[第7章 链接](#3948-1578662021442)

[7.1 编译器驱动程序](#8926-1578662021442)

[7.2 静态链接](#6439-1578662021442)

[7.3 目标文件](#4565-1578662021442)

[7.4 可重定位目标文件](#4993-1578662021442)

[7.5 符号和符号表](#2082-1578662021442)

[7.6 符号解析](#8335-1578662021442)

[7.6.1 链接器如何解析多重定义的全局符号](#4464-1578728385835)

[7.6.2 与静态库链接](#5296-1578734719462)

[7.6.3 链接器如何解析引用](#5775-1578735274060)

[7.7 重定位](#7497-1578662021442)

[7.7.1 重定位条目](#6079-1578662021442)

[7.7.2 重定位符号引用](#4585-1578750775173)

[7.8 可执行目标文件](#4320-1578662021442)

[7.9 加载可执行目标文件](#7197-1578662021442)

[7.10 动态链接共享库](#8230-1605790308777)

[7.11 从应用程序中加载和链接共享库](#8850-1615469247908)

[7.12 位置无关代码](#5783-1578662021442)

[7.13 库打桩机制](#6361-1578662021442)

[7.14 处理目标文件的工具](#7733-1578662021442)

[7.15 小结](#4764-1578662021442)

**本章学习期望**

链接器如何解析引用？什么是库？如何使用库来解析引用？

全局和局部变量之间的区别是什么？当定义一个 static 的变量或函数时，到底意味着什么？

理解概念：加载和运行程序、虚拟内存、分页、内存映射

掌握利用共享库。

**第7章 链接**

链接在以下**三个阶段**都可以执行：

**编译时**，即在源代码被翻译成机器代码时

**加载时**，即程序被加载器加载到内存并执行时

**运行时**，即由应用程序来执行

现代系统中，链接是由**链接器**自动执行的。

链接器使**分离编译**成为可能。

**7.1 编译器驱动程序**

**编译器驱动程序**可以使用户根据需要调用**语言预处理器**、**编译器**、**汇编器**和**链接器**。

通过**静态链接**，链接器将多个**可重定位目标文件**组合形成一个**可执行目标文件**。

**7.2 静态链接**

**静态链接器**以**一组可重定位目标文件**和**命令行参数**作为输入，生成一个**完全链接**的、**可以加载和运行**的**可执行目标文件**作为输出。

在构造可执行文件的过程中，链接器主要完成两个任务：

**符号解析**

**什么是符号：**目标文件**定义**和**引用**符号，每个符号对应着一个**函数**、**全局变量**或**静态变量**

**符号解析的目的**：将每个**符号引用**和一个**符号定义关联起来**

**重定位**

由编译器和汇编器生成的**可重定位目标文件**中的代码和数据节是从 0 开始的。可重定位目标文件中还包含重定位条目。

**如何实现重定位**：链接器通过把每个**符号定义**和一个**内存位置（运行时地址）关联起来**以实现重定位。然后修改所有对这些符号的**引用**，使它们指向这个内存位置。

**7.3 目标文件**

一个目标文件又称**目标模块**。目标文件纯粹是**字节块的集合**。目标文件本身是一个字节序列。

这些字节块中有些包含**程序代码**或**程序数据**，其他的则包含**引导链接器和加载器的数据结构**。

**链接器**把这些块连接起来，确定被连接块的运行时位置，并修改代码和数据块中的各种位置。

**目标文件有三种形式：**

**可重定位目标文件**：包含二进制的**代码**和**数据**。可以与其他可重定位目标文件合并成可执行目标文件。又称 obj 文件，gcc 经过预处理、编译、汇编后生成的 .o 文件即为可重定位目标文件。

**可执行目标文件：**包含二进制的代码和数据。可以被直接复制到内存并执行。简称可执行文件，gcc 经过链接后生成的 .out 文件以及无后缀名文件都是可执行文件。

在 linux 中，.out 文件和无后缀名文件基本意义一样，只是命名习惯的不一致而已，即 main.out 和 main 两个文件是一样的。

**共享目标文件**：特殊类型的可重定位目标文件，即**动态链接库**。可以在**加载**或者**运行时**被动态地加载进内存并链接。

**静态链接库属于哪种呢？**

编译器和汇编器生成可重定位目标文件和**共享目标文件**，链接器生成可执行目标文件。

目标文件是按照特定的目标文件格式来组织的，各个系统的目标文件格式都不相同。Windows 使用 **PE** 格式，Linux 使用 **ELF** 格式。

**7.4 可重定位目标文件**

本书以 Linux 中的 **ELF 可重定位目标文件**为例。

可重定位目标文件由**多个不同的节**组成，每一节都是一个连续的字节序列。指令、初始化了的全局变量、未初始化的的变量分别位于不同的节。

**一个 ELF 可重定位文件中包含以下节（按位置顺序排列）：**

**ELF 头**：特殊的节，包含文件的一些基本属性信息，用来**解释目标文件**和**帮助链接器进行语法分析**。

**包含内容**：生成该文件的系统的字的大小和字节顺序，ELF 头的大小，目标文件的类型，机器类型(如 x86-64)，节头部表的文件偏移，节头部表中条目的大小和数量。

**.text：**包含已编译程序的机器代码。即存放的是指令代码。

**.rodata：**包含一些特殊的只读数据。

**.data：**包含已初始化的全局和静态变量。

**.bbs：**包含未初始化的全局和静态变量，以及所有被初始化为 0 的全局或静态变量。

**注意：**.bss 节在目标文件中仅是一个占位符，不占据实际空间。这两类变量都是运行时在内存中为其分配变量，并初始化为 0

**.symtab：**包含一个**符号表**。存放了在程序中**定义**和**引用**的**符号 (即函数和全局变量)** 的信息。

**注意：**与可编译器中的符号表不同，.symtab 中的符号表不包含局部变量的条目。

**.rel.text：**包含一个 .text 节中位置的列表，当链接器把此目标文件与其他文件组合时，需要修改这些位置。

一般**任何调用外部函数或引用全局变量的指令都需要修改**，而调用本地函数的指令则不需要修改。**为什么不需要修改呢？**

**注意：**可执行目标文件不需要重定位，一般不包含 .rel.text 和 .rel.data 节。

**理解：**.rel.text 中包含的实际上是**代码的重定位条目**。

**.rel.data：**包含被模块引用或定义的所有全局变量的重定位信息。

如果一个已初始化的全局变量其初始值是一个全局变量地址或外部定义函数的地址，就需要被修改。

**理解**：.rel.data 中包含的实际上是**已初始化的数据的重定位条目**。

**.debug**：一个**调试符号表**，内部包含的条目是程序中定义的局部变量和类型定义，程序中定义和引用的全局变量，还有原始的 C 源文件。

**注意：**.debug 节并不总是存在，只有用 -g 选项来调用编译器驱动程序时，才会有这一节。

**.line：**包含原始 C 源程序中的**行号**和 .text 节中**机器指令**之间的**映射**。

**注意：**.line 节和 .debug 节一样，并不总是存在，只有用 -g 选项来调用编译器驱动程序时，才会有这一节。

**.strtab：**包含一个**字符串表**，其中包括 .symtab 和 .debug 节中的符号表，以及节头部中的节名字。

字符串表实际上就是一个以 null 结尾的**字符串的序列**。

**节头部表：**特殊的节，是一个用来描述目标文件的节。

内容：含有与目标文件中每个节相对应的一个**条目**，描述了对应节的位置和大小等信息。

注意局部变量在**运行时保存在栈中**，既不出现在 .data 节中，也不出现在 .bss 节中。

**伪节**

有三个特殊的**伪节**，它们在节头部表中是没有条目的：

**ABS：**代表不该被重定位的符号

**UNDEF：**代表未定义的符号，即在本目标模块中引用，但在其他地方定义的符号

**COMMON：**表示还未被分配位置的未初始化的数据目标。

这些伪节只有可重定位文件中才有，可执行文件中没有。

**COMMON 和 .bss 的区别很细微：**

**COMMON：**未初始化的全局变量

**.bss：**未初始化的静态变量，初始化为 0 的全局或静态变量

原因：未初始化的全局变量是全局符号中的弱符号，编译器将其分配为 COMMON 以表明是弱符号。

**7.5 符号和符号表**

**重定位的核心就是对符号表进行符号解析**

每个可重定位目标模块 m 都有一个符号表（即 .symtab 节），包含着 m 定义和引用的符号的信息。

**有三种不同的符号：**

由**模块 m 定义并能被其他模块引用**的**全局符号**。包括非静态的函数和全局变量

由**其他模块定义并被 m 引用**的全局符号，称之为**外部符号**。对应其他模块中定义的非静态函数和全局变量。

由**模块 m 定义且只能被 m 引用**的**局部符号。**包括带 static 属性的函数和全局变量。

**对照 C++ 的语法来理解什么是全局符号和局部符号**（static 对全局变量和函数的**隐藏效果**是一样的）：

C++ 中，static 变量只能在本文件中使用，即使外其他文件中用 extern 中声明也不行。属于这里的局部符号

C++ 中，非 static 的全局变量在其他文件中也能使用，只需在该文件中用 extern 声明即可。属于这里的全局符号

**注意：**符号表中没有非 static 局部变量的符号，非 static 局部变量在运行时在栈中被管理。这里的局部符号和程序中的局部变量是不同的。

**编译器**在 .data 或 .bss 中**为每个全局变量和 static 变量的定义分配空间**，并**在符号表中创建一个有唯一名字的符号**。

**符号表中的条目**

符号表实际上是一个**条目的数组**，每个条目描述一个符号的信息。

typedef struct{

int name;//name 是一个字符串表(.strtab节)中的字节偏移，指向符号的名字(用一个以 null 结尾的字符串表示)

char type:4;//表明符号的类型：数据或函数（4 bits）

binding:4;//表明符号是本地的还是全局的（4 bits）//这里的意思似乎是 type 和 binding 分别是一个 char 类型的高四位和低四位

char reserved;//

short section;//表明符号位于文件的哪个节中，section 是一个到节头部表的索引。

long value;//对于可重定位文件而言，value 是距定义目标的节的起始位置的偏移；对于可执行文件而言，value 是一个绝对运行时地址

long size;//对象的大小，以字节为单位

}

符号表中的条目除了符号外，还可以包含各个节的条目，对应原始源文件的路径名的条目。

**7.6 符号解析**

**链接器解析符号引用的方法：**将每个引用和它输入的可重定位文件的符号表中的一个确定的符号定义关联起来。

符号解析可以分为对局部符号的解析和对全局符号的解析：

局部符号：简单明了

备注：在每个模块中，编译器只允许每个局部符号有一个定义。并且会确保每个静态变量有唯一的名字。

全局符号：更复杂一些

方式：编译器遇到一个不是在当前模块定义的符号时，会假设该符号是在其他某个模块中定义的，在可重定位目标文件中生成一个符号表条目，并把它交给链接器处理。

特殊情况：多个目标文件中定义了相同名字的的全局符号。

**这里没有说清是如何完成对局部符号的解析的，也没有说清对全局符号的解析方式**

**重整**

链接器通过对重载函数的不同版本进行重整，将每个唯一的方法和参数列表的组合编码成一个对链接器来说唯一的名字。

**7.6.1 链接器如何解析多重定义的全局符号**

编译器和汇编器会把每个全局符号区分为强或弱，并将之隐含地编码在可重定位文件地符号表里。

**强符号**：函数和已初始化的全局变量

**弱符号**：未初始化的全局变量

**Linux 链接器使用以下规则来处理多重定义的全局符号：**

规则1：不允许有多个同名的强符号

规则2：如果一个全符号和多个弱符号同名，那么选择强符号

规则3：如果有多个弱符号同名，任意选择其中一个

注意：vs 的链接器并未遵守规则2，规则3：如果定义了同名的全局变量，链接器会直接报错，不论是强符号还是弱符号。

**7.6.2 与静态库链接**

可以将多个相关的目标模块打包成一个单独的文件，称为**静态库**。

通过静态库，相关的函数可以被编译为独立的目标模块，然后封装成一个单独的静态库文件。

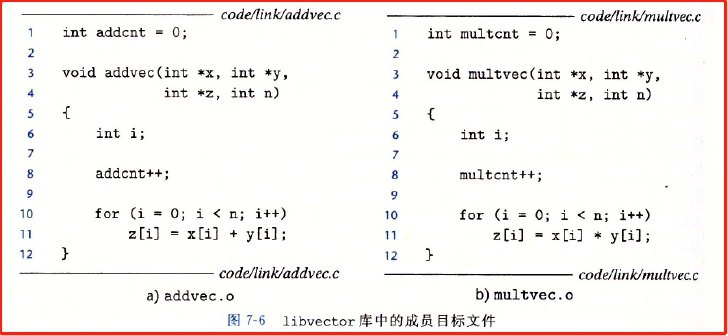
静态库可以用作链接器的输入。链接器在构造可执行文件时，从静态库中复制被应用程序引用的目标模块，其他未用到的模块则不会复制。

在 Linux 系统中，**静态库**以一种称为存档的特殊文件格式存放磁盘中。存档文件是**一组连接起来的可重定位目标文件的集合**，有一个头部用来描述每个成员目标文件的大小和位置。存档文件后缀名为 **.a** 。

**理解**：静态库和存档文件可以当作一个东西。存档是文件层面的描述，静态库是模块层面的描述。

在 linux 中，静态链接库是 **.a** 文件，动态链接库是 **.so** 文件。在windows 中，静态链接库是 .lib 文件，动态链接库是 .dll 文件。

**静态库的应用实例**



通过如下命令创建静态库：

linux> gcc -c addvec.c multvec.c //将 addvec.c 和 multvec 两个文件编译成两个可重定位目标文件

linux> ar rcs libvector.a addvec.o multvec.o //采用 ar 工具将上一步生成的两个可重定位目标文件 addvec.o 和 multvec.o 封装到静态库 libvector.o 中。

**7.6.3 链接器如何解析引用**

**符号解析的过程**

在符号解析阶段，链接器从左到右按照它们在编译器驱动程序命令行上出现的顺序来扫描可重定位目标文件和存档文件。

在扫描中，链接器会维护一个**可重定位目标文件的集合 E**，一个**未解析的符号 (即引用了但尚未定义的符号) 集合 U**，**已定义的符号集合 D**。初始时 E, U, D 都为空。

对于命令行上的每个输入文件 f，链接器会判断 f 是一个目标文件还是一个存档文件。

如果 f 是一个目标文件，链接器会把 f 添加到 E，修改 U 和 D 来反映 f 中的符号定义和引用，并继续下一个输入文件。

如果 f 是一个存档文件，链接器会尝试匹配 U 中未解析的符号和存档文件成员定义的符号。

如果 f 中的某个成员 m 定义了一个符号来解析 U 中的一个引用，就把 m 加到 E 中，并修改 U 和 D 来反映 m 中的符号定义和引用。

对存档文件中所有的成员目标文件都依次进行这个过程。之后任何不包含在 E 中的成员目标文件都简单地被丢弃。

处理完 f，链接器会继续处理下一个输入文件。

当链接器扫描完所有输入文件后，如果 U 是非空的，链接器会输出一个错误并终止。

**库在命令行中放在什么位置**

在命令行中，如果定义一个符号的库出现在引用这个符号的目标文件前，引用就不能被解析，链接会失败。因为初始时 U 是空的。

一般把库放在命令行的结尾。如果库之间相互依赖，则依赖者在前，被依赖者在后。如果双向引用，可以在命令行上重复库。

**7.7 重定位**

符号解析完成后，每个符号引用就和一个符号定义（即一个输入目标模块中的一个符号表条目）关联起来了。**到底是怎么关联起来的？**此时链接器已经知道它的输入模块中的代码节和数据节的确切大小（存储在节头部表中），接下来就是重定位步骤了。

重定位将合并输入模块并**为每个符号分配运行时地址**。

**重定位分为两步：**

重定位节和符号定义。分为两步：

链接器将所有相同类型的节合并为同一类型的新的聚合节。

链接器将运行时内存地址赋给新的聚合节，赋给输入模块定义的每个节，以及赋给输入模块定义的每个符号。

上面两步完成后，程序中的每条指令和全局变量都有唯一的运行时内存地址了。

重定位节中的符号引用。

链接器修改代码节和数据节中对每个符号的引用，是他们指向正确的运行时地址。链接器依赖于可重定位目标模块中的重定位条目。

**7.7.1 重定位条目**

重定位条目用来解决符号引用和符号定义的运行时地址的关联问题。

当汇编器遇到对最终位置的目标引用时，就会生成一个重定位条目，告诉链接器在合并目标文件为可执行文件时如何修改这个引用。

代码的重定位条目放在 .rel.text 中，已初始化数据的重定位条目放在 .rel.data 中。

每个重定位条目都代表了一个必须被重定位的引用。

**ELF 重定位条目的格式**

typedef struct{

long offset; //需要被修改的引用的节偏移（即该符号引用距离所在节的初始位置的偏移）。

long type:32, //重定位类型，不同的重定位类型会用不同的方式来修改引用

symbol:32; //symbol table index,指向被修改引用应该指向的符号

long addend; //一个有符号常数，一些类型的重定位要使用它对被修改引用的值做偏移调整

}

ELF 定义了 30 种不同的重定位类型。以下是其中最基本的两种：

R\_X86\_64\_PC32：重定位一个使用 32 位 PC 相对地址的引用。

什么是 **PC 相对地址**：一个 PC 相对地址就是距程序计数器的值的偏移量。当 CPU 执行到一条使用 PC 相对寻址的指令时，就将在指令中编码的 32 位偏移量值加上 PC 的当前运行时值，得到有效地址，PC 值通常是下一条指令在内存中的地址。

R\_X86\_64\_32：重定位一个使用 32 位绝对地址的引用。通过绝对寻址，CPU 直接使用在指令中编码的 32 位值作为有效地址。

这两种类型都使用了 x86-64 小型代码模型，该模型假设可执行目标文件中的代码和数据的总体大小小于 2GB，因此可以通过 32 位地址来访问。GCC 默认使用小型代码模型。此外还有中型代码模型和大型代码模型。

**7.7.2 重定位符号引用**

**1、重定位 PC 相对引用**

PC 相对引用的机制：在引用中存放着与 PC 的值偏移量。这实际上是符号定义的地址与符号引用的地址差。在实际运行时，当执行到了符号引用的指令时，PC 中的值就是符号引用的地址，加上 与 PC 的偏移量（即符号定义与符号引用的地址差）就得到了符号定义的地址。

备注：这里是相对粗糙的解释，详细的细节还要考虑到 attend 的值，具体的要看书上 481 页。

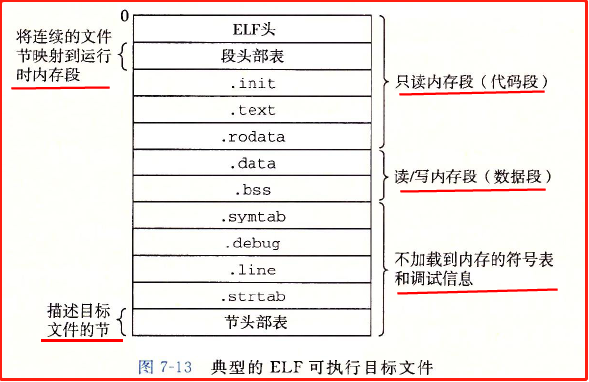
**2、重定位绝对引用**

绝对引用的机制：引用中存放的就是符号定义的绝对地址

**7.8 可执行目标文件**

**可执行目标文件**是一个二进制文件，包含加载程序到内存并运行它所需的所有信息。

可执行目标文件的格式与可重定位目标文件的格式类似。



其中 ELF头 描述了文件的总体格式，还包括**程序的入口点**，即程序运行时要执行的第一条指令的地址。

段头部表和节头部表描述了可执行文件中的片到内存映像中的段的映射关系。它描述了各节在可执行文件中的偏移、长度、在内存映射中的偏移等。

.text, .rodata, .data 节与可重定位目标文件中的节相似，但已经重定位到它们最终的运行时内存地址。

\_init 节定义了一个小函数 \_init，程序的初始化代码会调用它。

可执行文件是完全链接的，因此比可重定位目标文件少了 .rel 节。

**7.9 加载可执行目标文件**

Linux shell 中运行可执行目标文件的方式：**在命令行中输入文件的名字**（用带 ./ 的相对路径表示）。

>> ./prog.o

上面运行了文件 prog.o，因为 prog 不是一个内置的 shell 命令，所以 shell 会认为 prog 是一个可执行目标文件，通过调用**加载器**（是操作系统中的一个程序）来运行它。

**加载**：加载器将可执行目标文件的代码和数据从磁盘复制到内存，然后跳转到程序的第一条指令或入口点来运行程序。

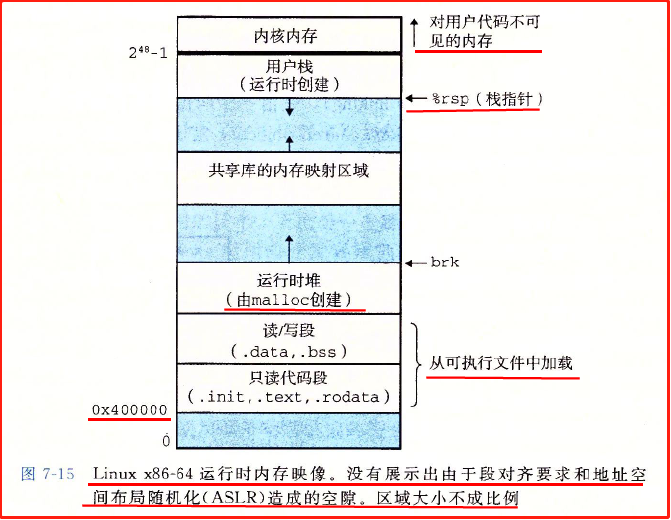
任何 Linux 程序都可以**通过 execve 函数来调用加载器**。

每个 Linux 程序都有一个运行时内存映像。**代码段总是从 0x400000 处开始**，后面是**数据段**，然后是**运行时堆段**，通过调用 malloc 库往上增长。堆后面的区域是为共享模块保留的。**用户栈**总是**从最大的用户地址** 2^48-1 开始，向较小内存地址增长。从地址 2^48 开始是留给内核的。

在分配栈、共享库、堆的运行时地址的时候，链接器还会使用地址空间布局随机化，所以每次程序运行时这些区域的地址都会改变。

**加载器的工作过程**

加载器运行时，创建一个内存映像（虚拟地址空间），在程序头部表的引导下，**将可执行文件的片复制到代码段和数据段**。然后加载器跳转到程序的入口点，即 \_start 函数的地址（函数在系统目标文件 ctrl.o 中定义），\_start 函数调用系统启动函数 \_\_libc\_start\_main（定义在 libc.o 中），\_\_libc\_start\_main 初始化执行环境，调用用户层的 main 函数，处理 main 函数的返回值，并在需要时把控制返回给内核。



**7.10 动态链接共享库**

静态库解决了如何让大量相关函数对应用程序可用的问题。

静态库的缺点：

静态库需要定期维护和更新。如果想要使用一个更新后的静态库，必须显式地将程序与更新了的静态库重新链接。

调用的静态库中的函数在运行时会被复制到每个运行进程的文本段中。

共享库是为了解决静态库缺陷的产物。也就是说共享库的主要目的就是

共享库与可执行文件相独立，只要输出接口不变（即名称、参数、返回值类型和调用约定不变），共享库更新不会对可执行文件造成任何影响。

允许多个正在运行的进程共享内存中相同的库代码，从而节约宝贵的内存资源。

共享库是一个**目标模块**，在运行或加载时，可以加载到任意的内存地址，并和内存中的程序链接起来。

动态链接：在程序运行或加载时，动态链接器将共享库加载到内存中并和程序链接起来。

共享库在 Linux 中以 **.so** 后缀表示，在 Windows 中以 **.dll** 表示。Windows 操作系统中大量使用了共享库。

共享库共享的方式：

一个共享库只有一个 .so 文件，所有引用该库的可执行目标文件共享这个 .so 文件中的代码和数据，而不是像静态库那样复制和嵌入到引用它们的文件中。

在内存中，一个共享库 .text 节的一个副本可以被不同的正在运行的进程共享。

**共享库实例**

生成共享库的方式：

linux> gcc -shared -fpic -o libvector.so addvec.c multvec.c //将 addvec.c 和 multvec.c 封装到动态库 libvector.so 中

// -fpic 选项指示编译器生成与位置无关的代码。

// -shared 选项指示链接器创建一个共享的目标文件。

**链接共享库**

在 main2.c 函数中，调用了共享库 libvector.so 中的 addvec 函数，因此要将 main2.c 和共享库 libvector.so 链接起来。

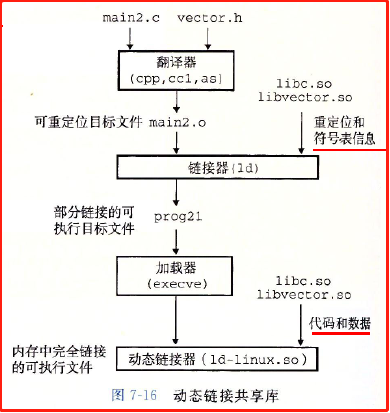
linux> gcc -o prog21 main2.c ./libvector.so //创建了一个可执行目标文件 prog21

将 main2.o 和 libvector.so 链接并不是将 libvector.so 中的内容拷贝到了可执行文件 prog21 中，而是链接器复制了一些 libvector.so 中的重定位和符号表信息，以便运行时可以解析对 libvector.so 中代码和数据的引用。

**理解：**动态链接库是在程序运行或加载时才动态链接的，但并不意味着在执行之前不需要进行其他操作：在链接时链接器要与动态链接库进行一次部分链接以获取到它的重定位和符号表信息。

理解：要在程序中使用动态链接库，也需要在源文件中包含相关的头文件。

下图是动态链接的过程，其中的 libc.so 是包含所有标准 C 函数的动态库。



动态链接器完成链接的操作：

重定位 libc.so 的文本和数据到某个内存段。（理解：这里的意思是将 libc.so 的内容加载到内存中？）

重定位 libvector.so 的文本和数据到另一个内存段。

重定位 prog21 中所有对由 libc.so 和 libvector.so 定义的符号的引用。

上述操作完成后，共享库的位置就固定了，且程序执行的过程中都不会改变。

**7.11 从应用程序中加载和链接共享库**

**动态链接**：应用程序在**运行时**要求动态链接器加载和链接某个共享库（共享库即动态链接库）。

动态链接的应用：

**分发软件**。软件开发者常利用共享库来分发软件更新，它们生成共享库的新版本，用户只需要下载共享库并替代当前版本，下一次运行应用程序时，应用将自动链接和加载新的共享库。

**构建高性能 Web 服务器**：许多 Web 服务器使用基于动态链接的方法来生成动态内容。将每个生成动态内容的函数打包在共享库中，当一个浏览器请求达到时，服务器就动态加载并链接相应函数，然后直接调用它，而非创建新的进程来运行函数。

**dlopen 函数**

Linux 系统为动态链接器提供了一个简单接口 **dlopen 函数**，允许应用程序在运行时加载和链接共享库。

#include <dlfcn.h>

void \*dlopen(const char \*filename, int flag); //若成功就返回指向句柄的指针，否则返回 NULL。

dlopen 函数加载和链接共享库 filename

**dlsym 函数**

dlsym 函数用来调用共享库中的函数

#include <dlfcn.h>

void \*dlsym(void \*handle, char \*symbol); //若成功，返回指向符号 symbol 的指针，若出错返回 NULL

两个输入参数中，handle 是一个指向前面已经加载链接了的共享库的句柄，symbol 是一个符号（可以是一个函数名）如果该符号存在，就返回符号的地址，否则返回 NULL。

以 symbol 是一个函数名为例，dlsym 返回该函数的地址，用户用一个函数指针接受返回的地址后，即可以通过该函数指针调用动态链接库中的函数。

**注意**：这要求提前知道动态链接库中的函数名及形参列表，返回类型。

**dlclose 函数**

如果没有其他共享库还在使用这个共享库，dlclose 函数就卸载该共享库。

#include <dlfcn.h>

int dlclose(void \*handle); //若成功返回 0，出错返回 -1。

**dlerror 函数**

#include <dlfcn.h>

const char \*dlerror(void); //如果前面对 dlopen, dlsym, dlclose 的调用失败，则返回用字符串表示的错误消息，否则返回 NULL。

**一个例子**

动态链接的过程需要**依次调用 dlopen, dlsym, dlclose, dlerror 函数**。

#include <stdio.h>

#include <stdlib.h>

#include <dlfcn.h>

int x[2] = {1, 2};

int y[2] = {3, 4};

int z[2];

int main()

{

void \*handle;

void (\*addvec)(int \*, int \*, int \*, int);

char \*error;

/\* Dynamically load the shared library containing addvec() \*/

handle = dlopen("./libvector.so", RTLD\_LAZY);

if (!handle){

fprintf(stderr, "%s\n", dlerror());

exit(1);

}

/\* Get a pointer to the addvec() function we just loaded \*/

addvec = dlsym(handle, "addvec");

if ((error = flerror()) != NULL) {

fprintf(stderr, "%s\n", error);

exit(1);

}

/\* Now we can call addvec() just like any other function \*/

addvec(x, y, z, 2);

printf("z = [%d %d]\n", z[0], z[1]);

/\* Unload the shared library \*/

if (dlclose(handle) < 0) {

fprintf(stderr, "%s\n", dlerror());

exit(1);

}

return 0;

}

Java 就是**通过 dlopen 接口加载共享库**来调用 C 和 C++ 函数的。

**7.12 位置无关代码**

共享库的一个主要目的就是允许多个正在运行的进程共享内存中相同的库代码，从而节约宝贵的内存资源。

多个进程如何共享动态库的同一个副本，两种方法：

给每个共享库分配一个事先预备的专用的地址空间片，然后要求加载器总是在这个地方加载共享库。这种方法问题很多。

**使用位置无关代码。**这种方法才是实际采用的方法，列出上面那个就是为了用来衬托这个方法的。

位置无关代码（PIC）可以加载而无需重定位。

用户可以对 GCC 使用 **-fpic 选项**来生成 PIC 代码。共享库的编译必须总是使用此选项。

**7.13 库打桩机制**

**7.14 处理目标文件的工具**

Linux 系统种可以帮助处理目标文件的工具：

AR：创建静态库，插入、删除、列出和提取成员。

LDD：列出一个可执行文件在运行时所需要的共享库。

其他工具省略，详见原书 496 页。

**7.15 小结**

链接可以在编译时由静态编译器完成（静态库的链接），也可以在加载和运行时由动态链接器完成（动态库的链接）。

链接器处理的文件是目标文件，目标文件是一种二进制文件，有 3 种不同形式：

可重定位目标文件：

可执行目标文件：静态链接器将多个可重定位目标文件合并成一个可执行目标文件，它可以加载到内存中并执行。.exe 文件就是可执行目标文件。

共享目标文件（共享库）：运行时由动态链接器链接和加载。

链接器的两个主要任务：

符号解析：将目标文件中的每个全局符号都绑定到一个唯一的定义。

重定位：确定每个符号的最终内存地址，并修改对那些目标的引用。

静态链接器是由 GCC 这样的编译驱动程序调用的。它们将多个可重定位目标文件合并成一个单独的可执行目标文件。多个目标文件可以定义相同的符号，链接器可以按照一定规则来解析这些相同的符号。

多个目标文件可以被连接到一个单独的静态库中。链接器用库来解析其他目标模块中的符号引用。许多链接器都是通过从左到右的顺序扫描库来解析符号引用。

加载器将可执行文件的内容映射到内存，并运行这个程序。链接器还可能生成部分链接的可执行目标文件，这样的文件中有对定义在共享库中的例程和数据的为解析的引用。在加载时，加载器将部分链接的可执行文件映射到内存，然后调用动态链接器，动态链接器通过加载共享库和重定位程序中的引用来完成链接任务。

被编译为位置无关代码的共享库可以加载到任何地方，也可以在运行时被多个进程共享。为了加载、链接和访问共享库的函数和数据，应用程序也可以在运行时使用动态链接器。