Chapter 7

2024年12月9日 10:39

1.理解从源文件到可执行目标文件的过程：

源文件->可重定位目标文件(.o)->可执行目标文件

预处理器：main.c->main.i　GCC 会处理源代码中的 **宏定义**、**文件包含（#include）**、**条件编译（#ifdef）**指令

编译器：main.i->main.s 翻译为汇编语言文件

汇编器：main.s->main.o 从汇编代码到机器语言（生成的是可重定位目标文件）

linker：将多个可重定位目标文件组合，得到可执行目标文件（linux中没有.exe后缀）

linux中判断文件是否可执行是基于文件的内部格式（下文的ELF文件）

3. 与 Windows 的对比

特性	Linux	Windows
是否需要后缀	不需要，可执行权限和文件格式决定	必须使用 `.exe` 后缀
判断可执行性	查看文件权限和文件格式（如 ELF 文件）	依赖后缀 `.exe`
运行方式	直接运行文件名（如 `./my_program`）	双击或运行 `.exe` 文件

2.目标文件

三种目标文件：可重定位目标文件、可执行目标文件、共享目标文件（是特殊的可重定位目标文件）

目标文件按照特定的文件格式组织，这使得linux可以区分它们（我们主要学ELF文件格式）

一个目标文件中含有若干个目标模块

左图为可重定位目标文件，右图可执行目标文件

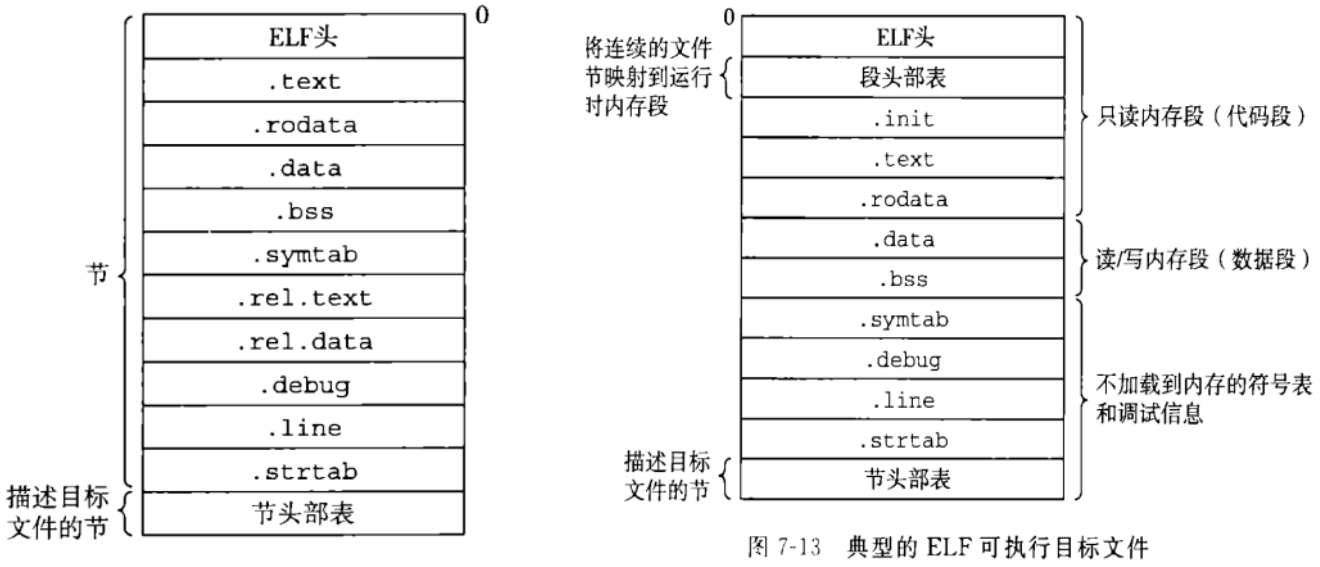


图 7-13　典型的 ELF 可执行目标文件

以左图为例：

1）ELF头：包括机器类型（machine）、elf头的大小（size of this header)、目标文件类型(Type)等信息

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1060
  Start of program headers:          64 (bytes into file)
  Start of section headers:          13968 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
```

2）.text：已编译程序的机器代码

3）.rodata:只读数据，比如printf语句中的字符串和switch的跳转表

4）.data:已初始化的全局和静态变量（不要求静态变量是全局的，局部也可以；P468）

5）.bss 未初始化的全局和静态变量以及被初始化为0的全局和静态变量

注：局部变量不在.data和.bss中，存在函数栈帧中

6）.symtab:符号表（下文展开说明）

7）.rel.text:该目标文件与其他文件组合时需要修改这些位置

8）.rel.data:被模块引用或定义的所有全局变量的重定位信息

可以看到可执行目标文件的rel.text和.rel.data消失了

链接器将.o中.text和.data节整合到一起时，会对整合后的.text和.data进行重定位，.text和.data节重定位时需要依赖.rel.text和.rel.data中的信息，一旦重定位结束后，这两个节的使命就完成了，所以可执行目标文件中不存在rel.text和rel.data节。

9）.debug:调试符号表，条目是程序中定义的局部变量和类型定义，定义和引用的全局变量和原始的源文件

-g编译才会得到这张表

10）.line:-g编译才会得到这张表 是源文件行号和.text机器指令的映射

11）.strtab:这个节也在节头部表中得以描述

内容包括.symtab和.debug中的符号表以及节头部表中的节名字

是（以null结尾的）字符串的序列

注：可执行目标文件增加了段头部表

段头部表的作用：用来辅助加载程序。当程序加载到内存时，需要将硬盘上的可执行目标文件，搬到“运行地址（虚拟内存的地址0x08400XXXX）”所指定的内存位置，段头部表中的作用就是用来辅助加载程序的。

对于linker而言，linker将可重定位目标文件视为节头部表描述的一系列section的集合

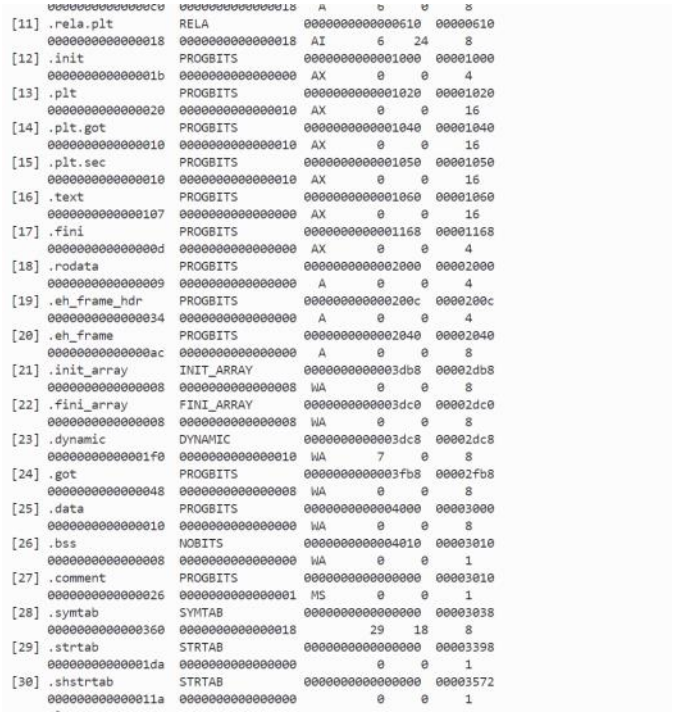
对于loader而言，loader将可执行目标文件视为段头部表描述的一系列segment的集合

目标文件需要链接器做进一步处理，所以一定有Section Header Table；可执行文件需要加载运行，所以一定有Program Header Table；而共享库既要加载运行，又要在加载时做动态链接，所以既有Section Header Table又有Program Header Table。

下面分别详细说明节头部表(section header table)和段头部表（program header table)

Section header table:

不同节（即上文所说的.data、.bss等）的位置和大小是节头部表描述的



```
[11] .rela.plt         RELA            0000000000000610  00000610
     0000000000000018  0000000000000018  AI       6    24     8
[12] .init             PROGBITS        0000000000001000  00001000
     000000000000001b  0000000000000000  AX       0     0     4
[13] .plt              PROGBITS        0000000000001020  00001020
     0000000000000020  0000000000000010  AX       0     0     16
[14] .plt.got          PROGBITS        0000000000001040  00001040
     0000000000000010  0000000000000010  AX       0     0     16
[15] .plt.sec          PROGBITS        0000000000001050  00001050
     0000000000000010  0000000000000010  AX       0     0     16
[16] .text             PROGBITS        0000000000001060  00001060
     0000000000000107  0000000000000000  AX       0     0     16
[17] .fini             PROGBITS        0000000000001168  00001168
     000000000000000d  0000000000000000  AX       0     0     4
[18] .rodata           PROGBITS        0000000000002000  00002000
     0000000000000009  0000000000000000  A        0     0     4
[19] .eh_frame_hdr     PROGBITS        000000000000200c  0000200c
     0000000000000034  0000000000000000  A        0     0     4
[20] .eh_frame         PROGBITS        0000000000002040  00002040
     00000000000000ac  0000000000000000  A        0     0     8
[21] .init_array       INIT_ARRAY      0000000000003db8  00002db8
     0000000000000008  0000000000000008  WA       0     0     8
[22] .fini_array       FINI_ARRAY      0000000000003dc0  00002dc0
     0000000000000008  0000000000000008  WA       0     0     8
[23] .dynamic          DYNAMIC         0000000000003dc8  00002dc8
     00000000000001f0  0000000000000010  WA       7     0     8
[24] .got              PROGBITS        0000000000003fb8  00002fb8
     0000000000000048  0000000000000008  WA       0     0     8
[25] .data             PROGBITS        0000000000004000  00003000
     0000000000000010  0000000000000000  WA       0     0     8
[26] .bss              NOBITS          0000000000004010  00003010
     0000000000000008  0000000000000000  WA       0     0     1
[27] .comment          PROGBITS        0000000000000000  00003010
     0000000000000026  0000000000000001  MS       0     0     1
[28] .symtab           SYMTAB          0000000000000000  00003038
     0000000000000360  0000000000000018          29    18     8
[29] .strtab           STRTAB          0000000000000000  00003398
     00000000000001da  0000000000000000           0     0     1
[30] .shstrtab         STRTAB          0000000000000000  00003572
     000000000000011a  0000000000000000           0     0     1
```

Program header table:

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000628 0x0000000000000628  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x0000000000000181 0x0000000000000181  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x00000000000000ec 0x00000000000000ec  R      0x1000
  LOAD           0x0000000000002db8 0x0000000000003db8 0x0000000000003db8
                 0x000000000000025c 0x0000000000000260  RW     0x1000
```

Offset:该段在文件中的偏移位置

VirtAddr:该段在内存中的虚拟地址

PhysAddr:该段的物理地址

FileSiz:表示当前段在文件中占多少字节

MemSiz:表示当前段在内存中占多少字节 （和filesiz大小可能不一样：

一方面，BSS 段在可执行文件中不包含实际数据，因此它的 FileSiz 为 0。但在程序运行时，操作系统会为其分配内存；另一方面，与内存中的对齐要求也有关）

Flags:读写权限

Align:对齐要求

Offset mod align=vaddr mod align

3.符号表与符号解析

（1）符号：

每个可重定位目标文件都有一个符号表，（书中对于目标模块和目标文件不加以区分）故下文也是

对于一个可重定位目标模块m而言（或者说对于一个可重定位目标文件文件而言），有三种符号：

❖ 使用 static 修饰的全局变量or函数的作用范围是 **限定在定义它的源文件内**

- 该模块自身定义的可被其他模块引用的符号：非静态的C函数和全局变量 称**全局符号**
- 其他模块定义的被该模块已引用的符号：**外部符号**，对应其他模块中的非静态的C函数和全局变量
- 该模块自身定义且只能自身引用的局部符号：对应该模块中的static的函数和static全局变量（由于是static属性，所以其他模块无法引用）称**本地符号** 事实上在后文中可以看到这一点也包括局部static变量
- 补充：在其他模块定义的非静态的变量和函数 m模块通过extern声明，但只声明不引用并不会导致对应符号出现在符号表中，只有在引用之后才会

1. **在一个文件中定义全局变量：**

```c
// file1.c
int global_var = 42;  // 非静态全局变量
```

2. **在另一个文件中引用这个全局变量：**

```c
// file2.c
extern int global_var;  // 外部声明，告诉编译器该变量在其他文件中定义

void some_function() {
    printf("%d\n", global_var);  // 可以访问 file1.c 中的 global_var
}
```

☺ 本地链接器符号和本地程序变量的区别：

教材P468:“本地链接器符号和本地程序变量不同”：本地链接器符号就是上方所说的三种符号，对应的那几种变量；本地程序变量就是局部变量。这段话意思是非静态局部变量对应的符号并不在符号表中

☺ 由static修饰的本地过程变量并不在栈中管理

P468的例子：局部变量如果被static修饰，就不在栈中管理了；而在.data或.bss中为每个定义分配空间，并在符号表中创建唯一名字的本地链接器符号（也就是说局部的静态变量在符号表中也有对应的符号）且即使是变量名字一样，但只要在不同函数，就有不同的符号名

（2）符号表：符号表由汇编器构造

符号表在.symtab中

例:

```c
#include<stdio.h>
int x=0;
int main()
{
        static int x=0;
        printf("haha\n");
        printf("haha\n");
        return 0;
}
```

```
ubuntu@liangshuyi2300013147:~/test$ nm test.o
0000000000000000 T main
                 U puts
0000000000000000 B x
0000000000000004 b x.0
```

全局变量x 在符号表中有对应的符号（这是上文所说的第一种符号）

而下面的局部变量x也有符号，因为它是static属性的

符号表的条目：每个条目是下面这样的一个struct

code/link/elfstructs.c

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

code/link/elfstructs.c

目标文件中的符号表

.symtab 节记录符号表信息，是一个结构数组

- 符号表（symtab）中每个条目的结构如下：

函数名在text节中

变量名在data节或bss节中

```c
typedef struct {
    Elf32_Word  st_name;   /*符号对应字符串在strtab节中的偏移量*/
    Elf32_Addr  st_value;  /*在对应节中的偏移量，可执行文件中是虚拟地址*/
    Elf32_Word  st_size;   /*符号对应目标所占字节数*/函数大小或变量长度
    unsigned char st_info; /*符号对应目标的类型：数据、函数、源文件、节*/
                           /*符号类别：全局符号、局部符号、弱符号*/
    unsigned char st_other; /*符号的可见性*/
    Elf32_Half  st_shndx; /*符号所在节在节头表中的索引*/
} Elf_Sym;
```

其他情况：ABS表示不该被重定位；UND表示未定义；COM表示未初始化数据（.bss），此时，value表示对齐要求，size给出最小大小

name是字符串表中的字节

符号所属的节or伪节：

符号表中的每个符号都被分配到某一个节中

.bss：未初始化的静态变量（包括局部未初始化的静态变量），初始化为0的全局或静态变量

```c
#include<stdio.h>
static int x;
int main()
{
        return 0;
}
```

```c
#include<stdio.h>
int main()
{
        static int x;
        return 0;
}
```

```
0000000000004014 b x
```

```
0000000000004014 b x.0
```

如图，全局和局部未初始化的静态变量均属于.bss(b代表.bss)

.data：已初始化的全局变量，全局静态变量，局部静态变量

```c
#include<stdio.h>
static int y=1;
int main()
{
        static int x=2;
        return 0;
}
```

```c
#include<stdio.h>
int y=1;
int main()
{
        //printf("%d\n",y);
        return 0;
}
```

```
0000000000000000 T main
0000000000000004 d x.0
0000000000000000 d y
```

```
0000000000000000 T main
0000000000000000 D y
```

三个伪节： 伪节在节头部表中没有条目，且只有可重定位目标文件才有

UNDEF：表示未定义的符号（即在这个目标模块中引用，定义在另一个模块的符号）

如下面的y 前面的U表示UNDEF

```c
#include<stdio.h>
extern int y;
int main()
{
        printf("%d\n",y);
        return 0;
}
```

```
0000000000000000 T main
                 U printf
                 U y
```

COMMON：未初始化的全局变量

ABS：不该被重定位的符号

（3）符号解析

linker解析符号引用：将每个引用与输入的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

分以下两种情况：

a.可重定位目标文件m引用变量x，且x对应的符号就在该目标模块（目标文件）的符号表之中：

b.可重定位目标文件m引用的变量x，但变量x对应的符号不在m目标文件的符号表中：

编译器会假设该符号在其他目标模块的符号表中定义，生成一个链接器符号表条目交给链接器处理

如下图：

```
ubuntu@liangshuyi2300013147:~/test$ nm test.o
                 U g
0000000000000000 T main
                 U puts
0000000000000000 B x
0000000000000004 b x.0
```

g函数在另一个源文件中定义，test源文件中引用了g,故生成的test可重定位目标文件中符号表中有g，但g这个条目前面是空的

```
0000000000001160 t frame_dummy
00000000000011a0 T g
0000000000001169 T main
                 U printf@GLIBC_2.2.5
                 U puts@GLIBC_2.2.5
00000000000010e0 t register_tm_clones
0000000000004014 B x
0000000000004018 b x.0
```

linker处理之后，得到的可执行目标文件中符号表中g所在条目前面不再空白了，有了那串数字（至于这串数字的含义就不再细说了）

复杂的情况是：多个目标模块可能定义了同名的全局符号

✸ 注：对于上面所说的全局符号、外部符号、本地符号 只有全局符号、外部符号会有这种问题

本地符号不参与讨论：因为本地符号只在单个目标文件内可见，不会被其他目标文件引用。因此，无论它是定义的还是引用的，都不需要考虑不同目标文件之间的冲突问题。

链接器处理多重定义的全局符号的规则：

1.强弱符号：

已初始化的全局变量名、全局函数名是强符号

未初始化的全局变量名，是弱符号（初始化为0也是初始化，属于强符号）即上文所说的属于common节中的符号

未定义的函数貌似不出现在符号表中

2.三条规则：

a.强符号只能有一个：不允许多个同名的强符号（这里的同名就是指名字一样，类型不同名字一样也是同名的强符号）

假如我在test.c中定义全局变量x并初始化，在test2.c中再定义并初始化，它们链接时报错

```
/usr/bin/ld: /tmp/cckFJsJb.o:(.data+0x0): multiple definition of `x'; /tmp/ccDxPfYk.o:(.bss+0x0): first defined here
collect2: error: ld returned 1 exit status
```

b.强符号和弱符号同时出现，选强符号

我在test.c中定义全局变量x并初始化，在test2.c中再定义但并不初始化

就选择test.c中变量对应的符号

如下图：x前面是D 表明x是在.data节中定义的，是已经初始化的 属于强符号

```
00000000000010c0 t register_tm_clones
0000000000004010 D x
```

c.只有弱符号，随机选一个弱符号

```c
#include<stdio.h>
int x;
void f()
{
        x=0;
}
```

test1.c

```c
#include<stdio.h>
void f(void);
int y=15212;
int x=15213;
int main()
{
        f();
        printf("%d\n",x);
        return 0;
}
```

与链接有关的错误：

如上图，在test2.c中全局变量x=15213,然而test中函数f没有定义（可以认为是弱符号，事实上没有定义的函数不会出现在符号表中），在test1.c中f有定义，是强符号，所以当二者组合在一起时，相当于执行了test1.c中的f函数，而f中的"x=0"语句中x变量对应的是test2.c被初始化为15213的变量x，所以原先值为15213的变量x的值现在变为0，覆盖了那块内存上存储的值

```c
#include <stdio.h>
int d=100;
int x=200;
void p1(void);
int main()
{
   p1();
   printf( "d=%d,x=%d\n" ,d,x);
   return 0;
}
```

main.c

p1.c

```c
double d;

void p1()
{
   d=1.0;
}
```

d最终输出结果如何?

单纯编译p1.c的时候，会认为把1.0给double d（汇编也是浮点指令）。但最终链接后，1.0作为浮点数，要放到int d中，因为main.c中d是强符号。具体如下：

p1执行后d和x处内容是什么？

	0	1	2	3
&x	00	00	F0	3F
&d	00	00	00	00

以上图为例：

d和x的地址是挨着的（差四位）

红色加粗字体为强符号，蓝色加粗字体为弱符号

永远可以先从main开始入手

gcc -fcommon test.c test1.c -o all命令执行之后，在main中调用了p1,由于p1既有强符号又有弱符号，所以选择强符号（即p1.c中对p1的定义），而p1.c中p1函数的语句是“d=1.0”，d既有强符号又有弱符号，所以选择强符号（即main.c中的d),但注意这里把main.c中的d视为double类型来看；所以最后结果冲掉了&x和&d的值

😐 理解：用以赋值的数到底是什么类型的机器数，由本来的c来决定（如double），因为这是在汇编及之前就完成的。而真正赋值给的对象（即修改了哪块内存），是强符号决定的（同一名称的符号若都是弱符号，则任选其一）

（4）与静态库链接

静态库

静态库：将所有目标模块打包在一起，可以像一个可重定位目标文件一样，作为linker的输入

函数可以被编译为独立的目标模块，

将所有相关的目标模块（.o）打包为一个单独的库文件（.a）【包含了许多.o文件】，称为静态库文件，也称存档文件（archive）

– 在构建可执行文件时只需指定库文件名，链接器会自动到库中寻找那些应用程序用到的目标模块，并只把用到的模块从库中拷贝出来

– 在gcc命令行中无需明显指定C标准库libc.a(默认库)

这样做的好处是：

另一种方法是把所有函数放在一起，生成一个目标模块.o

- 可执行文件不必包含所有标准函数集合的一个副本，减少了可执行文件在磁盘和内存占用的空间
- 对一个标准函数的改变，只需要重新编译这个函数，产生对应的目标模块再放入静态库中即可，节省时间

使用静态库解析引用

以gcc test1.c test2.c /usr/lib/x86_64-linux-gnu/libc.a为例（libc.a是标准库）：

编译器驱动程序命令行从左到右按照顺序扫描可重定位目标文件和存档文件

维护三个集合：

E：可重定位目标文件的集合

U：未解析的符号（即引用了但没有定义的符号）

比如一个main.c中使用了printf函数，但是在自身的符号表中没有对printf符号的定义（在标准库的符号表中有定义）

D：当前**已被加入到E**的所有目标文件中**定义符号**的集合

扫描过程中，

如果文件是目标文件，则直接将其加到E中，并相应修改U和D（原先U中的符号可能要剔除，加入D中）

如果文件是存档文件，比如上文的/usr/lib/x86_64-linux-gnu/libc.a，匹配U中未解析的符号和存档文件成员定义的符号，并修改U和D

如果匹配到了，就把这个存档文件成员加入E中，否则就简单地丢弃

最后，若U非空，linker报错

从左到右的顺序带来了问题：

引用符号应该在定义符号之前，否则，已经扫描过了定义该符号的目标模块，但是引用还没有出现，等到引用出现之后不一定能再扫到这个符号的定义了

解决方案：

1.库放在最后

2.库之间，如果不独立，需要使得对于每个存档文件成员外部引用的符号s，至少有一个s的定义再引用之后

简单来说，是一个递归的嵌套关系，最内层的放最后

比如：a b c d, a引用用了b和c中的符号，b和c又引用了d中符号，d相当于嵌套的最内层，d放最后，然后b和c放在a之后

4.重定位

理解：重定位到底在干什么？

假设有下面这样的例子：

test1.c

```c
#include<stdio.h>
extern int y;
extern void f();
int main()
{
        f();
        static int x=2;
        printf("%d\n",y);
        return 0;
}
```

test.c

```c
#include<stdio.h>
void f()
{
        printf("haha\n");
}
int y=15212;
int x=15213;
```

Test1在链接前得到的test1.o像下面这样

```
0000000000000000 <main>:
   0: f3 0f 1e fa           endbr64
   4: 55                    push   %rbp
   5: 48 89 e5              mov    %rsp,%rbp
   8: b8 00 00 00 00        mov    $0x0,%eax
   d: e8 00 00 00 00        call   12 <main+0x12>
      e: R_X86_64_PLT32 f-0x4
  12: 8b 05 00 00 00 00     mov    0x0(%rip),%eax        # 18 <main+0x18>
      14: R_X86_64_PC32 y-0x4
  18: 89 c6                 mov    %eax,%esi
  1a: 48 8d 05 00 00 00 00  lea    0x0(%rip),%rax        # 21 <main+0x21>
      1d: R_X86_64_PC32 .rodata-0x4
  21: 48 89 c7              mov    %rax,%rdi
  24: b8 00 00 00 00        mov    $0x0,%eax
  29: e8 00 00 00 00        call   2e <main+0x2e>
      2a: R_X86_64_PLT32  printf-0x4
  2e: b8 00 00 00 00        mov    $0x0,%eax
  33: 5d                    pop    %rbp
  34: c3                    ret
```

d: call 12<main+0x12> 此时f这个符号定义在另一个目标模块中，所以这里还无法跳到正确的位置执行函数f

linker重定位之后便得到：

```
0000000000001183 <main>:
    1183:   f3 0f 1e fa             endbr64
    1187:   55                      push   %rbp
    1188:   48 89 e5                mov    %rsp,%rbp
    118b:   b8 00 00 00 00          mov    $0x0,%eax
    1190:   e8 d4 ff ff ff          call   1169 <f>
    1195:   8b 05 75 2e 00 00       mov    0x2e75(%rip),%eax
    119b:   89 c6                   mov    %eax,%esi
    119d:   48 8d 05 65 0e 00 00    lea    0xe65(%rip),%rax
    11a4:   48 89 c7                mov    %rax,%rdi
    11a7:   b8 00 00 00 00          mov    $0x0,%eax
    11ac:   e8 bf fe ff ff          call   1070 <printf@plt>
    11b1:   b8 00 00 00 00          mov    $0x0,%eax
    11b6:   5d                      pop    %rbp
    11b7:   c3                      ret
```

这时便可以正确的执行函数f的指令了

链接器完成符号解析之后就要重定位

重定位主要是两个方面：重定位节和符号定义，重定位节中的符号引用

（1）重定位节和符号定义：

将所有可重定位目标文件相同类型的节合并为聚合节，并赋予这个聚合节新的内存地址。

（2）重定位节中的符号引用

a.重定位条目：

上文可重定位目标文件中有.rel.data节和.rel.text节，代码和数据重定位条目分别存在这两个节中，（如上文所说，当重定位完成后，便不再需要这两个节了，所以可执行目标文件中没有这两个节）

可重定位条目的结构：

```
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

Type:链接器重定位符号引用的方式，linux定义了32中方式，我们学两种

Offset:表示将要被修改的操作数相对于自身所在节的偏移量，比如上例中f在.text节中，f对应的可重定位条目的offset指的是f在.text节中的偏移量

如果得到.text的起始地址 加上offset就可以得到f引用的运行时地址

Addend:上例中addend是-0x4 原因在于 要修改的操作数地址（refaddr）和该操作数所在指令的下一条指令地址（即PC）相差4字节。因为操作数是4字节的，填充操作数之后，才是下一条指令的地址（PC），所以做差之后，要补偿操作数占用的4字节地址回去。 即通过知道call指令的下一条指令的地址，和符号引用的地址（refaddr），我们可以求出addend

b.重定位符号引用的算法

只说怎么做题

- 重定位PC 相对引用

  还是用上面的例子说明

  对于f来说，ADDR（s)即.text的地址，指的是test1中的main函数在实际的可执行文件中，位于内存的实际地址

  ADDR（r.symbol)指的是f符号引用在实际可执行文件中，位于内存的实际地址

  从上图可以看到 （要在重定位之后的图中看）

  main的起始地址为1183 ADDR(s)=1183; f在内存中的起始地址是1169（没放图）ADDR(r.symbol)=1169

  公式来啦：符号引用的实际内存地址（refaddr)=ADDR(s)+offset √

1183+e=1191 即上例中call指令的第二个字节

重定位更新这个符号引用：更新为：修改后的操作数=ADDR(r.symbol)+r.addend-refaddr √

上例中更新为：1169+(-0X4)-1191=d4

且还有：call的下一条指令地址+修改后的操作数=f函数的地址 √

记住上面三个公式并知道每个值的含义就可以做题了

- 重定位绝对引用

上例中没有这种情况，所以下文不再举例(我的电脑搞不出来这个绝对寻址)

ADDR(r.symbol)即符号引用在实际可执行文件中运行时的地址

有一点同pc相对引用一样：

符号引用的实际内存地址 refaddr=ADDR(s)+offset

ADDR(s)也是.text节的实际地址

不一样的是更新：修改后的操作数=ADDR(r.symbol）+r.addend

一篇文章说的比较好：

重定位符号引用--重定位绝对引用 - 知乎
https://zhuanlan.zhihu.com/p/639176481

5.loader

loader将可执行目标文件中的代码和数据从磁盘复制到内存，跳转到程序的入口点运行，这个过程叫做加载

linux程序可以通过调用execve函数来调用loader

运行时内存映像：

即进程的虚拟内存空间

loader加载并运行一个可执行目标文件的过程其实是创建一个进程的过程

最后附上几张小班课ppt的图：

复习：C语言的声明

- 声明的例子
  - 一个全局变量（可能来自其他文件）：extern int a;
  - 一个函数（可能来自其他文件）：int zero();
- 声明的通式
  - (static/extern) type name …;
  - static的用法：有两个含义，静态生命周期和内部链接
    - 对于函数：只可以在本文件（编译单元）内被引用 = **内部链接**
    - 对于变量：
      - 在函数外：在程序退出前一直存在 **静态生命周期**、**内部链接**
      - 在函数内：**静态生命周期**（类似全局变量），只可以在函数内部访问（类似局部变量）
    - 不可以出现在函数参数中

复习：C语言的声明

- extern的用法
  - 不能和static一起用
  - 对于变量
    - 一般用来标识非在本文件中定义的变量
    - 写上更清晰，建议写
  - 对于函数
    - 不加static的函数默认为extern的，可以不写extern

复习：C语言的定义

- 定义（同时也是声明）的例子
  - 变量：int a = 2;
  - 函数：int zero() { return 0; }
- 一般情况下，你会见到
  - (static) type name …;
- 对于函数，声明和定义很好区分
- 对于变量，情况比较复杂
  - 例子1：一个在函数外的extern int a; ——一定是声明、不是定义
  - 例子2：一个在函数外的int a = 2; ——一定是定义、也是声明
  - 例子3：一个在函数外的int a;
    - 首先，这是一个声明(他的定义可能来自其他文件)
    - 但这也有可能是一个定义(如果其他文件中都没有a的定义)
    - 这个时候，它的初始值为0
    - 避免混淆的最好方法：别这么干 出题人这么干你要知道