[读书笔记]CSAPP：17[VB]链接

视频链接：

【精校中英字幕】2015 CMU 15-213 CSAPP 深入理解计算机系统 课程视频_哔哩哔哩 (゜-゜)つロ 干杯~-bilibiliwww.bilibili.com/video/av31289365?p=13![img](https://pic4.zhimg.com/v2-82eac1470b916682f49fd18b47cf7d23_180x120.jpg)

课件链接：

http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/lectures/13-linking.pdfwww.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/lectures/13-linking.pdf

该部分对应于书中的第7章。

（最近在写毕设，耽误了点时间，这章的中文翻译看着好累。）

到了这一章，书中的内容也到了第二部分了，在前一部分中，主要介绍了程序和硬件之间的交互关系，而到了这部分后，将探索程序和操作系统之间的关系。操作系统给程序提供了一层抽象，让程序以为自己独占了处理器和内存。

由于局部非静态变量保存在寄存器或栈中，所以符号只包含函数、全局变量和静态变量。

判断符号的信息：

- **判断符号所在的节：**
  - 如果是函数，则在`.text`节中
  - 如果是未初始化全局变量，则在`COMMIN`节中
  - 如果是未初始化静态变量，以及初始化为0的全局变量或静态变量，则在`.bss`节中
  - 如果是初始化过的全局变量或静态变量，则在`.data`节中
- **判断符号类型：**
  - 如果是在当前文件中定义的函数或全局变量，则为全局符号
  - 如果是在当前文件定义的静态函数或静态全局变量，则为局部符号
  - 如果在别的文件中定义，则为外部符号
- **注意：**局部非静态变量保存在栈或寄存器中，所以不考虑

保存在什么表，是根据符号的类型和定义情况，而定义和引用是从变量或函数的角度来看的

判断符号采用哪种定义：

- 在各个文件中确定同名全局符号的强弱，其中符号和初始化的全局符号为强符号，未初始化的全局符号为弱符号
- 如果存在多个同名的强符号，则会出现链接错误
- 如果存在一个强符号和多个同名的弱符号，则会采用强符号的定义
- 如果存在多个同名的弱符号，则会随机采用一个弱符号的定义

需要根据目标文件和库之间对符号解析的依赖关系，来确定命令行中输入文件的顺序，保证前面文件中未解析的符号，能在后面文件中得到解析。

重定位包含两层意思：分配内存地址，根据重定位表条目来修改占位符。

链接（Linking）是将各种代码和数据片段收集并组合成一个单一的文件的过程，然后该文件会被加载到内存中执行。该过程由**链接器（Linker）**程序自动执行。**链接存在三种类型：**

- 执行于**编译时（Compile Time）**，即在源代码被翻译成机器代码时的传统静态链接
- 执行于**加载时（Load Time）**，即程序被**加载器（Loader）**加载到内存并执行时的动态链接

- 执行于**运行时（Run Time）**，即由应用程序来执行的动态链接

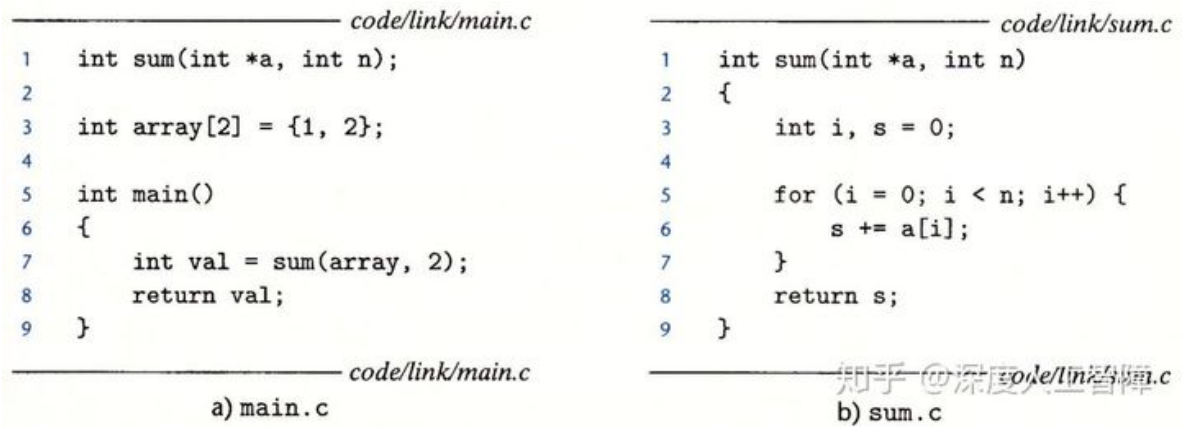
作用：链接的存在，使得**分离编译（Separate Compilation）**成为可能，一个大型应用程序可以分解成若干个小的模块，只需要对这些模块进行修改编译，然后通过链接器将其组合成大的可执行文件就行。

学习的意义：能帮助你构建大型程序，避免一些危险的编程错误，理解语言的作用域规则是如何实现的，理解其他重要的系统概念，比如加载和运行程序、虚拟内存、分页和内存映射，并且可以让你利用共享库。

本文使用的环境：运行Linux的x86-64系统，使用标准的ELF-64目标文件格式。

1 基本概念

大多数编译系统提供**编译器驱动程序（Compiler Driver）**，使得用户可以调用预处理器、编译器、汇编器和链接器对程序进行编译。



code/link/main.c

```
1  int sum(int *a, int n);
2
3  int array[2] = {1, 2};
4
5  int main()
6  {
7      int val = sum(array, 2);
8      return val;
9  }
```

code/link/main.c

a) main.c

code/link/sum.c

```
1  int sum(int *a, int n)
2  {
3      int i, s = 0;
4
5      for (i = 0; i < n; i++) {
6          s += a[i];
7      }
8      return s;
9  }
```

code/link/sum.c

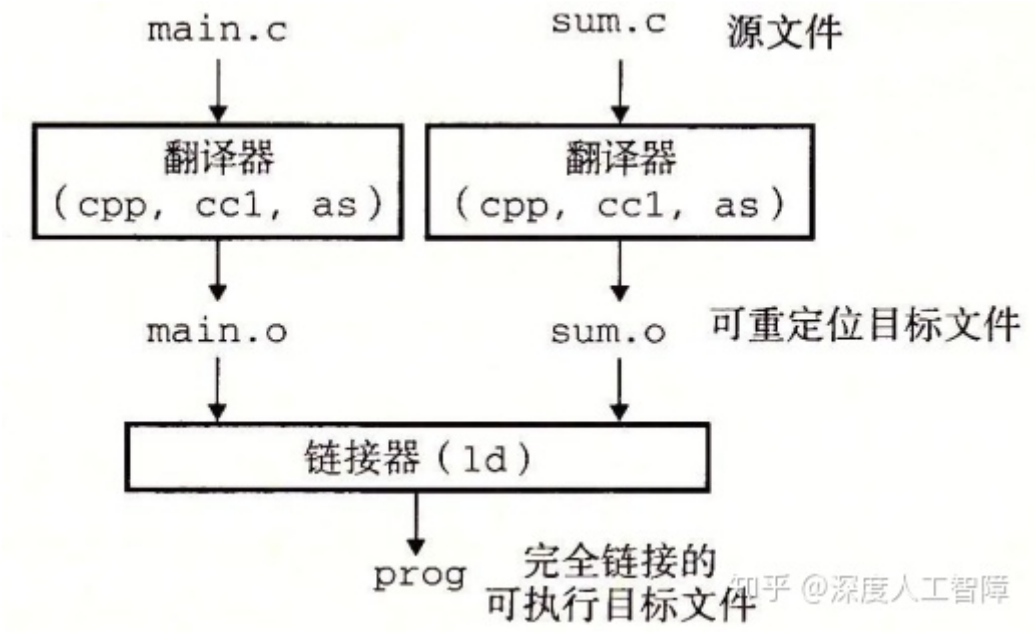
b) sum.c

如上图所示是一个示例程序，对于GNU编译系统，可以通过输入

```
gcc -Og -o prog main.c sum.c
```

来调用GCC驱动程序，然后执行以下过程：

1. GCC驱动程序运行**C预处理器（cpp）**，将C源程序`main.c`翻译为ASCII码的中间文件`main.i`。`cpp [other arg] main.c /tmp/main.i`
2. GCC驱动程序运行**C编译器（ccl）**，将`main.i`翻译成ASCII汇编语言文件`main.s`。`ccl /tmp/main.i -Og [other arg] -o /tmp/main.s`。
3. GCC驱动程序运行**汇编器（as）**，将`main.s`翻译成一个**可重定位目标文件（Relocatable Object File）**`main.o`。`as [other arg] -o /tmp/main.o /tmp/main.s`。
4. 对`sum.c`执行相同的过程，得到`sum.o`。
5. GCC驱动程序运行**链接器（ld）**，将`main.o`和`sum.o`以及其他必要的系统目标文件组合起来，得到**可执行目标文件（Executable Object File）**`prog`。`ld -o prog [system object files and args] /tmp/main.o /tmp/sum.o`。
6. 在shell中输入`./prog`，则shell会调用操作系统的**加载器（Loader）**函数，将该可执行文件`prog`复制到内存中，然后执行该程序。



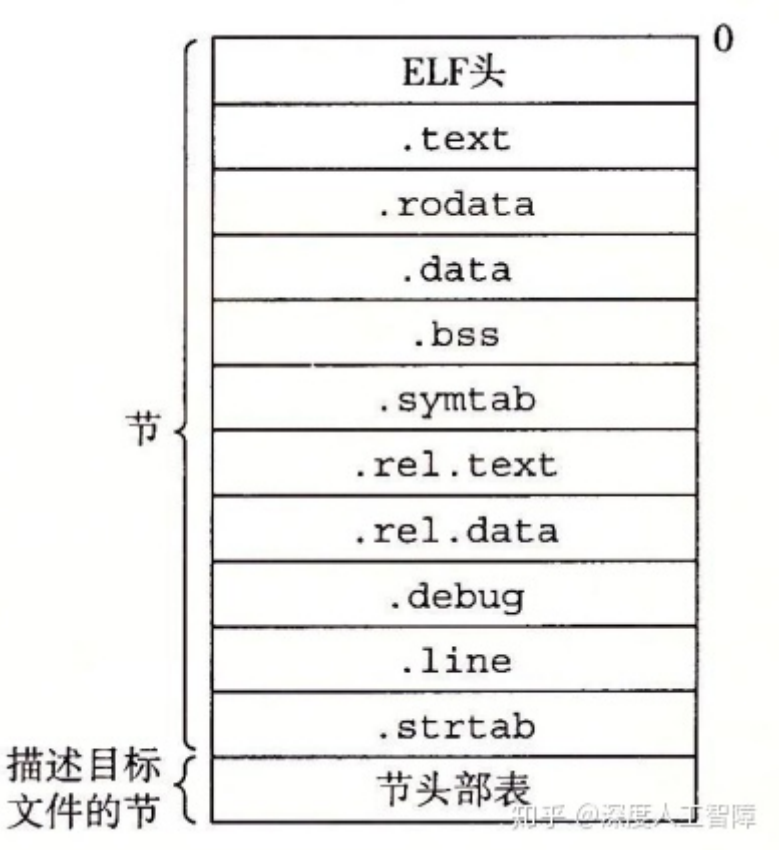
可以发现存在不同的目标文件，**主要包含**：

- **可重定位目标文件**：由各种不同的二进制代码和**数据节（Section）**组成，每一节是一个连续的字节序列，不同节存放不同的数据。
  - **生成**：由编译器和汇编器生成
  - **用处**：在编译时与其他可重定位目标文件合并起来，得到一个可执行目标文件。
- **可执行目标文件**：包含二进制代码和数据
  - **生成**：将可重定位目标文件和静态库输入到链接器中，可产生可执行目标文件
  - **用处**：可被加载器直接复制到内存中并执行
- **共享目标文件**：特殊类型的可重定位目标文件，可以在加载时或运行时被动态地加载进内存并链接。

注意：我们称**目标模块（Object Module）**为一个字节序列，而**目标文件（Object File）**是以文件形式存放在磁盘的目标模块。

目标文件是按照特定的目标文件格式进行组织的，Windews中使用**可移植可执行（Portable Executable，PE）**格式，Max OS-X使用**Mach-O**格式，x86-64 Linux和Unix使用**可执行可链接格式（Executable and Linkable Format，ELF）**。

以可重定位目标文件的ELF格式为例，如下图所示



- **ELF头（ELF header）：** 包含生成该目标文件的系统的字大小和字节顺序、ELF头的大小、目标文件类型、机器类型、节头部表的文件偏移，以及节头部表中条目的大小和数目。
- `.text`： 已编译程序的机器代码
- `.rodata`： 只读数据，比如跳转表等等
- `**.data**`： 保存已初始化的全局变量和静态变量（全局和局部）
- `**.bss**`： 保存未初始化的静态变量（全局和局部），以及被初始化为0的全局变量和静态变量（全局和局部）
- - **注意：**
  - - 在目标文件中`.bss`不占据实际的空间，只是一个占位符
    - 之所以要将初始化和未初始化分成两个节，因为在目标文件中，未初始化变量不需要占据任何实际的磁盘空间，运行时，再在内存中分配这些变量，初始值为0。
    - 局部变量在运行时只保存在栈中，不出现在`.data`和`.bss`中
    - 静态局部变量不受函数栈的管理，所以也要在这个位置创建
- `**.symtab**`： 符号表，存放在程序中**定义和引用**的函数和变量的符号信息
- - **注意：**不包含局部非静态变量条目，因为该变量是由栈管理的
- `.rel.text`： `.text`节的重定位信息，可执行目标文件中需要修改的指令地址
- `.rel.data`： `.data`节的重定位信息，合并后的可执行目标文件中需要修改的指针数据的地址
- - **注意：**
  - - 一般已初始化的全局变量，如果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
    - 可执行目标文件已完成重定位，就不需要`.rel.text`和`.rel.data`数据节了。
- `.debug`：调试符号表，其条目是程序中定义的局部变量和类型定义，程序汇总定义和引用的全局变量，以及原始的C源文件
- `.line`： 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：**只有以-g选项调用编译器驱动程序，才会出现`.debug`和`.line`
- `.strtab`： 字符串表，包括`.symtab`和`.debug`节中的符号表，以及节头部中的节名字
- **节头部表（Section Header Table）：** 给出不同节的大小和位置等其他信息

> 我们可以使用GNU READELF程序来查看目标文件内容。

每个可重定位目标模块都会有一个由汇编器构造的**符号表.symtab**，包含了当前模块中定义和引用的符号信息。在链接器的上下文中（链接器是对不同的可重定位目标文件进行操作的，所以它的上下文就是不同的可重定位目标模块），根据符号定义和引用的情况，可以将其分成以下类型：

- **全局链接器符号**：在当前可重定位目标模块中定义，并能被其他模块引用的符号。对应于非静态的函数和全局变量。
- **外部链接器符号**：在别的可重定位目标模块中定义，并被当前模块引用的符号。对应于在其他模块中定义的非静态的函数和全局变量。（外部连接器符号也是全局连接器符号）
- **局部链接器符号**：只在当前可重定位目标模块定义和引用的符号。对应于静态的函数和全局变量，这些符号在当前模块中任何位置都可见，但不能被别的模块引用。

注意：局部静态变量不在栈中管理，所以编译器在`.data`或`.bss`中为其分配空间，并在符号表`.symtab`中创建一个有唯一名字的局部链接器符号。

符号表`.symtab`中的每个条目具有以下格式

code/link/elfstructs.c

```
1  typedef struct {
2      int   name;      /* String table offset */
3      char  type:4,    /* Function or data (4 bits) */
4            binding:4; /* Local or global (4 bits) */
5      char  reserved;  /* Unused */
6      short section;   /* Section header index */
7      long  value;     /* Section offset or absolute address */
8      long  size;      /* Object size in bytes */
9  } Elf64_Symbol;
```


code/link/elfstructs.c

- **name**：保存符号的名字，是`.strtab`的字节偏移量
- **type**：说明该符号的类型，是函数、变量还是数据节等等
- **binding**：说明该符号是局部还是全局的
- **value**：对于可重定位目标文件而言，是定义该符号的节到该符号的偏移量（比如函数就是在`.text`中，初始化的变量在`.data`，未初始化的变量在`.bss`中）；对于可执行目标文件而言，是绝对运行形式地址。
- **size**：是符号的值的字节数目。（通过value和size就能获得该符号的值）
- **section**：说明该符号保存在哪个节中，是节头部表中的偏移量。
- - **注意**：可重定位目标文件中有三个无法通过节头部表进行索引的数据节，称为**伪节（Pseudosection）**
  - - **ABS**：不该被重定位的符号
    - **UNDEF**：未定义的符号，即在当前可重定位目标文件中引用，但在别的地方定义的符号
    - **COMMON**：表示未被分配位置的未初始化的全局变量。此时value给出对齐要求，size给出最小的大小。

对于像Linux LD这样的**静态链接器（Static Linker）**，是以一组可重定位目标文件和命令参数为输入，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。为了构造可执行目标文件，**链接器有两个任务**：

- **符号解析（Symbol Resolution）**：将每个符号引用和一个符号定义关联起来
- **重定位（Relocation）**：编译器和汇编器生成从地址0开始的代码和数据节，链接器会对代码、数据节、符号分配内存地址，然后使用汇编器产生的**重定位条目（Relocation Entry）**的指令，修改所有对这些符号的引用，使得它们指向正确的内存位置。

2 符号解析

定义：链接器符号解析是将每个符号引用与输入的所有可重定位目标文件的符号表中的一个确定的符号定义关联起来。

- 对于局部链接器符号，由于符号定义和符号引用都在同一个可重定位目标文件中，情况相对简单，编译器只允许每个可重定位目标文件中每个局部链接器符号只有一个定义。而局部静态变量也会有局部链接器符号，所以编译器还要确保它有一个唯一的名字。
- 对于全局符号（包括全局符号和外部符号），编译器可能会碰到不在当前文件中定义的符号，则会假设该符号是在别的文件中定义的，就会在重定位表中产生该符号的条目，让链接器去解决。而链接器可能还会碰到在多个可重定位目标文件中定义相同名字的全局符号，也要解决这些冲突。

2.1 链接器解析多重定义的全局符号

编译器会向汇编器输出每个全局符号是**强（Strong）**还是**弱（Weak）**，而汇编器会把这些信息隐式编码在可重定位目标文件的符号表中。函数和已初始化的全局符号是强符号，未初始化的全局符号是弱符号。

然后链接器通过以下规则来处理在多个可重定位目标文件中重复定义的全局符号：

1. 不允许有多个同名的强符号，如果存在，则链接器会报错
2. 如果有一个强符号和多个弱符号同名，则符号选择强符号的定义
3. 如果有多个弱符号同名，符号就随机选择一个弱符号的定义

我们从编译器的角度来看，当编译器看到一个弱全局符号时，它并不确定是否会在别的文件中对该符号进行定义，也无法确定链接器会采用多重定义的哪个定义。所以编译器将未初始化的全局符号放在`COMMON`表中，让链接器去决定。而当全局符号初始化为0时，它就是一个强全局符号，根据规则1可知该符号是唯一的，所以编译器可以直接将其分配到`.bss`中。而对于静态变量，由于其符号也是唯一的，所以编译器也可以直接将其放到`.bss`或`.data`中。

特别要注意的是，当同名符号的类型不同时，规则2和3可能会导致意想不到的错误，比如以下代码

```c
/* foo5.c */
#include <stdio.h>
void f(void);

int y = 15212;
int x = 15213;

int main()
{
    f();
    printf("x = 0x%x y = 0x%x \n",
           x, y);
    return 0;
}
```

```c
/* bar5.c */
double x;

void f()
{
    x = -0.0;
}
```


这里在文件`foo5.c`中，`x`和`y`是强全局符号，而在文件`bar5.c`中，`x`是弱全局符号，所以链接器会选择`foo5.c`中对`x`的定义，将其定义为`int`类型。但是`f`函数会用`double`类型的`-0.0`对`x`进行赋值，而`int`是4字节，`double`是8字节的，可能会造成错误。

2.2 静态库的链接与解析引用

链接器除了能将一组可重定位目标文件链接起来得到可执行目标文件以外，编译系统还提供一种机制，将所有相关的目标模块打包为一个单独文件，称为**静态库（Static Library）**，可以作为链接器的输入。静态库是以**存档（Achive）**的文件格式存放在磁盘的，它是一组连接起来的可重定位目标文件的集合，有一个头部来描述每个成员目标文件的大小和位置，后缀为`.a`。**使用静态库的优点有：**

- 相关的函数可以被编译为独立的目标模块，然后封装成一个独立的静态库文件。
- 链接时，链接器只会复制静态库中被应用程序引用的目标模块，减少了可执行文件在磁盘和内存中的大小
- 应用程序员只需要包含较少的库文件名就能包含很多的目标模块，比如ISO C99中在`libc.a`静态库中包含了`atoi.o`、`scanf.o`、`strcpy.o`等可重定位目标模块，在`libm.a`静态库中包含了数学函数的目标模块。

比如我们有以下函数

code/link/addvec.c

```
1   int addcnt = 0;
2
3   void addvec(int *x, int *y,
4               int *z, int n)
5   {
6       int i;
7
8       addcnt++;
9
10      for (i = 0; i < n; i++)
11          z[i] = x[i] + y[i];
12  }
```

code/link/addvec.c

a) addvec.o

code/link/multvec.c

```
1   int multcnt = 0;
2
3   void multvec(int *x, int *y,
4                int *z, int n)
5   {
6       int i;
7
8       multcnt++;
9
10      for (i = 0; i < n; i++)
11          z[i] = x[i] * y[i];
12  }
```

code/link/multvec.c

b) multvec.o

我们可以用AR工具创建包含这些函数的静态库，首先需要得到这两个函数的可重定位目标文件

```
gcc -c addvec.c multvec.c
```

由此可以得到`addvec.o`和`multvec.o`，然后创建包含这两个可重定位目标文件的静态库

```
ar rcs libvector.a addvec.o multvec.o
```

由此就得到了静态库`libvector.a`。

为了便于说明静态库中包含了那些函数，以及这些函数的原型，我们会创建一个头文件来包含这两个函数的函数原型，便于想要使用该静态库的人员查看**（不确定？）**

code/link/main2.c

```
1   #include <stdio.h>
2   #include "vector.h"
3
4   int x[2] = {1, 2};
5   int y[2] = {3, 4};
6   int z[2];
7
8   int main()
9   {
10      addvec(x, y, z, 2);
11      printf("z = [%d %d]\n", z[0], z[1]);
12      return 0;
13  }
```


code/link/main2.c

如上面的代码，在头文件`vector.h`中给出了函数`addvec`和`multvec`的函数原型。想要创建可执行目标文件，就要编译和链接`main2.o`和`libvector.a`。首先先产生可重定位目标文件

```
gcc -c main2.c
```

由此可以得到`main2.o`，然后运行以下代码：

```
gcc -static -o prog2c main2.o ./libvector.a
```

由此就能得到一个可执行目标文件`prog2c`。这里的`-static`表示链接器需要构建一个完全链接的可执行目标文件，可以加载到内存并运行，无需进一步链接。我们同样可以使用以下方法：

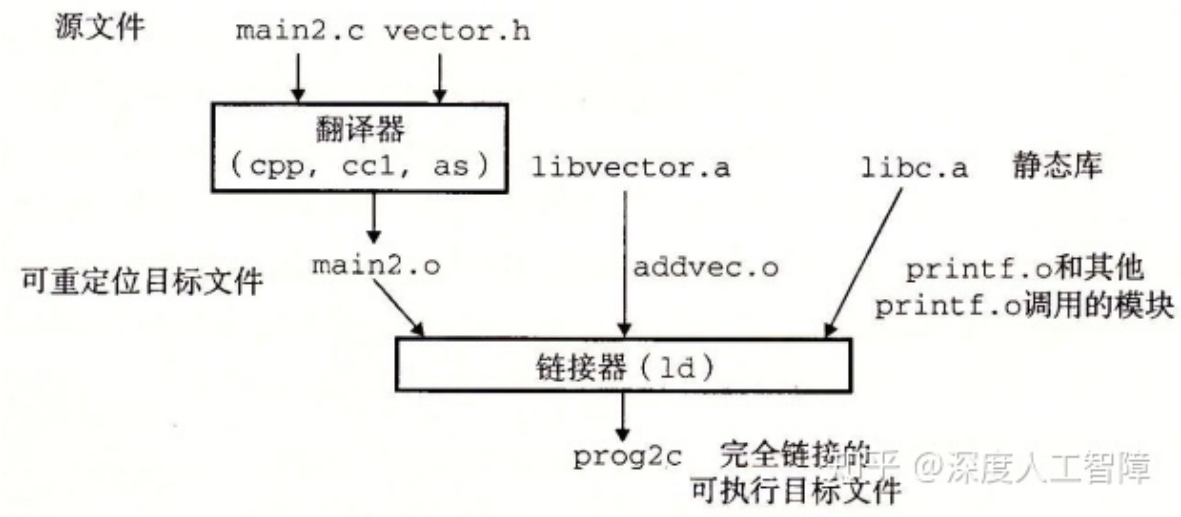
```
gcc -static -o prog2c main.o -L. -lvector
```

这里的`-lvector`是`libvector.a`的缩写，`-L.`告诉链接器在当前目录中查找`libvector.a`静态库。

当运行了该命令行，在符号解析阶段，链接器会维护一个可重定位目标文件的集合`E`，一个引用了但是还未定义的符号集合`U`，一个前面输入文件中已经定义的符号集合`D`，然后在命令行中从左到右依次扫描可重定位目标文件和存档文件：

- 如果输入文件是可重定位目标文件，链接器就将其添加到`E`中，然后根据该文件的符号表来修改`U`和`D`，然后继续下一个输入文件。
- 如果输入文件是存档文件，则链接器会依次扫描存档文件中的成员`m`，如果`m`定义了`U`中的一个符号，则将`m`添加到`E`中，然后根据`m`的符号表来修改`U`和`D`。最后没有包含在`E`中的成员就会被丢弃，然后继续下一个输入文件。
- 如果链接器扫描完毕，`U`中还存在没有确定定义的符号，则链接器会报错并终止，否则链接器会合并和重定位`E`中的目标文件，得到可执行目标文件。

在这个例子中，链接器首先得到输入文件`mian2.o`，其中存在未解析的符号`addvec`，则会将该符号保存在集合`U`中，然后扫描下一个输入文件`libvector.a`时，由于是存档文件，就会依次扫描其中的成员，首先扫描到`addvec.o`时，能对符号`addvec`进行解析，则将`addvec.o`保存在`E`中，并将符号`addvec`从`U`中删除，扫描到`multvec.o`时，由于`U`中已不存在未解析的符号了，所以不会将`multvec.o`包含在`E`中，最终链接器会合并和重定位`E`中的目标文件，得到可执行目标文件。所以链接器最终只会从静态库`libvector.a`中提取`addvec.o`



根据以上过程的描述，我们**需要小心命令行上库和目标文件的顺序**，要保证前面输入文件中未解析的符号能在后续输入文件中进行解析，否则会出现链接错误，一般是将库放在后面，如果库之间存在依赖，也要注意库之间的顺序，并且为了满足依赖关系，可以在命令行上重复库。

特别的：首先输入目标文件，由于目标文件会直接包含在`E`中，所以可以得到目标文件中所有未解析的符号，并且提供了该目标文件中的所有解析的符号，相当于“无条件加入”的，如果存在库依赖目标文件，就无需再输入目标文件了。然后根据库之间的依赖来排序库，存档文件会根据`U`的内容来确定是否将成员`m`保存在`E`中，相当于“按序加入”的，所以需要重复输入库来满足依赖关系。

比如`p.o -> libx.a -> liby.a`且`liby.a -> libx.a -> p.o`。此时我们先输入`p.o`，就包含了解析`lib.a`符号的定义了，然后我们根据依赖输入`libx.a liby.a`，此时由于第一个`libx.a`只是解析了`p.o`中未定义的符号，而`liby.a`中还存在由`libx.a`解析的符号，所以我们还需输入`libx.a`来解析`liby.a`的符号。

3 重定位

当链接器完成符号解析时，就能确定在多个目标文件中重定义的全局符号的解析，以及获得静态库中需要的目标模块，此时所有符号引用都能和一个符号定义关联起来了。此时开始重定位步骤，**包括：**

- 链接器将所有目标模块中相同类型的节合并成同一类型的新的聚合节，比如将所有输入目标模块的`.data`节聚合成可执行文件中的`.data`节，其他节也如此操作。
- 此时链接器知道代码节和数据节的确切大小，就将运行时内存地址赋给新的聚合节，以及输入模块定义的每个符号。此时程序的每条指令和全局变量都有唯一的运行时内存地址了。
- 记得之前可重定位目标文件中，由于编译器和汇编器并不知道符号的运行时内存地址，所以使用一个占位符来设置符号引用的地址，而当前链接器已为符号分配了内存地址，所以链接器需要修改代码节和数据节中对每个符号的引用，使它们指向正确的运行时内存地址。

当汇编器生成目标模块时，它无法确定数据和代码最终会放在内存的什么位置，也无法确定该模块引用外部定义的函数和全局变量的位置，所以汇编器先用占位符来占领位置，然后对地址未知的符号产生一个**重定位条目（Relocation Entry）**，代码的重定位条目会保存在`.rel.text`节中，已初始化数据的重定位条目会保存在`rel.data.`节中。重定位条目的数据结构如下所示

code/link/elfstructs.c

```
1  typedef struct {
2      long offset;    /* Offset of the reference to relocate */
3      long type:32,   /* Relocation type */
4           symbol:32; /* Symbol table index */
5      long addend;    /* Constant part of relocation expression */
6  } Elf64_Rela;
```


code/link/elfstructs.c

其中，`offset`表示要修改符号引用的内存地址，`type`表示重定位的类型，`symbol`是符号表的索引值，表示引用的符号，可以通过该符号获得真实的内存地址，`addend`是一个有符号常数，有些重定位需要使用这个参数来修改引用位置。

我们通过以下代码来介绍两个重定位类型：`R_X86_64_PC32`和`R_X86_64_32`。

code/link/main.c

```
1  int sum(int *a, int n);
2
3  int array[2] = {1, 2};
4
5  int main()
6  {
7      int val = sum(array, 2);
8      return val;
9  }
```


code/link/main.c

我们可以通过 `objdump -dx main.o` 来得到 `main.o` 的反汇编代码，可以发现该函数中无法确定 `array` 和其他目标模块中定义的函数 `sum` 在内存中的地址，所以会对 `array` 和 `sum` 产生重定位条目

code/link/main-relo.d

```
1  0000000000000000 <main>:
2     0:   48 83 ec 08             sub    $0x8,%rsp
3     4:   be 02 00 00 00          mov    $0x2,%esi
4     9:   bf 00 00 00 00          mov    $0x0,%edi        %edi = &array
5                          a: R_X86_64_32 array           Relocation entry

6     e:   e8 00 00 00 00          callq  13 <main+0x13>   sum()
7                          f: R_X86_64_PC32 sum-0x4       Relocation entry
8    13:   48 83 c4 08             add    $0x8,%rsp
9    17:   c3                      retq
```


code/link/main-relo.d

3.1 R_X86_64_PC32

该重定位条目主要用来产生32位PC相对地址的引用，即函数调用时的重定位。

其中 `call` 指令的开始地址处于节偏移 `0xe` 处，然后有一个字节的操作码 `e8`，后面跟着的就是函数 `sum` 的32位PC相对引用的**占位符**，所以链接器修改的位置在当前节偏移 `0xf` 处。该重定位条目 `r` 包含以下字段

```
r.offset = 0xf //该值是当前节的偏移量，定位到重定位的位置
r.symbol = sum //保存的是要重定位的符号
r.type = R_X86_64_PC32 //保存的是重定位的类型
r.addend = -4
```

当前链接器已经确定了各个节和符号的的内存地址，该代码处于 `.text` 节中，则我们可以通过 `.text` 和 `r.offset` 的值来确定占位符的内存地址

```
ADDR(s) = ADDR(.text) = 0x4004d0
refaddr = ADDR(s) + r.offset
        = 0x4004d0 + 0xf
        = 0x4004df
```

然后我们需要计算占位符的内容，根据相对地址的计算方法，可以知道占位符的内容是目标地址减去当前PC的下一条指令的地址。可以通过 `ADDR(r.symbol)` 来获得目标地址，即 `sum` 函数的地址，可以通过 `refaddr` 减去4字节来获得下一指令的地址，然后可以通过以下计算公式来计算占位符内容

```
refptr = s + r.offset //占位符的指针
ADDR(r.symbol) = ADDR(sum) = 0x4004e8
*refptr = (unsigned)(ADDR(s.symbol) + r.addend - refaddr)
        = (unsigned)(0x4004e8 + (-4) - 0x4004df)
        = (unsigned) 0x5
```

由此在可执行目标文件中，`call` 指令就有如下的重定位形式：

```
4004de:   e8 05 00 00 00          callq  4004e8 <sum>      sum()
```

3.2 R_X86_64_32

该重定位条目主要用来产生32位绝对地址的引用，即数组的重定位。

使用数组 `array` 的指令处于 `.text` 节偏移 `0x9` 处，后面有一个字节的操作码，后面跟着的就是数组 `array` 的32位绝对地址的引用的**占位符**，所以链接器修改的位置在当前节偏移 `0xa` 处。该重定位条目 `r` 包含以下字段

```
r.offset = 0xa
r.symbol = array
r.type = R_X86_64_32
r.added = 0
```

我们可以通过`r.symbol`的地址来确定数组`array`的内存地址，然后直接将该内存地址保存到占位符中，即

```
refptr = s + r.offset //占位符的指针
*refptr = (unsigned)(ADDR(r.symbol) + r.addend)
        = (unsigned) 0x601018
```

由此在可执行目标文件中，该引用有以下重定位形式

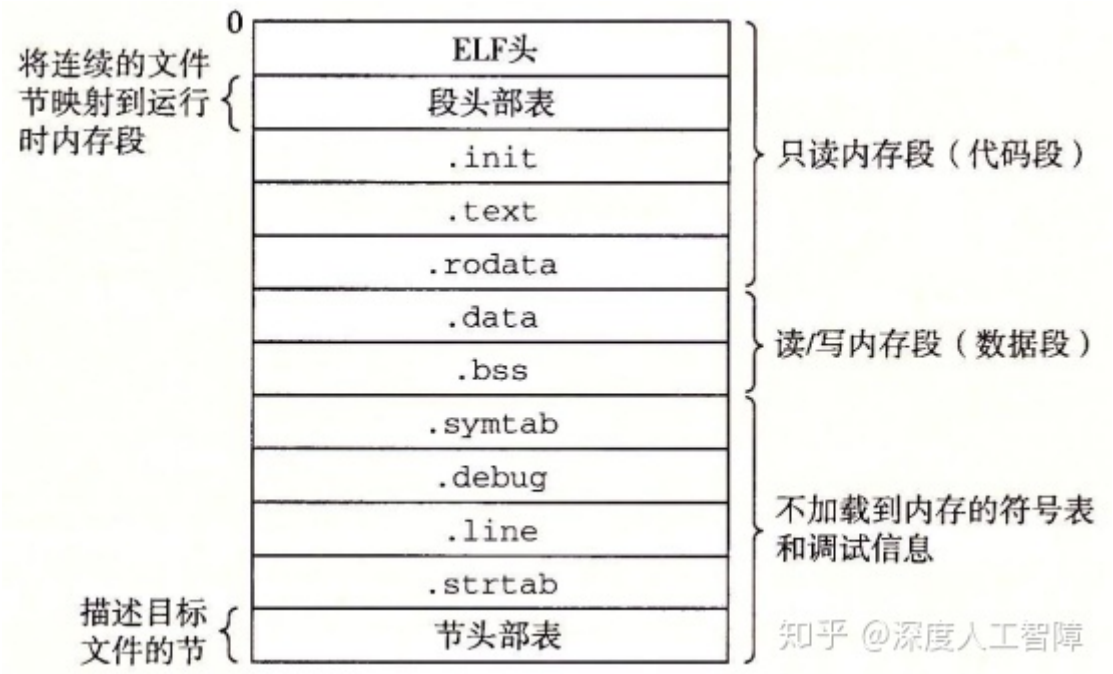
```
4004d9:   bf 18 10 60 00          mov    $0x601018,%edi   %edi = &array
```

重定位后，加载器就会将这些节的字节直接复制到内存中，可以直接执行。

4 可执行目标文件

4.1 ELF格式

通过以上符号解析和重定位过程，链接器已将可重定位目标文件和库合并成一个可执行目标文件了，目标文件的ELF格式如下所示



- **ELF头：**描述了文件的总体格式，还包括程序的**入口点（Entry Point）**，即当程序运行时要执行的第一条指令的地址。
- `.init`：定义了一个小函数`_init`，程序的初始化代码会调用
- `.text`、`.rodata`和`.data`和可重定位目标文件中的类似，只是这里被重定位到了最终的运行时内存地址
- 由于可执行目标文件是完全链接的，已经不需要重定位了，所以不需要`.rel`节了。

段头部表（Segment Header Table）：包括页大小、虚拟地址内存段（节）、段大小等等。描述了可执行文件连续的片到连续的内存段的映射关系，如下图所示是通过`OBJDUMP`显示的`prog`的段头部表

code/link/prog-exe.d

```
Read-only code segment
1 LOAD off    0x0000000000000000 vaddr 0x0000000000400000 paddr 0x0000000000400000 align 2**21
2      filesz 0x000000000000069c memsz 0x000000000000069c flags r-x

Read/write data segment
3 LOAD off    0x0000000000000df8 vaddr 0x0000000000600df8 paddr 0x0000000000600df8 align 2**21
4      filesz 0x0000000000000228 memsz 0x0000000000000230 flags rw-
```


code/link/prog-exe.d

在可执行目标文件中，根据不同数据节对读写执行的不同要求，将不同的数据节分成了两个段：代码段和数据段，其中**代码段**包含ELF头、段头部表、`.init`、`.text`和`.rodata`，**数据段**包括`.data`和`.bss`。然后段头部表中就描述了代码段和数据段到内存段的映射关系，其中`off`是目标文件中的偏移，表示要从目标文件的什么位置开始读取该段；`vaddr/paddr`是内存地址，表示要将该段加载到的内存地址；`align`是对齐要求；`filesz`是目标文件中的段大小，则通过`off`和`filesz`就能确定我们要加载的段的内容；`memsz`是内存中的段大小，表示我们养将目标文件中的该段加载到多大的内存空间中；`flags`表示该段运行时的访问权限。

比如第1行、第2行描述的就是代码段，表示将目标文件中从`0x0`开始的`0x69c`个字节数据保存到从`0x400000`开始的，大小为`0x69c`字节的内存空间中，并具有读和可执行权限。第3行、第4行描述的是数据段，表示将目标文件从`0xdf8`开始的`0x228`个字节数据保存到从`0x600df8`开始的，大小为`0x230`字节的内存空间中，并具有读写权限。

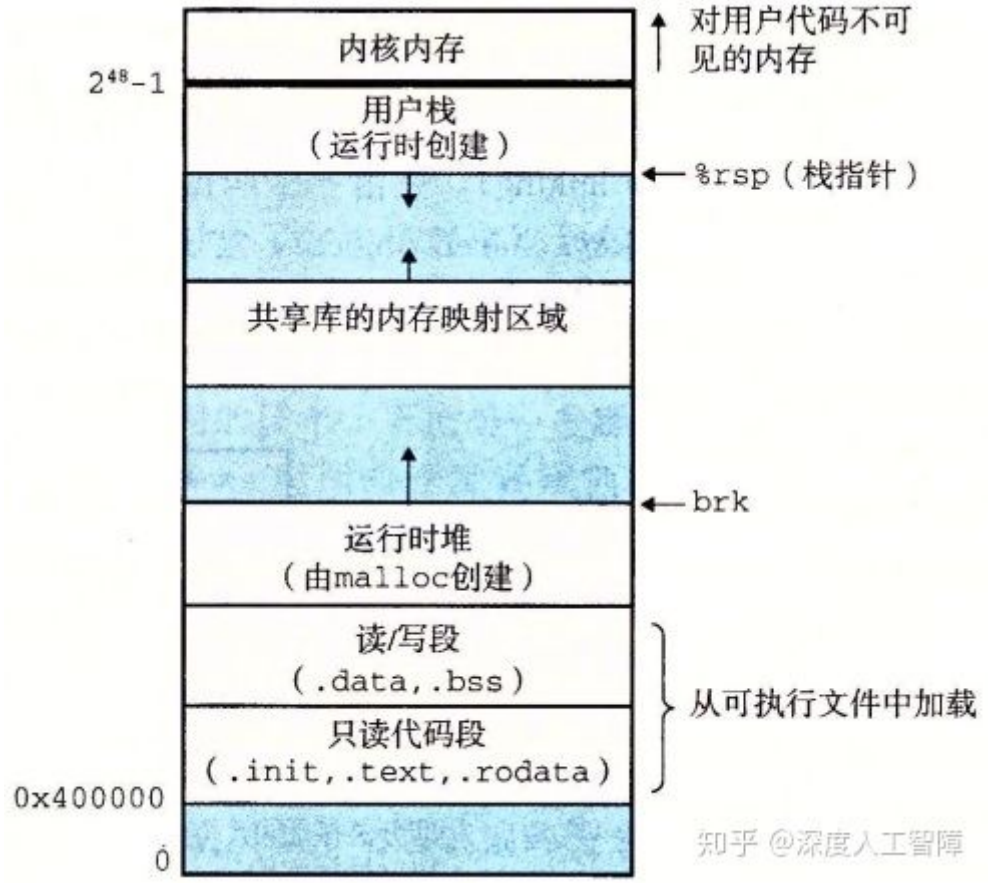
这里为了使得程序执行时，目标文件中的段能高效地传送到内存中，要求

$$vaddr \ mod \ align = off \ mod \ align$$

4.2 加载可执行目标文件

当我得到可执行目标文件`prog`时，我们可以在`shell`中输入`./prog`。

由于`prog`不是内置的shell命令，所以shell会认为`prog`是一个可执行目标文件，就通过调用`execve`函数来调用内核中的**加载器（Loader）**，则加载器会在可执行目标文件的段头部表的引导下，将可执行文件中的数据段和代码段复制到对应的内存位置，然后加载器会创建如下运行时内存映射



- **代码段和数据段**：x86-64通常将代码段保存在`0x400000`处，所以会将可执行目标文件的代码段和数据段映射为如上形式。**注意**：这里数据段为了满足对齐要求，会和代码段之间存在间隙。
- **运行时堆**：在数据段之后会有一个运行时堆，是通过调用`malloc`库动态往上增长的
- **共享库**：在堆之后是一个共享库的内存映射区域
- **用户栈**：用户栈是从最大的合法用户地址开始，向较小的地址增长
- **内核**：最上方的是位内核中的数据和代码保留的，是操作系统驻留在内存的位置

注意：链接器通常会使用地址空间布局随机化（ASLR）来修改堆、共享库和栈的地址，但是会保持三者相对位置不变。

随后加载器会跳转到程序的入口点，到达`_start`函数的地址，然后该函数代用系统启动函数`_libc_start_main`，然后该函数初始化执行环境，然后调用用户层的`main`函数。其中，`_start`定义在系统目标文件`ctrl.o`，`__libc_start_main`定义在`libc.so`中。

5 共享库

静态库具有以下缺点：需要定期维护和更新，并且几乎所有C程序都会使用标准I/O函数，则运行时这些函数的代码会被复制到每个运行进程的文本段中，占用大量的内存资源。

为了解决静态库的问题提出了**共享库（Shared Library）**，它是一个目标模块，不会在产生可执行目标文件时将数据段和代码段复制到可执行目标文件中进行静态链接，而是等到程序要加载时或要运行时才进行链接，我们可以提供最新的共享库，使得可执行目标文件可以直接和最新的共享库在加载或运行时链接，无需重新产生可执行目标文件。共享库由**动态链接器（Dynamic Linker）**加载到任意的内存地址，并和一个在内存中的程序链接起来，该过程称为**动态链接（Dynamic Linking）**。动态链接器本身就是一个共享目标，Linux中为`ld-linux.so`。

共享库的"共享"具有**两层含义**：

- 在任意文件系统中，一个库只有一个`.so`文件，所有引用该共享库的可执行目标文件都共享该`.so`文件中的代码和数据，不像静态库的内容会被复制到可执行目标文件中。
- 在内存中，一个共享库的`.text`节可以被不同正在运行的进程共享。

5.1 加载时动态链接

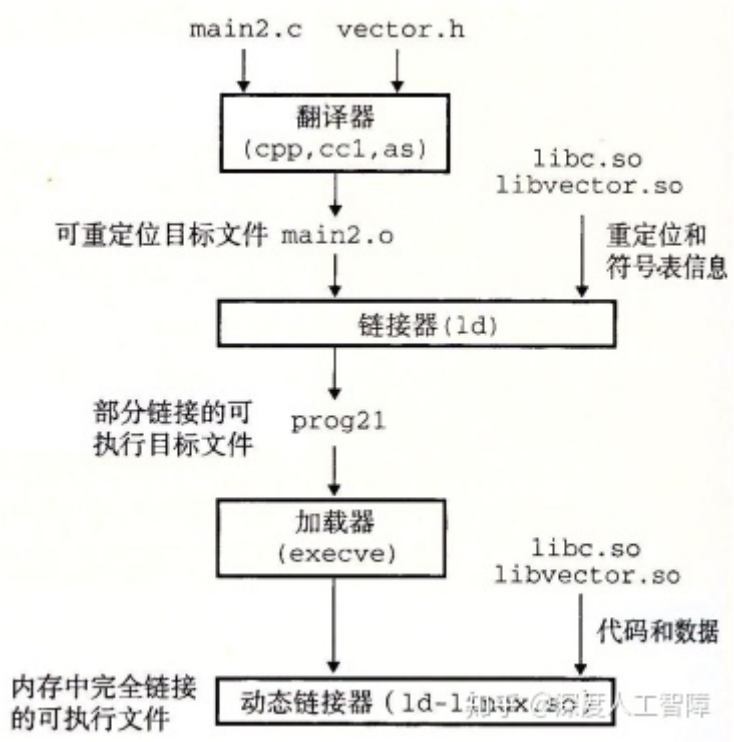
我们可以通过以下形式产生共享库

```
gcc -shared -fpic -o libvector.so addvec.c multvec.c
```

其中，`-shared`指示链接器创建一个共享的目标文件，`-fpic`指示编译器生成与位置无关的代码。然后我们可以通过以下形式利用该共享库

```
gcc -o prog2l main2.c ./libvector.so
```

由此就创建了一个可执行目标文件`prog2l`，其过程如下图所示



- 在创建可执行目标文件时，链接器会复制共享库中的重定位`.rel`和符号表`.symtab`信息，使得运行时可以解析对共享库中代码和数据的引用，由此得到部分链接的可执行目标文件。**注意：**此时没有将共享库的代码和数据节复制到可执行文件中。
- 调用加载器加载部分链接的可执行目标文件时，加载器会在段头部表的引导下，将可执行文件中的数据段和代码段复制到对应的内存位置。
- 加载器可以在`prog21`中发现`.interp`节，其中保存了动态链接器的路径，则加载器会加载和运行这个动态链接器
- 动态链接器会将不同的共享库的代码和数据保存到不同的内存段中
- 动态链接器还会根据共享库在内存的位置，来重定位`prog21`中所有对共享库定义的符号的引用
- 最后加载器将控制权传递给应用程序，此时共享库的位置就固定了，并在程序执行的过程中不会改变。

此时就能在应用程序被加载之后，在运行之前动态链接器加载和链接共享库。

5.2 运行时动态链接

应用程序还可以在它运行时要求动态链接器加载和链接某个共享库。

Linux为动态链接器提供一个接口，使得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);
```

`dlopen`函数可以打开`filename`指定的共享库，并返回句柄指针，而参数`flag`可以用来确定共享库符号解析方式以及作用范围，两个可用`|`相连，包括：

- `RTLD_NOW`：在dlopen返回前，解析出全部没有定义符号，假设解析不出来，则返回NULL
- `RTLD_LAZY`：在dlopen返回前，对于共享库中的没有定义的符号不运行解析，直到执行来自共享库中的代码（仅仅对函数引用有效，对于变量引用总是马上解析）。
- `RTLD_GLOBAL`：共享库中定义的符号可被其后打开的其他库用于符号解析
- `RTLD_LOCAL`：与`RTLD_GLOBAL`作用相反，共享库中定义的符号不能被其后打开的其他库用于重定位，是默认的。

```
#include <dlfcn.h>
void *dlsym(void *handle, char *symbol);
```

该函数返回之前打开的共享库的句柄中`symbol`指定的符号的地址

```
#include <dlfcn.h>
void dlclose(void *handle);
```

用来关闭打开的共享库句柄

```
#include <dlfcn.h>
const char *dlerror(void);
```

如果`dlopen`、`dlsym`或`dlclose`函数发生错误，就返回字符串。

如下图所示的代码示例

code/link/dll.c

```c
#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 = dlerror()) != NULL) {
    addvec = dlsym(handle, "addvec");
    if ((error = dlerror()) != 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;
}
```


code/link/dll.c

该程序就会在运行时动态链接共享库`libvector.so`，然后调用`addvec`函数。

我们可以用以下的编译方式

```
gcc -rdynamic -o prog2r dll.c -ldl
```

其中，`-rdynamic`通知链接器将全部符号加入到动态符号表中，就可以通过使用`dlopen`来实现向后跟踪，`-ldl`表示程序运行时会动态加载共享库。

5.3 位置无关代码

当链接器产生可执行目标文件时，已为目标文件中的数据节和符号分配好了内存地址，如果可执行目标文件有引用共享库中的符号时，就需要假设共享库符号的地址。较早存在**静态共享库（Static Shared Library）**方法，即操作系统会在某个特定的地址中划分一部分，为已知的共享库预留空间，则共享库会被加载到对应的地址空间中，而可执行目标文件就可以在对应的地址空间中找到想要的共享库。但是该方法会造成地址冲突，并造成地址空间的浪费，以及维护的困难。

所以就想能否将共享库加载到任意的内存位置，还能使得可执行目标文件能找到。类似于使用静态库时，链接器会根据重定位表和分配好的内存地址来替换编译时未知的地址，这里可以使用**加载时重定位（Load Time Relocation）**方法，由于编译、汇编和链接时对共享库在内存的位置是未知的，所以可执行目标文件对共享库的符号的引用也用占位符代替，当加载器加载可执行目标文件进行加载时，会调用动态链接加载器将共享库加载到内存中，此时就能根据共享库被加载的内存地址，对可执行目标文件中的占位符进行重定位。但是该方法会对共享库中的指令进行修改，由于指令被重定位后对于每个进程是不同的，所以该共享库无法在多个进程中共享。但是共享库中的数据部分在多个进程中是有自己备份的，所以可以通过该方法来解决。

我们的目的其实就是希望共享的指令部分在装载时不需要因为装载地址的改变而改变，所以实现的基本想法就是把指令中那些需要被修改的部分分离出来，跟数据部分放在一起，这样指令部分就可以保持不变，而数据部分可以在每个进程中拥有一个副本。这种方案就是目前被称为**地址无关代码(PC, Position-independent Code)**的技术。

注意：对于动态库的创建，`-fpic`选择地址无关代码是必须的编译选项。

5.3.1 实例介绍

当你在代码中调用共享库中的函数或全局变量时，编译器、汇编器以及链接器并不知道该函数和全局变量在内存中的位置，只有当可执行目标文件加载时或运行时动态链接共享库，动态链接器将共享库加载到内存时，才知道共享库中的地址。

但是可执行目标文件中的代码段是不可写的，所以无法通过动态链接器对可执行目标文件的代码段进行修改，使其指向共享库的函数或变量的地址；其次，比如动态库A的函数调用了动态库`glibc.so`中定义的`printf`函数时，只有在动态链接器加载了`glibc.so`时，动态库A才能知道`printf`函数所在的内存地址，但是我们也不能对动态库A的代码段进行修改，否则动态库A就无法在各个进程中共享了。

这里链接器使用**全局偏移量表（Global Offset Table，GOT）**和**过程链接表（Procedure Linkage Table，PLT）**来解决这个问题。我们以下方的例子进行介绍

code/link/addvec.c

```c
int addcnt = 0;

void addvec(int *x, int *y,
            int *z, int n)
{
    int i;

    addcnt++;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

code/link/addvec.c

a) addvec.o

code/link/multvec.c

```c
int multcnt = 0;

void multvec(int *x, int *y,
             int *z, int n)
{
    int i;

    multcnt++;

    for (i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}
```

code/link/multvec.c

b) multvec.o

code/link/main2.c

```c
#include <stdio.h>
#include "vector.h"

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);
    return 0;
}
```


code/link/main2.c

我们可以通过以下命令行来产生位置无关代码的共享库 `libvector.so`

```
gcc -shared -fpic -o libvector.so addvec.c multvec.c
```

然后通过以下命令行产生可执行目标文件 `prog`

```
gcc -o prog main2.c ./libvector.so
```

首先，由于可执行目标文件无法对代码段进行修改，所以链接器无法对引用了共享库的函数或变量的地址进行修改，所以链接器在可执行目标文件中的数据段新建了一个数据节 `.got`，用来保存共享库函数的地址，由于数据节是可写的，所以就能在知道共享库函数地址时，对该数据节进行修改，使其指向正确的共享库函数的地址。比如我们这里可以通过 `readelf -a -x .got prog` 来解读可执行目标文件，得到以下 `.got` 数据节

```
Hex dump of section '.got':
 NOTE: This section has relocations against it, but these have NOT been applied to this
dump.
  0x00200fb0 b00d2000 00000000 00000000 00000000 .. .............
  0x00200fc0 00000000 00000000 46060000 00000000 ........F.......
  0x00200fd0 56060000 00000000 00000000 00000000 V...............
  0x00200fe0 00000000 00000000 00000000 00000000 ................
  0x00200ff0 00000000 00000000 00000000 00000000 ................
```

其次，由于链接器无法修改编译器产生的汇编代码，所以无法修改调用共享库的函数的 `call` 指令，所以链接器在可执行目标文件的代码段新建一个 `.plt` 节，然后对所有引用了共享库中的函数都在该数据节中创建一个新函数 `xxx@plt`，然后将代码中调用地址替换成 `call xxx@plt`，所以就能通过函数 `xxx@plt` 来完成对 `.got` 的更新，以及指向正确的地址。

我们这里可以通过 `objdump -dx prog` 来将其转化为汇编代码。首先查看 `main` 函数中对共享库 `libvector.so` 的 `addvec` 函数和共享库 `glibc.so` 的 `printf` 函数的调用

```
000000000000077a <main>:
 77a:   48 83 ec 08             sub    $0x8,%rsp
 77e:   b9 02 00 00 00          mov    $0x2,%ecx
 783:   48 8d 15 9e 08 20 00    lea    0x20089e(%rip),%rdx        # 201028 <z>
 78a:   48 8d 35 7f 08 20 00    lea    0x20087f(%rip),%rsi        # 201010 <y>
 791:   48 8d 3d 80 08 20 00    lea    0x200880(%rip),%rdi        # 201018 <x>
 798:   e8 a3 fe ff ff          callq  640 <addvec@plt>
 79d:   8b 0d 89 08 20 00       mov    0x200889(%rip),%ecx        # 20102c <z+0x4>
 7a3:   8b 15 7f 08 20 00       mov    0x20087f(%rip),%edx        # 201028 <z>
 7a9:   48 8d 35 a4 00 00 00    lea    0xa4(%rip),%rsi        # 854
<_IO_stdin_used+0x4>
 7b0:   bf 01 00 00 00          mov    $0x1,%edi
 7b5:   b8 00 00 00 00          mov    $0x0,%eax
 7ba:   e8 91 fe ff ff          callq  650 <__printf_chk@plt>
 7bf:   b8 00 00 00 00          mov    $0x0,%eax
 7c4:   48 83 c4 08             add    $0x8,%rsp
 7c8:   c3                      retq
 7c9:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)
```

可以发现，这里将对 `addvec` 函数和对 `prinrf` 函数的调用转化为了对 `addvec@plt` 和对 `__printf_chk@plt` 函数的调用，这两个函数就是在 `.plt` 节中定义的，而 `.plt` 节中的内容如下所示

```
 Disassembly of section .plt:

0000000000000630 <.plt>:
 630:   ff 35 82 09 20 00       pushq  0x200982(%rip)        # 200fb8
<_GLOBAL_OFFSET_TABLE_+0x8>
 636:   ff 25 84 09 20 00       jmpq   *0x200984(%rip)        # 200fc0
<_GLOBAL_OFFSET_TABLE_+0x10>
 63c:   0f 1f 40 00             nopl   0x0(%rax)

0000000000000640 <addvec@plt>:
 640:   ff 25 82 09 20 00       jmpq   *0x200982(%rip)        # 200fc8 <addvec>
 646:   68 00 00 00 00          pushq  $0x0
 64b:   e9 e0 ff ff ff          jmpq   630 <.plt>

0000000000000650 <__printf_chk@plt>:
 650:   ff 25 7a 09 20 00       jmpq   *0x20097a(%rip)        # 200fd0
<__printf_chk@GLIBC_2.3.4>
 656:   68 01 00 00 00          pushq  $0x1
 65b:   e9 d0 ff ff ff          jmpq   630 <.plt>
```

这里首先需要介绍一个**位置无关代码的性质**：我们将可以加载而无需重定位的代码称为位置无关代码。由于我们无论在内存什么位置加载该目标模块（包括共享目标模块），数据段与代码段的距离总是保持不变的。所以我们可以让处于代码段的 `plt` 函数通过距离常量来访问处于数据段中对应的 `got` 中保存的地址。

比如上面我们调用 `addvec@plt` 函数时，会执行 `0x640` 处的 `jmpq *0x200982(%rip)` 指令， 这里的 `0x200982` 就是上面所说的距离常量，用来指向特定的 `got` 项，这里可以得到访问的 `got` 项的地址为 `0x200982+0x646=0x200fc8`，而该地址对应的 `got` 内容如下所示

```
0x00200fc0 00000000 00000000 46060000 00000000 ........F.......
```

根据小端法可以只为 `0x664`，即跳转回到下一条指令，然后调用 `.plt` 函数

```
0000000000000630 <.plt>:
 630:   ff 35 82 09 20 00       pushq  0x200982(%rip)        # 200fb8
<_GLOBAL_OFFSET_TABLE_+0x8>
 636:   ff 25 84 09 20 00       jmpq   *0x200984(%rip)        # 200fc0
<_GLOBAL_OFFSET_TABLE_+0x10>
 63c:   0f 1f 40 00             nopl   0x0(%rax)
```

其中第一条指令是将地址`0x200982+0x636=0x200fb8`作为参数压入栈中，而第二条指令是跳转到`0x200984+0x63c=0x200fc0`处保存的地址，我们通过上面可以看到，在未运行可执行目标文件时，该地址的值为`0`，而当运行了可执行目标文件时，该地址的值会修改到动态链接器中的`_dl_runtime_resolve`函数，来进行地址解析，查看共享库的`addvec`被加载到什么内存地址。那该函数是如何知道要获得哪个函数的地址，以及要将函数地址保存到哪个`got`项呢?

我们观察可执行目标文件中以下共享库的函数

```
0000000000000640 <addvec@plt>:
 640:   ff 25 82 09 20 00       jmpq   *0x200982(%rip)        # 200fc8 <addvec>
 646:   68 00 00 00 00          pushq  $0x0
 64b:   e9 e0 ff ff ff          jmpq   630 <.plt>

0000000000000650 <__printf_chk@plt>:
 650:   ff 25 7a 09 20 00       jmpq   *0x20097a(%rip)        # 200fd0
<__printf_chk@GLIBC_2.3.4>
 656:   68 01 00 00 00          pushq  $0x1
 65b:   e9 d0 ff ff ff          jmpq   630 <.plt>
```

可以发现每个函数的第一条指令是跳转到对应的`got`项，而对应的`got`项被初始化为下一条指令的地址，当`got`项没有被修改时，就自动跳转到下一条指令。而第二条指令在不同函数中是不同的，其实对应的是`.rela.plt`的索引

```
Relocation section '.rela.plt' at offset 0x5e8 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000200fc8  000300000007 R_X86_64_JUMP_SLO 0000000000000000 addvec + 0
000000200fd0  000500000007 R_X86_64_JUMP_SLO 0000000000000000 __printf_chk@GLIBC_2.3.4
+ 0
```

其中，`offset`表示对应的`got`项的地址，`Sym.Name`就是函数的名字。所以动态链接器通过索引值和`.rela.plt`数据组就能确定要定位哪个动态库函数，以及将其内存地址保存到哪个`got`项。

当动态链接后的`addvec`函数的内存地址保存到对应的`got`项时，下次再调用`addvec`函数时，就能直接通过该`got`项直接获得`addvec`函数的内存地址。

我们可以发现，第一次调用共享库的函数时，对应的`xxx@plt`函数并不会跳转到正确的函数地址，而是调用动态链接器来获得函数的地址，然后将其保存到`got`项中，下一次再运行时，才会跳转到正确的函数地址，该方法称为**延迟绑定（Lazy Binding）**，只有共享库的函数要用时，才会重定位它的地址，否则不会，由此防止可执行目标文件加载时需要对大量的共享库的地址进行重定位。

综上所述：当函数要访问共享库中的函数时，实现执行`call xxx@plt`，访问该函数的封装函数，然后该`plt`函数会访问对应的`got`项，如果`got`项被赋值为对应的`xxx`函数的地址，则会调用该函数，否则会调用`.plt[0]`中的动态链接器，来定位`xxx`函数的内存地址，然后将其保存到对应的`got`项中。

6 库打桩

Linux链接器支持**库打桩（Library Interpositioning）**技术，允许你截获对共享库函数的调用，替换成自己的代码。**基本思想**为：创建一个与共享库函数相同函数原型的包装函数，是不不同的库打桩技术，使得系统调用包装函数，而不是调用目标函数。

6.1 编译时打桩

code/link/interpose/int.c

```
1  #include <stdio.h>
2  #include <malloc.h>
3
4  int main()
5  {
6      int *p = malloc(32);
7      free(p);
8      return(0);
9  }
```


code/link/interpose/int.c

我们以以上代码为例，说明编译时打桩技术，替换动态库 `libc.so` 的 `malloc` 和 `free` 函数的调用。

首先，我们可以定义一个本地的头文件 `malloc.h`，如下所示

```
//本地malloc.h
#define malloc(size) mymalloc(size)
#define free(ptr) myfree(ptr)
```

然后在编译 `int.c` 时，使用 `-I.` 编译选项，使得预处理器首先从本地查找 `malloc.h` 文件，由此就能将共享库的 `malloc` 和 `free` 函数替换成我们自己的 `mymalloc` 混合 `myfree` 函数。

而我们需要自己实现 `mymalloc` 和 `myfree` 函数，其中需要调用原始的 `malloc.h`，所以需要先将该函数进行编译，所以创建以下文件

code/link/interpose/mymalloc.c

```
1  #ifdef COMPILETIME
2  #include <stdio.h>
3  #include <malloc.h>
4
5  /* malloc wrapper function */
6  void *mymalloc(size_t size)
7  {
8      void *ptr = malloc(size);
9      printf("malloc(%d)=%p\n",
10             (int)size, ptr);
11     return ptr;
12 }
13
14 /* free wrapper function */
15 void myfree(void *ptr)
16 {
17     free(ptr);
18     printf("free(%p)\n", ptr);
19 }
20 #endif
```


code/link/interpose/mymalloc.c

所以我们可以通过以下代码得到该函数的可重定位目标文件 `mymalloc.o`

```
gcc -DCOMPILETIME -c mymalloc.c
```

然后在本地的 `malloc.h` 中给出包装函数的函数原型，即

code/link/interpose/malloc.h

```
#define malloc(size) mymalloc(size)
#define free(ptr) myfree(ptr)

void *mymalloc(size_t size);
void myfree(void *ptr);
```


code/link/interpose/malloc.h

然后就可以通过以下命令行进行编译时打桩

```
gcc -I. -o intc int.c mymalloc.o
```

此时，由于`-I.`编译选项，对于`int.c`中的`malloc.h`，预处理器会首先从本地搜索`malloc.h`文件，而在本地`malloc.h`文件中，对`malloc`和`free`函数重新包装成`mymalloc`和`myfree`函数，而这两个函数在之前编译好的`mymalloc.o`可重定位目标文件中，此时就完成了编译时打桩。

综上所述：想要在编译时打桩，意味着要通过`#define`来使用预处理器将目标函数替换成包装函数。

6.2 链接时打桩

Linux静态链接器也支持使用`--wrap f`标志进行链接时打桩，此时会将符号`f`解析为`__wrap_f`，而将对`__real_f`符号的引用解析为`f`， 意味着原始对函数`f`的调用，还会替换成对`__wrap_f`函数的调用，而通过`__real_f`函数来调用原始函数`f`。

我们定义以下函数

code/link/interpose/mymalloc.c

```
#ifdef LINKTIME
#include <stdio.h>

void *__real_malloc(size_t size);
void __real_free(void *ptr);

/* malloc wrapper function */
void *__wrap_malloc(size_t size)
{
    void *ptr = __real_malloc(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void __wrap_free(void *ptr)
{
    __real_free(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```


code/link/interpose/mymalloc.c

然后我们可以同时进行编译

```
gcc -DLINKTIME -Wl,--wrap,malloc -Wl,--wrap,free -o intl int.c mymalloc.c
```

也可以分开编译

```
gcc -DLINKTIME -c mymalloc.c
gcc -c int.c
gcc -Wl,--wrap,malloc -Wl,--wrap,free -o intl int.o mymalloc.o
```

其中，`-Wl`表示传递链接器参数，而这些参数通过`,`相连。

由此，`int.c`中对`malloc`和`free`函数的调用，会变成对`__wrap_malloc`和`__wrap_free`函数的调用。

综上所述：想要在链接时打桩，意味着在对可重定位目标文件的符号进行解析时，进行替换。

6.3 运行时打桩

想要在运行时进行打桩，意味着是对共享库的函数进行打桩，这里使用动态链接器提供的`LD_PRELOAD`环境变量，通过该变量设置共享库路径列表，执行可执行目标文件时，动态链接器就会先搜索`LD_PRELOAD`共享库。

我们可以定义以下函数

code/link/interpose/mymalloc.c

```c
#ifdef RUNTIME
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

/* malloc wrapper function */
void *malloc(size_t size)
{
    void *(*mallocp)(size_t size);
    char *error;

    mallocp = dlsym(RTLD_NEXT, "malloc"); /* Get address of libc malloc */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    char *ptr = mallocp(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void free(void *ptr)
{
    void (*freep)(void *) = NULL;
    char *error;

    if (!ptr)
        return;

    freep = dlsym(RTLD_NEXT, "free"); /* Get address of libc free */
    if ((error = dlerror()) != NULL) {
        fputs(error, stderr);
        exit(1);
    }
    freep(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```


code/link/interpose/mymalloc.c

然后通过以下命令行将其编译成共享库

```
gcc -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl
```

然后在运行时指定环境变量`LD_PRELOAD`

```
gcc -o intr int.c
./LD_PRELOAD="./mymalloc.so" ./intr
```

此时运行到`malloc`和`free`函数时，就会调用动态链接器搜索该符号的定义，此时会先搜索`LD_PRELOAD`指定的共享库，而`mymalloc.so`中定义了这两个符号，所以就替换了这两个函数的具体实现。**注意：**如果想要调用原始的定义，就需要用运行动态链接的方式，通过指定`dlsym`的参数为`RTLD_NEXT`，来在后续的共享库中获得`malloc`的定义。

参考：

符号解析与重定位 - yooooooo - 博客园

dlopen 方式调用 Linux 的动态链接库

地址无关码 - yooooooo - 博客园

聊聊Linux动态链接中的PLT和GOT（１）--何谓PLT与GOT*运维*海枫的专栏-CSDN博客