char pad[7];
{ // 17-72 four or eight-byte relocation entries
    short t_esid; // target , ESID of referenced csect o
r symbol
    // or zero for CXD (total size of PR defs)
    short p_esid; // pointer , ESID of csect with refere
nce
    char flags; // type and size of ref ,
    char addr[3]; // csect-relative ref address
}
```

每个入口都有目标和指针的ESID，一个标志字节，和指针的csect相对地址。标志字节的位标志了引用的类型(代码，数据，PR或CXD)，长度(1, 2, 3或4字节)，一个符号位说明是加上还是减去重载值，还有一个”相同”位。如果”相同”位置1，下一个入口忽略两个ESID，并用这个入口的ESID。

END 记录

结束记录，如图21所示，给出了程序的开始地址，要么是一个csect中的地址或着一个外部符号。

图3-21: END格式

```
char flag = 0x2; // 1
char rtype[3] = "END"; // 2-4 three letter type
char pad;
char loc[3]; // 6-8 csect relative start address or zero
char pad[6];
short esid; // 15-16 ESID of csect or symbol
```

小结

尽管80列记录已经非常过时了，IBM的目标格式仍然是令人惊讶地简单和灵活。极其小地连接器和载入器就能处理这种格式；在360的某个型号上，我曾经用过一个可以装载一个80打孔卡上的绝对载入器，可以载入一个程序，翻译TXT和END记录，忽略其他的记录。

基于磁盘的系统要么将目标文件存成一个卡映像，或存成一个同样记录类型的变种格式，但是一个没有序列号的更长的记录。DOS(IBM 360的一种轻量级操作系统)的连接器生成一种简化的格式，带有一个csect和一个剥离了ESID的RLD记录。

在目标文件中，独立的命名的csect可以让一个程序员或连接器把需要的模块放到程序中，如将所有的代码csect放在一起。这种格式显示它的时间的主要地方是最大8字符的符号，没有关于每个csect的类型信息。

微软可移植执行体格式

微软的Windows NT继承了很多系统的特点，包括MS-DOS的早期版本和Windows，Digital的VAX VMS(很多程序员都在上面工作过)，和Unix系统V(很多其他的程序员在上面工作过)。NT的格式是从COFF演化来的，是Unix在a.out之后和ELF格式之前用的格式。我们会讨论PE格式，以及微软版本的COFF和PE的区别。

Windows是在一个带有慢的处理器，有限内存和硬件换页的能力低下的系统中开发的，所以强调了共享库来节省内存，特殊的技巧来提高性能，有些在PE/COFF的设计中是明显的。多数Windows的可执行文件包含了资源，一个指向诸如光标、图标、位图、菜单和字体的目标，这些目标在程序和GUI环境中共享。一个PE文件可以包含一个文件中程序代码所要使用的所有资源的资源目录。

PE可执行文件最初是为一个分页环境设计的，所以一个PE文件的页面通常直接映射到内存页面中并运行，非常象一个ELF可执行文件。PE文件可以是EXE

程序或DLL共享库(也就是dynamic link libraries)。这两种文件的格式相同，用一个标志位来互相区分。它们都可以包含一个可以被其他PE文件载入到相同的地址空间的导入的功能和数据的列表，和一个在载入的时候需要从其他文件解析的导入的功能和数据。每个文件还含有一组类似于ELF段的块，被称为区、段和目标。在这里我们把它们称为区，微软现在使用的术语。

一个PE文件，如图22所示，由一个小的DOS .EXE文件开始，这个文件打印出诸如”This program needs Microsoft Windows”的提示。(微软对这种向后的兼容性的贡献令人印象深刻。)在EXE头部的最后有一个以前没有使用过的域，指向PE签名，接下来是一个文件头，包含了一个COFF区和”可选的”头部，尽管它在所有的PE文件中都出现，然后是一个区头部的列表。区头部描述了文件中不同的区。一个COFF目标文件由COFF头部开始，忽略可选头部。

图3-22：微软PE和COFF文件格式

DOS header (PE only)
DOS program stub (PE only)
PE signature (PE only)
COFF header
Optional header (PE only)
Section table
Mappable sections (pointed to from section table)
COFF line numbers, symbols, debug info (optional in PEfile)

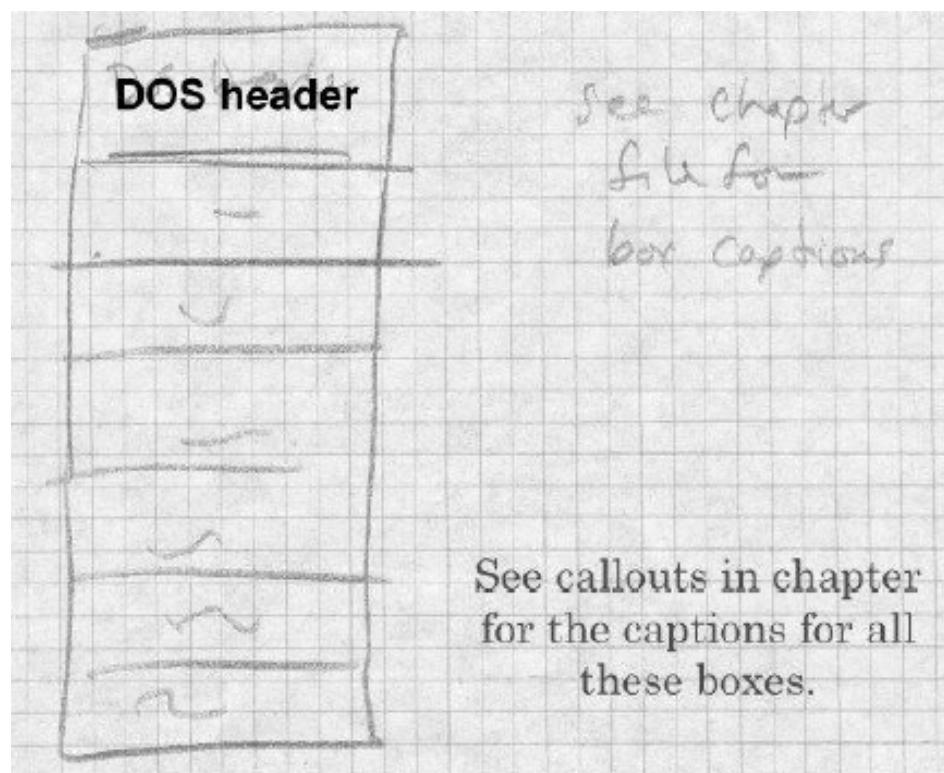


图23显示了PE、COFF和”可选的”头部。COFF头部描述了文件的内容，包括区表中入口的数目的重要的数值。”可选的”头部包含指向最常使用的文件的区。地址都是程序调入内存后的地方的偏移量，也称为相对虚拟地址，或RVA。

图3-23: PE和COFF头部

PE签名

```

char signature[4] = "PE\0\0"; // magic number, also shows byte order

COFF头部
unsigned short Machine; // required CPU, 0x14C for 80386, etc.
unsigned short NumberOfSections; // creation time or zero
unsigned long TimeDateStamp; // creation time or zero
unsigned long PointerToSymbolTable; // file offset of symbol table in COFF
unsigned long NumberOfSymbols; // # entries in COFF symbol table or zero
unsigned short SizeOfOptionalHeader; // size of the following optional header
unsigned short Characteristics; // 02 = executable, 0x200 = nonrelocatable,
// 0x2000 = DLL rather than EXE
PE头部后面的可选头部, 不含在COFF目标中
// COFF fields
unsigned short Magic; // octal 413, from a.out ZMAGIC
unsigned char MajorLinkerVersion;
unsigned char MinorLinkerVersion;
unsigned long SizeOfCode; // .text size
unsigned long SizeOfInitializedData; // .data size
unsigned long SizeOfUninitializedData; // .bss size
unsigned long AddressOfEntryPoint; // RVA of entry point
unsigned long BaseOfCode; // RVA of .text
unsigned long BaseOfData; // RVA of .data
// additional fields.
unsigned long ImageBase; // virtual address to map beginning of file
unsigned long SectionAlignment; // section alignment, typically 4096, or 64K
unsigned long FileAlignment; // file page alignment, typically 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;// total size of mappable image, rounded to SectionAlignment
unsigned long SizeOfHeaders;// total size of headers up through section table
unsigned long CheckSum;// often zero
unsigned short Subsystem;// required subsystem: 1 = native, 2 = Windows GUI,
// 3 = Windows non-GUI, 5 = OS/2, 7 = POSIX
unsigned short DllCharacteristics;// when to call initialization routine (obsolescent)
// 1 = process start, 2 = process end, 4 = thread start, 8 = thread end
unsigned long SizeOfStackReserve;// size to reserve for stack
unsigned long SizeOfStackCommit;// size to allocate initially for stack
unsigned long SizeOfHeapReserve;// size to reserve for heap
unsigned long SizeOfHeapCommit;// size to allocate initially for heap
unsigned long LoaderFlags;// obsolete
unsigned long NumberOfRvaAndSizes;// number of entries in following image // following pair is repeated once for each directory
{
    unsigned long VirtualAddress;// relative virtual address of directory
    unsigned long Size;
}

```

目录的顺序是，
 导出目录
 导入目录
 资源目录
 例外目录
 安全目录
 基重载目录
 调试目录
 映像描述字符串
 机器相关数据
 线称载入存储目录
 载入配置目录

每个PE文件都建立成可以让系统直接将其映射到内存中。每个区都对齐到磁盘块的边缘，或者更大(文件对齐值)，逻辑对齐到一个内存页面的边缘(在x86上是4096)。连接器为文件要映射的特定的目标地址(影响基础)生成一个PE文件。如果在那个地址有可用的一块地址空间，通常总是如此，无需载入时的修改。

在很少的时候，如旧式的win32的兼容的系统目标地址不可用，载入器需要将文件映射到其他的地方，在这种情况下，文件需要在. reloc区中包含重载信息来告诉载入器要改变那些地址。共享DLL库也需要重载，因为DLL映射的地址取决于已经有什么其他程序占用了地址空间。

PE头部后面是区表，如图24所示的入口数组。

图3-24：区表

```
// array of entries
unsigned char Name[8];// section name in ASCII
unsigned long VirtualSize;// size mapped into memory
unsigned long VirtualAddress;// memory address relative to image base
unsigned long SizeOfRawData;// physical size , multiple of file alignment
unsigned long PointerToRawData;// file offset
// next four entries present in COFF , present or 0 in PE
unsigned long PointerToRelocations;// offset of relocation entries
unsigned long PointerToLinenumbers;// offset of line number entries
unsigned short NumberOfRelocations;// number of relocation entries
unsigned short NumberOfLinenumbers;// number of line number entries
unsigned long Characteristics;// 0x20 = text , 0x40 = data , 0x80 = bss , 0x200 // 0x800 = don't link , 0x10000000 = shared ,
// 0x20000000 = execute , 0x40000000 = read , 0x80000000 = write
```

每个区都有一个文件地址和尺寸(PointerToRawData和SizeOfRawData)和内存地址和尺寸(VirtualAddress和VirtualSize)，它们并不一定相同。CPU的页面尺寸通常要比磁盘块大，通常是4K的内存页面和512字节的磁盘块，在一个页面中间结束的区不一定位剩下的页面空间分配磁盘块，节省了一些磁盘空间。每个区都为页面标志了适当的硬件许可，如为代码页面标志读+执行许可，数据页面标志读+写许可。

PE 特殊区

PE文件象Unix可执行文件(实际上仅是那些名字)一样有. text, . data有时还有. bss区，以及很多Windows特定的区。

- **导出区：**在这个模块中定义并在其他模块中可见的符号。EXE文件通常不导出符号，或仅为调试器导出一两个。DLL文件为例程导出符号和它

们所提供的数据。导出区包含一个导出的符号的RAV的数组，还有两个符号的名字的并行数组(就是一个ASCII串的RVA)，还有按照串的名字排序的符号的导出顺序表。要用名字查找一个符号，对串名字表执行对分查找，在顺序表中找到对应已经找到的名字的位置，用顺序表为RVA数组进行索引。(经过论证，这种方法要比重复一个三字入口的表要快。)导出区也可以是”转发”区，即RVA指向一个串，这个串就是在另一个库中的符号的实际名字。

- **导入区：**导入表列出了在载入的时候需要从DLL中导入的所有符号。连接器预先确定那些符号可以在那些DLL中找到，所以导入表由一个导入目录开始，每个要引用的DLL为一个入口。每个目录入口包括DLL文件的名字，还有一个并行数组，一个确定需要的名字，另一个映像文件中保存符号值的位置。第一个值中的入口可以是一个顺序(如果高位置1)，或一个指向一个名字串的指针，这个指针是符号在顺序表中的位置的一个猜测，以加快查找速度。第二个数组包含了存贮符号值的位置；如果符号是一个过程，连接器必须调整所有对这个符号的调用，改为对这个位置的间接调用，导入模块中对符号的引用改为用这个位置作为指向实际数据的指针。(一些编译器自动提供简介调用转换，其他的需要显式的程序代码。)
- **资源区：**资源表被组织为一个树的数据结构。该结构支持任意深度的树，但实际中树为三层结构，资源类型，名字和语言。(这里的语言指自然语言，这样就可以定制可执行文件在英语之外的环境中运行。)每个资源要么有一个名字或数字。典型的资源如DIALOG类型(对话窗)，名字是ABOUT(About This Program窗口)，英语。资源使用Unicode名字来支持非英语语言，不象符号使用ASCII字符。实际的资源是一块二进制数据，格式根据资源类型而不同。
- **线程本地存储区：**Windows支持进程的多个线程的执行。每个线程可以有自己的私有存储，或TLS。这个区有指向一块在线程开始运行的时候用来初始化一个TLS的映像的指针，还包含指向在每个线程开始的时候需要调用的初始化例程的指针。这个区通常出现在EXE文件而不是DLL文件中，因为当一个程序动态连接一个DLL库时，Windows不为其分配TLS存储块。(见第10章。)
- **修正区：**如果可执行文件被移动了，整个文件作为一个整体移动，所以所有的修正都有相同的值，即实际的装载地址和目标地址的不同。如果存在着修正表，则它包含了一个修正块的数组，每条为要映射的可执行代码的4K页面。(没有修正表的可执行文件只能装载的连接的目的地址。)每个修正块包含页面的基RVA，修正的数目，和一个有16位修正入口的数组。每个入口的低12位是块中需要重载的偏移量，高4位是修正类型，如32位值，调整高16位或低16位(用于MIPS体系结构)。这种一块一块的机制在重载表中节省了大量的空间，因为每个入口可以压缩到2个字节，而不象ELF中需要8到12个字节。

运行 PE 可执行文件

运行一个PE可执行过程是非常简单的。

- 读入文件中带有DOS头部、PE头部和区头部的第一个页面。
- 确定地址空间中的目标地址是否可用，如果不可用再分配另外一个区域。
- 使用区头部中的信息将文件中所有的区映射到分配的地址空间中的适当的位置。
- 如果文件没有装载到它的目标地址，进行修改。
- 遍历导入区中的DLL列表，把那些还没有装载的库装载进来。（这个过程可以是递归的。）
- 解析导入区中所有的导入符号。
- 利用PE头部中的值建立初试栈和堆。
- 建立初试线程并启动进程。

PE 和 COFF

Windows的COFF可重载目标文件的文件头和区表和PE文件一样，但机构更象一个可重载ELF文件。COFF文件没有DOS头，也没有在PE头后面的可选头。每个代码或数据区还带有重载信息和行号。（在EXE文件中如果有行号信息，行号信息保存在一个调试区，系统装载器不与处理。）COFF目标文件有相对与区的重载信息，就象ELF文件一样，而不是相对RVA的重载信息，总带有一个需要的符号的符号表。由各种语言编译器生成的COFF文件不包含任何资源，资源都在由特定的资源编译器生成的分离的资源文件中。

COFF文件还有一些在PE文件中没有使用的区。最常用的是. directive区，包含了连接器用的文本命令串。编译器通常使用. directive区告诉连接器来查找特定语言使用的库。包括MSVC的一些编译器在生成DLL文件的时候还有对连接器的导出代码和数据的指示。（这种把命令和目标代码混合的做法又回到了以前，早在60年代，IBM连接器接受混合的命令和目标文件的打孔卡。）

PE 文件小结

对于一个带有虚拟内存的线性寻址的系统而言，PE文件格式可以胜任的，仅带有很多一些从DOS继承下来的内容。它还有一些其他的特性，如顺序导入导出，最初是为了加快在小系统中的程序装载速度，但在现代32位系统中这个特性的有效性有很大折扣。16位分段可执行系统使用的早期NE格式远比PE格式复杂，PE格式绝对是一个改进。

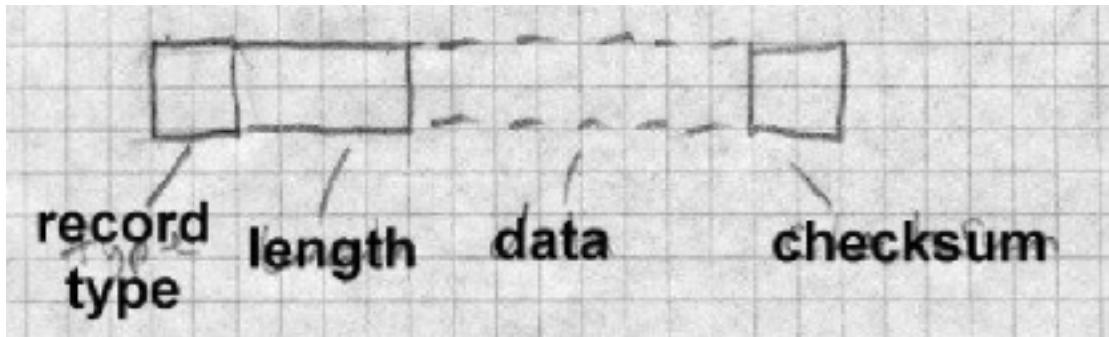
Intel/Microsoft 的 OMF 文件格式

我们在这章将要看到的倒数第二种格式是仍在使用的最老的格式，即Intel Object Module Format。Intel在70年代为8086定义了OMF格式。在接下来的几十年中，很多厂商，包括Microsoft，IBM和Phar Lap（曾经为DOS写了很多被广泛使用的32位扩展工具），都定义了各自的扩展。目前的OMF格式是原来的规范和多数扩展的集合，去除了一些已经融合到其他扩展或从未使用过的扩展。

到目前为止，我们看到的所有的文件格式都是为那些具有随机存取磁盘和足够的RAM用简单的方式来完成编译器和连接器处理的系统而设计的。OMF是为早期的带有极少内存和使用打孔纸带作为存储设备的系统的消失而设计的。所以OMF文件将目标文件分为一系列小的记录，如图25所示。每条记录包括一个类型字节，一个两字节的长度，内容和将整个记录内容的所有位归零的校验字节。(打孔纸带设备没有内建的错误检测机制，而且由于灰尘和粘连的部分造成的错误并不少见。)OMF被设计成在一个没有大容量存储设备的系统上的连接器可以用最少的遍数完成对文件的处理。通常使用1%遍处理的技巧，用一半的处理来解析放在每个文件前面的符号名字，然后用一遍处理完成连接并生成输出文件。

图3-25：OMF记录格式

```
picture of
-- type byte
-- two-byte length
-- variable length data
-- checksum byte
```



OMF格式由于8086的分段体系结构而变得非常复杂。OMF连接器的一个主要功能就是将代码和数据封装到最少的段和段组中。OMF中的每块代码或数据都分配到一个段中，每个段然后可以分配到一个段组或段类中。(一个段组必须足够小以便可以使用一个段值来寻址，一个段类可以为任何尺寸，所以段组通常用于寻址和存储管理中，而段类仅用于存储管理中。)代码可以用名字来引用段和段组，也可以用相对于段或段组的基的段相对地址来引用代码。

OMF格式还可以提供一些对覆盖连接的支持，尽管我所知道的OMF连接器还没有支持这个特性的，而是用一个分离的指示文件中的覆盖指令来完成。

OMF 记录

目前的OMF定义了至少40种记录类型，在这里都列出来就太多了，所以我们来看一个简单的OMF文件。(完整的规范可参见Intel的TIS文档。)

OMF文件用几种编码技术来使记录尽可能的短。所有的名字串都是变长的，在字符的前面保存着串的长度字节。一个空名字(在有些内容中有效)是一个零字节。一个OMF模块将每个名字在LNAMES记录中保存一遍，然后再用这个表来索引查找定义段、段组和符号的名字的名字；而不是用名字来引用段、符号或组。第一个名字是1，第二个名字是2，这样一直将整个名字集列出，不论需要多少个L

NAMES记录。(在很多的情况下这么做可以节省很少的空间, 即一个段和一个外部符号可以有相同的名字, 因为它们的定义可以指向相同的东西。)从0到0x7f的索引保存在一个字节中, 如果一个字节的高位为1, 则说明这是一个双字节序列。奇怪的是, 这种情况下第一个字节的低7位是这个值的高7位, 而第二个字节是值的低8位, 正好和Intel的顺序相反。段、段组和外部符号也用索引来引用, 每种都有各自的索引序列。如, 假定一个模块列出了DGROUP、CODE和DATA的名字, 定义名字的索引1、2和3。然后模块定义了叫做CODE和DATA的段, 指向名字2和3。因为CODE是第一个定义的段, 它的段索引是1, DATA的段索引是2。

最初的OMF格式是为16位的Intel的体系机构设计的。对于32位程序, 对于地址尺寸会对记录类型的定义有影响记录使用新的OMF记录格式。所有的16位记录格式碰巧都使用偶数值代码, 所以相对应的32位记录类型使用比16位类型大的奇数类型。

OMF 文件的细节

图26列出了一个简单的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、段组和类的名字列表

SEGDEF 段(一个段一个记录)

GRPDEF 段组(一个组一个记录)

PUBDEF 全局符号

EXTDEF 未定义的外部符号(一个符号一个记录)

COMDEF 公共块

COMENT 第一遍扫描信息

LEDATA 代码或数据块(多个)

LIDATA 分离的数据块(多个)

FIXUPP 重载和外部引用修正, 在每个引用的LEDATA或LIDATA后面

MODEND 模块结尾

文件由THEADR记录开始, 标志着模块的开始并用模块的源文件名为模块命名。(如果这个模块是一个库的一部分, 它也会由一个类似的LHEADR记录开始。)

第二个记录是一个有着糟糕的名字的COMENT记录, 包含给连接器使用的配置信息。每条COMENT记录有一些标志位说明在连接的时候是否保留这条注释, 还有一个类型字节和注释文本。一些注释类型确实是注释, 如, 编译器版本号或版权说明, 但有很多为连接器传送信息, 如要使用的内存模型(从微到大模式), 在处理完这个文件后要搜索的库的名字, 弱外部符号的定义, 还有厂商们在OMF格式中挤入的其他数据类型的收集。

接下来是一系列的LNAMES记录, 列出了在这个模块中的段、段组、类和覆盖中所使用的所有名字。就象上面所说, 所有LNAMES记录中的所有名字都逻辑地为一个从1开始的数组。

在LNAMES后面是SEGDEF记录, 模块中每个段有一个。SEGDEF记录包含一个

段的名字的索引，如果有类和覆盖的话，还有类和覆盖的名字的索引。还有段的属性，包括对齐要求，和与其他模块中有相同名字的段的合并规则，还有段的长度。

接下来是GRPDEF记录，如果文件中有的话，定义了模块中的组。每个GRPDEF记录都有段组的名字的索引和组中的段的名字的索引。

PUBDEF记录定义了对其他模块可见的”公共”符号。每个PUBDEF记录定义了同一个段组或段中的一个或多个符号。记录中还有段或段组的索引，在段或段组中每个符号的偏移量，名字和单字节编译器相关的类型域。

EXTDEF记录定义了未定义的外部符号。每条记录包含了一个符号的名字和一个或两个字节的调试器符号类型。COMDEF记录定义了公共块，类似于EXTDEF记录，除了它们还定义了符号的最小尺寸。模块中所有的EXTDEF和COMDEF符号逻辑上都是一个数组，所以可以根据它们的索引进行修正。

解下来是一个可选的特殊COMENT记录，标记出第一遍扫描的结尾。它告诉连接器可以在连接过程的第一遍扫描中忽略文件的剩下的部分。

文件中剩下的部分是程序的实际的代码和数据，和包含重载和外部引用的修正记录混在一起。有两种数据记录，LEDATA(枚举)和LIDATA(重复)。LEDATA记录仅有段的索引和起始的偏移量，后面就是数据。LIDATA记录也是由段索引和起始偏移量开始，但后面可以跟一个重复数据的嵌套集。LIDATA记录可以高效地处理象下面的Fortran语句生成的代码：

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

~~例如一个单一的LIDATA可以有一个两位字节的块，包含12并将其重复100次~~

每个需要修正的LEDATA或LEDATA后面必须紧跟着FIXUPP记录。FIXUPP记录是目前最复杂的记录类型。每个修正需要三个内容：首先是目标，即引用的地址，然后是帧，在一个段或段组中的相对于要计算的地址的位置，最后是要修正的位置。因为在很多修正中都指向同一个帧是非常常见的，并且在很多修正中引用同一个目标也是比较常见的，OMF定义的修正线索，一个两位的标志用作帧或目标的缩略，所以在任何时刻，设定了线索值后，可以有四个帧或目标的修正。每个线索值可以按照需要随时进行重定义。如，如果一个模块有一个数据的段组，这个组通常作为模块中几乎每个数据引用中的帧，所以为那个组的基地址定义一个线索数可以节省大量的空间。在实际中，一个GRPDEF记录总是跟着一个定义那个组的帧线索数的FIXUPP记录。

每条FIXUPP记录都有一系列的子记录，每个子记录要么定义一个线索或一个修正。一个线索定义子记录有标志位说明它定义的是一个帧还是目标线索。一个目标线索定义包含了线索数目，索引的类型(相对段，相对段组，外部相对)，段、段组或符号的基的索引，还有一个可选的基偏移量。一个帧线索定义包含线索数值，索引的种类(所有目标定义的种类加上两个公共特殊情况，对位置的相同的段和对目标的相同段。)

一旦定义了线索，修正的子记录就相对简单了。它包含修正的位置，一个定义修正类型的代码(16位偏移量，16位段值，全段:偏移量，8位相对地址，等等)，还有帧和目标的位置。帧和目标可以引用已经定义的线索或在原地定义。

在LEDATA，LIDATA和FIXUPP记录的后面是标记模块结尾的MODEND记录，如果模块是一个程序的主例程还可以可选的定义一个入口点。

一个实际的OMF文件还有用来保存本地符号、行号和其他调试器信息的更多的记录类型，在Windows环境下面，还有在目标NE文件(PE的16位分段方式的前身)

中建立的导入导出区，但模块的结构没有变化。文件中记录的顺序非常灵活，尤其是如果没有第一遍扫描截止标记。最硬性和快速的规定是THEADER和MODEND必须是第一个和最后的记录，FIXUPP必须紧跟在它所引用的LEDATA和LIDATA记录的后面，不允许模块内的前向引用。在实际中，在定义符号、段和段组的时候可以生成记录，只要它们在他们所引用的其他记录之前。

OMF 格式小结

OMF格式与我们看到的其他格式相比非常复杂。复杂性一部分是由于数据的压缩技巧造成的，部分是由于近年来增加的特性，部分是由于分段程序寻址的继承的复杂性。不同类型的记录的统一的记录格式是它的一个重要优点，因为可以用非常简单的方式进行扩展，并允许程序在处理OMF文件的时候忽略不理解的记录。

尽管如此，即使仅有数兆内存和大容量的硬盘的小的桌面计算机，OMF将目标切分为小的记录带来的麻烦也比它的价值大。目标模块的小的记录类型在70年代之前非常普遍，但现在已经过时了。

不同目标格式的比较

在这章中我们已经研究了不同的目标和可执行格式，从简陋的(.COM)格式到复杂的(ELF和PE)格式，到过度复杂的(OMF)格式。现代目标文件格式，如ELF试图将所有的数据组合到单一的类型中，便于连接器处理。它们还考虑了在虚拟内存中的文件的分布，这样系统在将文件映射到程序的地址空间中只需很少的工作。

每种目标格式都带有定义它的操作系统的风格。Unix系统传统上将其接口设计为简单和精密设计的，a.out和ELF格式显示了它们相对的简单性并缺少特殊情况的处理。Windows是另一个方向，将进程管理和用户界面考虑到文件格式中。

项目

这里我们为本书的项目作业定义一个简单的目标格式。不象其他所有的目标文件格式，这个格式都是由ASCII文本组成的。这样可以通过文本编辑器建立目标文件，对项目连接器的输出文件的检查变得简单。图27是格式得概略图。段、符号和重载入口都用文本行表示，不同的域用空格分割。每行在结尾都可以有额外的域，程序需要忽略。数字都用十六进制表示。

图3-27：项目目标文件格式

```
LINK
nsegs nsyms nrels
-- segments --
-- symbols --
-- rels --
```

-- data --

第一行是”神奇数”，单词LINK。

第二行包含至少三个十进制数，文件中段的数量，符号表的入口数，重载入口的数量。在这三个数后还可以有其他的信息，用于连接器的扩展版本。如果没有符号或重载，那么相对的数就是零。

接下来是段的定义。每个段定义都有段的名字，段逻辑开始的地址，段的字节数，还有一个描述段的代码字符串。带有R的代码字符代表可读的，W代表可写的，P代表在目标文件中出现的。（还可能有其他的字符。）一组典型的类似a.out的文件的段可以是：

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

段按照它们的定义的出现顺序编号，第一个段为1。

下面是符号表，每个入口的格式是：

```
name value seg type
```

name是符号的名字。Value是符号的十六进制值。Seg是相对段定义的地方的段值，0代表绝对或未定义符号。Type是一个字符串，D代表已定义的，U代表未定义的。符号也按照出现的顺序编号，从1开始。

下面是重载信息，一行一条重载信息：

```
loc seg ref type ...
```

loc是需要重载的位置，seg是这个位置出现的段，ref是那里需要重载的段或符号数，type是一个体系结构相关的重载类型。通用类型A4代表一个四字节的绝对地址，或R4是四字节相对地址。有些重载类型在type后面有额外的域。

在重载信息的后面是目标数据。每个段的数据是一个很长的十六进制串，跟着一个换行符。（这样就很容易在perl中读写区的数据。）每对十六进制代表一个字节。段的数据串和段表的顺序一样，每个”出现”的段必须有段数据。数据串的长度是由段定义的长度确定的，如果段是100字节长，段数据的行是200个字符，不包括结尾的换行符。

项目3-1：写一段perl程序读入一个这种格式的目标文件，将内容保存到一个perl表和数组的适合格式中，然后将文件写回。输出文件不需要和输入文件一样，尽管语法上相同。如，符号无需和读入的顺序相同，尽管它们的顺序不同，重载入口必须调整过来反映符号表的新的顺序。

练习

1. 象这个项目中的这种文本目标文件格式是否实用？（提示：见Fraser和Hanson的论文”一个独立于机器的连接器”。）

第 4 章 存储空间分配

\$Revision: 2.3 \$
\$Date: 1999/06/15 03:30:36 \$

连接器和载入器的首要任务是存储分配。一旦存储资源被分配好了，连接器可以继续进行后续的符号绑定和代码修正处理。多数在可连接目标文件中定义的符号都定义为文件中的存储空间中的相对位置，所以只要空间是未知的，符号就无法解析。

就像连接的其他方面一样，存储分配的基本问题比较简单，但不同的计算机体系结构的特性和编程语言的语义不同(还有两者直接的交互作用)可以使问题变得复杂。存储分配的多数工作可以用一种精致的和相对与体系结构无关的方法，但总有一些细节需要特殊的和机器相关的技巧。

段和地址

每个目标或可执行文件都用一种目标地址空间的模型。通常这个目标就使目标计算机系统的程序地址空间，但有的时候还可能使其他的模型，如共享库。重载连接器或载入器的基本问题是保证程序中的所有的段都有定义和地址，但这些地址在不该覆盖的时候不覆盖。

连接器的每个输入文件都有一系列不同类型的段。不同种类的段是用不同的方法处理的。某一种类型的所有的段，如可执行代码，几乎都连接到输出文件的同一个段中。有时候，段都合并到其他类型之上，如Fortran的公共块；越来越多情况下，对于共享库和C++的特殊特性，连接器本身需要建立一些段并布置到文件中。

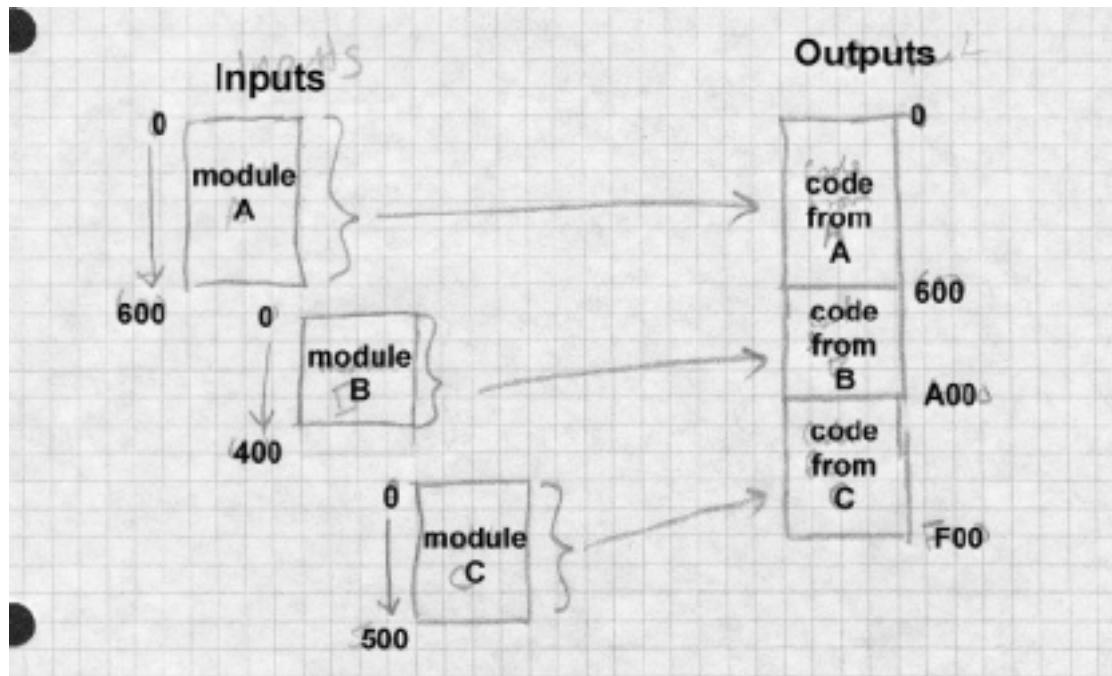
存储布置是一个两遍过程，因为每个段的位置在所有的段的尺寸都确定之前无法分配。

简单存储布置

一种简单但不实际的情况下，连接器的输入包含一组模块，从M1到Mn，每个模块包含一个段，从位置0开始，从L1到Ln，目标地址空间也从零开始，如图1所示。

图4-1：单一段存储分配

bunch of segments all starting at zero are relocated one after another



连接器和载入器依次检查每个模块，顺序分配存储空间。Mi的起始地址是L₁到L_{i-1}的和，连接后的程序的长度是L₁到L_n的长度和。

多数体系结构需要将数据对齐到字边界，或者至少在对齐的情况下运行的更快，所以连接器一般将每个L_i长度对齐到体系结构要求的最严格的对齐要求的数倍，典型长度是4或8字节。

例1：假定一个主程序调用的main要与三个子例程连接，分别是calif，mass和newyork。(将风险资金按照地理分配。)每个例程的长度是(十六进制)：

name	size
------	------

main	1017
calif	920
mass	615
newyork	1390

假定存储的分配是从十六进制1000开始，对齐到4字节。那么存储的分配是：

name	location
------	----------

main	1000 - 2016
calif	2018 - 2937
mass	2938 - 2f4c
newyork	2f50 - 42df

由于对齐的原因，在2017的一个字节和2f4d处的三个字节浪费掉了，但不值得考虑。

多种段类型

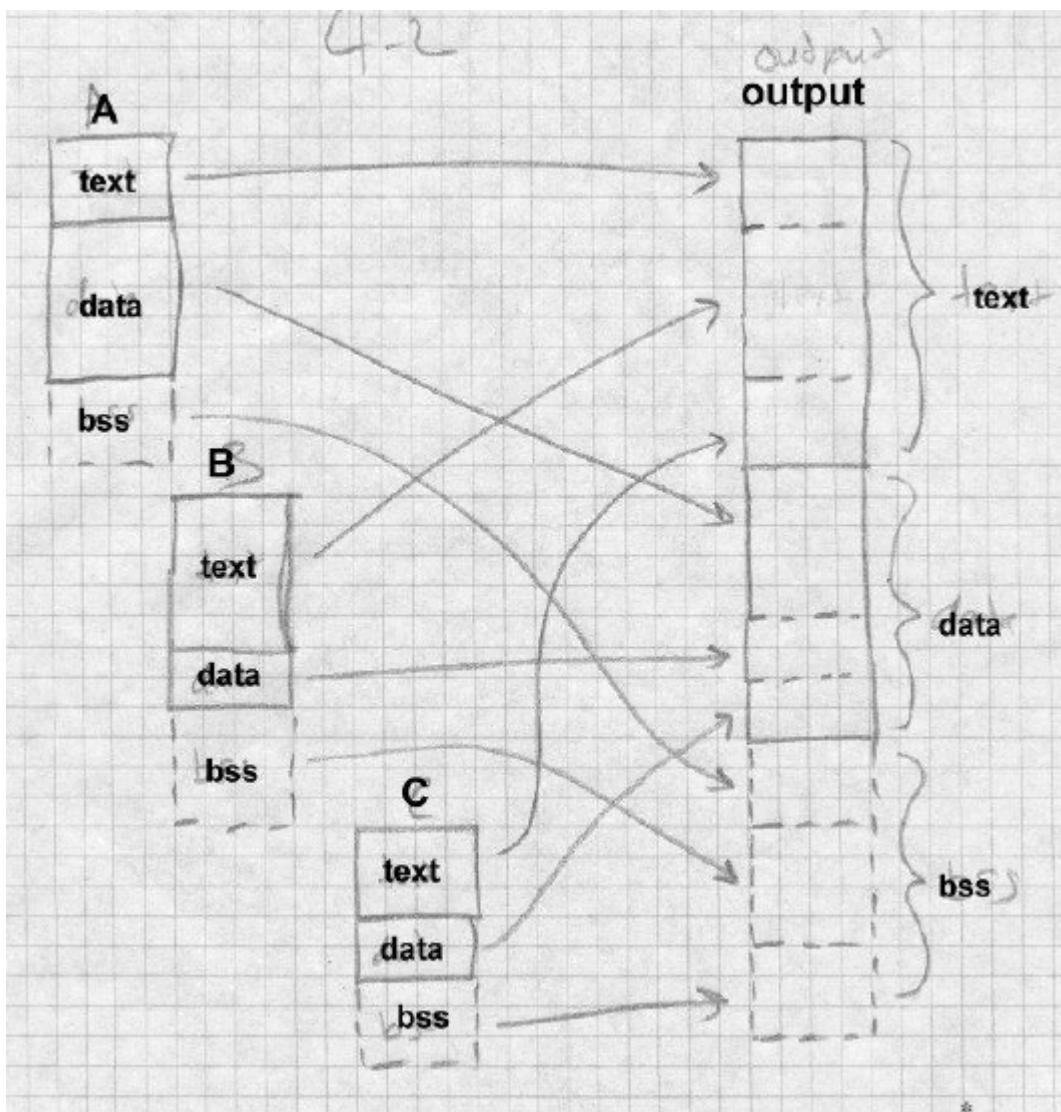
除了最简单的目标格式，还有几种段的类型，连接器需要将输入模块中相

关的段组合到一起。在带有文本和数据段的Unix系统中，连接器需要将所有的文本段集合到一起，然后是所有的数据段，逻辑上跟着BSS段。（尽管BSS段在输出文件中不占空间，但需要分配空间来解析BSS符号，而且在载入输出文件的时候需要对BSS的长度的说明。）这就需要一种两层存储分配的机制。

在这里每个模块Mi有长度为Ti的文本段，长度为Di的数据段，长度为Bi的BSS段，如图2所示。

图4-2：多种段的存储分配

text, data, and BSS segments being combined separately



在读入每个输入模块的时候，连接器为每个 T_i , D_i 和 B_i 分配空间，尽管每个段原来都分别从零开始分配的空间。当读入所有的输入文件后，连接器就知道了这三个段的总长度， T_{tot} , D_{tot} 和 B_{tot} 。因为数据段是在文本段之后，连接器将 T_{tot} 加到为每个数据段分配的地址上，因为BSS段在文本和数据段的后面，连接器把 T_{tot} 和 D_{tot} 的和加到BSS段分配的地址上。

同样，连接器通常需要将每个分配的空间对齐。

段和页面对齐

如果文本段和数据段要装载到分离的地址空间中，实际中通常都是这种情况，文本段需要对齐到一个完整的页面，数据和BSS段的位置也需要做相应的调整。很多Unix使用一种技巧来节省文件空间，就是在目标文件中将数据段紧跟着文本段，将文件中的这个页面在虚拟内存中映射两次，一次映射为只读用于文本，一次映射为写入时拷贝(copy-on-write)用于数据。在这种情况下，数据段的地址逻辑上是从文本段的结尾上的页面开始的，所以无需进行对齐，数据段从4K位置开始，或者紧接文本段的结尾而无论页面的大小。

例2：我们把例1进行扩展，每个例程都有文本、数据和bss段。字对齐要求还是4字节，但页面的尺寸是0x1000字节。

Name	text	data	bss
main	1017	320	50
calif	920	217	100
mass	615	300	840
newyork	1390	1213	1400

所有的数都是十六进制

连接器首先布置好文本段，然后是数据段，最后是bss段。需要注意的是数据区从0x5000的页边界开始，但bss紧接着数据区，因为在运行的时候数据和bss逻辑上是一个段。

name	text	data	bss
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

在页面的结尾处的0x42e0到0x5000之间有一些浪费的空间。Bss段在页面中间0x86eb处结束，但典型的程序在这之后还要分配堆的空间。

公共块和其他特殊的段

上面这种简单的段分配机制在连接器处理的存储中的80%的时候都工作得很好。剩下得需要特殊的技巧。这里我们看一些更加普通的情况。

公共

公共存储是50年代的Fortran I的一个特性。在原始的Fortran系统中，每个子程序(主程序、函数或子例程)各自都有其静态声明和分配的标量和数组变量。还有一个带有标量和数组的公共区域供所有的子程序使用。公共存储被证明非常有用，在Fortran的后续版本中从一个单一的公共块(现在为空公共块，因为它的名字为空)演变成多个命名的公共块，每个子程序都声明它所使用的公共块。

在Fortran出现后的前40年间，Fortran不支持动态存储分配，公共块是程序员绕过这个限制的主要工具。标准的Fortran允许在不同的子程序中声明不同大小的公共块，最大的块具有最高的优先级。所有的Fortran系统都将这个特性扩展为允许所有的块都可以有不同的大小，仍然是最大的块有最大的优先级。

大型的Fortran程序经常突破它们所运行的系统的内存限制，因为没有动态内存分配，程序员经常要重新编译这个软件包，将尺寸压缩来适应这个包所要解决的问题。软件包中除了一个之外的所有子程序都把每个公共块定义为一维数组。

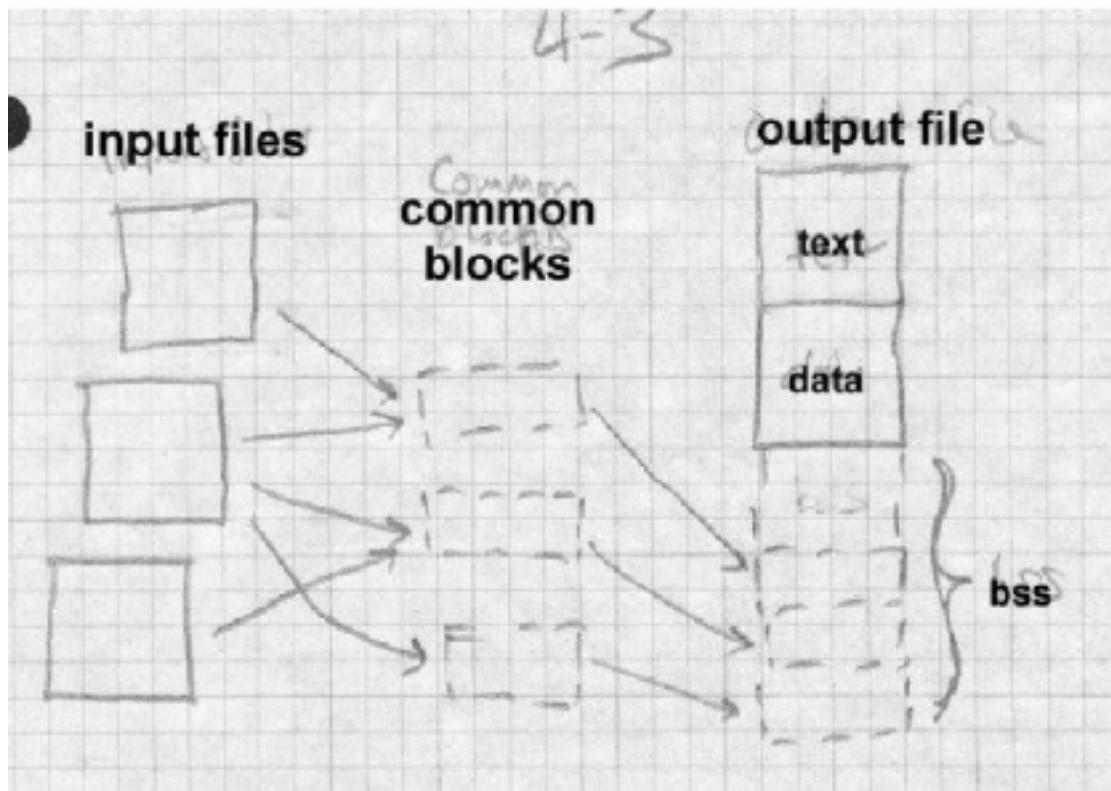
子程序中的一个定义所有公共块的精确尺寸，并在开始的时候将尺寸放入一个变量(也是另一个公共块)，软件包中其他程序可以使用这个变量。这样就可以通过改变并重新编译一个定义公共块的例程来改变块的大小，然后再连接其他的程序。

60年代后Fortran增加了BLOCK DATA，为任何公共块(除了空公共块，一个很少要求的限制)的所有或部分指定静态的初试数据。通常用BLOCK DATA初始化的公共块的尺寸是连接的时候该块的实际尺寸。

在处理公共块的时候，连接器将输入文件中的一个公共块的声明作为一个段来处理，但将所有的块覆盖在同一个名字下，而不是将这些段连接起来。连接器用所有声明中最大的尺寸作为段的尺寸，除非输入文件中的某个文件有一个段的初始化版本。在某些系统中，初始化的公共存储是一种单独的段的类型，而其他的存储仅是数据段的一部分。

Unix的连接器一直以来都支持公共块，因为最早期的Unix版本都带有一个Fortran编译器的子集，而且C的Unix版本传统上对非初始化的全局变量的处理非常象对公共块的处理。但是Unix的目标文件在ELF格式之前只有文本、数据和bss段，没有直接的方法声明一个公共块。作为一种特殊的技巧，连接器把一个标志为非定义但有一个非零值的符号作为一个公共块来处理。连接器将这种符号中的最大的值作为公共块的尺寸。对于每个块，连接器在输出文件中的bss段定义符号，在每个符号后面分配需要的空间，如图3所示。

图4-3: Unix公共块
common at the end of bss



C++重复代码消除

在一些编译系统中，c++的编译器会由于虚拟功能表，模板和外部嵌入功能而生成大量的重复代码。这些设计期望程序的这些部分在系统中同步运行。一个虚拟函数表(通常缩写为vtbl)包含一个C++类中所有虚拟函数(和可以被子类覆盖的例程)的地址。需要虚拟函数的类都需要一个vtbl。模板本质上是带有数据类型的参数的宏，并且对于不同类型的参数展开为不同的例程。尽管要保证一个对被调用的普通例程的引用是程序员的工作，如hash(int)和hash(char*)，每种hash都有一种精确的定义，但会自动为每个数据类型的hash函数建立一个模板hash(T)。

在一个源程序是被分别编译的环境下，一种简单的技巧是将该文件使用的所有vtbl、扩展开的模板例程和外部的嵌入语句放到每个目标文件中，所以产生了很多的重复代码。

连接的时候的最简单的办法就是保留这些重复的代码。虽然程序可以正确地工作，但代码会膨胀到其应该的尺寸的三倍甚至更多。

在那些带有简单连接器的系统中，一些C++系统使用叠代连接方法，对在哪里扩展哪里或增加的语义(pragma，用于提示编译器的源代码)使用分离的数据，为编译器输入足够的信息仅仅生成需要的代码。我们会在11章讨论这个。

很多现代的C++系统正面解决这个问题，要么让连接器更加智能化，或者将连接器和程序开发系统的其他部分结合起来。(我们在11章也会讨论后一种方法。)用智能连接器的方法就是让编译器在每个目标文件中生成可能的重复代码，连接器再确认并去除重复的部分。

微软Windows的连接器为每个代码区定义一个COMDAT标志，告诉连接器去除一个之外的所有有相同名字的区。编译器用模板的名字为区命名，并适当地改变名字来将参数类型包含进去，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连接器通过定义一个类似于公共块的”一次连接”区来解决模板问题。如果连接器看到有相同的以格式为.*gnu.linkonce.name*的名字的段，则丢弃该段，除了第一次看到该名字的时候。编译器将一个模板扩展到一个.gnu.linkonce区中，并冠以调整过的名字。

这种机制可以很好地工作，但它并不是灵丹妙药。一种情况是，它并不能保护功能不相同的vtbl和扩展模板。有些连接器对丢弃的段和保留的段要进行每个字节的比较。这种方法非常保守，但如果两个文件是用不同的优化参数或编译器的不同版本编译的话会产生假错误。另一种情况是，它基本上不能丢弃它所能丢弃的重复代码。在多数C++系统中，所有的指针都有相同的内部表示。这就意味着，如，一个用指向int类型的指针表示的模板和用指向float类型的指针表示的相同的模板通常会生成相同的代码，尽管它们在C++中的类型是不同的。有些连接器会将带有相同代码的link-once区丢弃到另一个区中，甚至它们的名字并不完全匹配，但是这个问题还是没有很好地解决。

尽管我们到现在讨论了关于模板的问题，但对于嵌入函数还有缺省建构(constructors)，拷贝和赋值例程也有完全相同的问题，也可以用同样的方法处理。

初始器和终结器

另一个问题并不是C++所独有的，但C++的初始器和终结器使其更加严重。通常而言，如果在程序开始的时候能够运行一个初始器例程；在程序要结束的时候运行一个终结器例程的话，写一个库比较容易一些。C++允许使用静态变量。如果一个变量的类有一个建构器，这个建构器需要在启动的时候被调用来初始化变量，如果这个类有一个解构器(destructor)，在推出的时候需要调用这个解构器。有很多方法可以完成这个功能而无需连接器的支持，我们会在11章讨论，但现代连接器通常支持这个特性。

通常的方法是在每个目标文件中的一个匿名例程中放入启动代码，然后在一个称为.`init`的段或类似的段中放入一个指向这个例程的指针。连接器将所有的`.init`段连结在一起，这样就建立了一个指向所有启动例程的指针的列表。程序在启动的时候只需遍历这个表然后依次调用所有的例程即可。推出时的代码也可以用基本相同办法处理，相应的段叫`.fini`。

但是这种方法也不是完全另人满意的，因为某些启动代码需要比其他的启动代码预先启动。C++的定义说应用级的初始器的运行顺序不可预测，但I/O操作和其他的系统库初始器需要在对C++应用中的初始器被调用之前运行。”完美”的办法是为每个初始例程都明确列出其依赖性，然后进行拓扑排序。BeOS的动态连接器基本上就是这么做的，通过使用库引用的依赖性检查。（如果A库依赖于B库，B库的初始器需要首先运行。）

一种简单得多的办法是使用多个初始段，`.init`和`.ctor`，这样启动过程可以先调用库一级的`.init`初始化例程，然后再调用C++的构建器的`.ctor`例程。在程序结束的时候也会有类似的问题，也分别用相应的`.dtor`和`.fini`段来解决。这样的系统允许程序员为代码分配优先级，用户代码的优先级从0到127，系统库代码的优先级从128到255，连接器在合并它们之前对其进行排序，这样最高优先级的初始器最先运行。这种办法仍然不能令人十分满意，因为构建器之间会有顺序依赖性，并产生难以发现的bug，但对于这些问题C++将解决依赖性的责任交给了程序员。

这种机制的一个变种就是在`.init`段中放入初始化代码的实际代码。在连接器合并它们的时候段就是所有的初始器的嵌入代码。只有很少的系统试着使用了这种方法，但在没有直接寻址的机器中很难使用这种方法，因为每个目标文件的代码块需要可以对其文件中的数据进行寻址，通常需要指向地址数据表的寄存器。匿名例程和其他例程一样建立它们的地址，通过已经解决的问题来减少寻址的问题。

IBM 精灵寄存器

IBM主机系统的连接器提供一种称为”外部傀儡”区或者”精灵寄存器”的有趣的特性。360系列是早期没有直接寻址的主机体系结构，也就是说小块的共享数据区域实现起来是非常昂贵的。每个需要引用一个全局目标的例程都要有一个指向这个目标的4字节的指针，如果这个目标仅是4个字节的开始的话，带来的开销就太大了。例如，PL/I程序对每个打开的文件和其他的全局目标都需要一个4字节的指针。（PL/I曾经是唯一的使用精灵寄存器的高级语言，尽管应用程序员不能直接使用。PL/I使用精灵寄存器作为指向打开的文件的控制块的指针，这样应用程序代码可以嵌入对I/O系统的调用。）

另一个相关的问题是OS/360不能提供现在我们称为每个进程或任务的本地存储的支持，对共享库也只提供有限的支持。如果两个任务运行相同的程序，其中一个标为可重入的，这样他们要么共享整个程序、代码和数据，要么不可重入，所以他们不能共享任何东西。所有程序都载入到同一个地址空间中，所以同一个程序的多个例程需要各自安排和其相关的数据。（360系统没有硬件重载，尽管370增加了这个功能，但直到OS/VS操作系统的几个版本后，系统才能提供每个进程的独立的地址空间。）

精灵寄存器帮助解决了这两个问题，如图5所示。每个输入文件都可以声明

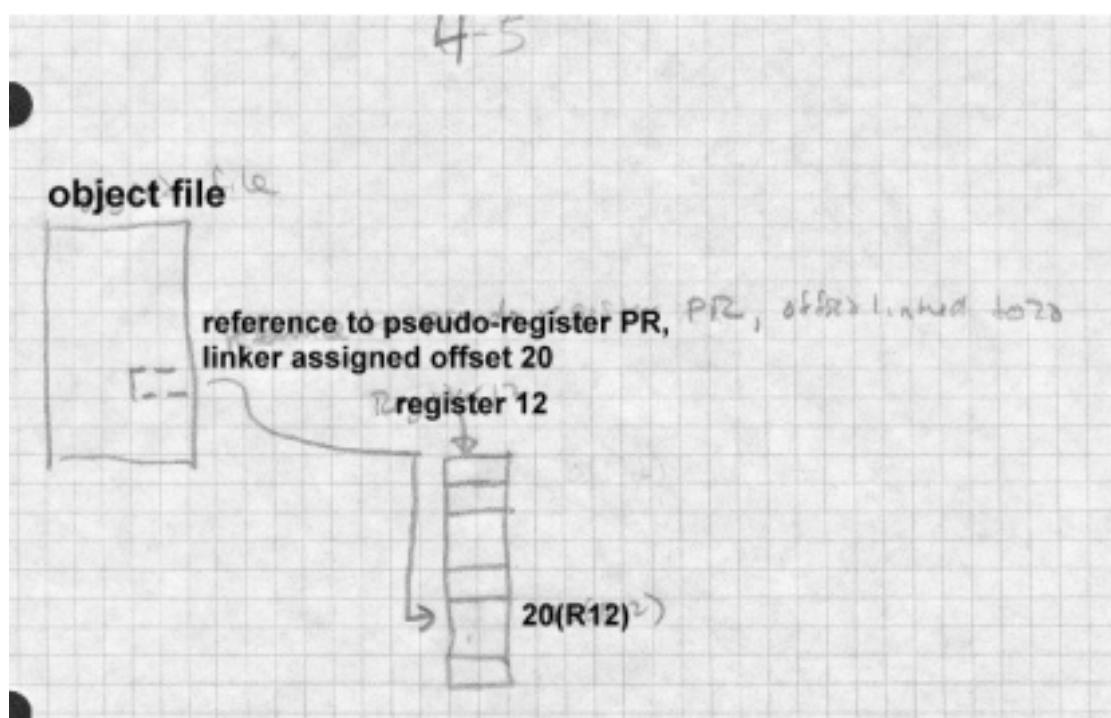
精灵寄存器，也称为外部傀儡区。（在360的汇编语言中，一个傀儡区等同于一个结构声明。）每个精灵寄存器都有一个名字，长度和对齐要求。在连接的时候，连接器收集所有的精灵寄存器，并放到一个逻辑段中，用最大的尺寸和最严格的对齐要求作为每个寄存器的尺寸和对齐要求，然后再给他们分配在这个段中的非覆盖的偏移量。

但连接器并不为精灵寄存器段分配空间。它仅仅计算段的长度，然后将其存入程序的数据空间中一个用特殊的CXD（累计外部傀儡）标出的地址，以及相应的重载信息。要引用某个特定的精灵寄存器，程序代码可以使用另一个特殊的XD（外部傀儡）标记和重载类型来标示在逻辑段中哪里保存着该寄存器的偏移量。

程序的初始化代码为精灵寄存器动态地分配空间，使用CXD来知道需要多少空间，按照惯例将地址存入寄存器R12，该寄存器在程序运行的过程中保持不变。程序的任何一个部分都可以通过将R12的内容和寄存器的XD值相加得到某个精灵寄存器的地址。要完成这个功能的通常的办法是用load或store指令，把R12作为索引寄存器，将XD项嵌入指令中作为地址替换域。（地址替换域只有12位，但XD将16位半字的高4位置为0，就是基寄存器为0，这样仍然能生成正确的结果。）

图4-5：精灵寄存器

bunch of chunks of space pointed to by R12. various routines offsetting to them



所有这些的结果就是程序的所有部分都可以通过load、store和其他RX格式指令对所有的精灵寄存器进行直接存取。如果一个程序有多个活动的例程，每个例程都用不同的R12分配一个不同的空间。

尽管精灵寄存器的最初的产生原因现在基本上没有了，但连接器为对线程本地数据提供支持的概念是非常好的，并且在现代的系统中有各种应用，最主要的就是Windows 32系统。而且现代的RISC计算机也有360的有限地址空间的问题，

需要内存指针表来提供任意内存位置的寻址。在很多RISC UNIX系统中，编译器在每个模块中生成两个数据段，一个用于传统的数据，另一个用于”小”的数据、在一定大小之下的静态目标。连接器将所有的小的数据段收集起来，然后通过程序启动代码将合并的小的数据段放到一个保留的寄存器中。这样就可以通过相对于这个寄存器的基寻址方法对小的数据直接引用。需要注意的是，与精灵寄存器不同的是，小的数据存储是通过连接器规划和分配的，每个进程只有小数据的一个拷贝。有些UNIX系统支持线程，但每个线程的存储是由显示的程序代码处理的，不需要连接器的支持。

特殊表

连接器分配的存储的最后一个资源是连接器本身。尤其是当一个程序使用共享库或覆盖技术的时候，连接器需要生成指针、符号和其他必须的段来支持共享库和覆盖技术。一旦这些段已经生成，连接器用和其他段一样的方法为其分配存储。

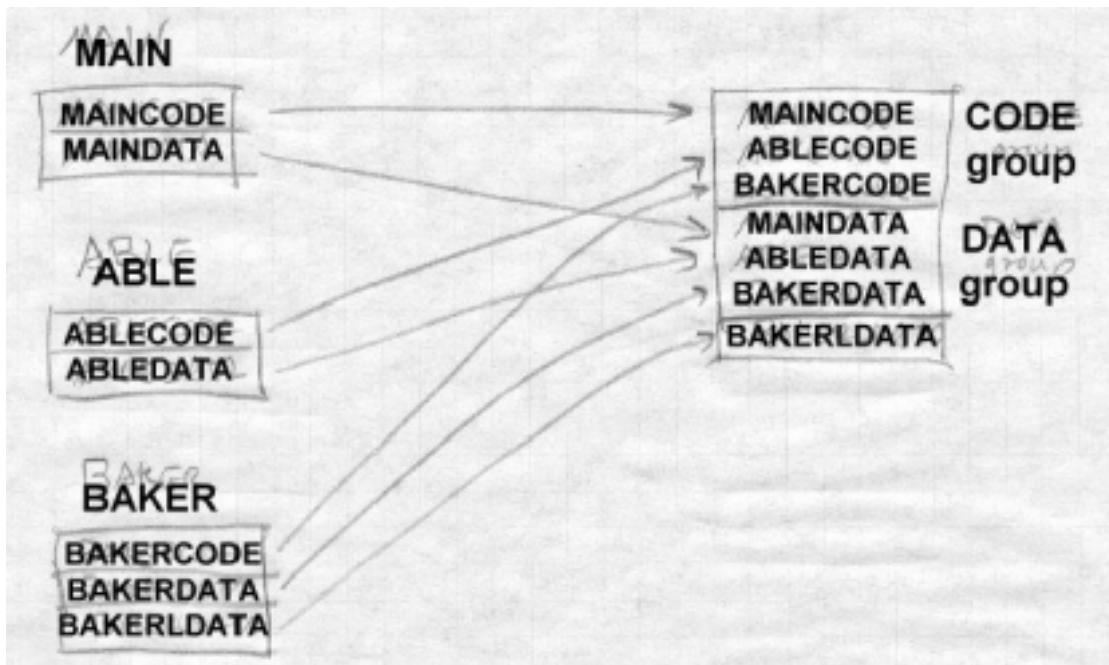
X86 的分段存储分配

对8086和80286关于分段内存寻址的特定需求需要特殊的功能来处理。X86的OMF目标文件为每个段命名，还有可选的类别。所有有相同名字的段根据一些由编译器或汇编器设置的标志位，合并到一个大的段中，然后一个类别中的所有的都分配到一个块中的连续空间中。编译器和汇编器使用类的名字来标记段的类型，如代码和静态数据，这样连接器可以将某个类的所有段分配到一起。只要一个类中的所有段的总长小于64K，它们可以看成一个单一的”寻址”组，使用一个段寄存器，可以节省可观的时间和空间。

图6显示了一个由三个输入文件连接起来的程序，分别是main、able和baker。Main包含MAINCODE段和MAINDATA段，able包含ABLECODE和ABLEDATA段，baker包含BAKERCODE和BAKERDATA和BAKERLDATA段。每个代码区都在CODE类中，数据在DATA类中，但BAKERLDATA中的”大数据”没有分配到类中。在连接好的程序中，假设CODE区总长为64K或更少，它们在运行时可以作为一个单一的段，使用短而不用长调用和转跳指令，以及同一个无需改变的CS代码段寄存器。类似的，如果所有的DATA的总和可以放在64K中，可以作为一个单一的段来处理，只需使用短内存引用，和无需改变的DS数据段寄存器。BAKERLDATA段在运行的时候作为一个单独的段来处理，使用载入一个段寄存器(通常是ES段)的代码来引用。

图4-6: X86

```
CODE class with MAINCODE, ABLECODE, BAKERCODE  
DATA class with MAINDATA, ABLEDATA, BAKERDATA  
BAKERLDATA
```



在实模式和286保护模式下的程序是用几乎相同的方法连接的。主要的区别在于一旦连接器在保护模式下生成了连接的段，连接就完成了，把实际分配的内存地址和段号留到程序载入的时候完成。在实模式下，连接器需要额外的步骤来将段分配到线性地址中，并且为段分配一个相对于程序起始地址的“段落”号。然后在载入的时候，程序载入器需要在实模式下的程序修正所有的段落号，或者在保护模式下的段号到程序实际载入的地址。

连接器控制脚本

传统上，连接器对输出数据的分配仅提供有限的用户控制。由于连接器开始要面对一个混乱的内存环境，如嵌入式微处理器；和多个目标环境，连接器需要提供在输出文件中对数据和目标地址空间的更加细粒度的控制。一些只有固定段的连接器通常有用于确定每个段的基址的开关，用于将程序载入到与标准应用环境不同的地址中。（操作系统通常使用这些开关。）有些连接器有数量巨大的命令行开关，由于系统对命令行的长度的限制，可以放到一个文件中。如微软的连接器可以有约50个命令行开关，可以设定文件中每个区的特性，还有很多输出文件的其他细节。

其他的连接器定义一种脚本语言来控制连接器的输出。GNU连接器带有很多命令行开关，也定义了一种语言。图7显示了一个简单的连接器脚本，用来为系统VR3.2的系统，如SCO Unix，生成COFF可执行文件。

图4-7：用于生成COFF的GNU连接器控制脚本

```
OUTPUT_FORMAT( "coff-i386" )
SEARCH_DIR( /usr/local/lib );
ENTRY( _start )
SECTIONS
{
```

```

.text SIZEOF_HEADERS : {
    *(.init)
    *(.text)
    *(.fini)
    etext = .;
}
.data 0x400000 + (. & 0xffffc00fff) : {
    *(.data)
    edata = .;
}
.bss SIZEOF(.data) + ADDR(.data) :
{
    *(.bss)
    *(COMMON)
    end = .;
}
.stab 0 (NOLOAD) :
{
    [ .stab ]
}
.stabstr 0 (NOLOAD) :
{
    [ .stabstr ]
}
-----
```

前几行定义了输出的格式，它们必须出现在编译到连接器中的格式表中，然后是要查找目标代码库的地方，缺省入口点的名字，在这里就是_start。然后再列出输出文件中的区。在区名字后面的可选的值是说这个区从哪里开始，这样.text区紧接着文件头部。在输出文件的.text区中含有所有输入文件中的.init区，然后是.text区和.init区。连接器定义在.fini区后面的符号etext。然后脚本将原来的.data区在文本区后面大概0x400000左右的4k页边界处开始，然后是所有输入文件中的.data区，后面定义符号edata。然后是紧接着数据区的.bss区，包含所有的输入的.bss区以及任何带有end标记bss的结束的公共块。(COMMON是脚本语言中的一个保留字。)在这之后是从输入文件中相关部分收集的用于符号表入口两个区，但在运行的时候并不载入到系统中，因为只有调试器才会使用这些符号。连接器脚本语言要比这个简单的例子灵活的多，足够描述从简单的DOS可执行文件到Windows PE可执行文件到复杂的覆盖分配中的所有内容。

嵌入式系统存储分配

在嵌入式系统中的分配和我们讨论的机制类似，只是由于程序必须运行在复杂的地址空间中而更加复杂。嵌入式系统的连接器提供的脚本语言可以让程序员定义地址空间的区域，可以将特定的段或目标文件分配到这些区域中，还可以

指定每个区域中段的对齐要求。

特殊处理器如DSP的链接器支持每个处理器特有的特殊性。例如Motorola 5600X DSP支持环缓冲区，必须设置和缓冲区一样的对齐边界。56K的目标文件将对这些缓冲区有特殊的段，连接器会自动把它们放到一个正确边界上，并且是各自使用单独的缓冲区。

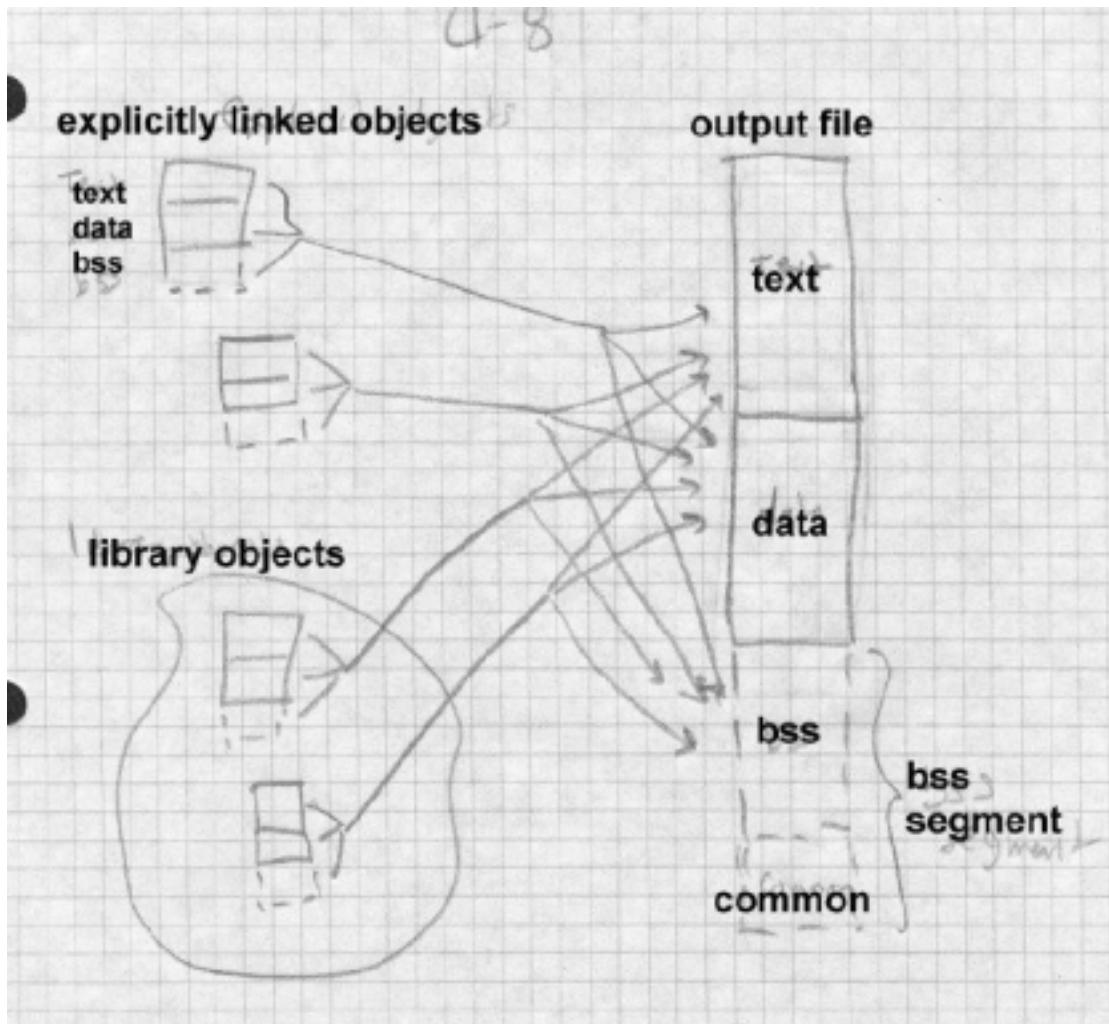
实际的存储分配

我们用一些流行的连接器对存储的分配来结束这篇。

Unix a.out 连接器的存储分配

在ELF前的Unix连接器的存储分配仅比我们这章开始讨论的理想化例子复杂一点点，因为在连接之前段集已经确定，如图8所示。每个输入文件都有文本、数据和bss段，也许还会有作为外部符号的公共块。连接器从每个输入文件中收集文本段、数据段和bss段，还有从库中导入的任何目标模块的长度。在读入了所有的目标后，带有非零值的未解析的外部符号作为公共块，并分配到bss的结尾。

图4-8: a.out连接
picture of text, data, and bss/common from explicit and library
objects being combined into three big segments



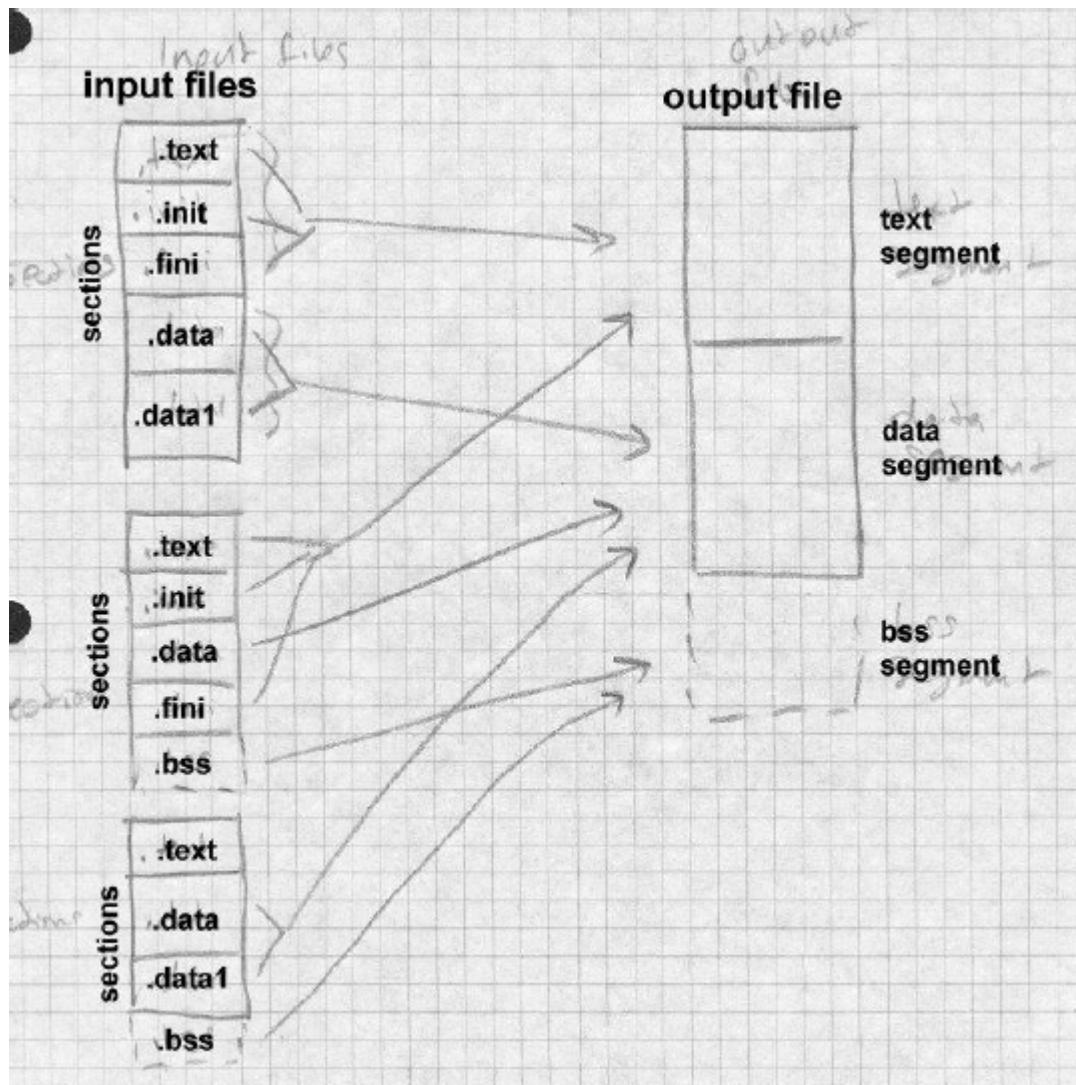
到目前为止，连接器能够为所有的段分配地址。文本段从一个固定的位置开始，这个位置取决于建立的a.out的不同，要么是位置零(最老的格式)，或者是位置零后的一个页面(NMAGIC格式)，或者是一个页面之后再加a.out头部的长度(QMAGIC格式)。数据段紧接着文本段(老式的非共享a.out格式)，或者是文本段的下一个页边界(NMAGIC格式)。在所有的格式中，bss段紧接着数据段。在每个段中，连接器将每个输入文件中的段分配到前面的段的下一个字边界。

ELF 格式中的存储分配

ELF的连接比a.out要复杂一些，因为输入的段可以非常大，连接器需要将输入的段(在ELF的术语中称为区)变成可载入的段(在ELF的术语中称为段)。连接器还需要建立程序载入器所需要的程序头部表，还有一些用于动态连接的特殊的区，如图9所示。

图4-9： ELF的连接

Adapt figs from pages 2-7 and 2-8 of TIS ELF doc
show input sections turning into output segments.



ELF 的目标文件有传统的文本、数据和 bss 区，这里为. text、. data 和. bss。通常它们还有. init 和. fini，用于程序开始和结束时的代码，以及各种零碎的内容。. rodata 和. data1 区用于一些编译器处理只读数据和字面线外数据。(有些编译器还有处理只读字面线外数据的. rodata1 区。)一些 RISC 系统象 MIPS 只有有限的地址偏移量，. sbss 和. scommon 区是“小” bss 区和公共块帮助将小的目标分组到可以直接寻址的区域，就像我们前面讨论的精灵寄存器。在 GNU C++ 系统中，文本区、rodata 和数据区中还有一次连接区。

不论区的类型有多么丰富，连接的过程基本还是一样的。连接器从输入文件中将每种区都收集到一起，还有库目标文件中的区。连接器还会记下哪些符号需要在运行的时候从共享库中解析，并且建立. interp、. got、. plt 区和符号表区以支持运行时连接。(我们会在第9章中详细讨论这些细节。)一旦这一切完成，连接器会按照传统的顺序安排空间。与 a.out 不同，ELF 目标不会载入到靠近地址零的任何地方，而是载入到地址空间的中间位置，这样栈可以在文本段的下面向下生长，堆可以在数据段的上面向上生长，这样总的地址空间保持相对紧凑。在 386 系统中，文本段的基地址是 0x08048000，允许在文本段的下面有相对大的栈而地址仍然可以保存在 0x08000000 之上，允许多数程序使用一个两级页面表。

(记住在386中，每个两级表映射0x00400000地址。) ELF使用QMAGIC技巧将头部放到文本段中，这样实际的文本段在ELF头部后面和程序头部表后面开始，通常在文件的0x100偏移量处。然后再分配数据段. interp(动态连接器的逻辑连接，需要首先运行)，动态符号表区. init，. text和一次连接文本，然后是只读数据。

接下来是数据段，逻辑地在文本段后一个页面处开始，因为在运行地时候页面既作为文本的最后一个页面也作为数据的第一个页面映射到内存中。连接器分配各种. data和一次连接数据，. got区，根据程序运行的平台，还有. sdata小数据和. got全局偏移量表。

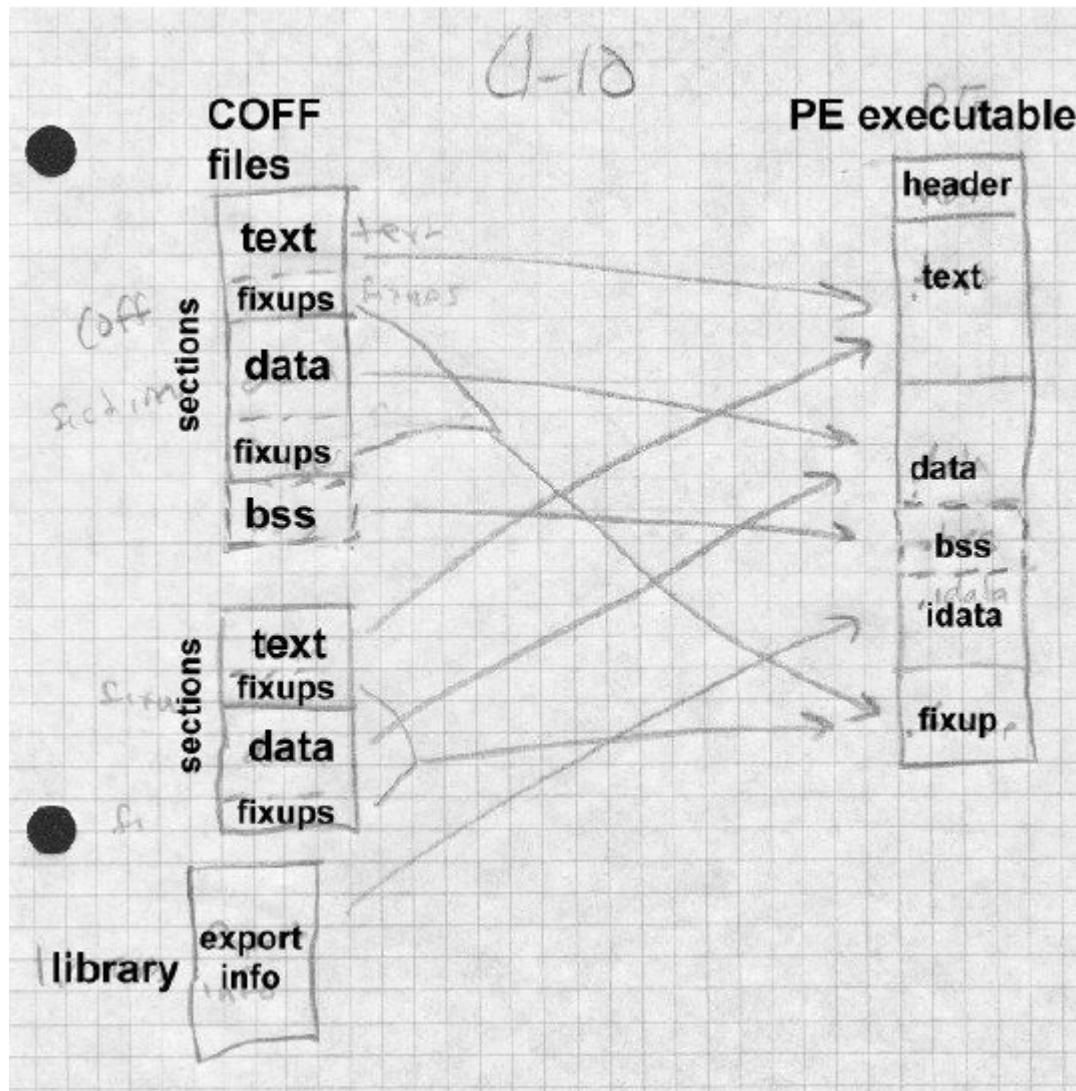
最后是bss区，逻辑地紧跟在数据区后面，由. sbss开始(如果有. sdata和. got就放在其后)，bss段和公共块。

Windows 连接器的存储分配

对Windows PE文件的存储分配比ELF文件简单一些，因为PE格式的动态连接需要比较少的连接器的支持，而需要更多的编译器的支持，如图10所示。

图4-10: PE文件的存储分配

adapt from MS web site



PE可执行文件传统上都载入到0x400000，文本区从这里开始。文本区包含输入文件中的文本，还有初试化和终结区。下面是数据区，对齐到一个逻辑磁盘块的边界。（磁盘块通常比内存页面小，在Windows机器中多为512字节或1K，而不是4K。）跟着是bss区和公共块，.rdata重载修正（用于通常不能在载入到希望的目标地址的DLL库），用于动态连接的导入和导出表，以及其他区如Windows资源。

一个不寻常的区的类型是.tls，线程本地存储。一个Windows进程可以而且在实际中会有多个线程控制同时活动。PE文件为每个线程分配.tls数据区。还有用于初试化的数据块和在线程开始和结束的时候要调用的函数序列。

练习

1. 为什么连接器要打乱段并且把相同类型的段放到一起？保留它们原来的顺序是不是更加容易一些？
2. 在什么时候连接器为例程分配存储用一个什么顺序会有影响？在我们的例子中，如果连接器分配newyork、mass、calif、main而不是main、calif、ma

ss、newyork会有什么不同。(在后面当我们讨论覆盖和动态连接的时候还会问这个问题，所以你可以忽略关于这些的考虑。)

3. 在很多情况下，连接器按顺序分配类似的区，如，calif的数据区，然后是newyork和其他的。但连接器会将有相同名字的公共区分配到其他区的前面。为什么？

4. 允许在不同的输入文件中声明有相同名字但不同长度的公共块是不是一个好主意？为什么是或不是？

5. 在例子1中，假设程序员重新写了calif例程，这样目标长度为十六进制1333长。重新计算分配的区的位置。在例子2中，再假设重新写的calif例程的数据和bss的长度为975和120。重新计算分配的段的位置。

项目

项目4-1：将项目3-1的连接器原型扩展，来支持UNIX风格的存储分配。假设考虑的段只有.、.data和.bss。再输出文件中，文本在十六进制1000处开始，数据在文本后的1000边界开始，bss在数据后面的4字节边界开始。你的连接器需要根据输出文件段定义输出局部目标文件。(现在你还无需输出符号、重载或数据。)在你的连接器中，要注意使用一种数据结构可以让你确定每个输入文件中的每个区都分配到哪些地址中，因为你需要在后续的章节中需要这个功能。使用例子2来测试你的分配器。

项目4-2：实现Unix风格的公共块。就是说，扫描符号表查找带有非零值的未定义符号，再在.bss段中增加适当的空间。不用担心调整符号表入口，我们下一章讨论。

项目4-3：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 \$

符号管理是连接器的关键功能。如果没有一种在模块之间相互引用的方法，连接器的其他功能就不会有什么用处。

绑定和符号解析

连接器要处理很多种符号。所有的连接器都要处理模块之间的符号引用。每个输入的模块都有一个符号表。符号包括：

- 在模块内定义的全局符号，该符号可能在这个模块内引用。
- 在模块内引用的全局符号，但没有在这个模块内定义（通常称为外部符号）。
- 段的名字，通常也作为在段的开始处定义的全局符号。
- 非全局符号，通常用于调试器和崩溃时的转储（dump）分析。并没有连接时所需要的符号，但有时它们也和全局符号混在一起，这样连接器至少可以跳过它们。有些时候它们会在文件中另一个表中，或另一个调试信息文件中。（可选的）
- 行号信息，告诉源语言调试器源程序行和目标代码之间的关系。（可选的）

连接器读入输入模块中的所有符号表，导出有用的信息，有时候就是所有的输入信息，通常仅仅是连接所需要的信息。然后再建立连接时符号表，并用这个表来指导连接过程。根据输出格式的不同，连接器会将部分或全部符号信息放到输出文件中。

在有些格式中，每个文件都有多个符号表。例如，ELF共享库可以有一个仅包含动态连接器所需要的信息的符号表，然后再有一个分离的更大的表用于调试和再连接。这并不是一个坏的设计，动态连接器表通常比完整的表要小，把它分离出来可以加快动态连接的速度，而动态连接要比调试库和再连接库要发生的多的多。

符号表格式

连接器符号表和编译器的表类似，尽管要简单一些，因为连接器所需要的符号通常比编译器需要的符号要简单。在连接器中，有一个表列出了输入的文件和库模块，记录每个文件的信息。还有一个符号表处理全局符号，连接器需要在输入文件中解析这些全局符号。第三个表处理模块间的调试符号，尽管连接器多数情况下不需要为调试符号建立一个全面的符号表，只要将调试符号从输入文件中传到输出文件中。

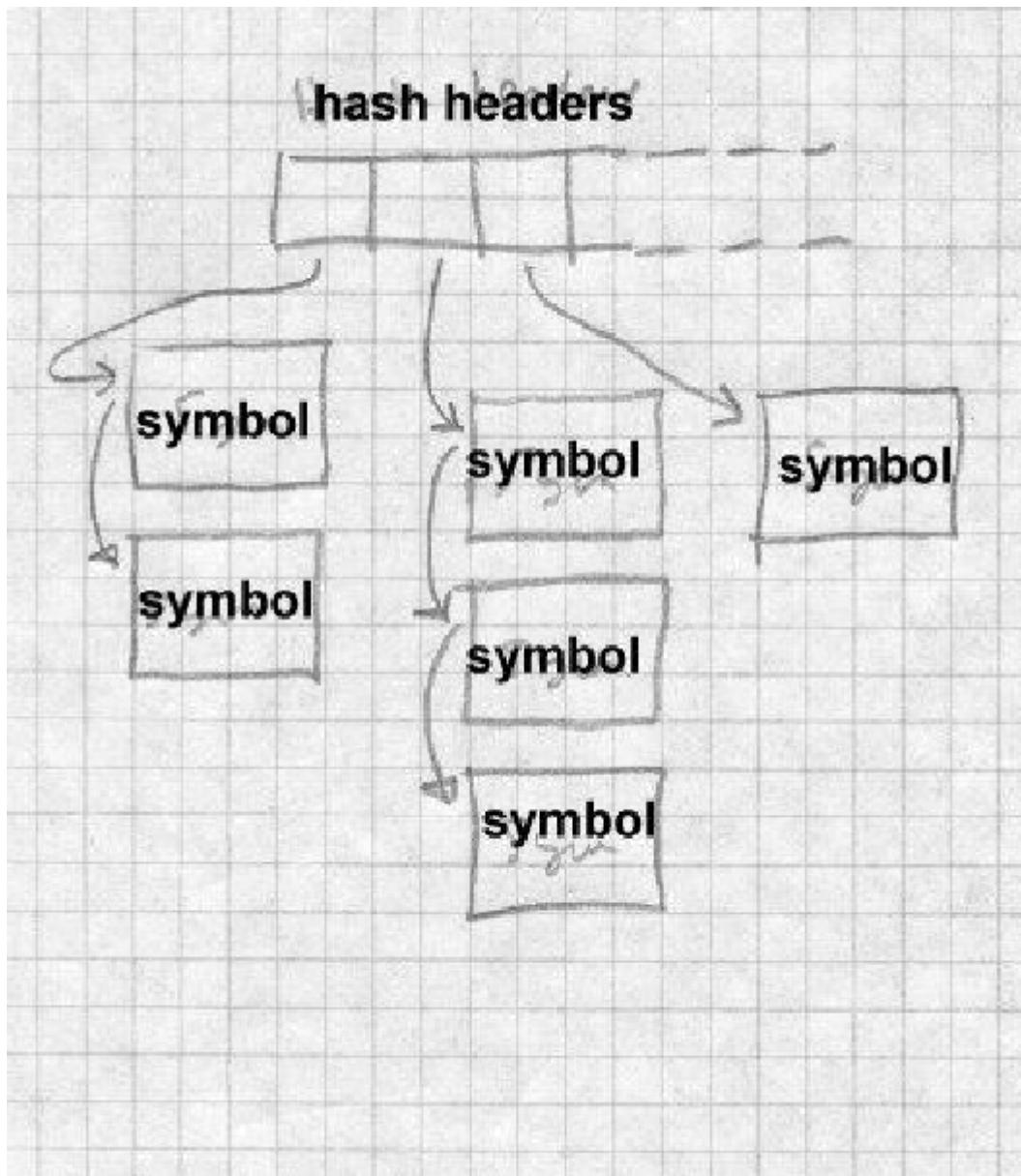
在连接器本身内部通常保存一个符号表作为一个表入口的数组，使用哈希功能来定位入口；或者作为一个指针的数组，用哈希功能来索引，所有哈希值相同的入口串在每个文件头部，如图1所示。要查找表中的一个符号，连接器计算出符号名的哈希值，使用这个哈希值对桶的数目求模，来选择哪一个桶（就是图中的`symhash[h%NBUCKET]`，这里`h`就是哈希值），在符号串中遍历查找这个符号。

传统的连接器仅支持短的名字，从IBM的主机系统和早期UNIX系统的8个字节到多数DEC系统的6个字节，可以少到一些正在消失的小型计算机的2个字节。现代的连接器支持长的名字，既因为程序员使用比以前长的名字（或者，对于Cobol而言，不愿意继续改变名字来保证前八个字节为唯一的。），也因为编译器通过额外的字符“调整”名字来将类型信息编进去。

带有有限名字长度的早期连接器对哈希查找串中的每个符号名字进行字符串比较，知道找到匹配的名字或查完所有的符号。现在，一个程序可以很容易就含有很多长的名字，只有最后几个字符不相同，通常在C++中的名字“修改”就是如此，这让字符串比较变得昂贵。一种简单的变通方法是在符号表中保存完整的哈希值，只有在哈希值相同的情况下才进行字符串比较。根据内容的不同，如果一个符号没有找到，连接器要么把它加到串中，要么报告一个错误。

图5-1：符号表

```
Typical symbol table with hashes or hash headers with
chains of symbols
struct sym *symhash[NBUCKET];
struct sym {
    struct sym *next;
    int fullhash; /* full hash value */
    char *symname;
    ...
};
```



模块表

连接器在连接过程中需要跟踪每个输入模块，既有显式连接的模块，也有从库中导出的模块。图2显示了GNU连接器在生成a.out目标文件时的模块表的简化的版本的结构。因为每个a.out文件的多数关键信息都在文件头中，这个表仅仅是保留了头部的一个拷贝。

图5-2：模块表

```
/* Name of this file. */
char *filename;
/* Name to use for the symbol giving address of text start */
char *local_sym_name;
```

```

/* Describe the layout of the contents of the file */
/* The file's a.out header. */
struct exec header;
/* Offset in file of debug symbol segment, or 0 if there is none */
e. */
int symseg_offset;
/* Describe data from the file loaded into core */
/* Symbol table of the file. */
struct nlist *symbols;
/* Size in bytes of string table. */
int string_size;
/* Pointer to the string table. */
char *strings;
/* Next two used only if 'relocatable_output' or if needed for
*/
/* output of undefined reference line numbers. */
/* Text and data relocation info */
struct relocation_info *textrel;
struct relocation_info *datarel;
/* Relation of this file's segments to the output file */
/* Start of this file's text seg in the output file core image.
*/
int text_start_address;
/* Start of this file's data seg in the output file core image.
*/
int data_start_address;
/* Start of this file's bss seg in the output file core image.
*/
int bss_start_address;
/* Offset in bytes in the output file symbol table
of the first local symbol for this file. */
int local_syms_offset;
-----
```

模块表还包含了指向了符号表和字符串表的内存中拷贝的指针(因为在[一个a.out文件中](#), 符号名字串是在一个与符号表分离的表中), 还有重载表, 还有输出文件中文本段、数据段和bss段的偏移量。如果文件是一个库, 每个连接的库的成员都有自己的模块表入口。(这里没有显示细节。)

在第一遍扫描过程中, 连接器从每个文件中读入符号表, 一般仅仅将其拷贝到一个内存中的缓冲区内。在那些将符号名字放入另一个分离的字符串表中的符号格式中, 连接器还会读入符号名字, 并且为了便于后续的处理, 遍历符号表, 并将每个名字串的偏移量编程一个指向内存中该串的指针。

全局符号表

连接器保存一个全局符号表，该表为任何输入文件中引用的或定义的符号有一个入口，如图3所示。每当连接器读取一个输入文件，将文件中的全局符号加到全局符号表中，保持一串符号定义或引用的地方。当完成第一遍扫描后，每个全局符号应该有一个定义和零个或多个引用。（这是一种最小的过度简化的描述，因为UNIX的目标文件将公共块作为带有非零值的未定义的符号，但只是连接器要处理的一种简单的特殊情况。）

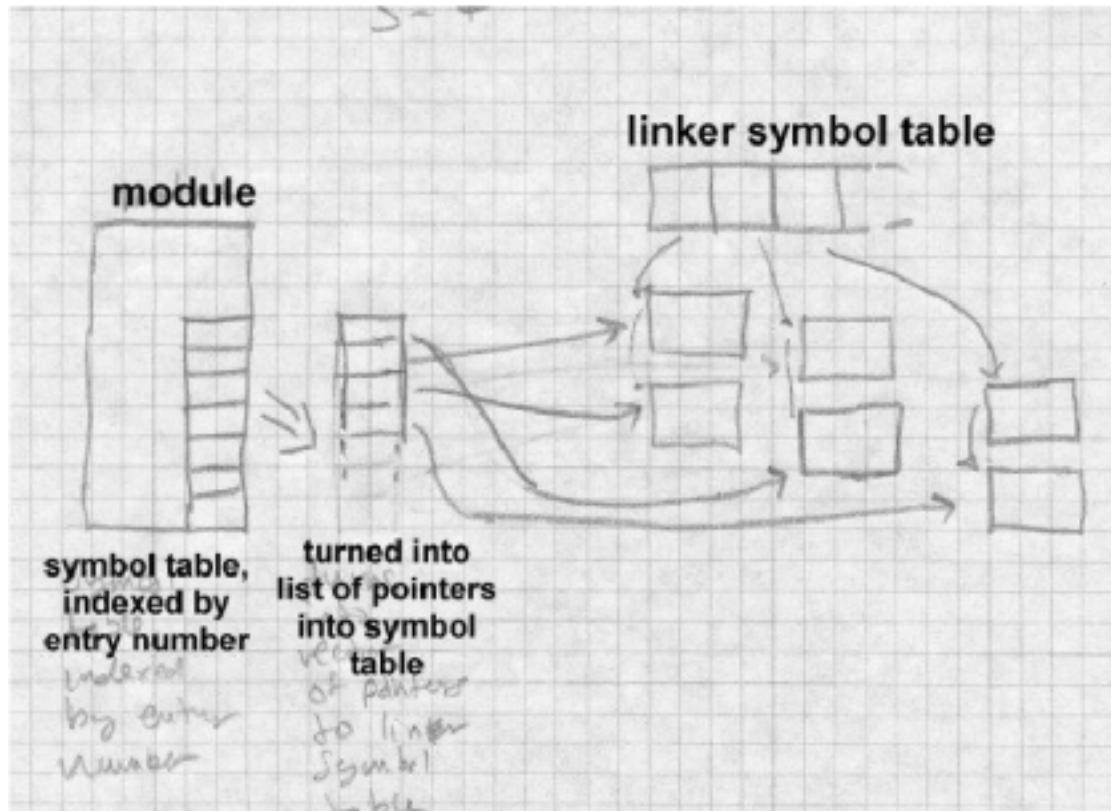
图5-3：全局符号表

```
/* abstracted from gnu ld a.out */
struct glosym
{
    /* Pointer to next symbol in this symbol's hash bucket. */
    struct glosym *link;
    /* Name of this symbol. */
    char *name;
    /* Value of this symbol as a global symbol. */
    long value;
    /* Chain of external 'nlist' s in files for this symbol, both
     * defs and refs. */
    struct nlist *refs;
    /* Nonzero means definitions of this symbol as common have been
     * seen, and the value here is the largest size specified by any of
     * them. */
    int max_common_size;
    /* Nonzero means a definition of this global symbol is known to
     * exist. Library members should not be loaded on its account. */
    char defined;
    /* Nonzero means a reference to this global symbol has been seen
     * in a file that is surely being loaded. A value higher than 1 is
     * the n_type code for the symbol's definition. */
    char referenced;
    /* 1 means that this symbol has multiple definitions. 2 means
     * that it has multiple definitions, and some of them are set elements,
     * one of which has been printed out already. */
    unsigned char multiply_defined;
}
```

在将每个文件中的全局符号加到全局符号表的时候，连接器将文件中的每个符号入口连接到全局符号表的相应入口中，如图4所示。重载信息通常用模块中自己的符号表的所以来指向相应的符号，所以对每个外部引用，连接器要能够知道，例如，在模块A中的符号15的名字是fruit，而模块B中的符号12的名字也

是fruit，就是说，是相同的符号。每个模块都有自己的索引集，并且需要各自的指针向量。

图5-4: 将文件中的符号解析到全局符号表
 Each module entry points to vector of symbols from input file, each of which is set to point to global symbol table entry.



符号解析

在第二遍连接过程中，连接器在建立输出文件的时候解析符号引用。解析细节和重载交互在一起(第7章描述)，因为在多数目标格式中，重载入口确定程序对符号的引用。在最简单的例子中，连接器建立一个带有绝对地址的输出文件(如在Unix连接器中的数据引用)，符号的引用简单地用符号地址来替换。如果符号被解析到地址20486，连接器将引用替换为20486。

实际的情况要更加复杂一些。其一，一个符号有很多种方法来引用，在数据指针中，在指令中，甚至由多个指令组合来引用。其二，连接器的输出文件本身通常是可重载的。这就是说，如果一个符号解析到数据区的偏移量426出，输出文件必须包含一个对符号所指向的data+426处的可重载引用，

输出文件通常有自己的符号表，因此连接器需要为输出文件中所使用的符号的索引建立一个新的向量，然后将需要重载的入口的符号编号映射到新的索引中。

特殊符号

很多系统使用一些由连接器自己定义的特殊符号。Unix系统都需要连接器定义etext, edata和end分别作为文本段、数据段和bss段的结束。系统的sbrk()使用end作为运行时堆的开始的地址，因此可以和已有的数据和bss段连续地分配下来。

对于带有构建器和解构器例程的程序，很多连接器都建立指向每个输入文件中的例程的指针表，带有象__CTOR_LIST_的连接器建立的符号，程序启动子程序可以查找这个列表并调用所有的例程。

名字调整(搅拌)

在目标文件符号表中和连接过程中使用的名字通常和编译目标文件的源程序中的名字是不一样的。这么做有三个原因：避免名字冲突，名字覆盖，和类型检查。将源程序中的名字转换为目标文件中的名字的过程叫做名字调整。这一节我们讨论在C、Fortran和C++程序中的典型的名字调整。

简单的 C 和 Fortran 名字调整

在早期的目标格式(在1970年左右之前)，编译器将源程序中的名字直接作为目标文件中的名字，也许会将长的名字截断到名字长度的限制。这个方法曾经可以工作得相当好，但会由于与编译器和库保留的名字冲突而造成一些问题。例如，完成格式化I/O的Fortran程序显式调用库中的子程序。其他的子程序处理算术错误，复杂的算术计算以及其他在编程语言中由于太复杂无法生成行内代码所有内容。

所有这些子程序的名字都是保留名字，编程传统的一部分就是记住不能用哪些名字。一个尤其异乎寻常的例子是，下面这个Fortran程序在很多年间都可以让一个OS/360系统崩溃：

```
CALL MAIN  
END
```

这是为什么呢？OS/360的编程传统是每个子程序，包括主程序都有一个名字，而主程序的名字是MAIN。当一个Fortran程序开始的时候，它调用操作系统来捕捉各种算术错误，每个捕捉陷阱都在系统表中分配一些空间。但这个程序一遍又一遍地递归调用自己，每次都生成另一套嵌套地陷阱调用，系统表空间用尽空间，然后系统就崩溃了。OS/390比其30多年前的机型要可靠得多，但仍然存在着名字解析问题。在混合语言程序下，情况更加糟糕，因为所有语言的代码都需要避免所使用的其他语言运行时库的名字冲突。

解决名字解析问题的一个办法是用其他的方法来完成对运行时库的调用。例如，在PDP-6和-10系统中，Fortran I/O包的接口是通过各种系统调用指令陷入回程序而不是操作系统。这是一种聪明的技巧，但太限定于PDP-6/10的体系机构，而不能很好地扩展，因为混合语言代码无法共享陷阱，对I/O包所需要的最必须的部分也不能连接，因为很难知道在程序中使用的输入模块要用什么来捕捉

陷阱。

在 Unix 系统上曾经使用的方法是调整 C 和 Fortran 过程的名字，这样就不会在不经意间将库函数和其他子程序的名字产生冲突。C 过程名字是用前置的下划线来修饰的，这样 main 就成了 _main。Fortran 名字被进一步调整，前面和后面都加上了下划线，这样 calc 就成了 _calc_。（这种方法可以在 Fortran 中调用名字后面带有下划线的 C 子程序，就可以用 C 来写 Fortran 的库函数。）这种机制唯一的缺点是将 C 语言的名字空间从原来的目标文件格式所允许的 8 个字符减少到 7 个，对于 Fortran 则减少到 6 个。而在当时，Fortran-66 标准只需要 6 个字符，所以这种方法并不难以接受。

在其他的系统中，编译器的设计者使用相反的方法。多数的汇编器和连接器都允许符号字符，如 . 和 \$，而在 C 和 C++ 的标识符中是禁止的。与将 C 和 Fortran 程序中的名字进行调整不同，运行时库函数使用带有禁止的字符的名字，这样就不会与应用程序的名字冲突了。在调整名字和使用无法冲突的库函数名字之间如何选择可以根据开发者的喜好进行。在 1974 年 Unix 用 C 进行重写的时候，它的作者已经有了相当多的用汇编语言写的库函数，把新的 C 和 兼容 C 的例程名字进行调整要比修改所有已有的代码要容易。现在，二十年过去了，汇编代码已经被进行了五次重写，而 UNXI C 编译器，尤其是生成 COFF 和 ELF 目标文件的编译器已经不再考虑下划线了。

C++ 类型编码：类型和范围

名字调整的另一个用途是对范围和类型信息进行编码，这样可以用现有的连接器来连接 C++、Ada 和其他语言的程序，它们的命名规则要比 C、Cobol 或 Fortran 复杂。

在 C++ 程序中，程序源可以定义很多有相同名字但不同范围的函数和变量，对于函数，还有参数类型。同一个程序可以有一个全局变量 V，还有一个类的静态成员 C::V。C++ 允许函数名字覆盖，多个程序可以有相同的名字但不同的参数，如 f(int x) 和 f(float x)。类的定义可以包含函数，包括覆盖的名字，甚至对内建操作符重新定义的函数，即一个类可以包含一个函数，而其名字实际上是 >> 或其他内建操作符。

C++ 最初被开发成一个叫做 cfront 的程序，就是生成 C 代码并使用已有的连接器，所以它的作者使用名字调整来产生可以通过 C 编译器并被连接器使用的名字。所有的连接器所做的就是通常的匹配定义的相同的名字和未定义的全局名字的工作。从那以后，几乎所有的 C++ 编译器都直接生成目标代码或至少是汇编代码，但名字调整作为处理名字覆盖的标准方法而保留了下来。现代连接器可以根据名字调整规则来在报告的错误信息中将名字进行反调整，但其他的时候则保留调整的名字。

很有影响力的 C++ 参考手册描述了 cfront 使用的名字调整机制，事实上的标准与其仅有很少的不同。我们在这里论述一下。

在 C++ 类外的数据变量的名字完全不进行调整。一个叫做 foo 的数组的名字调整后还是 foo。没有与类相关的函数名字的调整是根据参数类型用 _F 后缀和一串代表参数类型的字母进行的，类型修饰符如图 5 所列。例如，一个函数是 func (float, int, unsigned char) 变成了 func_FfiUc。类的名字被认为是有类型的，被编码成类的名字的长度后面跟着名字，如 4Pair。类还可以含有多个层

次的内部类的名字；这些”合法的”名字用Q来编码，后面是一个数字标明层次的数目，然后是类的名字，这样First::Second::Third变成了Q35First6Second5Third。也就是说一个有两个类参数的函数f(Pair, First::Second::Third)变成了f__F4PairQ35First6Second5Third。

图5-5: C++中调整后的名字的类型

Type	Letter
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 <i>n</i>	An_
function	F
pointer to nth member	MnS

类成员函数编码成函数名字，然后是两个下划线，然后是编码后的类的名字，然后是F和参数，这样，c1::fn(void)变成fn__2c1Fv。所有的操作符也都编码成4或5个字符的名字，如_ml代表*，而_aor代表|=。特殊的函数，包括构建器，解构器，new和delete也都编码成_ct，_dt，_nw和_d1。一个类Pair的构建器需要两个字符指针参数，Pair(char*, char*)变成了_ct__4PairFPcPc。

最后，因为调整后的名字会非常长，对带有多个相同类型参数的函数还有两种快速编码方法。代码Tn代表”和第n个参数相同的类型”，而Nm代表”n个和第m个参数类型相同的参数”。一个函数segment(Pair, Pair)会变成segment_F4PairT1，而一个函数trapezoid(Pair, Pair, Pair, Pair)会变成trapezoid_F4PairN31。

名字调整完成了对每个可能的C++对象都有一个唯一的名字的任务，代价就是在错误信息和列表的时候的极其长而且(如果没有连接器和调试器的支持的话)很难读的名字。

虽然如此，C++本来就存在着一个可能的巨大的名字空间的问题。任何表示

C++对象的名字的机制都会和名字调整一样复杂，而调整的名字还有对某些人类是可读的优点。

名字调整的早期用户发现尽管连接器理论上支持长的名字，在实际中很长的名字并不能很好地工作，而当连接带有很多直到最后几个字符才不同的长名字程序的时候性能是极其差的。幸运的是，符号表算法是一个很容易理解的题目，现在可以期望连接器在处理长名字的时候没有任何问题。

连接时类型检查

尽管只是随着C++的出现，调整名字变的流行起来，但连接器类型检查的想法已经存在很久了。（我第一次在1974年的Dartmount PL/I连接器中碰到它。）连接器类型检查的想法相当简单。多数语言都有带有类型声明的过程，如果调用者没有向被调用者传送其要求的数目和类型的参数，这就是一个错误，如果调用者和被调用者是在不同的文件中分别编译的话，这个错误是很难排除的。对于连接器类型检查，每个定义的或未定义的全局符号都和一个代表参数和返回类型的字符串相关联的，类似与调整的C++参数类型。当连接器解析一个符号的时候，它会比较引用中的类型字符串和符号的定义中的类型字符串，如果不匹配就报告一个错误。这个机制的一个很很好的特性是连接器完全不需要理解类型编码，只是比较字符串是否相同。

甚至在使用C++名字调整的环境下，这种类型检查还是非常有用，因为不是所有的C++类型都编码到一个调整的名字中。函数返回的类型，全局数据的类型可以用类似这种机制来检查。

弱外部和其他种类的符号

到目前为止，我们已经探讨了所有用相同方法处理的连接器全局符号，每次我们提到一个名字要么是一个符号的定义，要么是一个符号的引用。很多目标文件格式可以将一个引用分成一个弱或强的引用。强引用必须被解析，而弱引用如果有定义可以被解析，如果没有定义的话不会造成错误。连接器对弱符号的处理非常类似于对强符号的处理，除了在第一遍扫描结束的时候如果没有定义的话也不会产生错误。通常连接器将未定义的弱符号定义为零，这个值可以由应用程序代码检查。弱符号主要用于与库函数的连接，所以我们在第6章中再来研究它。

维护调试信息

现代编译器都支持对源程序的调试。也就是说，程序员可以根据源程序中函数和变量名字来对目标代码进行调试，并且可以设置断点和单步执行程序。编译器通过在目标文件中保留信息来实现这些功能，这些信息可以提供一个源程序文件行号和目标代码地址的映射关系和对程序中使用的所有函数、变量、类型和结构的描述。

UNIX编译器有两种略微不同的调试信息格式，即主要用在a.out、COFF和非

系统V的ELF文件中的stab(符号表的缩写)格式和为系统V的ELF格式定义的DWARF格式。微软为其Codeview调试器定义了其自己的格式，目前最新的是CV4格式。

行号信息

所有的符号调试器都需要将程序地址和源程序行号映射起来。这样，用户可以根据行号设置断点，而调试器可以在代码中适当的位置放置断点，还允许调试器根据调用堆栈跟踪到程序地址，然后根据相关的信息用行号来报告错误。

行号信息非常简单，除了一种情况，就是优化编译器可以将代码来回调整，这样目标文件中的代码序列与源程序文件中的行不能匹配。

源程序文件中的每一行，如果编译器生成任何代码，编译器都会生成一个行号入口，包括行号和代码的开始位置。如果一个程序地址在两个行号入口之间，调试器将其报告为二者中的小者。行号需要用文件名来界定，既有源程序文件名也有include的文件名。有些格式通过建立一个文件列表，然后在每个行号入口放入文件名索引的方法实现这个功能。其他的格式在行号列表中放入”begin include” 和” end include”，显式地维护一个行号堆栈。

当编译器的优化将同一条语句生成不连续的代码的时候，有些目标文件格式(主要式DWARF)让编译器将目标代码的每个字节都映射回到一个源文件行，在这个过程中消耗大量空间，而其他的格式仅是使用大概的位置。

符号和变量信息

编译器还需要为每个程序变量生成名字，类型和位置。这些调试信息比名字调整要复杂一些，因为不仅要对类型名字编码，对于结构类型还有类型的定义，这样调试器才能正确理解结构中所有的子项。

符号信息是一个隐式或显式的树。在每个文件的顶层式类型，变量和在顶层定义的函数的列表，其中每个都还有结构的域，函数中定义的变量，等等。在函数中，树还有” begin block” 和” end block” 标记，来映射到行号，这样调试器可以知道在程序的每个点中变量是否还在作用范围之内。

符号信息中最复杂的部分是位置信息。一个静态变量的位置不会变化，但在一个子程序中的局部变量可以是静态的、在栈中、在寄存器中、或者在优化代码中，在子程序的不同地方挪来挪去。在(树)叶子程序或没有分配局部静态变量的子程序中，一种常用的优化方法是跳过帧指针的设置。调试器需要知道这些，来正确地解释栈的跟踪(traceback)，并在一个没有帧指针的子程序中找到一个局部变量。Codeview通过一个没有帧指针的子程序的特殊的列表来实现这个功能。

实际的问题

对多数情况而言，连接器仅仅将调试信息不做翻译地传递过去，也许会在过程中进行段相关地址的重载。

连接器需要开始做的一件事是查找并删除重复的调试信息。在C尤其是C++

中，程序通常都有一系列头文件来定义类型和声明函数，每个源文件都含有这些头文件，定义了所有这个文件可能用到的类型和函数。

编译器将每个源文件包含的所有头文件中所有的调试信息都传递过去。这就是说，如果一个头文件被20个源程序文件包含，它们都被编译和连接到一起，连接器会收到这个文件的调试信息的20个拷贝。尽管调试器可以毫无问题地忽略重复的信息，头文件，尤其是C++程序中，重复的头文件将是巨大的。连接器可以安全地忽略重复的内容，这么做即可以加快连接器和调试器的速度，还可以节省空间。在有些情况下，编译器将调试信息直接存入文件或数据库中，由调试器直接读取，跳过连接器，这样，连接器只需要增加或更新段的相对位置信息，这些段是由每个源文件，还有由连接器本身生成的诸如转跳表等的数据文件生成的。

如果调试信息储存在一个目标文件中，有的时候调试信息和连接器符号混在一个大的符号表中，有时是存在不同的表中。Unix系统这些年来每次逐渐在编译器中增加一些调试信息，最后变成了现在的一个巨大的符号表。其他包括微软的ECOFF格式更趋向于把连接器符号和调试符号以及根据行号的信息分开。

有的时候最终的调试信息存入输出文件，有时候是一个分离的调试文件，或者两者都有。将所有调试信息都存入输出文件的好处在于生成过程非常简单，因为用来调试程序的信息都在一个地方。最明显的缺点是这样让可执行文件变得巨大。同样，如果调试信息分离出来，非常容易生成一个程序的最终版本，在发布的时候只发布可执行文件而没有调试文件。这样就能保持发布程序的尺寸很小，而且避免了非常容易进行的反向工程，但开发者还保留着在发布的程序出现错误时进行调试所需要的调试文件。UNIX系统有一个”strip”命令可以删除目标文件中的调试符号而一点也不改变代码。开发者可以保留未剥离的版本而发布剥离的版本。尽管两个文件是不同的，运行的代码是相同的，调试者可以用未剥离版本中的符号来调试一个由剥离版本生成的内存转储。

练习

1. 写一个C++程序，带有很多函数，这些函数的修改后的名字仅在最后几个字符不同。看看编译需要多少时间。改变函数的名字，使它们的调整的名字在开始几个字符就不同，再进行编译和连接，并记时。你需要一个新的连接器么？

2. 研究你最喜欢的连接器使用的调试符号格式。(有些在线资源列在书后的资源表中。)写一个程序将一个目标文件中的符号信息转出，看看你可以根据它重新建立多少源程序。

项目

项目5-1：扩展连接器，使之可以处理符号名字解析。使连接器可以从每个文件中读取符号表，然后生成一个全局符号表，连接器的后续部分可以继续使用。全局符号表中的每个符号都需要有，名字，符号是否经过定义，还有哪个模块定义的。注意要检查未定义和重复定义的符号。

项目5-2：为连接器增加符号值解析。因为多数符号都是在连接器输入文件中相对于段定义的，每个符号的值都需要根据每个段重载的地址来修改。例如，

如果一个符号定义在一个文件文本段的42位置处，而这个段重载到3710，符号变成3752。

项目5-3：完成项目4-2的工作；处理Unix风格的公共块。为每个公共块分配位置值。

第 6 章 链接库

每一种当前流行的连接器都可以处理库文件，它作为目标文件的集合，程序连接时需要它。本章将涉及传统的静态连接库，我们将在第9, 10章讨论更复杂的共享库。

连接库的目的

在上世纪40年代和50年代早期，程序操作台(或“软件工作室”，原文：programming shops)有实际的代码库，包含磁带卷轴和卡片盒，程序员可以查看并挑选加载某个例程和他的程序一起运行。当加载器和连接器开始解析符号引用后，自动地根据未定义的符号从库中选择例程的过程就变为可能。

一个库文件本质上只是一个目标文件的集合，通常额外还有一些使它能够更快速查找的目录信息。当然细节要比基本思想复杂的多，我们将在这章阐述这些细节。我们使用术语“文件”代表一个独立的目标文件，“模块”代表一个包含在库文件中的目标文件。

库文件格式

最简单的库文件格式就是目标模块的序列。在诸如磁带或纸带机这样的连续介质上，因为连接器不得不读取整个库文件，并且忽略库文件成员当然比把它们读入要快，因此增加一个目录是很重要的。在磁盘介质上，目录可以显著的加速库文件的查找速度，这现在已经是一个标准了。

使用操作系统

OS/360和它的后续产品(包括MVS)都提供了“partitioned data sets”(PDS)，它包含已命名成员，每一个命名成员都被认为是一个连续的文件。系统给单一成员提供了多别名的特性，这是为了在程序运行过程中、在列举一个逻辑PDS中的成员名称以及在读写成员时，将多个PDS当作一个单一的逻辑PDS对待。成员名称有8个字节，它或许和连接器引用的外部符号的长度不一致。(MVS引入一个扩展PDS，也称为PDSE，它支持长达1024个字节的名字，这对c, c++和Cobol程序员来说很有用)

一个连接库只不过是一个PDS，它里面每一个成员都是一个目标文件，并通过入口点命名。目标文件定义多个全局符号，当库创建时，会给每一个创建的全局符号分配一个别名。连接器会根据那些成员名称与未定义符号相匹配的库来搜索指定的逻辑PDS。这种方案的好处就是不需要维护目标库文件升级的程序，因为PDS的标准文件维护工具已经足够了。

虽然我从没见过一个连接器这样做，但是在类unix系统上的一个连接器可以用同样的方式处理库文件；库文件可以是一个目录，成员目标文件在这个目录

中，每一个文件名是一个在文件中定义的全局符号。(unix允许一个文件有多个文件名。)

Unix 和 Windows 档案文件

Unix连接库使用"archive"文件格式，它实际上可以用作任何文件的集合，虽然实际应用中它很少用作其它东西。库文件由一个档案文件头，一个可选择的文件头和目标文件组成。早期的档案文件没有符号目录，就是一些目标文件的集合，后继版本有多种类型的目录，经过了大约10年的时间逐渐稳定到BSD版本的类型（文本档案头和一个目录叫做_.SYMDEF），而现在，在System 4、后继版本的BSD和Linux上主要使用COFF和ELF类型的库（文本档案头和一个对应长文件的扩展名，目录称作/）。Windows的ECOFF库使用和COFF类似的文档格式，但是他的目录虽然也称作/，却使用不同的格式。

Unix 档案格式

所有现代的Unix系统都使用同一个档案格式的衍生版，见图1. 这种格式在档案头中只使用文本字符，这意味着一个文本文件自己就可以构成文本形式的档案文件（这种特性在实际应用中是没有用处的）。每一个档案文件以8个字节的“幻数”开头:!\n，\n代表换行。每一个档案文件成员都以一个包含如下信息的60个字节的头部信息开头:

- 成员名称，如果不够16字节，填充到16字节。
- 修改时间，以十进制表示的从1970年以来的秒数。
- 以十进制表示的用户和组ID
- 以八进制表示的Unix文件模式。
- 以十进制表示的文件大小。如果文件大小是奇数，就向文件结尾添加一个换行符使文件长度变为偶数，这个填充字节不记入文件大小域。
- 两个字节（反引号和换行符），使头部信息成为一行文本，这样可以对该头部是否真是一个头部提供一种简单的检查方法。

每一个成员头部都包含修改时间，用户和组ID和文件访问模式，虽然连接器忽略它们。

图6-1 Unix 档案文件格式

File header:

!<arch>\n

Member header:

```
char name[16]; /* member name */  
char modtime[12]; /* modification time */  
char uid[6]; /* user ID */  
char gid[6]; /* group ID */  
char mode[8]; /* octal file mode */  
char size[10]; /* member size */  
char eol[2]; /* reverese quote, newline */
```

成员名就是不多于15个字节的字符串，后面跟随一些空间使长度填充到16个字节，或者对COFF和ELF文件是在一个斜杠后面跟随足够的空间将长度填充到

16个字节。 (unix和windows都使用斜杠区别文件名中的不同部分) a.out文件使用的档案文件格式不支持超过16个字节的成员名，这是由于在BSD之前的UNIX文件系统将文件名中的每一部分都限制在14字节内 (一些BSD档案文件格式确实支持更长的文件名，但是由于连接器没有办法正确的处理长文件名，所以没人采用)。COFF, ELF和windows档案文件在名为“//”成员里存储超过16字节的文件名. 这个成员包含被斜杠分开的长文件名， 在Unix系统上还包括换行符，在windows系统上包含NULL字符。对于一个具有较长名称的成员，他的头部结构的name域(属于成员头部结构中)包含一个斜杠，后面跟着一个指示名字串在//成员中的位置偏移量的十进制数(这里得做试验验证一下，我心里也没有底，只是根据自己对ELF的理解翻译的)。对于window档案文件，//成员必须是档案文件的第三个成员. 而对于Unix档案文件，如果没有长文件名这个成员不必存在，如果有，则要紧跟在符号目录后面。

虽然符号目录格式有很多种，但从功能上来讲都是一样的，即把成员名称和成员位置进行映射以便连接器可以直接移动到他需要使用的成员并进行读取。

a.out格式文件把目录信息存储在__.SYMDEF成员里，此成员必须是文件中第一个成员，见图2. 这个成员以一个大小为2个字节的变量开头，这个变量包含着紧跟在它后面的符号表的长度(以字节为单位)。因此在符号表中的入口项数值是这个字的1/8(符号表每一项的长度是8个字节)。符号表后面是大小为2个字节，代表字符串表长度的变量。在字符串表中每一个字符串都要以null字符结尾. 每一个符号表表项都包含这个符号名在字符串表中的从0开始的偏移量，以及定义这个符号的头部的成员在文件中的位置. 符号表表项排列顺序和成员在文件中的顺序一样。

图 6-2 SYMDEF 目录格式

```
int tablesiz; /* size in bytes of following table */
struct symtable {
    int symbol; /* offset in string table */
    int member; /* member pointer */
} symtable [];
int stringsize; /* size of string table */
char strings[]; /* null terminated strings */
```

COFF和ELF档案文件格式使用另外的不可能出现在文件名中的符号“/”作为符号目录的名称，而不使用__.SYMDEF，并且使用更加简单的格式，见图3. 前四个字节代表符号数量. 接下来是一个标识成员在文件中位置偏移量的数组，然后是一系列以null结尾的字符串数组。第一个偏移量指向那个定义了以第一个字符串命名的符号的成员，然后依此类推。COFF档案文件的符号表通常使用大端字节序，而忽略宿主架构的字节顺序。

Figure 6-3: COFF / ELF 目录格式

```
int nsymbols; /* number of symbols */
int member[]; /* member offsets */
char strings[]; /* null terminated strings */
```

微软ECOFF档案文件有第二个符号目录成员，见图4，令人迷惑的是，它也叫"/".

Figure 6-4: ECOFF 第二个符号目录

```
int nmembers; /* count of member offsets */
int members[]; /* member offsets */
int nsymbols; /* number of symbols */
ushort symndx[]; /* pointers to member offsets */
char strings[]; /* symbol names, in alphabetical order */
```

ECOFF目录由一些成员入口项和一个成员偏移量数组构成，数组中每一个元素对应一个档案文件成员。接下来是一些符号，一个成员偏移量指针数组，以及一些以字母顺序排序的符号名。成员偏移量指针包含从1开始的成员偏移量表的索引，该偏移量索引表属于定义了相应符号的成员。举例来说，如果要定位对应第五个符号的成员，就引用数组中第五项，它包含定义这个符号的成员所在数组的索引值。理论上已排序过的符号搜索速度会更快，但实际上速度提升不可能很大，因为不论怎样连接器查找一个符号时，都要扫描整个符号表。

扩展到 64 位

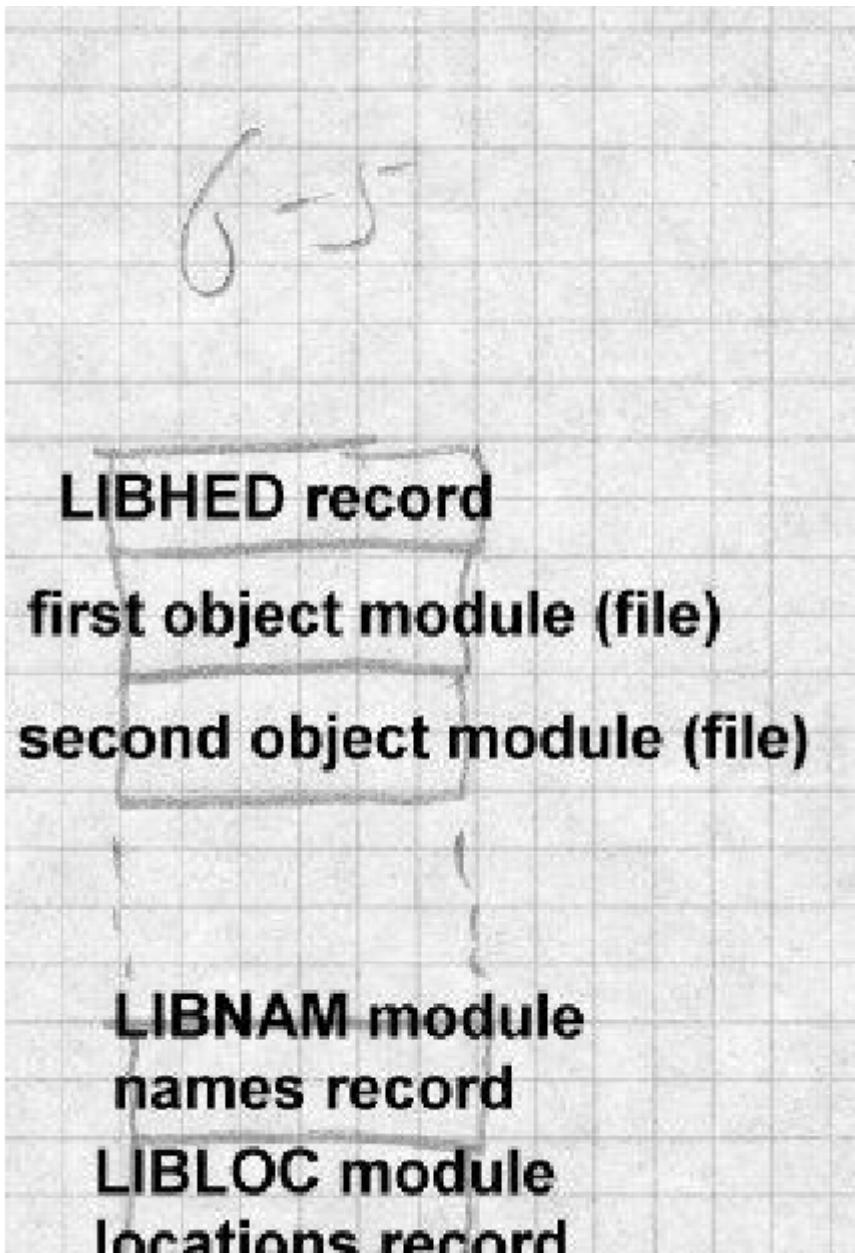
即使一个档案文件包含64位架构的目标文件，也不需要改变ELF和ECOFF档案文件格式，除非这个档案文件超过4GB。但是一些64位架构有不同的符号目录格式，并有不同的成员名，如 /SYM64/。

intel OMF 库文件

我们最后关注的库文件格式是intel OMF 库文件。再次声明，库文件是一些目标文件和符号目录的集合。不象unix库文件，intel OMF库文件的目录在文件的底部，图5。

Figure 6-5: OMF libraries

```
LIBHED record
first object module (file)
second object module (file) ...
LIBNAM module names record
LIBLOC module locations record
LIBDIC symbol directory
```



库文件以一个LIBDIC记录开始，它包含了Intel ISIS操作系统使用的以 (block, offset) 格式表示的LIBNAM记录的文件偏移量。LIBNAM记录只包含一个模块名称列表，每个名称前面有一个计数字节标识这个名字的长度。LIBLOC记录包含一个以 (block, offset) 格式指示各个模块在文件中起始位置的列表。LIBDIC包含一个在各个模块中定义的名字的字符串组成的字符串组的列表（字符串长度也会被记录），并且每个组末尾会有一个null字节以将他和后面的组区分开。

虽然这种格式有点晦涩，但它包含必要的信息，并且它能很好地工作。

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。根据系统对档案文件的

支持程度，建立一个库文件可能要用到所有标准的系统文件管理程序以及特定的库文件工具。

作为一种极端情况，IBM MVS库文件的建立方法是通过标准的IEBCOPY工具，它建立分段的数据区。作为一个趋中情况，Unix库文件通过“ar”命令建立，它整合文件到一个档案文件中。对于a.out档案文件，有一个叫ranlib的程序加载符号目录，从每一个成员里读符号信息，创建_.SYMDEF成员，并且把它拼接到文件里。理论上ranlib能够把符号目录作为独立的文件来创建，然后调用ar把它插入到档案文件里，但是实际上ranlib直接操作档案文件本身。对于COFF和ELF文件，ranlib的功能已经合并到ar了，如果任何一个成员是目标模块，ar就建立符号目录，尽管ar仍然可以创建非目标文件的档案文件。

作为另一种相反的极端情况，OMF档案文件和windows的ECOFF档案文件通过专门的库文件创建，因为除了目标库文件以外，很少有其它文件使用这些格式。

库文件创建的一个小问题是目标文件的顺序，特别是那些没有符号目录的老文件格式。在ranlib出现之前的unix系统上，有一对叫“lorder”和“tsort”的程序用来创建档案文件。Lorder处理输入的目标文件（不是库文件），并且输出一个符号依赖性列表，说明所需的某个文件的符号在哪个文件里。（这并不难做；lorder作为一个shell脚本通过一个符号列表工具提取符号，对符号作一个简单的文本处理，然后使用标准排序连接工具建立最终的输出。）Tsort根据lorder的输出作一个拓扑分类，产生一个文件分类列表，这样每个在引用之后才被定义的符号允许***忽略这个文件以便解析所有未定义符号引用。lorder输出是用来控制ar的。

虽然目前使用的库文件符号目录无论库文件中目标文件的顺序如何，都允许连接过程工作，但是大多数库文件仍旧使用lorder和tsort工具用来加速连接过程。

搜索库文件

库文件创建后，连接器必须能够搜索它。对库文件的搜索通常发生在连接器的第一次处理中，就是在连接器读取每一个输入文件之后。如果库文件有符号目录，连接器就读取这个目录，并依照连接器符号表依次查找每一个符号。

如果这个符号在程序中被引用但没有定义，连接器就从库文件中加入那个符号所在的文件。但对于后继的加载过程来说，只标记这个文件是不够的；连接器必须对库文件中各个段的符号进行处理，就好像这些符号在一个已连接程序中。段会被放入段表中，所有的已定义符号和未定义符号都要加入到全局符号表中。一个库文件中的函数引用另一个库文件中的函数非常普遍，举个例子，一个高级的i/o函数printf就要引用低级的putc或write函数。

库文件符号解析是一个交互过程。当连接器对目录中的符号进行过一遍扫描以后，如果这次扫描中它包括在这个库文件中的任何文件，它就会再进行一次扫描去解析被包含在内的文件所需的任何符号，直到它对整个目录完成了一次完全的扫描，并且没有发现任何需要再加入的文件为止。并不是所有连接器都这样；不少连接器仅仅对目录进行一次顺序的扫描，并忽略掉库中一个文件对先扫描的另一个文件的向前依赖关系。tsort和lorder等工具可以减少由于单次扫描连接器导致的困难。但是在加载器命令行中多次精确的列出同样的库，

以强制多次扫描和解析所有符号，这对程序员来说并不是难办的事情。

Unix连接器和许多windows连接器能够以命令行或控制文件的方式读取一个混合的目标文件和库文件列表，并且依次处理每一个，因此程序员可以控制加载目标文件和搜索库文件的顺序。虽然理论上这将表现出较大的灵活性，并且可以通过在实际库文件加载前列出用户定制库文件的方式，提交用户定制的库文件，但实际上顺序查找过程并没有提供额外的工具让程序员这样做。程序员总是列出他们所有的目标文件，任何特定应用程序二进制代码，以及提供数学运算的系统库，网络设备相关的库文件等等，最后是标准系统库文件。

程序员使用多个库文件时，当库文件中有循环依赖关系时经常需要不止一次地列出库文件。也就是说，如果一个在库文件A中的例程依赖在库文件B中的某个例程，但另一个在库文件B中的例程依赖库文件A中的某个例程的话，按顺序先查找A再查找B，或者先查找B再查找A都不能一次找到所有所需的例程。这个问题在依赖关系涉及三个库文件以上时更糟糕。通知连接器去查找ABA或BAB，或ABCDABCD是不雅的，但是却解决了这个问题。因为在这些库文件中很少有重名的符号，如果连接器能像IBM大型主机的连接器或AIX的连接器那样简单的对所有的符号都进行搜索，程序员就很舒服了。

一个主要的例外情况是应用程序有时在程序内部定义与库文件中函数同名的本地函数，并且用他们代替系统库文件中的函数，特别是malloc和free，这些是关于堆区管理的函数。在这种情况下，将设置一个连接器标识，表示“不要在库文件中查找这些符号”。这种方法通常比把局部函数放到库文件搜索顺序前面的效果更好。

性能问题

关于库文件的主要性能问题以前主要是顺序扫描库文件要耗费太多时间。当符号目录成为了标准后，从一个库中读入一个文件和读入一个独立的文件相比，就慢的不是那么明显了，并且只要库是经过拓扑排序的，连接器就很少需要对符号目录进行一遍以上的扫描。

如果一个库文件包含很多较小的函数的话，库文件搜索仍会很慢。一个典型的Unix系统库文件有超过600个成员。更一般的情况是，所有库文件成员在运行时整合在一个单一的共享库里，或许这比建立一个定义所有库文件中符号的单一目标文件要快，而且连接时使用这种方法比搜索一个库文件要好。我们将在第九章详细讨论这个方法。

弱(隐式)外部符号(Weak external symbols)

符号解析所和库文件成员选择所使用的简单的“声明-引用”模型对于一些应用程序来说，显得不是很灵活。比如，大多数C程序都调用printf类函数去格式化输出数据。Printf能够格式化所有类型的数据，包括浮点类型，这意味着所有使用printf函数的程序都要和浮点库连接在一起，即使这个程序实际上不使用浮点。

很多年来，PDP-11机的程序必须“欺骗”连接器，使得只使用整数的程序不连接浮点库。C编译器在很多使用浮点相关代码的程序中引用一个特殊的符号“f

ltused”。 C库文件布局见图6， 它利用了连接器顺序搜索库文件的优点。如果程序使用浮点，对f1tused符号的引用将使得程序和实际的浮点库连接在一起，包括fcvt，它是浮点数据输出函数。当I/O模块被连接以提供printf函数时，在这之前fcvt已经存在并且它满足I/O模块对底层函数的引用。在不使用浮点的程序中，实际的浮点函数将不被加载，因为在这种情况下不会有任何关于浮点的未定义符号，在I/O模块中对fcvt的引用将被定义为库中紧跟在I/O例程后面的空定义浮点例程。

图6-6: Unix 经典C库

...
实际的浮点模块，定义f1tused和fcvt
I/O模块，定义调用fcvt的printf函数
空浮点例程，定义了空fcvt
...

虽然这个窍门可以工作，但对于一个或两个以上符号来说，它并不实用，并且它的正确操作依靠于库中模块的顺序，所以当库文件重建时，它就很容易出错了。

解决这种问题的方法是弱(隐式)外部符号，即不会导致库成员被加载的外部符号。如果一个符号定义是有效的，或者在一个显式的被连接文件里，或者由一个常规外部符号而导致加载的库文件成员里，一个弱(隐式)外部符号就会象一个标准的外部符号那样被解析。但是如果没有一个定义是有效的，这个弱(隐式)外部符号就被当成一个未定义符号，实际上会被填充为0。但这并不被认为是一个错误。在上面的情况下，这个I/O模块将对fcvt进行弱(隐式)引用，实际的浮点模块紧跟在I/O库中模块后面，并且不需要空例程。这时如果有一个对f1tused的引用，那么这个浮点例程将被连接并定义了fcvt。如果没有对它的引用，对fcvt的引用将保留未解析状态。这将不再依赖于库的顺序，并且对于使用多次解析扫描的库也是可用的。

ELF规范里有另外一种弱(隐式)符号，一个弱定义或称弱引用。如果没有有效的常规定义，一个弱定义就会定义一个全局符号。如果存在一个有效的常规定义，这个弱定义将被忽略。弱定义很少使用，但是当不需要将报错空例程放置在单独模块的前提下定义报错空例程时，是非常有用的(译者注：这里的报错空例程，是指程序中用到的，在调试模式下可以输出报错信息，在发布模式下被定义为空的例程)。

练习

如果两个在不同的库文件中的模块定义了相同符号，连接器将怎样做？这是一个错误吗？

库文件符号目录通常只包括已定义的全局符号。包括一个未定义的全局符号是否也有用？

当使用lorder和tsort排序目标文件时，tsort可能不能做这些文件的全序排序。当它发生时，这是否是个错误？

一些库文件格式把目录放置在库文件的前部，而有些格式把它放置在库文

件底部。这实际上有什么不同吗？

描述一些隐式外部符号和隐式定义有有用处的其它情形。

项目

这部分项目给连接器增加了库文件搜索的功能。我们将拿两种不同格式的库文件进行试验。第一个是IBM类的目录格式，在本章上半部分我们介绍过它。一个库文件就是一个目录，每个成员是这个目录中的一个文件，(each file having names for each of the exported files in the directory.) 如果你使用的系统不支持Unix式样的多名字，你可以伪造它。给每个文件命名一个名字(从输出符号中选择一个)。然后把一个文件命名为MAP并包含如下所示的格式：

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

这里的name是文件名，sym是输出符号的其它部分。

第二种库文件格式是一个单一文件。这类库文件以下列行开头：

```
LIBRARY nnnn pppppp
```

这里的nnnn是库文件里面的模块编号，pppppp代表库文件目录开始的偏移量(and pppppp is the offset in the file where the library directory starts.)接下来是库成员，一个接一个的排列。在文件的底部，在偏移量pppppp处是库文件目录，它由几行组成，以下列格式排列：

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

where pppppp is the position in the file where the module starts, 111111 is the length of the module, and the symi are the symbols defined in this module.

Project 6-1: Write a librarian that creates a directory-format library from a set of object files. Be sure to do something reasonable with duplicate symbols. Optionally, extend the librarian so it can take an existing library and add, replace, or delete modules in place.

Project 6-2: Extend the linker to handle directory-format libraries. When the linker encounters a library in its list of input files, search the library and include each module in the library that defines an undefined symbol. Be sure you correctly handle library modules that depend on symbols defined in other library members.

Project 6-3: Write a librarian that creates a directory-format library from a set of object files. Note that you can't correctly write the LIBRARY line at the front of the file until you know the sizes of all of the modules. Reasonable approaches include writing a dummy library line, then seeking back and rewriting line in place with the correct values, collecting the sizes of the input files and computing the sizes, or buffering the entire file in main memory. Optionally, extend the librarian to update an existing library, and note that it's a lot harder than updating a directory format library.

Project 6-4: Extend the linker to handle file-format libraries. When the linker encounters a library in its list of input files, search the library and include each module in the library that defines an undefined symbol. You'll have to modify your

routines that read object files so that they can read an object modules from the middle of a library.

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，这个问题就困惑着我，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。*Library "bit creep" is a chronic and very hard to debug source of program errors when a program runs using different versions of libraries than its authors anticipated*(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成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。

Figure 7-1: Simple relocation entry
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。

Figure 7-2: a. out relocation entry

```
int address /* offset in text or data segment */
unsigned int r_symbolnum : 24, /* ordinal number of add symbol */
r_pcrel : 1, /* 1 if value should be pc-relative */
```

```
r_length : 2, /* log base 2 of value's width */
r_extern : 1, /* 1 if need to add symbol to value */
-----
```

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远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代替。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。

Figure 7-3: SPARC relocation entry

```
int r_address; /* offset of of data to relocate */
int r_index:24, /* symbol table index of symbol */
r_type:8; /* relocation type*/
int r_addend; /* datum addend*/
```

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，但是链接器为了一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。

Figure 7-4: MS COFF relocation entry

```
int address; /* offset of of data to relocate */
int index; /* symbol index */
short type; /* relocation type */
```

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_RELA。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_RELA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。

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LIDATA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向，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，*with elaborate reverse polish strings representing link-time expressions to be resolved and stored into program memory.* 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，因此现代连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SETH I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REFLO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：让连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15000，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复本的问题，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字大小的寄存器保存区域。T

SS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

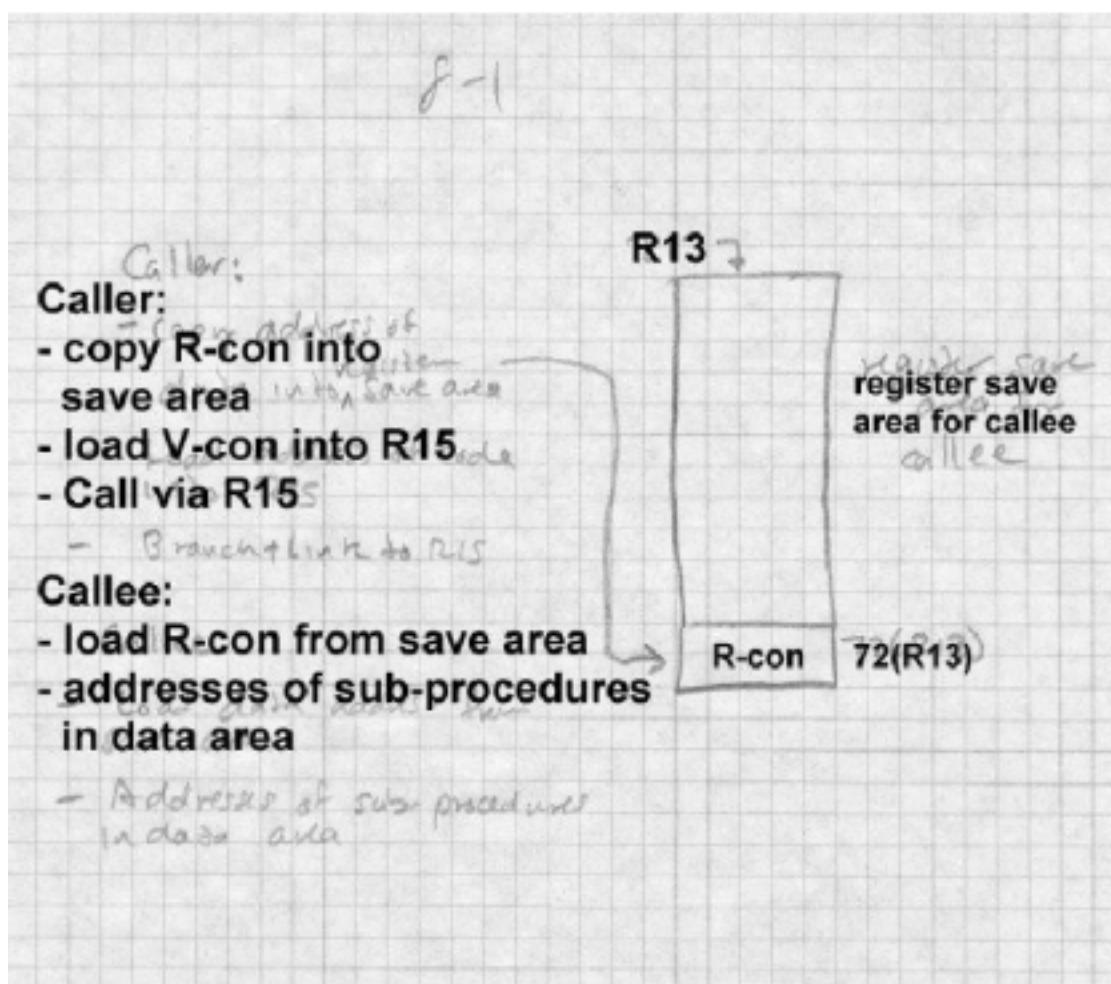
将V-con加载到寄存器R15

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（根据作者的上下文来看，当时使用的2个字长的指针是不同于同时代的普通指针字长的，那时候的指针字长可能是1个字？得确认

一下) (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]
[ROMP style data table with code pointer at the beginning.]

调用者:

将指针表地址加载到指针寄存器RP
按寄存器0(RP)的内容将代码地址加载到寄存器RC。
通过RC调用。

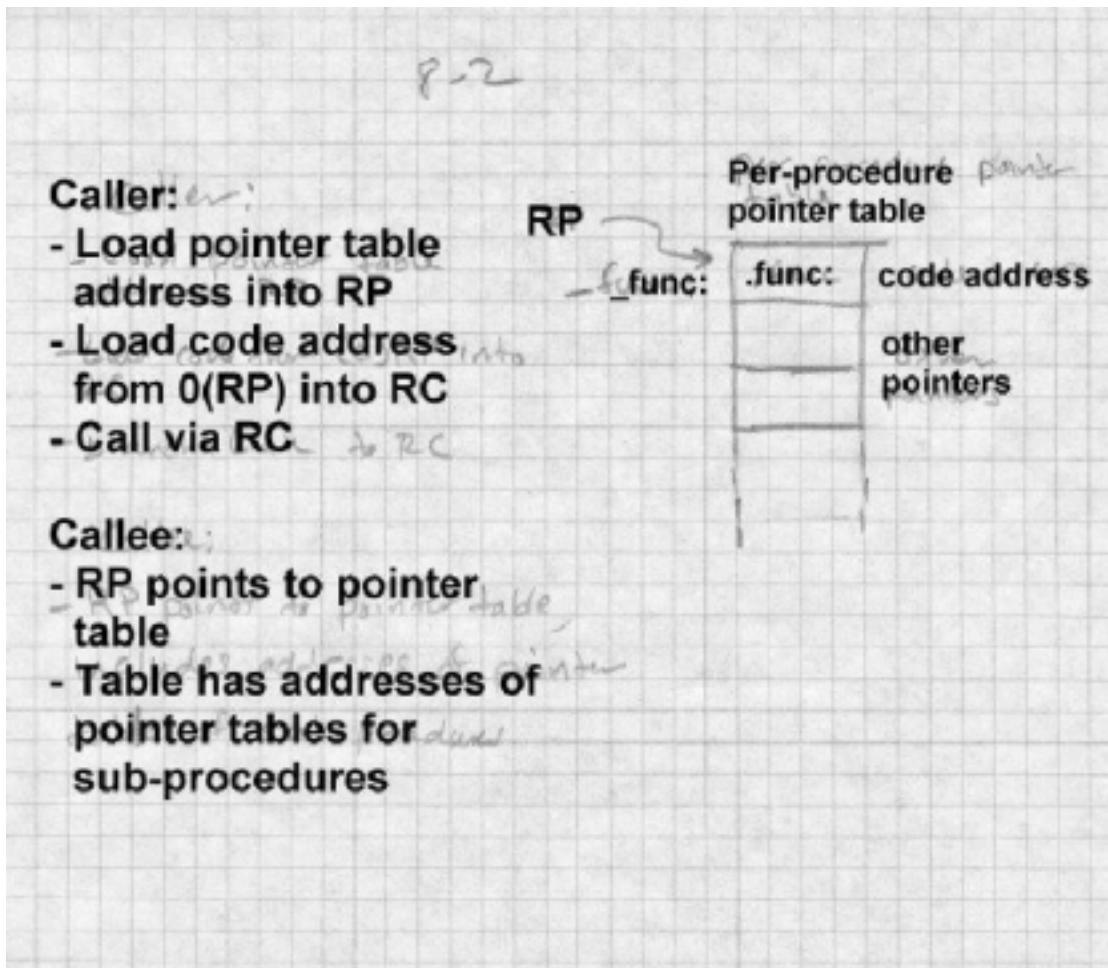
被调用者:

RP指向指针。

表:

table

表具有子过程的指针表地址



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
```

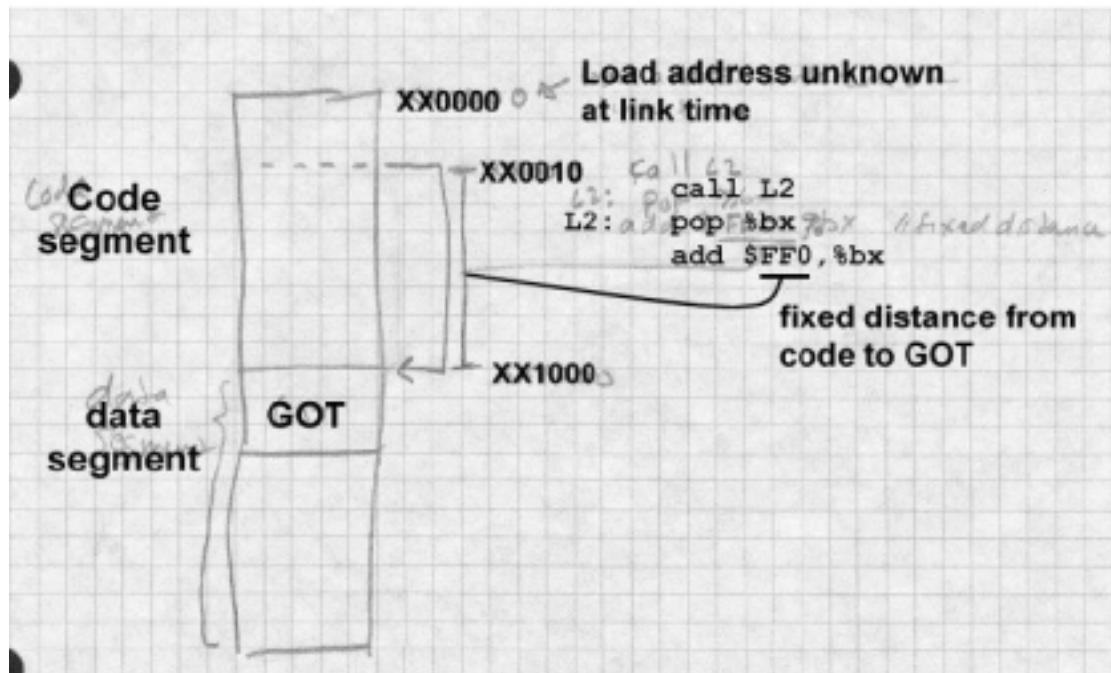
它存在一个对后面紧接着位置的call指令，这可以将PC(译者注：程序计数器，即指令指针寄存器)压入栈中而不用跳转，然后用pop指令将保存的PC加载到一个寄存器中并立刻加上call的目标地址和GOT地址之间的差。在一个由编译器生成的目标文件中，专门有一个针对addl指令操作数的R_386_GOTPC重定位项。它告诉链接器替换从当前指令到GOT基地址的偏移量，同时也是告诉链接器在输出文件中建立GOT的一个标记。在输出文件中，由于addl到GOT之间的距离是固定的，所以就不再需要重定位了。

(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，在链接的时候连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_BLOBAL_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64 K，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

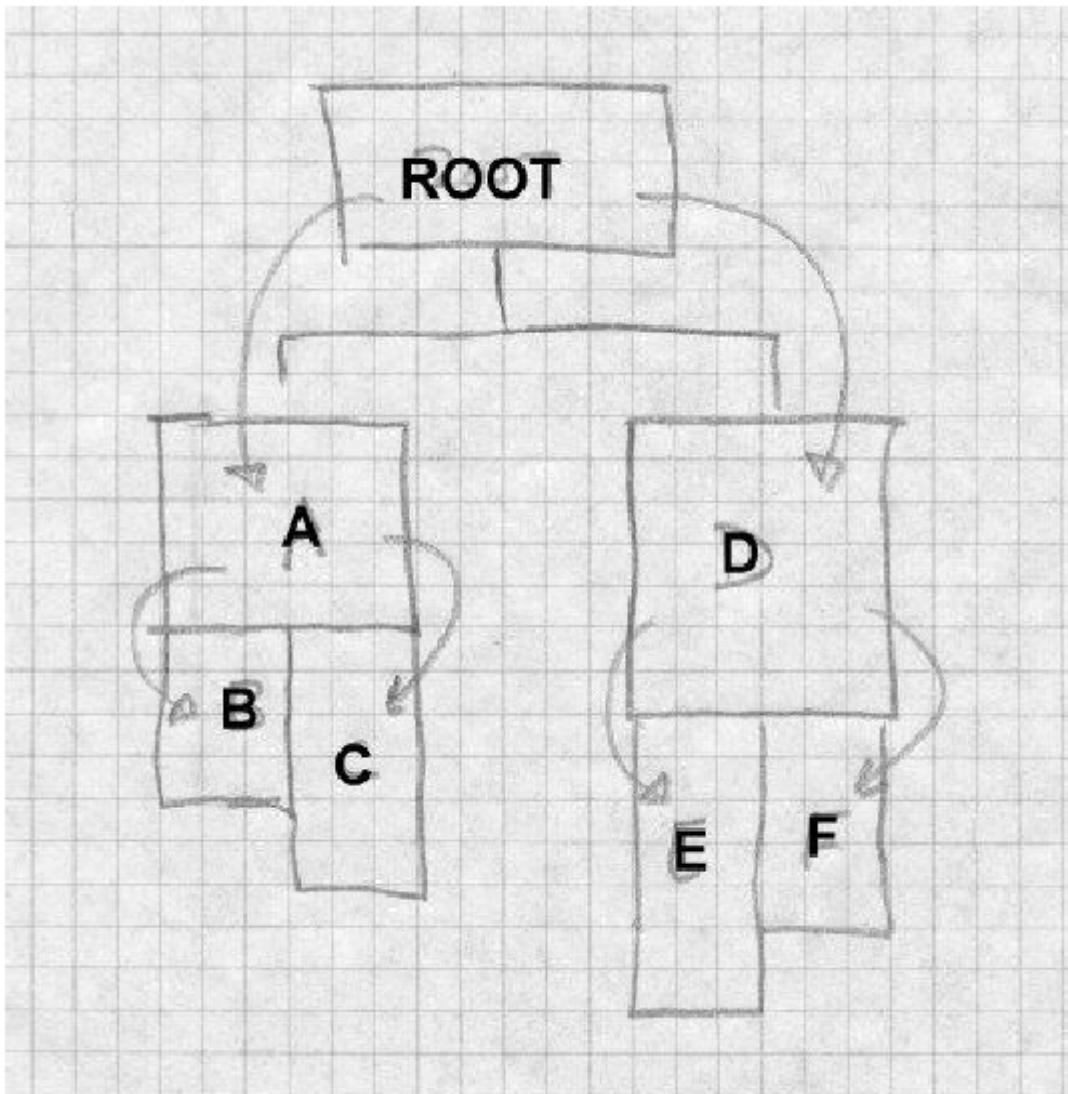
我们通过一个对树状结构的覆盖来结束这一章，这是在虚拟内存出现之前让程序运行在比自己小的内存中而广泛使用的一种方案。覆盖是另一个使用在1960年代以前的技术，并且在一些内存限制严格的系统中仍然使用。1980年时就有一些MS-DOS链接器像25年前就采用此技术的大型主机计算机那样支持树状结构覆盖。虽然现在常用的系统上已经很少使用覆盖技术了，这种链接器用以创

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）

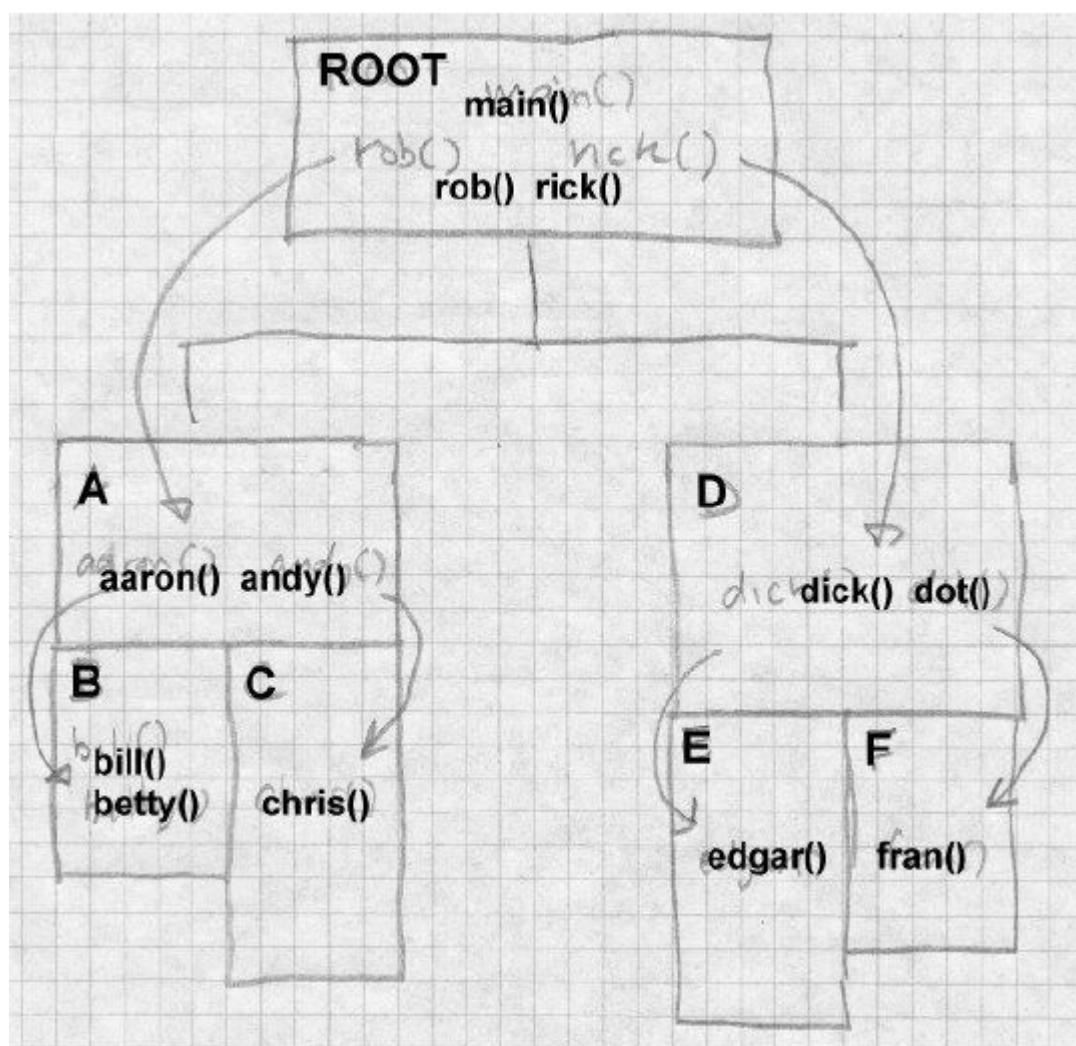


图6展示了可以告诉IBM 360链接器以创建这种结构的链接器命令。空格并不碍事，所以我们对命令进行了缩进以对应这种树状结构。OVERLAY命令定义了每一个段的开头；相同覆盖名的段定义了覆盖彼此的段。所以第一个OVERLAY A D定义了段A，第二个OVERLAY AD定义了段D。覆盖段按照从左到右的深度优先顺序在树中进行定义。INCLUDE命令为链接器要读取的逻辑文件命名。

（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（overlay fine points，不知道翻译的是否准确）

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，覆盖连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 origa

mi” 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 \$

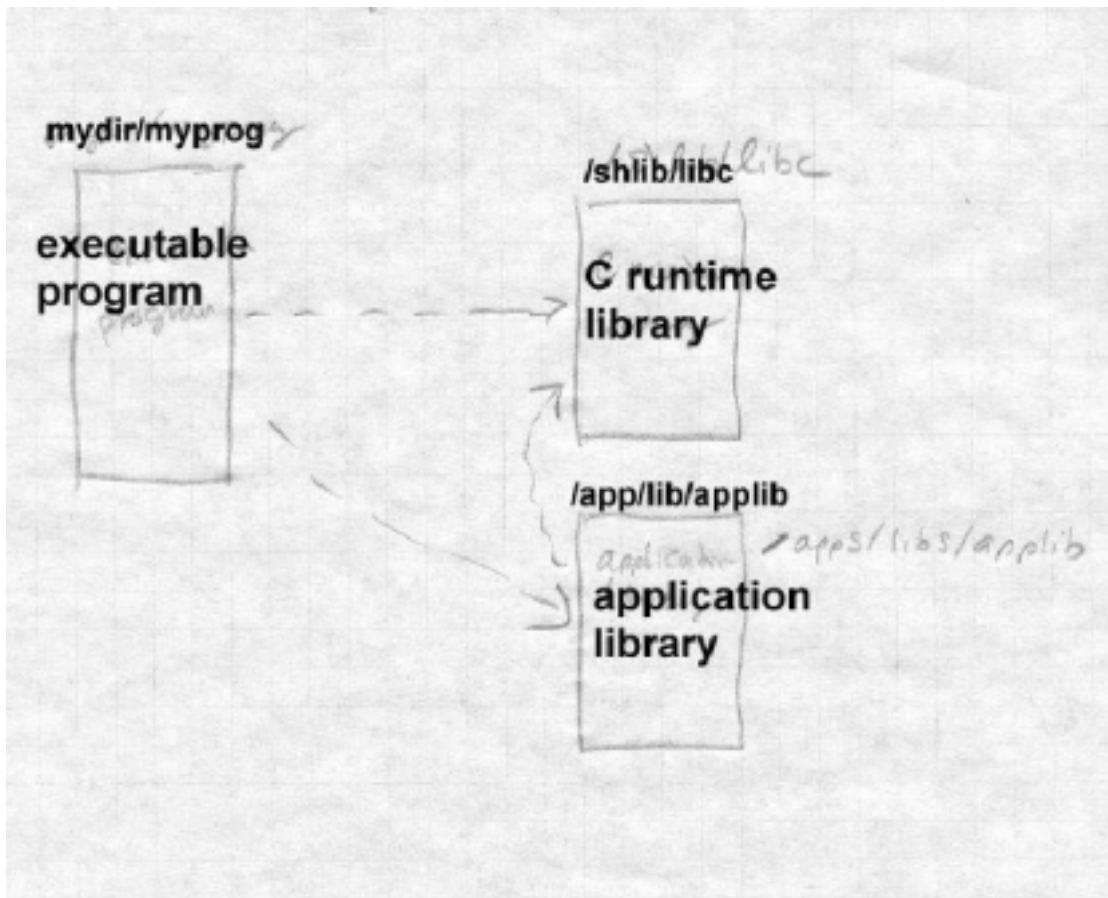
\$Date: 1999/06/15 03:30:36 \$

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 and 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。

Figure 9-1: Program with shared libraries
Picture of executable, shared libraries
main executable, app library, C library
files from different places
arrows show refs from main to app, main to C, app to C



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 (Binding time)

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，有时一个静态共享库就可以被更新而不至于破坏程序对它的调用（这句话我建议后半句可以翻译的顺口一些，不必太严格按照原句，建议翻为：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。实际上，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。SVR 3.2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，即共享库本身和给连接器用的空占位库(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（可能会包含一个小的自举例程），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连接器，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。假定当连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。

数据全局符号会被连接器放在共享库中任何地方，获取它们的数值的最合理的方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地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/shlib/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：就是说得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存在于多个目录中。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连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。连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uselib()。

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。

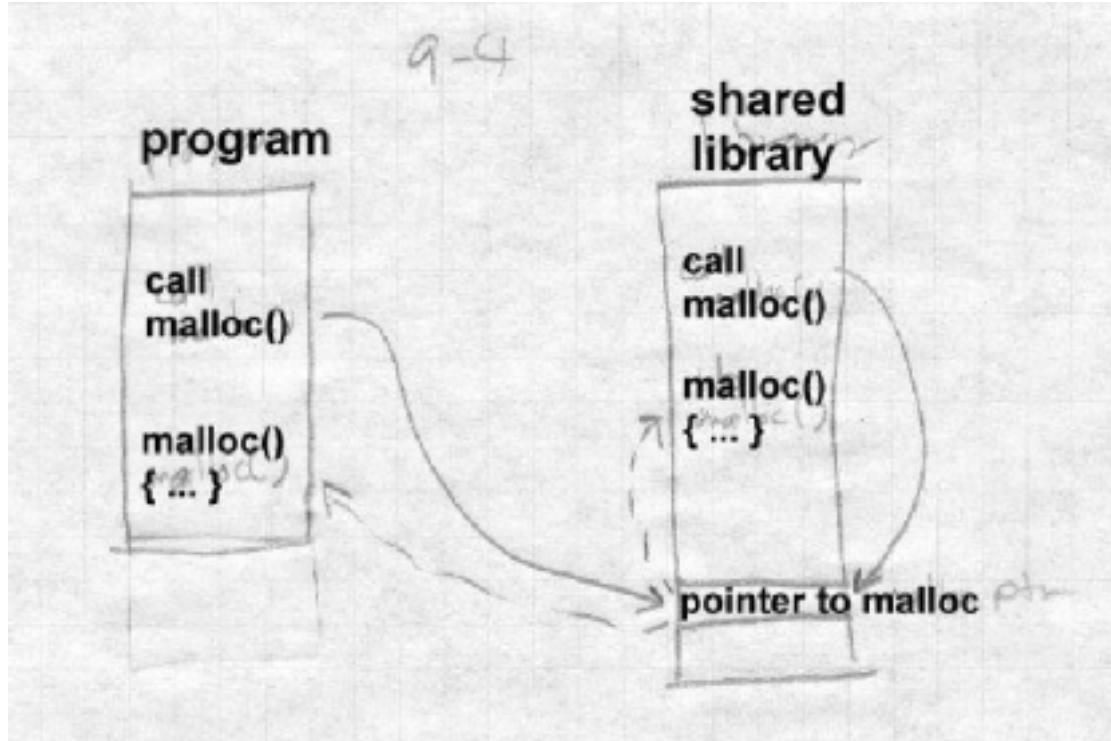
但是很多程序定义了它们自己的malloc()和free()版本，这是分配堆存储的例程；如果在一个程序中存在这些例程的多个版本，那么程序将不能正常工作。例如，标准 strdup() 例程，返回一个指向用 malloc 分配的字符串指针，当程序不再使用它时可以释放它。如果库使用 malloc 的某个版本来分配字符串的空间，但是应用程序使用另一个版本的 free 来释放这个字符串的空间，那么就会发生混乱。

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)
```

*Figure 9-4: The malloc hack
 picture of program, shared C library.
 malloc pointer and init code
 indirect calls from library code*



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 gggggg hhhhhh ...
```

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 动态连接和装载

\$Revision: 2.3 \$
\$Date: 1999/06/15 03:30:36 \$

动态连接使得大部分的连接过程延迟，直到程序开始运行。这种做法做提供了许多其它方法难以实现的优点：

- 动态连接库比静态连接库更容易创建。
- 动态连接库比静态连接库更容易更新。
- 动态连接库的和非共享库的语义非常的接近。**(这里语义指什么)**
- 动态连接库容许程序在运行时装载和卸载例程，这是其它方式很难提供的一个功能。

当然，它也有一些缺点。相对那些静态连接，动态连接运行时的性能耗费是很多的，因为大部分的连接过程在每次程序运行时都必须重做。程序中所有的动态连接符号都得通过一个符号表 ([symbol tables](#)) 来查找和解析。（Windows DLLs 稍微减轻了这些消耗，我们将在下面解释）由于动态库必须包含了这些符号表，因此它们比静态库要大些。

除开调用兼容性问题 (???)，**a chronic source of problems is changes in library semantics**。相对于静态连接库和非共享库而言，动态连接库是很容易更新的，所以更改程序的库就比较容易，这就意味着有可能这些程序的状态在不知情的状况下发生了改变。在微软的Windows平台下这是一个常见的问题，Windows平台使用了大量共享库，它们有很多版本，且缺乏较完善的版本控制。大多数程序附带了他们所需要的库，而安装程序经常在不经意间将老版本的共享库覆盖掉新版本，使得那些需要新版本共享库特性的程序无法运行。虽然在旧版本的库覆盖新版本库之前，一些友好的安装程序会弹出一个警告，但是，如果新版本库取代旧版本的库，依赖旧版本库的程序将无法运行。

ELF 动态连接

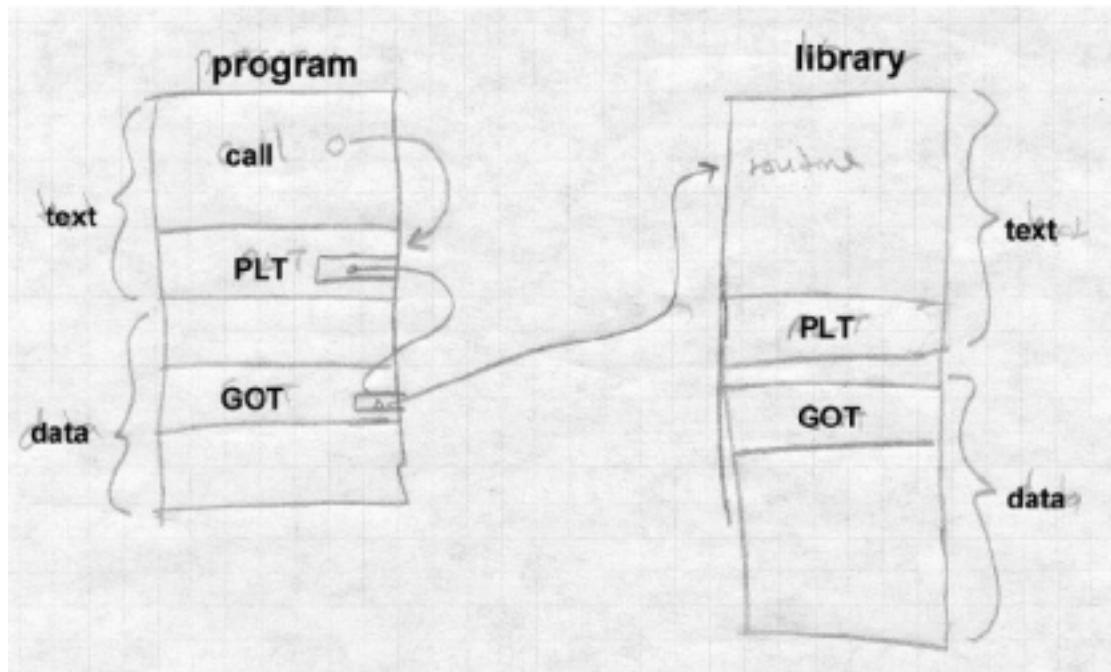
SUN公司的SunOS在20世纪80年代末将动态连接库引入到了UNIX。随后，Sun协助开发的UNIX System V Release 4中引入了ELF，并使它与Sun的系统相适应。相比过去的目标文件格式，ELF无疑是一个进步，到了20世纪90年代末，ELF成为了UNIX和类UNIX操作系统（如Linux和BSD派生系统）的标准。

ELF 文件内容

如第三章所提到的，一个ELF文件从连接器 ([linker](#)) 的角度看，是一些节 ([sections](#)) 的集合；从程序装载器 ([loader](#)) 的角度看，它是一些段 ([segments](#)) 的集合。ELF格式的程序和共享库具有相同的结构，只是段的集合和节的集合上有些不同。

ELF格式的共享库可以装载到任何地址，所以共享库使用PIC（位置无关代码），使得文件的text page（如何翻译？）不需要重定位，并且可以被多个进程共享。如第八章所说，ELF格式的连接器通过GOT（Global Offset Table）来支持PIC代码。GOT包含了动态库所有被引用的静态数据地指针，动态连接器通过GOT解析和重定位所有的指针。这些可能带来一些性能问题，但实际上，除开一些大的库，GOT通常很小；一般一个标准的C库只有350k的代码和180个GOT入口。

Figure 10-1: PLT and GOT
picture of program with PLT
picture of library with PLT and GOT



GOT和调用它的代码是在同一个ELF文件中，因此不管程序装载到地方，GOT的相对位置不会改变，所有代码可以通过相对地址来定位GOT，把GOT的地址装到寄存器，然后装载GOT的内容，它指向一个静态数据。如果一个库没有引用静态数据，可以不包含GOT，但是事实上，所有的库都包含了一个GOT。

为了支持动态连接，每个ELF共享库和每个使用共享库的可执行文件都有一个PLT（Procedure Linkage Table）。同GOT访问数据的方法类似，PLT提供了一个间接访问函数的方法。PLT支持懒惰模式（lazy evaluation），即直到函数首次被调用才解析。考虑到PLT项比GOT项要多得多（在一个通用的C库里面有近600个），而大部分的例程根本不会被调用，所以使用惰性模式可以节省启动时间和加速整个程序。

下面将讨论PLT的细节：

一个动态连接的ELF文件包含了所有的连接信息（原文说是连接器信息，按照原文的说法，上下读起来意思就无法连贯，不知道你的理解是什么样的），这些信息用来在运行时帮助连接器重定位文件和解析所有的符号。其中，.dynsym节（dynamic symbol table），包含了文件所有的导入和导出符号；.dynstr节

和 .hash 节包含了符号的名字，哈希表可以用来加速符号查找。

动态连接的 ELF 文件尾有一个特别的地方，DYNAMIC 段（或者叫 [dynamic 节](#)），动态连接器用它来查找一些必要的信息。它做为数据段装载，不过 ELF 文件头中包含了一个指向它的指针，保证运行时的动态连接器能够找到它。.dynamic 节是指针和标签的列表。有些项只出现在程序中，有些项只出现在库中，还有一些在两者中都有。

- NEEDED：必要的库文件名（一般在程序中，当一个库依赖另外一个库的时候，也可以出现在库中，出现次数可以多于一次）
- SONAME (shared object name)：连接器要用到的文件名。（在库）
- SYMTAB, STRTAB, HASH, SYMENT, STRSZ：分别指向符号表，关联字符串表，哈希表，符号表大小，字符串表大小。（程序和库）
- PLTGOT：指向 GOT，在某些平台上指向 PLT。（程序和库）
- REL, RELSZ, RELENT（也可能叫 RELA, RELASZ, RELAENT）：指向重定位入口，重定位入口的数量和大小。REL 和 RELA 的区别是 RELA 有附加物。（程序和库）
- JMPREL, PLTRELSZ, and PLTREL: pointer to, size, and format (REL or RELA) of relocation table for data referred to by the PLT. (Both.)（中间的 format，您是怎么理解的？）
- INIT 和 FINI：指向程序启动和结束的时候调用的初始化例程和结束清理例程（可选，但是通常程序和库中都有）
- 除开上面的，还有一些不常用的项就不提了

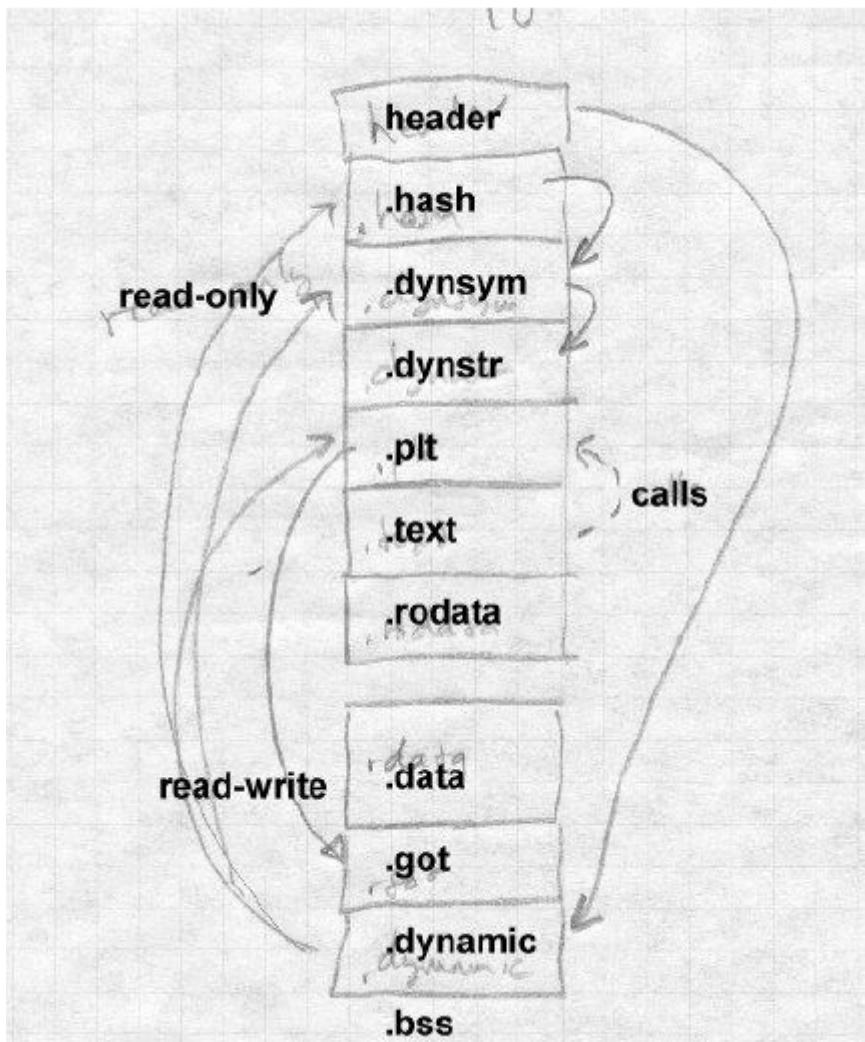
一个完整的 ELF 共享库如下图所示。首先是只读部分，包含符号表，PLT，代码，只读数据，然后是可读写部分，包含一般数据，GOT，.dynamic 节，最后一个只读节的后面一般跟随 bss 段，但是通常它不出现在文件中

Figure 10-2: An ELF shared library

(Lots of pointer arrows here)

read-only pages:

```
.hash
.dynsym
.dynstr
.plt
.text
.rodata
read-write pages:
.data
.got
.dynamic
.bss
```



装载一个动态连接程序

装载一个动态连接的ELF程序是一个漫长但是直接的过程。

启动动态连接器

当操作系统运行程序，`it maps in the file's pages as normal`（此处应该指的是将硬盘的文件镜像映射到内存，但是此处的`file's page`不好怎么翻译），注意可执行文件中的INTERPRETER节，它所指的解释程序就是动态连接器：ld.so，ld.so本身就是ELF格式的共享库。操作系统启动程序的时候，首先将动态连接器映射到一个合适的地址，然后传递一个连接器所需要的辅助向量（辅助）信息到堆栈，最后启动ld.so。

辅助向量（辅助信息）包括：

- `AT_PHDR`, `AT_PHENT` 和 `AT_PHNUM`: 程序文件头的地址，程序头中入口项的大小和数目。入口项描述了文件的段信息。如果系统还没有把程

序映射到内存，那么就会有一个AT_EXECFD入口项，它包含了程序文件打开后的文件描述符。

- AT_ENTRY: 程序开始地址，动态连接器在初始化完毕后就会跳到这个地方。
- AT_BASE: 动态连接器被装载的地址。

此时，ld.so 开始部分的自引导代码首先查找自己的GOT，GOT的第一个入口项指向ld.so文件中的dynamic段。通过dynamic段，连接器可以找到自己的重定向入口，在自己的数据段中重定向指针，解析装载其它符号的基本例程的代码位置。[\(linux ld.so 用 _dt_ 给这些基本例程取名，使用特定的代码会去寻找以 _dt_ 开始的符号，然后将它们解析\)](#)

然后，连接器用程序符号表指针和连接器符号表指针来初始化符号表链。从概念上说，程序和装载到进程中的所有库共享一个符号表，与其在运行时将所有的符号表合并，不如由连接器维护一个每个文件符号表的链表。每个文件包含一个哈希表，通过一些哈希头和每个头对应的哈希链表，连接器只要计算符号的哈希值一次，就可以迅速查到所需的符号，加速符号了查询[then running through appropriate \(这个单词好像拼写有误\) hash chain in each of the symbol tables in the list.](#)

库的查找

一旦连接器完成了自己的初始化工作，就查找程序所需要的库的名字。程序头中有一个指针指向dynamic段，它包含了动态连接的信息，dynamic段中的DT_STRTAB指向一个字符表，而其中的DT_NEEDED包含了一个相对于字符表的偏移，指向所需要的库名。[\(注：假设字符表的用数组 strtab\[N\] \[\] 来描述，那么DT_NEEDED字段就是一个大于0，小于N的数字，设为x，那么DT_NEEDED所指定的库名就是strtab\[x\]\)](#)

对于每一个库，连接器首先查找库文件位置，从本质上说是一个相当复杂的过程。DT_NEEDED所描述的库文件名一般类似libXt.so.6 (Xt开发包，版本6)，库文件可能在任意的库文件目录中，还也可能有重名的文件。在我的系统中，这个库的实际文件名是/usr/X11R6/lib/libXt.so.6.0，最后的“.0”表示次版本号。

连接器查找下列几个地方：

- 首先查看 .dynamic 段是否包含了一个叫DT_RPATH的项（它是一个以冒号分隔的库文件搜索目录列表）。这个项是在程序被连接器连接时[\(这里所说的不是动态连接器 ld.so，而是所谓的静态连接器 ld\)](#)，由命令行开关或者环境变量添加上去的。它常应用于子系统中，比如像数据库应用，我们要装载一些程序集合以及支持库到一个目录中去的时候。
- 查看是否存在环境变量 LD_LIBRARY_PATH（它是一个以冒号分隔的库文件搜索目录列表）。这个项可以帮助开发者建立一个新版本的库，把他的路径添加到LD_LIBRARY_PATH中，把它和现存的可连接程序一同使用，用来测试新的库，[or equally well to instrument the behavior of the program. \(It skips this step if the program is set-up, for security reasons.\)](#)

- 连接器查看库高速缓存文件 /etc/ld.so.conf，它包含了库名和路径的一个对应列表，如果库名存在，连接器就使用它对应的路径，用这个查找方法能够找到大部分的库（文件名不需要和要求完全符合，这点可以参考接下来的“库的版本”）。
- 如果上面的查找都失败，连接器就查找默认路径 /usr/lib，如果库文件依旧没有找到，则显示一个错误然后退出。
- 连接器找到了库文件后，先打开它，然后读取ELF头，找到指向各个段的指针 (*to find the program header which in turn points to the file's segments including the dynamic segment, 应该还有更好的翻译方式*)。连接器为库的代码段和数据段分配空间并映射到内存，随后是bss（不分配空间）。通过库的 .dynamic 段，连接器添加这个库的符号表到符号表链，如果库所依赖的其它库没有装载的话，则添加那个库到装载队列中。

完成这个过程后，所有的库都已经被映射，loader在逻辑上拥有了一个全局的符号表，它是全部程序和被映射库的符号表的联合。

共享库的初始化

现在loader再次访问每个库，处理库的重定向入口，填充库的GOT，将库的数据段进行重定位。x86下的装载时重定位类型包括：

- R_386_GLOB_DAT，用来初始化GOT项，符号地址定义在另外一个库中。
- R_386_32，非GOT项，符号地址定义在另外一个库中。*（和上面的区别是什么？）*
- R_386_RELATIVE，重定向数据，比较典型的是一个字符串的指针或者一个局部定义的静态数据。
- R_386 JMP_SLOT，用来初始化PLT的GOT项，后面有详解。

如果一个库有 .init 节，则它作为库的特定初始化例程被 loader 调用，如C++的静态构造器；同样，程序在退出的时候要运行 .fini 节的例程。*（上面的工作不是由主程序main来完成的，而是程序自启动代码来处理，在linux中这些代码在 ld.so 中的_glibc_start_main 中）*。当上面的工作完成以后，所有的库都被完全装载并准备运行，loader最终调用程序的开始地址 *(AT_ENTRY)*，程序开始执行。

使用 PLT 的惰性连接过程

使用动态库的程序一般包含了很多的函数调用，但是它们中的许多在某次运行中根本不会被调用，这些函数可能是错误处理例程或者此程序不使用的函数。此外，每个共享库也包含了其它库函数的调用，但是在某次程序的运行中要用到的就更加少了，因为程序没有直接或者间接的调用它们。

为加速程序的启动，动态连接的ELF程序使用惰性模式来绑定地址。就是说，一个函数（或者变量）在它第一次被调用的时候才确定它的地址。

ELF通过PLT（Procedure Linkage Table）来支持惰性绑定，每个动态连接的程序和共享库都有一个PLT，程序或共享库中每一个非本地例程 *(函数??)*，

在PLT包含了一个入口项，Figure 3。要注意的是，PLT里面使用的是PIC代码，其本身也是PIC代码，所以它可以成为只读代码段的一部分。

Figure 10-3: PLT structure in x86 code

```
Special first entry
PLT0:    pushl GOT+4
        jmp *GOT+8
```

Regular entries, non-PIC code:

```
PLTn:   jmp *GOT+m
        push #reloc_offset
        jmp PLT0
```

Regular entries, PIC code:

```
PLTn:   jmp *GOT+m(%ebx)
        push #reloc_offset
        jmp PLT0
```

在一个程序或共享库中对某个例程的访问都被调整成为对PLT入口的访问。程序或共享库第一次访问一个例程，PLT入口调用动态连接器解析这个例程的实际地址。此后，PLT

入口直接跳转到实际地址，所以在第一次调用完成以后，使用PLT的消耗只是一个额外的跳转指令，返回的时候并没有多余的消耗。

PLT中的第一个入口，叫做PLT0，它是一段访问动态连接器的特殊代码，在装载时，动态连接器自动在GOT里面设置两个值。在GOT+4（是GOT[1]），它放了一些特定的库标识。在GOT+8（即GOT[2]），它放置了动态连接器的符号解析例程的地址。（在整个第十章中大量出现了routine和function两个单词，严格的翻译应该是“例程”和“程序”，如此读起来不怎么方便，在一般的情况下，我认为没有必要严格区分，笼统的翻译成“程序”，不知可否？）

PLT剩下的入口项称之为PLTn，每个项以一个直接跳转开始（参见Figure 10-3, **Regular entries**）。每个PLT入口项对应一个GOT入口项，GOT项初始化为一个指针，指向PLT直接跳转指令紧接下来的PUSH指令的地址。（在PIC文件中，装载时还需要的重定位，不过减少了符号查询花费），跳转指令后面是一个PUSH指令，将一个偏移值（#reloc_offset）压入堆栈，偏移值指定文件的重定位表中，一个类型为R_386_JMP_SLOT的重定位入口项，重定位入口项的符号参考（[r_offset](#), 参考下面地注释）指向符号表的一个符号，其符号地址（[r_info](#), 参考下面的注释）指向一个GOT的入口项。

在linux中一般有个重定位信息的表，通常叫做REL_PLT，它包含了某个符号的相对地址和其它的信息，它的数据结构是

```
typedef struct {
    Elf32_Addr    r_offset;
    Elf32_Word    r_info;
} Elf32_Rel;
```

r_offset指向对应的GOT的入口项，r_info指向了符号表的一个符号。这样，我们通过#reloc_offset，也就是这个重定位入口项的地址，把它压入堆栈以后，后面的程序就可以通过它找到相应的符号信息，解析符号，然后存放到指定的GOT中去了）

这个紧凑但是比较变态的安排意味着：第一次程序或者共享库访问PLT入口项的时候，由于GOT项里面的指针指回了PLT项，在PLT的入口项的第一个跳转实际没有任何的效果。PUSH指令将一个偏移值压入堆栈，这个偏移值间接的指明了要解析的符号，和符号解析到哪个GOT项中去，然后跳转到PLT0，PLT0里面的代码首先将程序或者库的标识符压入堆栈，然后跳转到动态连接器的符号解析代码部分，这样在栈顶就压入了两个值。（注意，使用的是跳转而不是调用，压入的两个值的上面是调用PLT代码的返回地址）

现在（*stub code*）保存所有的寄存器，调用动态连接器的一个内部例程来解析符号。堆栈中的两个标识符足够找到库的符号表和在符号表的入口。动态连接器使用符号表链查找符号值，找到后，将例程的地址保存在指定的GOT项中。然后（*stub code*）恢复寄存器，弹出PLT压入的两个值，退出此段代码。GOT项到此时已经被更新，在接下来的调用里面，PLT直接跳转到例程，而不需要动态连接。

动态连接的其它特殊点

为了保持运行时的语义尽可能的合那些非共享库相似，ELF连接器和动态连接器有许多隐藏的代码处理各种特殊情况，

静态初始化

如果一个程序引用了一个定义在共享库的全局变量，由于程序的数据地址必须在连接时确定，连接器必须为之创建一个变量的拷贝，如Figure4。由于动态连接器可以通过一个GOT指针来修正这些地址，一般不会出现什么问题。但是，如果共享库初始化了这个变量，那么程序中将无法知道。为了解决这个问题，连接器在程序的重定位表中设置一个类型为 R_386_COPY(否则只包含 R_386_JMP_SLOT, R_386_GLOB_DAT, R_386_32, 和R_386_RELATIVE)的入口项，指向程序中变量拷贝的地址，然后告诉动态连接器从共享库重将变量的初始化值拷贝过来。

Figure 10-4: Global data initialization

Main program:

```
extern int token;
```

Routine in shared library:

```
int token = 42;
```

虽然这个特性在某些类型的代码中是很基本的，但是在实际中出现得确非

常少。由于只作用于单个数据，所以是个权宜之计。但是被初始化的通常是函数或数据的指针，所以这种折衷就足够了。

库版本

动态库的名字有主版本号和次版本号组成，如 libc.so.1.1 但是程序只识别到主版本号，像 libc.so.1。次版本号只是为了支持兼容。

为了让程序迅速的加载，系统维护了一个库缓存文件，包含了每个库的最新版本的全路径名，当一个新的库安装的时候，由配置程序负责来更新。

为了支持这种设计，每个动态连接库有一个“真名”叫SONAME，在库创建时分配。举个例子：一个库叫libc.so.1.1它的“真名”叫做libc.so.1 (SONAME通常默认为库名)。当连接器创建一个使用共享库程序的时候，它列出使用的库的SONAME，而非全称。库缓存创建程序扫描包含共享库的所有路径，找出这个共享库，解出SONAME，看是否有多个版本，选择最高版本，然后将SONAME和最高版本的全路径写入缓存文件。这样就能够保证在运行时，动态连接器能够迅速的找到每个库的当前版本。

运行时的动态连接

尽管ELF动态连接器在程序运行时经常隐式的调用PLT入口，程序也可以显式的使用dlopen()来装载一个共享库，用dlsym()来查找一个符号(通常是一个函数)的地址。这两个例程在动态连接器被简单的包装成回调函数。当动态连接器通过dlopen()来装载一个库的时候，像它在其它库上的操作一样，做同样的重定位和符号解析。所以，所有动态装载程序，可以在不借助任何特殊的回调函数的情况下，在程序运行时装载和引用全局数据。
(so the dynamically loaded program can without any special arrangements call back to routines already loaded and refer to global data in the running program.
最后这一句看能不能翻译得更好一些，我的翻译有些问题的：)

上面这些特性容许用户不要修改程序的原代码，就可以为程序添加额外的功能，甚至可以不要停止和重启程序(这种程序在数据库和web服务器中有用)。早在20世纪60年代初，大型机的操作系统提供了一个“exit routines”实现类似的功能，虽然没有提供像现在这么方便的接口，但是很长一段时间给打包的应用程序带来了相当大的弹性。它提供给程序扩充自己的途径，用C或C++来写一段新的程序，运行编译器和连接器创建一个共享库，然后动态的装载并运行新的代码。
(Mainframe sort programs have linked and loaded custom inner loop code for each sort job for decades.)

Microsoft 动态连接库

Microsoft windows 提供了叫做动态连接库或者DLLs的共享库。它们和ELF共享库非常的相似，只不过要简单一些罢了。windows3.1的16位动态连接库和windows NT/95下的32位动态连接库有本质上的变化。在这里我们仅仅讨论新的w

in32库。DLLs的导入过程类似PLT。尽管从设计上说，可以使用类似GOT的方法来导入数据，实际上采用了一个更加简单的方案，使用显式的代码（`to dereference imported pointers, dereference` 如果用本义，有点说不通）来共享数据。

在Windows中，所有的程序和DLLs都是PE格式(portable executable)文件，这些文件作为虚拟内存被映射到进程空间。和windows3.1中所有应用程序共享一个地址空间不同，win32为每一个应用程序分配一个地址空间，然后将可执行程序和库映射到各自使用的地址空间上。对于只读代码段，这两种方法并没有什么区别，但对于DLLs中的数据，系统将为程序分配一个该数据的拷贝。（这种方法有些过于简单，虽然PE文件也可以指定某个节为共享数据段，将单个数据共享给所有使用该库的程序，但是，大多数情况下并不这样）

在windows中，动态连接器虽然是内核的一部分（Linux中，动态连接器是一个应用程序），但是其装载可执行程序和DLLs的过程，同装载一个动态连接的ELF程序是相似的。首先，内核通过PE头中的节信息映射可执行文件，然后同样使用DLL文件的PE头信息来映射动态连接库。

PE文件也可以包含重定位信息。不过一个可执行文件一般不包含重定位信息，它的映射地址在连接时就已经确定。而所有的DLLs都包含重定位入口，如果DLLs在连接时确定的映射地址不可用的话，将使用重定位项来重定位自己。（Microsoft 称运行时的重定位为 *rebasing*）

可执行文件和DLLs两种PE文件都有一个入口点。当DLLs被装载或卸载时，以及进程每次绑定和卸载DLLs时，装载器都调用DLLs的入口点（装载器会传递一个参数来说明调用原因）。这种方式和ELF中的.init和.fini节功能类似，使得程序员能够在程序初始化之前以及退出之前做一些事情。（This provides a hook for static initializers and destructors analogous to the ELF .init and .fini sections. 上面的句子完全采取意译）

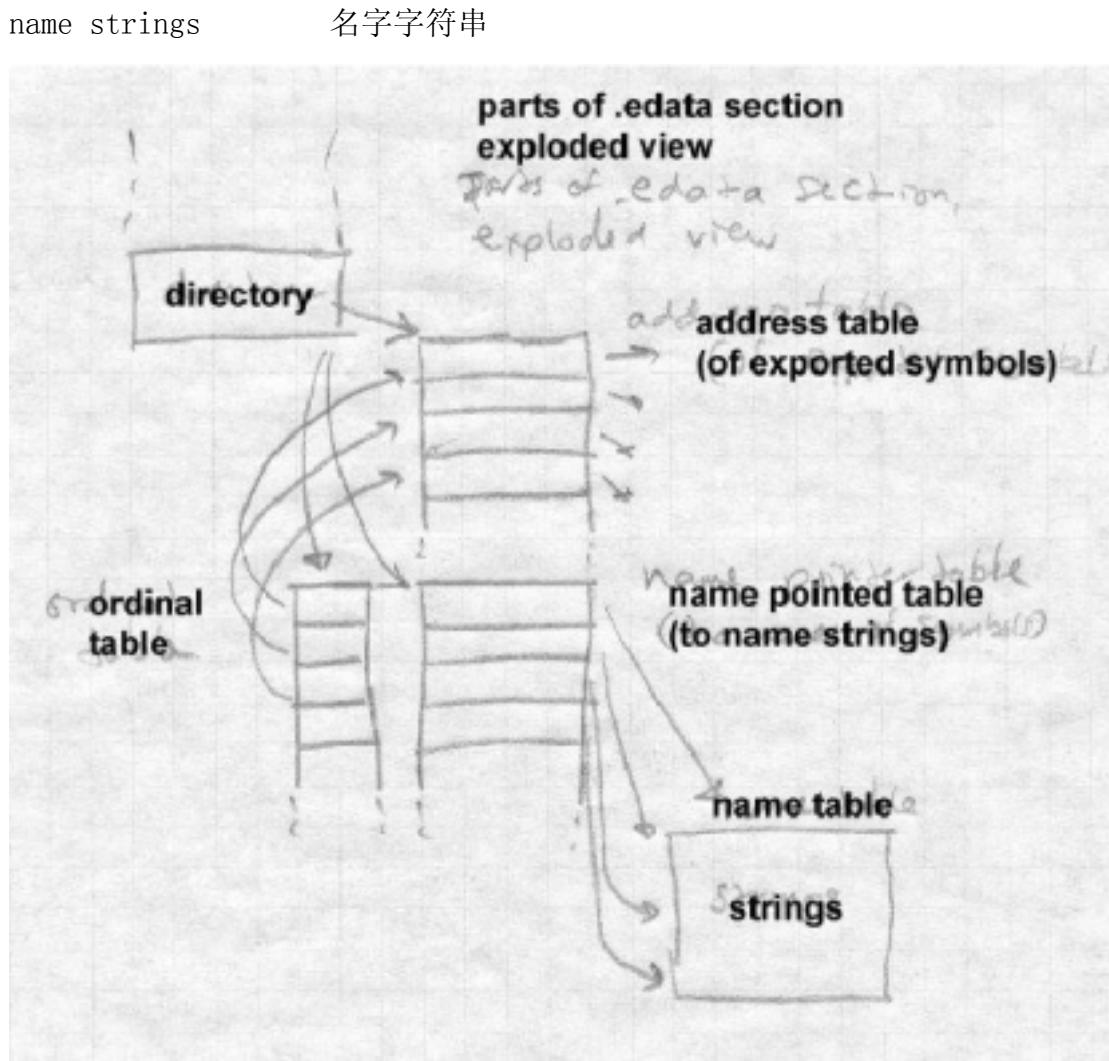
PE 文件中的输入和输出符号

PE通过文件中两个特殊的段来支持共享库，.edata（用来输出符号）列出一个文件的输出符号；.idata（用来输入符号）列出一个文件的输入符号。程序文件通常只有.idata段，而DLLs总是具有.edata段，如果DLLs使用了其它的DLLs，也有可能有.idata。一个符号可以通过符号名或者序数（指出符号在符号地址表中位置的小整数）导出。通过序数查找符号可以提高连接效率，但是，开发者在用序数创建DLLs时，很难保证各个版本的序数一致，犯错的可能性较大。所以在实际中，只有在更改很少的系统服务时才使用序数，其它情况都使用符号名。

.edata段包含了一个导出目录表，描述其它的节，紧接着就是符号导出表。

图10-5：.edata 节结构

export directory pointing to:	
export address table	导出地址表
ordinal table	序数表
name pointer table	名字指针表



导出地址表包含符号的RVA(relative virtual address, 相对虚拟地址, 相对于PE文件加载地址的偏移)。如果RVA指回到.edata节, 它就是一个“forwarder”引用, and the value pointed to is a string naming the symbol to use to satisfy the reference, probably defined in a different DLL. (这一句如何理解?)序数表和名字指针表是并行的, 名字指针表的每一个项是符号名的RVA, 序数表的项值是符号在地址导出表的编号(序数不需要以0开始, 将序数减去序数的基准值(这个值通常为1)就是导出符号表的编号)。导出符号不一定需要名字。不过它们通常都有名字。名字指针表中的符号以字母顺序排列, 以便装载器使用二分法查找。

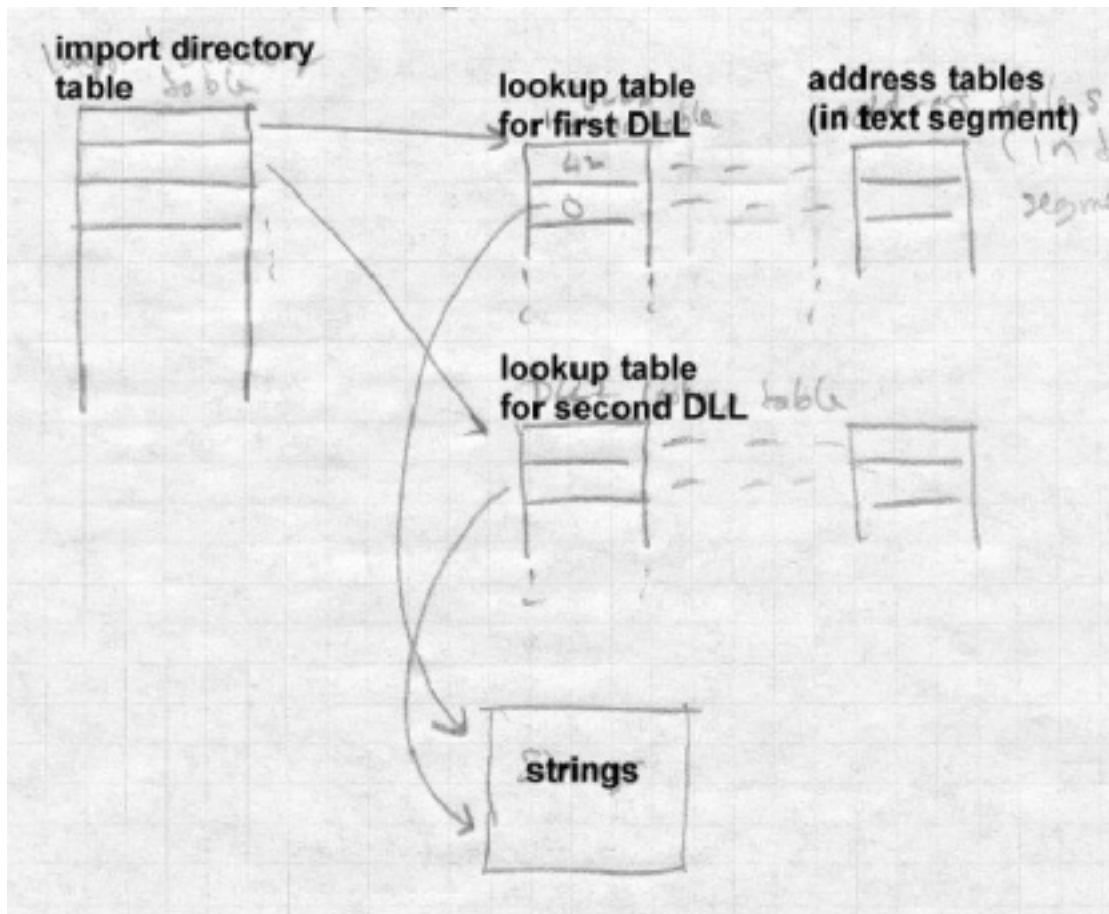
.idata节的用途和.edata节相反, 它们将符号或者序数映射回一个虚拟地址。.idata节若干0结尾的数组构成, 包括导入目录表, 每个DLL的导入查找表, 最后是名字表。

图 10-6: .idata节的结构

import directory tables, with lotsa arrows
each has import lookup table RVA, time/date stamp, forwarder chain (unused?), DLL name, import address RVA table

NULL

import table, entries with high bit flag (table per DLL)
hint/name table



在程序的代码段中有一个数组维护导入DLL的地址，程序装载器在装载时要解析这个地址。导入查找表标记了要导入的符号，它的入口项和导入地址表的项是相关联的。每个查找表项由32比特构成。如果最高位为1，则低31位是一个符号的序数，否则低31位是hint/name table中的RVA地址。每个hint/name项有4个字节的hint，猜测符号在DLL导出名字指针表的编号，紧接着是以NULL结尾的符号名。程序装载器通过hint来查找导出符号表，如果符号名匹配，就使用这个符号，否对整个导出符号表做二分查找。（如果DLL没有改变过，或者至少导出符号表没有发生过改变，只要DLL被连接了，利用hint来猜测就可以找到符号）

和ELF导入符号不同，通过 .idata 导入的符号地址只放置在导入符号表中，且导入文件的任意地方都不会被修正。对于代码地址，稍微有些不同，当连接器创建一个执行体或者 DLLs 的时候，要在代码段建立一个无名字的“thunks”表，通过它间接跳转到导入符号表，将“thunks”作为导入例程的地址，这些对程序员来说是透明的。*(The thunks as well as most of the data in the .idata section actually come from a stub library created at the same time as the DLL.)* 在微软最新版本的c/c++编译器中，如果程序员知道要调用DLL中一个例程，可以将这个例程申明为“d11import”，编译器将产生一个对地址表相应项的间接跳转指令，以避免额外的间接跳转。对于数据地址，这样会有些问

题, since it's harder to hide the extra level of indirection required to address a symbol in another executable. Traditionally, programmers just bit the bullet and explicitly declared imported variables to be pointers to the real values and explicitly dereferenced the pointers. (我对这一段文字理解还不够) 在最新版本的微软c 和 c++编译器中, 容许程序员声明全局变量为“`dllimport`”, 编译器将使用额外的指针引用, 和ELF代码中通过GOT中指针来引用数据非常的相似。

惰性绑定

新版本的windows编译器添加了延时装载的功能, 容许进程惰性绑定符号, 有点像ELF的PLT, 延时装载的DLL和.idata导出目录表有类似的结构, 但是由于不在.idata节, 所以装载器不会自动处理它。其导出目录表的每个项都被初始化为同一个例程的指针, 这个例程用来查找和装载DLLs, 替换每个项的内容为实际地址。延时装载的目录表有一处存放导入表的原始内容, 使得DLL即使随后卸载, 可以将其还原。Microsoft 提供了标准的例程, 但是其接口是公开的, 如果有需要的话程序员也可以自己实现一个版本。

Windows容许程序使用LoadLibrary和FreeLibrary显式的装载和卸载DLLs, 使用GetProcAddress来查找符号地址。

DLLs 和线程

如果DLL使用TLS ([thread local storage-线程本地存储](#)), 在这种状况下运转将不会很好。一个Windows程序在一个进程里可以启用多个线程, 它们共享进程的地址空间。每个线程有一块TLS来存放线程私有数据, 如线程使用的资源和数据结构的指针等。TLS需要为可执行程序和每个使用TLS的DLL的数据准备slot ([见小字注释](#))。Windows的连接器能够在PE可执行文件中创建一个.tls节, 定义了可执行程序和任意直接被引用的DLLs所需要的TLS中数据存放次序。每当进程创建线程的时候, 新的线程以.tls节为摸板创建自己的TLS。

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。

问题是大多数的DLLs既可以隐式的被执行体连接, 也可以通过LoadLibrary 显式的装载。由于无法预测一个DLLs将被显式还是隐式的调用, 所以DLLs不能去依赖.tls节, 显式连接的DLLs不会自动获取TLS.

幸运的是, Windows为TLS定义了运行时分配slots的API。写DLL最好使用这些API, 而不是.tls节, 除非你知道DLLs是隐式的调用。

OSF/1 准共享库

OSF/1 是来自与 OSF ([Open Software Foundation](#))的一个 UNIX 变种, 它使用了一种介于动态连接和静态连接之间的共享库方案。它的开发者说, 由于静态连接只需要很少的重定位, 所以它比动态连接的速度要快; 同时, 由于库的更新并不频繁, 因此系统管理员即算在

更新库的时候要重连接系统中所有的程序，他们也愿意忍受。

因此 OSF/1 使用一个对所有进程可见的全局符号表的方法，将在系统启动的时候将所有的共享库装载到一个共享的地址空间。这就表示所有的库分配了地址后，在系统的运行时也不会发生改变。每当一个程序启动的时候，如果它使用了共享库，它就通过全局符号表映射共享库和符号以及解析未定义的引用。由于程序都是联接存在的某地址空间，重定位都是在启动时完成，所以运行时不要消耗装载时间。

当一个共享库改变，系统就得重启，然后装载新的库，并创建一个新的全局符号表。

这个方法很巧妙，但是确不是令人很满意。其一，程序查找符号的时间比重定位时间还要长，所以节省重定位时间不一定能够提高性能。其二，OSF/1 不能够提供运行时装载和运行库的功能。

让共享库加速

共享库，特别是ELF共享库，可能会非常的慢。原因来自多个方面，有几个在第八章提到过：

- 装载时库重定位
- 装载时符号解析
- PIC的prolog代码开销
- PIC的间接数据引用开销
- PIC的保留地址寄存器

前两个问题可以通过缓存来改善，后两个可以通过将纯净的PIC代码退化来改善。

近代的计算机上有大量的地址空间，为某个共享库选择一个地址范围，对绝大部分使用该共享库的进程来说是可行的。Windows采取的一个比较有效率的方法是，无论库被连接还是第一次被装载，将它暂时绑定到一个地址上。然后，每次程序连接库的时候，尽可能的使用同样的地址，这就意味着不需要重定位。如果在某个进程中此地址不可用，则库要如前那样重定位。

SGI系统使用术语 (QUICKSTART) 来描述进程在连接时的预重定位，(or in a separate pass over the shared library)。BeOS在库第一次装载到进程的时候将它缓存。即使多个库之间互相依赖，理论上是可能在多个库之中预重定位和预解析符号的，虽然我并没发现有这样的连接器。

如果一个系统使用预重定位的库，PIC就显得不那么重要了。所有的进程装载库到预重定位地址，不管库是不是PIC，其代码都可以被共享。所以在一个合适地址的非PIC库实际上也可以被共享，且不会像PIC那样损失性能。这些基于第九章所说的静态连接库，即算地址冲突，动态连接器将库移动到其它的地址，损失一些性能，也好过连接失败，Windows就采用这种方式。

BeOS将需重定位的库彻底缓存起来，同时在库变化时还能保持正确的语义。当BeOS发现一个新版本的库安装了的话，它就在程序员引用这个库的时候为之创建一个新版本的缓存。库的改变具有连锁反应，如果库A库B的符号，库B更新的时候，库A也会创建一个新的缓存。这些使得程序员的工作变得简单。但是我不明白的是，(libraries are in practice updated often enough to merit the considerable amount of system code needed to track library updates.)

几种动态连接方法的比较

Unix/ELF 和 Windows/PE 的动态连接在几个方面都不同。

ELF对每个程序使用一个名字空间，而 PE对每个库都有一个名字空间。一个ELF程序知道它所需要的符号和库，但是并没有记录符号和库的对应关系。另一个方面，PE知道某个符号来自与某个库，这使得PE缺乏灵活，但是对不经意的欺骗有更大的抵抗力。想象一个可执行文件调用一个例程AFUNC和BFUNC，AFUNC在库A中，BFUNC在库B中，如果一个新版本的A库恰好定义了BFUNC，则ELF程序会优先选择新的BFUNC，而PE程序不会这样。当有大量的库的时候，对于ELF来说是个麻烦；一个局部解决问题的方法是，使用DT_FILTER和DT_AUXILIARY告诉动态连接器这个符号来自哪个库，让连接器先搜索这些库，然后才搜索可执行体和其它的库。DT_SYMBOLIC域告诉动态连接器首先寻找库本身的符号表，因此其它的库就不会覆盖库内部符号引用。[（这些并不是总是需要的，考虑一下前面章节所描述的malloc hack）](#) 这些特别的方法可以尽量避免相关库的符号之间的覆盖，不过还是缺乏像第11章所看到的java那样的层次结构的名字空间的替代品。

维护静态连接程序的语义，EFL比PE要困难得多。在一个LEF程序中，对外部库中数据的引用是自动解析的，而PE将它作为导入数据而特别处理。PE在比较一个指针和函数有些困难，因为一个导入函数的地址是“thunk”，而不是在库中的实际地址。而ELF文件将所有的指针同样对待。

在运行时，几乎所有的Windows动态连接器都是在内核中，而EFL的动态连接器是作为应用程序的一部分，内核只是将它映射到初始化文件中。Windows的方法显然要快，因为它不用每次在进程开始连接之前，将动态连接器映射和重定位。ELF的定义则更加灵活。因为每个可执行体使用叫做“interpreter”的程序来做连接器(现在一般的连接器叫做 ld.so)不同的可执行程序可以使用不同的翻译程序而不需要操作系统做改变。这样实际上就使得支持各种各样版本的Unix变得非常的容易，比如Linux and BSD，可以通过制造一个兼容库的动态连接器来支持非本地的可执行文件。

练习

在ELF共享库中，库中的函数调用一般是通过PLT来实现的，而函数地址是在运行时才绑定的，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plugh()，然后程序员创建了一个使用该库的动态连接程序，后来系统管理员注意到plugh()是一个笨名字，装了一个新版本的库，这个例程在此库中叫做xsazq，当下次程序员运行这个程序时，将发生什么。

如果一个运行时的环境变量LD_BIND_NOW被设置，EFL动态连接器在装载时将程序所有的PLT入口绑定。在前一个问题中，如果LD_BIND_NOW被设置，结果将会如何？

Microsoft 靠在连接器中一些额外的技巧，和操作系统现存的一些机制，在没有内核的支持下实现了惰性绑定。那么要提供对共享数据透明的访问，避免在现在方案中使用多余的指针，有多困难？

工程

创建一个完整的动态连接器是不切实际的，因为大部分的工作动态连接在运行时，而不是连接时。在8-3我们创建一个PIC可执行体实际上已经为创建一个共享库做了大量的工作。一个动态连接的共享库就是一个PIC可执行体和定义好的导入和导出符号清单，以及它所依赖的库。将这样一个文件创建成为一个共享库或者使用共享库的可执行文件：

```
LINKLIB lib1 lib2 ...
```

```
LINK lib1 lib2 ...
```

其中lib1 lib2是这个动态连接的共享库所依赖的库

Project 10-1：扩展project 8-3版本的连接器的功能，让它处理共享库和需要共享库的可执行体。连接器将一系列的文件以及共享库作为输入，将它们合并成为一个可执行体或者库。输出文件包含一个包含已定义符号（导出）和未定义符号（导入）的符号表。*Relocation types are the ones for PIC files along with AS4 and RS4 for references to imported symbols.*

Project 10-2：写一个运行时的binder，以一个使用动态连接库的可执行体为输入，解析其所有的引用。它首先应该读取可执行体，然后读取必要的库，将它们重定位到合适的非重叠地址，随后创建一个逻辑的符号表（也可以真正的创建这个表，使用链表或者如ELF的方式来做），最后解析所有的内部和外部引用。完成这些后，所有的代码和数据都应该被安置到内存空间，而且它们的地址都应该被解析和重定位到指定的地址。

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(dumb 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。

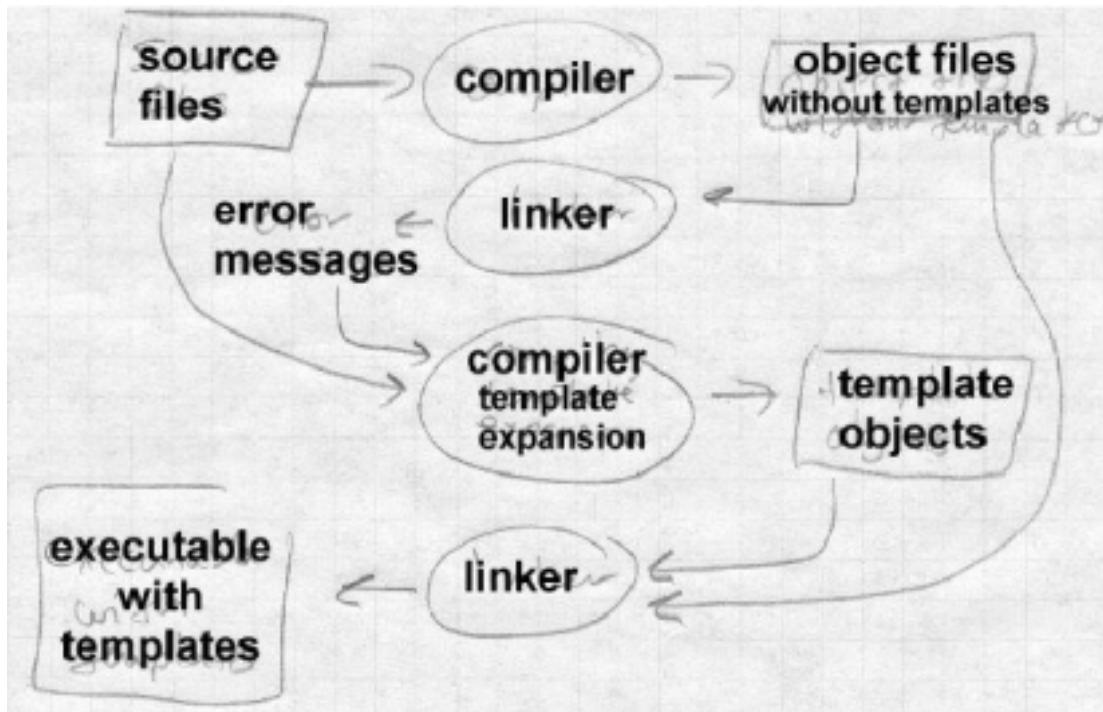
试验链接

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

Figure 11-1: Trial linking

input files pass through linker to trial output plus errors, then inputs plus info from errors plus maybe more generated objects pass through linker to final object



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_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。

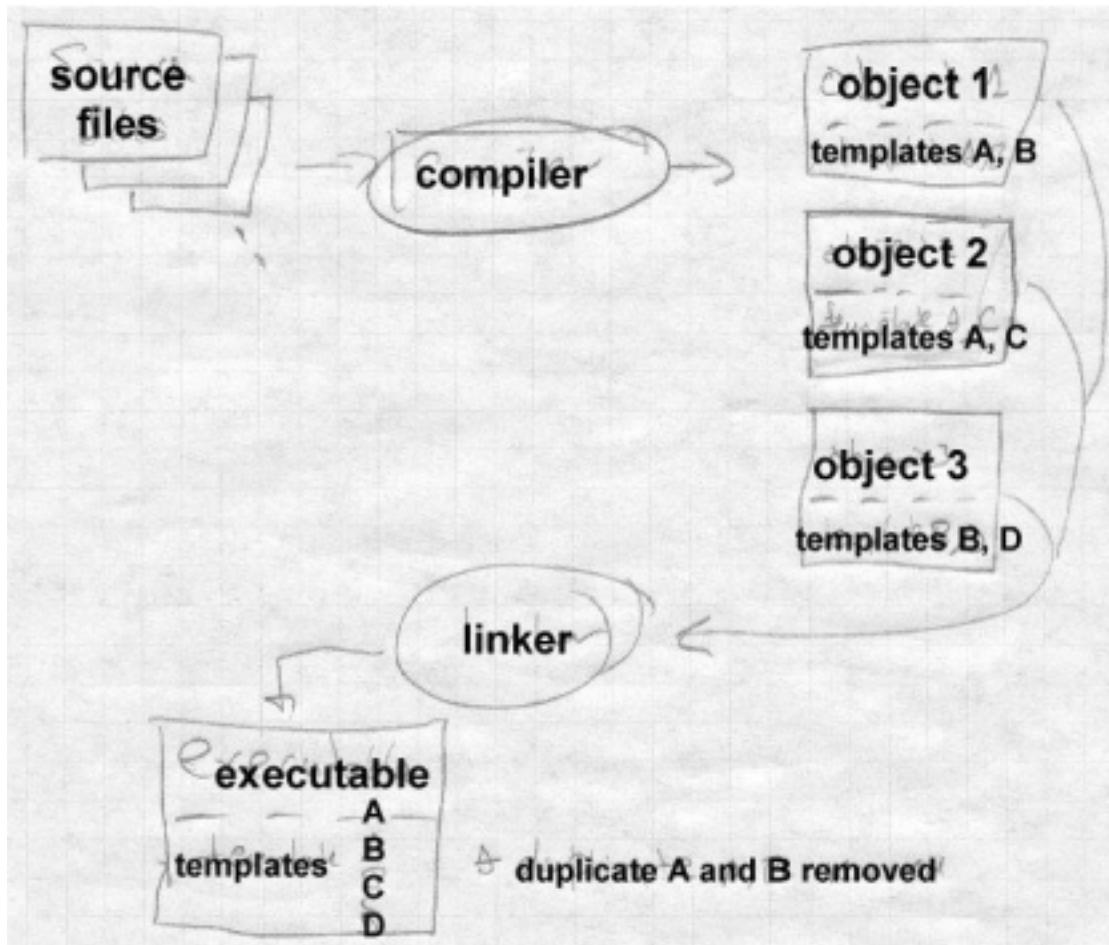
这里会有一个小问题是为模板寻找对应的源代码，因为所要找寻的目标可能潜伏在非常大量的源代码文件中。C前端程序使用了一种简单而特别的技术：扫描头文件，然后猜测一个在foo.h中声明的模板会定义在foo.cc中。新近版本的GCC会使用一种在编译过程中生成，以注明模板定义代码的位置的小文件，称之为“仓库”（repository）。在试验链接后，编译器驱动程序仅需要扫描这些小文件就可以找到模板对应的源代码。

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在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据的模板，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

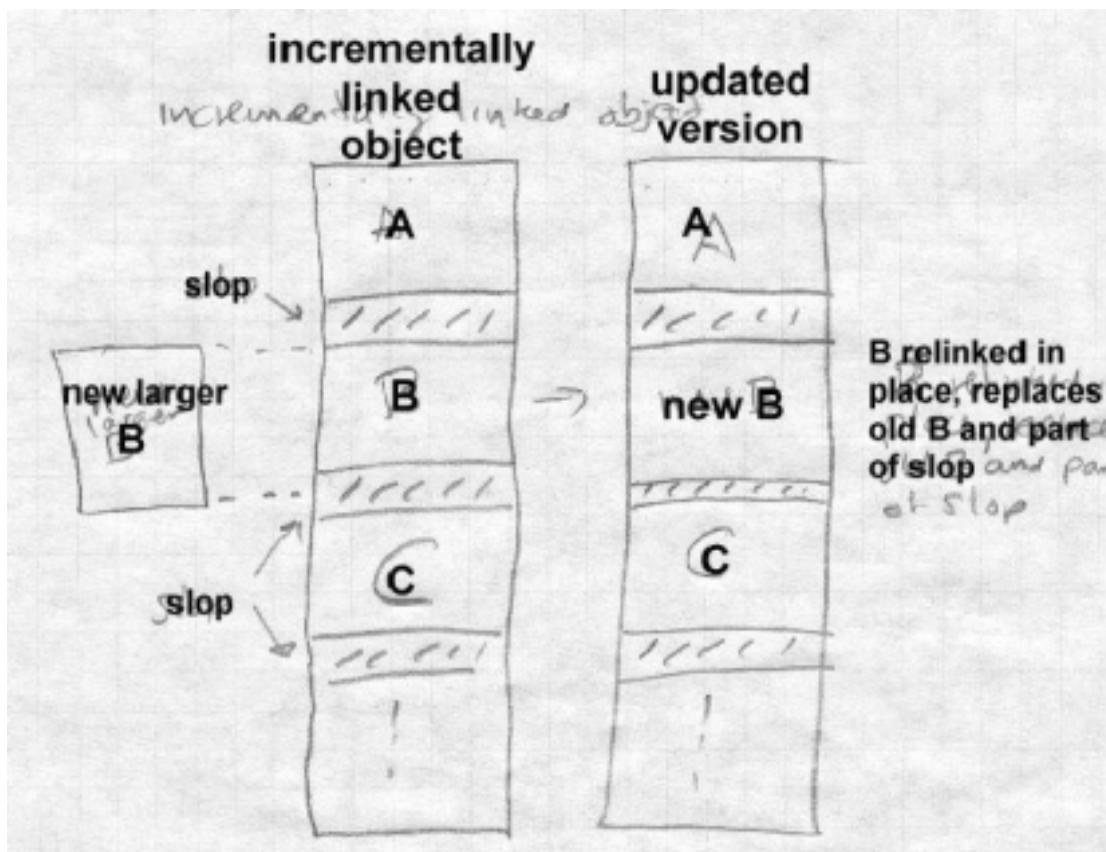
GCC所用的“仓库”实际上就是一个小的数据库。最终，工具开发者都会转而使用数据库来存储源代码和目标代码，就像IBM的Visual Age C++的Montana开发环境一样。数据库跟踪每一个声明和定义的位置，这样就可以在源代码改变后精确的指出哪些例程会对此修改具有依赖关系，并仅仅重新编译和链接那些修改了的地方。

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处来的空间时，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。

Figure 11-3: Incremental linking
picture of inclink-ed object file with slop between segments, a
nd new version's segments pointing to replace old ones



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的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。原因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。

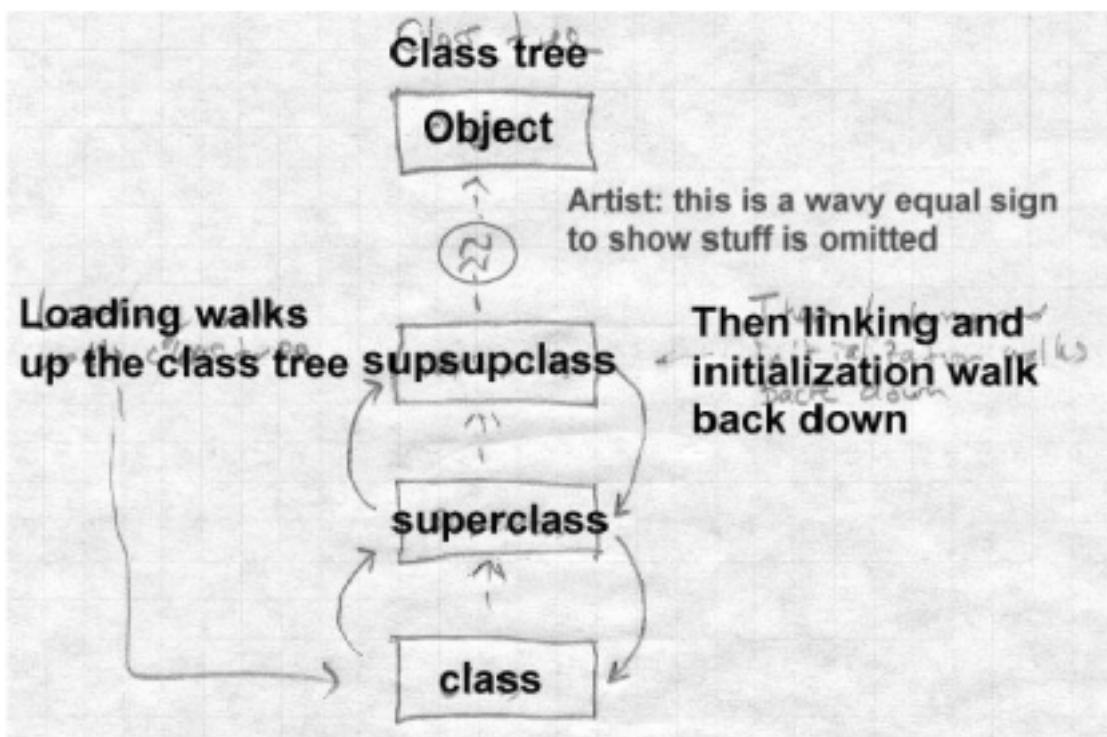
Java每次加载一个类。一个Java程序最开始会以一种依赖于具体实现的方法加载一个最初的基类。如果这个类引用了别的类，则那些类也会在需要的时候被加载。一个Java应用程序可以使用内建的自举类加载器从磁盘上的文件中加载类，也可以使用自己的类加载器以任何自己想采用的方法来创建和恢复类。多数情况下一个定制的类加载器通过网络链接获得类文件，同样也可以在运行时生成代码，以及从压缩或加密的文件中提取代码。当一个类由于其它类的引用而被加载时，系统会使用与加载引用它的类相同的加载器。每一个类加载器各自具有独立的名字空间，因此即使一个从磁盘上运行的应用程序和另外一个从网络上运行的应用程序具有相同的类或类成员名称，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。

Figure 11-4: Loading and linking a Java class file
crawling up and down the tree



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。