

1 链接

在上一章（??）中，我们讨论了标准形式（Standard Form）的程序 $P = I_1, I_2, \dots, I_s$ ，满足条件 ($\forall 1 \leq t \leq s$) $(I_t = J(m, n, q)) \rightarrow (q \leq s)$ 。现在我们假设有限长度的程序 $P = I_1, I_2, \dots, I_s$ 并非标准程序，也即存在跳转指令 I_t 可以跳转到程序之外：

$$(\exists 1 \leq t \leq s) \quad ((I_t = J(m, n, q)) \rightarrow (q > s))$$

现在，我们考虑从 P 中生成新的程序 $P^* = I_1^*, I_2^*, \dots, I_s^*$ ，使得 P^* 为标准形式。生成的方式是这样的：考虑到 Z, S, T 指令都是顺序执行的，因此我们不修改这三类指令，也即 $I_t^* = I_t$ ；对于 $I_t = J(m, n, q)$ 指令，我们对 P^* 增加一个指令位置 I_{s+1}^* ，如果 J 指令跳转到 I_s 以外，我们将指令指针转移到 I_{s+1}^* ：

$$I_t^* = \begin{cases} J(m, n, q) & q \leq s + 1 \\ J(m, n, s + 1) & q > s + 1 \end{cases}$$

显然，对于非标准程序 P ，当发生非标准的跳转时，程序进入了未定义的状态，因为 I_q 是未定义的。在我们生成的标准程序 P^* 中，我们额外增加了一个指令 I_{s+1}^* ，它并不起任何作用，只是使得程序 P^* 有定义并且停止： $(P \uparrow) \Rightarrow (P^* \downarrow)$ 。

上述的转换过程正是我们链接（Linking）程序的核心：对于一切未定义的数据与指令，我们提供一个有定义，但是对程序执行没有意义的地址，让每一个未被定义的数据与指令都引用一个有效的地址（这正是 I_{s+1}^* 的作用）。这样的设计使得我们能够增加 C 语言编程的扩展能力，我们在.c 源文件中引用未定义的数据或函数，而这一数据或函数则由其他库（Library）提供。将函数与数据分散在不同的.c 源文件中，使得我们在工程上能够模块化，而链接则将若干.c 源文件生成的可执行可链接文件结合为一份体积更大的可执行目标文件，后者的未定义数据与未定义函数要少于前者（考虑动态链接）。

因此，对于链接，我们所需要考虑的核心问题如何设计一个可执行文件，并且该文件中包含格式 I_{s+1}^* ，使得我们能够进行链接。这种文件格式被称为可执行可链接格式（Executable and Linkable Format, ELF），ELF 会被静态链接（Static Linking）为可执行目标文件（Executable Object File, EOF）。

1.1 可执行可链接格式

在这一章中，我们可以建立自定义的 ELF 文件。我们用.txt 文本文件来描述 ELF，形成可执行可链接的文本。读者会发现，其实可执行文件与可链接文件本质上与一行一行的文本文件并没有区别，只是在 Linux 等操作系统中按照 Byte 翻译，而我们按照字符翻译。在第??章中，我们实现了指令的字符串化，因此在这一章我们有能力实现可执行文件的文本化。更进一步，读者在后续讨论虚拟内存时还将发现，就连内存中的页，也可以作为文本文件储存。在 Linux 等类 Unix 操作系统中，一切数据都是文件，而在我们的模拟中，一切数据都是文本。

首先，我们考虑什么是文件系统（File System）上的文件。我们知道，文件都是持久地储存在磁盘等设备上的，但它的实质仍然是二进制序列。只要我们对序列进行恰当的描述，我们就可以获得想要的全部信息。为了将某一文件的二进制串与其他文件的二进制串区别开，我们需要指定这一文件的类型与大小等信息，这些信息存放在文件二进制串的起始地址，也即首部（Header）。

1.1.1 ELF Header

对于 ELF，首部的信息包括文件类型、机器类型等信息。在 Linux 读取 ELF 文件时，Linux 将 ELF Header 从 Byte 翻译为内存中的数据结构：

```
/usr/include/elf.h

1 typedef uint16_t Elf64_Half;
2 typedef uint32_t Elf64_Word;
3 typedef uint64_t Elf64_Xword;
4 typedef uint64_t Elf64_Addr;
5 typedef uint64_t Elf64_Off;
6 typedef uint16_t Elf64_Section;
7
8 #define EI_NIDENT (16)
9
10 typedef struct
11 {
12     unsigned char    e_ident[EI_NIDENT];      /* Magic number and other info      */
13     Elf64_Half      e_type;                  /* Object file type                */
14     Elf64_Half      e_machine;               /* Architecture                   */
15     Elf64_Word      e_version;                /* Object file version             */
16     Elf64_Addr     e_entry;                 /* Entry point virtual address   */
17     Elf64_Off       e_phoff;                 /* Program header table file offset */
18     Elf64_Off       e_shoff;                 /* Section header table file offset */
19     Elf64_Word      e_flags;                 /* Processor-specific flags        */
20     Elf64_Half      e_ehsize;                /* ELF header size in bytes       */
21     Elf64_Half      e_phentsize;              /* Program header table entry size */
22     Elf64_Half      e_phnum;                 /* Program header table entry count */
23     Elf64_Half      e_shentsize;              /* Section header table entry size */
24     Elf64_Half      e_shnum;                 /* Section header table entry count */
25     Elf64_Half      e_shstrndx;               /* Section header string table index */
26 } Elf64_Ehdr;
```

数据结构 `Elf64_Ehdr` 包含了很多信息，但对我们而言，我们只需要知道 Header 是如何定位到节头表 (Section Header Table, SHT) 的。如图1，Header 中的 `Elf64_Ehdr.e_shoff` 是 SHT 对 ELF 起始地址的 Byte 偏置，`Elf64_Ehdr.e_shentsize` 描述 SHT 每一项的 Byte 大小，`Elf64_Ehdr.e_shentnum` 描述 SHT 描述表项的数量，这样，我们就可以通过后两项计算出 SHT 的大小。Header 中的 `Elf64_Ehdr.e_shstrndx` 则是 SHT 中一个特殊节——字符串表——的段索引，Header 可以借此计算 ELF 中每一项的字符串。

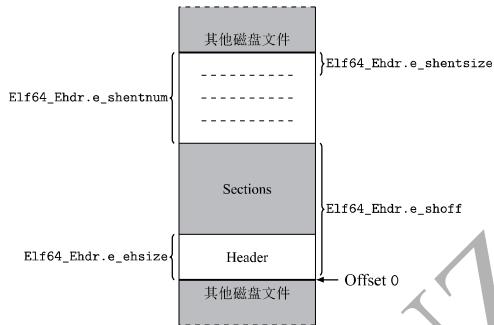


图 1: ELF Header 到 Section Header Table

我们举一个例子，使我们洞见 ELF 文件的数据结构。考虑如下的 `elf.c` 源文件：

```

elf.c
1 unsigned long long data1 = 0xd1d1d1d1d1d1d1d1;
2 unsigned long long data2 = 0xd2d2d2d2d2d2d2d2;
3 void func1() {};
4 void func2() {};

```

使用`/usr/bin/gcc-7`编译得到它的 ELF 文件：`gcc-7 -c elf.c -o elf.o`，其中`-c`的含义是仅编译源文件生成 ELF 文件，但不对 ELF 文件进行链接（当然这里我们只有一个文件，因此并不能链接）。生成 ELF 文件以后，利用`/usr/bin/readelf`或`/usr/bin/hexdump`查看 ELF 文件 `elf.o` 的内容。

`hexdump`获得 `elf.o` 的二进制内容以后，按照 `Elf64_Ehdr` 的结构格式，我们可以解读 ELF 的 Header。首先，我们可以计算得知 `sizeof(Elf64_Ehdr)` = 64，因此 Header 占据 ELF 文件的低 64Bytes。这样，我们就得到了 `elf.o` 的 Header：

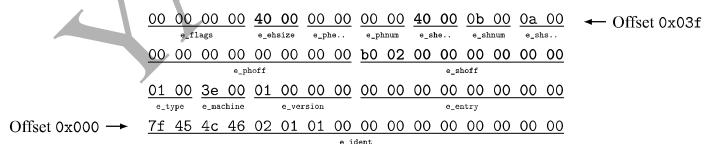


图 2: elf.o Header

起始位置的 16 Bytes 为 Magic Number，在这之后偏移 $2 + 2 + 4 + 8 + 8 = 24$ Bytes，大小为 8 Bytes 的数据为 `Elf64_Ehdr.e_shoff`，也即 SHT 起始地址对 ELF 文件的偏移，数值为 `0x00000000000002b0`。在这之后，对 `Elf64_Ehdr.e_shoff` 偏移为 4 Bytes 的位置储存了 ELF 文件 Header 的大小：`Elf64_Ehdr.e_ehsize = 0x0040`。再之后 $2 + 2 = 4$ Bytes 即 `Elf64_Ehdr.e_shentsize`，也即为 SHT 每一项的大小，数值为 `0x0040`。最后两个 2 Bytes 的数分别是 `Elf64_Ehdr.e_shnum = 0x000b` 以及 `Elf64_Ehdr.e_shstrndx = 0x000a`。

1.1.2 节头表 SHT

SHT 描述了 ELF 中不同的节 (Section)，包括数据节 (.data)、代码节 (.text) 等。这些 Section 中的数据是由编译器生成的，按照 Section 的组织写入到磁盘上的 ELF 文件中。SHT 的每一项可以被数据结构 Elf64_Shdr 所描述：

```

/usr/include/elf.h

1 typedef struct
2 {
3     Elf64_Word sh_name;      /* Section name (string tbl index) */
4     Elf64_Word sh_type;      /* Section type */
5     Elf64_Xword sh_flags;    /* Section flags */
6     Elf64_Addr sh_addr;     /* Section virtual addr at execution */
7     Elf64_Off sh_offset;     /* Section file offset */
8     Elf64_Xword sh_size;     /* Section size in bytes */
9     Elf64_Word sh_link;      /* Link to another section */
10    Elf64_Word sh_info;      /* Additional section information */
11    Elf64_Xword sh_addralign; /* Section alignment */
12    Elf64_Xword sh_entsize;   /* Entry size if section holds table */
13 } Elf64_Shdr;

```

这样，我们可以通过 SHT 查找到 ELF 内的任一 Section，常见的 Section 有 .text, .rodata, .data, .bss, .symtab, .rel.text, .rel.data, .strtab，它们可以被 Elf64_Shdr.sh_name 确定：

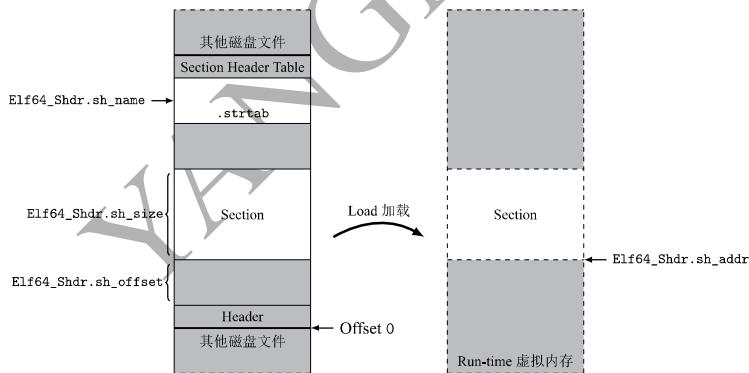


图 3: ELF Section 格式

在 `elf.o` 的 Header 中，我们已经发现 SHT 距离 ELF 文件的起始地址偏移为 688 Bytes，并且每一表项的大小为 64 Bytes，共 11 项，那么我们可以找到 SHT 对应的二进制段。但是由于这段代码太长了，因此我们只截取 .text 与 .data 两节：

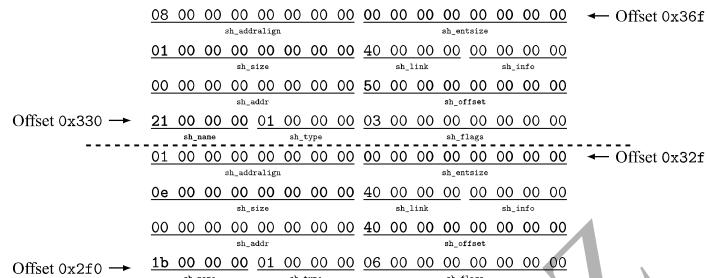


图 4: elf.o Section Header Table: .text, .data

和解析 Header 一样，对于 SHT 的每一个表项，按照结构 `Elf64_Shdr` 逐个 Byte 解析，我们就得到了.text 对 ELF 文件的偏移：`.sh_offset = 0x0000000000000040`。回忆起我们先前得知 Header 的大小就是 `0x40`，也即 64 Bytes，因此.text 所处的位置正是 Header 之后。并且，它的大小为 `.sh_size = 0x0000000000000010`，就说明 `func1()` 与 `func2()` 两个函数占据了 16 Bytes（但目前我们并不知道每一个函数的真正位置）。对于.data，我们可以同样进行解析。`readelf -S elf.o` 会给我们相同的结论（我们省略了其他表项）。

值得一提的是，SHT 中总存在 [0] 项，它被视为未定义的符号所在 Section，SHN_UNDEF，也就是 0。同时，这一 SHT 表项的值总是 64 Bytes 的 0：

[Nr]	Name	Type	Address		Offset	
Size		EntSize	Flags	Link	Info	Align
[0]	NULL		0000000000000000	00000000	00000000	00000000
0000000000000000	0000000000000000		0000000000000000	00000000	00000000	00000000

1.1.3 符号表

最后，我们要定位到每一个符号在 ELF 中所处的位置，这就需要一个特别的 Section, .symtab, 也即符号表 (Symbol Table)。符号表被用来描述.c 源文件中可以被其他 ELF 使用的符号。对于什么是符号 (Symbol)，其实有必要先从编译器的视角去观察.c 源文件。这部分工作主要是编译中语义分析 (Semantic Analysis) 的过程，通过语法中的环境 (Environment) 和范围 (Scope) 来进行管理。其中 Environment 映射在编译器中也被称为符号表，我们很快便能发现编译和链接语境下的符号表其实是含义相同的。

考虑赋值的语句 `int a = 0xaaaaaaaa;`，在这里，终结符 `a` 是我们通常所说的左值，其实它会被处理为一个字符串；终结符 `0xaaaaaaaa` 是我们所说的右值，被处理为数值。当我们在.c 源文件中引用终结符 `a` 时，其实做了两步映射（Two-Stage Mapping）：

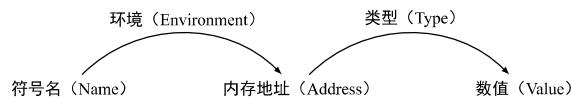


图 5: 引用对象的两步映射

其中第二步的 Type 映射，我们在第??章中已经简要地讨论过了。更复杂的内容，例如类型转换等，已经超出了我们的讨论范围。在链接中，我们只需要考虑 Environment 映射。在 C 语言中，每一个对象（函数、变量）有它可见的 Scope，也就是说在 Scope 内，这个对象可以被引用。而在不同的 Scope 内，Environment 未必相同。我们仔细考虑 C 语言的语法，很容易发现源文件可以被区分为不同的块（Block），它们通常被一组花括号（'{'，'}'）所包围，天然地对应一棵有关 Block 的树（考虑最外层为根节点）。例如考虑如下的源文件：

```
1 int      g_var_init = 0xffffffff;
2 int      g_var_uninit;
3
4 void     func_undef();
5 void     func_def(int func_param)
6 {
7     int func_var = 0;
8     do
9     {
10         int loop_var    = 1;
11         func_var      += 1;
12         g_var_init     += 1;
13
14         func_undef();
15     }
16     while (func_var < func_param)
17 }
18
19 void main()
20 {
21     func_def();
22 }
```

相应的，考虑它的 Block 树，以及每一个 Block 内可见的对象：

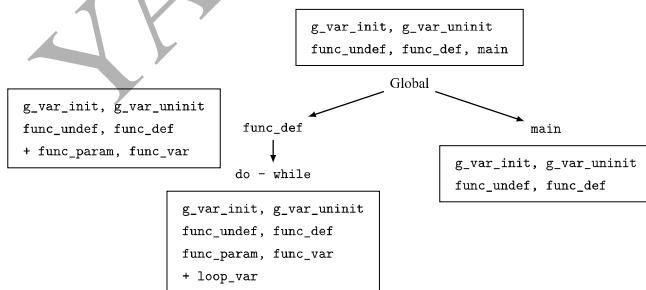


图 6: 源文件中 Block 结构，以及 Block 内可见的标识符

我们可以发现，每当进入一个 Sub-Block 以前，Sub-Block 会复制当前 Block 的 Environment，同时，在 Sub-Block 以内，新定义的变量、传入函数的参数，也都会被视为在 Sub-Block 内新的映射。因此，我们在进入新的 Sub-Block 时，可以更新 Environment，从而实现了对 Environment 的维护。

我们已知 C 语言不允许在函数内定义函数，这个简单的事实将整个.c 源文件分为扁平的两层：函数内部 (Internal)、函数外部 (External)，如图7。在第??章中，我们已知函数 Scope 内的数据是在 Run-Time Stack 上分配的，因此函数内 Scope 的 Environment 映射到栈。而函数外部，也就是函数本身以及全局变量，它们的 Environment 需要我们额外维护，这也就是编译器在 ELF 文件中生成的符号表。

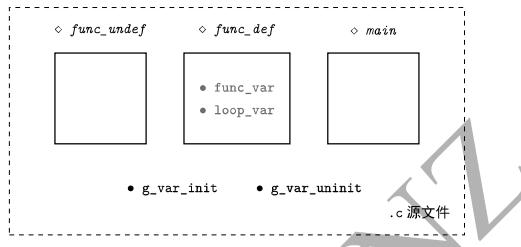


图 7: 链接器的可见范围 (函数外部)

因此，外部的函数与变量是对链接可见的，也因此它们能够被其他 ELF 文件所引用。这就是我们在链接时所说的符号——一个函数 (Function)，一个全局变量 (Global Variable)，或一个静态变量 (Static Variable)。

根据我们对符号的理解，按照定义和引用，可以将符号分为以下三种类型。假定 source1.c 是我们当前可见的源文件，source2.c 是被链接的 ELF 的源文件。

	source1.c 有定义	source2.c 有定义
source1.c 仅引用	static int var = 0xf;	extern int var;
source2.c 仅引用	int var = 0xf;	

其中被 static 修饰的，被称为局部符号 (Local Symbol) 或静态符号 (Static Symbol)，只对同一源文件中的函数可见。在本地有定义，但没有被 static 修饰的，是普通的全局符号 (Global Symbol)，对链接中所有的源文件可见。被 extern 修饰的，说明在当前的源文件中没有定义，而只在 extern 处有声明，方便当前源文件中的函数对它进行引用。

全局符号 我们考察 elf.o 中的例子，data1 与 data2 是作为变量的符号，func1 与 func2 是作为函数的符号，它们是典型的全局符号。我们来观察全局符号是怎样在 ELF 文件中被找到的。从 ELF Header 计算偏移到 Section Header Table，当我们读取到表项为 .symtab 时，就开始处理符号表，符号表表项的数据结构如下：

```

/usr/include/elf.h

1  typedef struct
2  {
3      Elf64_Word      st_name;        /* Symbol name (string tbl index) */
4      unsigned char   st_info;        /* Symbol type and binding */
5      unsigned char   st_other;       /* Symbol visibility */
6      Elf64_Section  st_shndx;       /* Section index */
7      Elf64_Addr     st_value;       /* Symbol value */
8      Elf64_Xword    st_size;        /* Symbol size */
9  } Elf64_Sym;

```

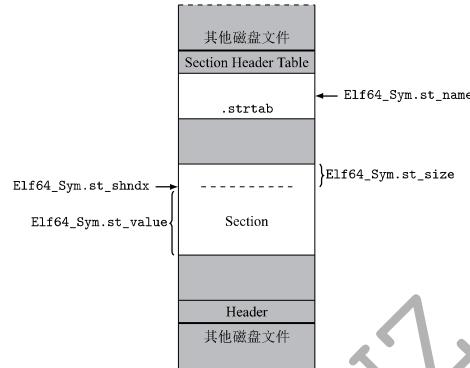


图 8: ELF 符号表

.symtab 是 elf.o SHT 中的 [8] 表项，它对 ELF 文件的偏移为 0x000000e8，占据 0x120 Bytes，并且.symtab 的每一个元素都有固定的大小，Elf64_Shdr.sh_entsize = 0x0000000000000018。这样，我们就可以计算得到.symtab 一共有 12 项，其中四项关于 data1, data2, func1, func2:

func2:	19 00 00 00 12 00 01 00 07 00
func1:	13 00 00 00 12 00 01 00
data2:	0d 00 00 00 11 00 02 00 08 00 00 00 00 00 00 00 00 00 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00
data1:	07 00 00 00 11 00 02 00 00 00 00 00 00 00 00 00 00 00 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00

图 9: elf.o .symtab

其中 Elf64_Sym.st_name 查到符号在字符串表中的名称，例如对于 data1，Elf64_Sym.st_name = 0x7，字符串表从偏移 0x7 的 Byte 开始，直到遇到第一个字符串终结符 00 ('\0')，所得的字符串为: 64 61 74 61 31 00，转译为 ASCII 码的字符即为'd', 'a', 't', 'a', '1', '\0'。符号 data1 的 Section 索引为 Elf64_Sym.st_shndx = 0x2，那么在 SHT 中，编号为 [2] 的是.data 节，因此我们确定了 data1 所在的 Section。

再根据 Elf64_Sym.st_value = 0x0，我们知道 data1 对.data 节的偏移为 0，到此为止是 Environment 映射。由符号表，我们得知 data1 占据 Elf64_Sym.st_size = 0x8 Bytes。至此，我们也完成了 Type 的映射，可以获得 data1 的数值了：

d1 d1 d1 d1 d1 d1 d1 d1 d2 d2 d2 d2 d2 d2 d2	← Offset 0x5f
Offset 0x40 → 55 48 89 e5 90 5d c3 55 48 89 e5 90 5d c3 00 00	func1 func2

图 10: elf.o 定位 data1

静态符号 静态符号的定位过程与全局符号是一样的，所不同的是对符号的描述——Elf64_Sym.st_info ——并不相同。我们考虑下面的例子：

```
1 static unsigned long long data3 = 0x1;
2 static void func3(){};
```

在符号表中，它们的表项为：

func3:	0c 00 00 00 02 00 01 00 00 00 00 00 00 00 00 00 07 00 00 00 00 00 00 00 00 00	st_name	BT .. st_shndx	st_value	st_size
data3:	06 00 00 00 01 00 02 00 00 00 00 00 00 00 00 00 08 00 00 00 00 00 00 00 00 00	st_name	BT .. st_shndx	st_value	st_size

和全局变量相比，静态变量 data3 与 func3 依然分别处在.data 节与.text 节，它们的大小与位置也分别由 Elf64_Sym.st_size 和 Elf64_Sym.st_value 给定。Elf64_Sym.st_other 是一直被置为 0x00 的。我们用以描述符号性质的，是 8 位的 Elf64_Sym.st_info。Elf64_Sym.st_info 的 8 个 Bits 被分为两个部分：低 4 位的被解释为符号的类型（Type），高 4 位为符号在源文件中的关联（Bind），这里取值为 0x1。Bind 目前在 Linux 中有九种取值，我们主要关心前三种：

```
/usr/include/elf.h
1 #define STB_LOCAL    0      /* Local symbol      */
2 #define STB_GLOBAL   1      /* Global symbol     */
3 #define STB_WEAK    2      /* Weak symbol       */
```

由于是静态符号，因此它们的 Bind 都是 STB_LOCAL(0x0)。对比全局符号 data1, data2, func1, func2，它们的 Bind 则都是 STB_GLOBAL(0x1)，这是静态符号和全局符号的主要差别。而 Type 则依据它们是如何被定义的而设定，data* 被定义为变量，因此 Type 为 0x1，func* 被定义为函数，因此 Type 为 0x2，具体请参见下面的几种情况：

```
/usr/include/elf.h
1 #define STT_NOTYPE  0      /* Symbol type is unspecified      */
2 #define STT_OBJECT   1      /* Symbol is a data object        */
3 #define STT_FUNC     2      /* Symbol is a code object       */
```

外部符号 被 `extern` 修饰的情况更为复杂，这主要是由于我们在申明来自其他源文件的外部符号时，可以省略关键字 `extern`。但“省略 `extern`”这一行为对函数和变量会造成不同的后果。外部符号是没有定义的，它的定义直到链接时才对当前的.c 源文件可见。因此我们无法在.text 节和.data 节中给出外部符号的 Bytes，而只能使用符号表和重定位表描述符号，于是我们发现，符号表和重定位表正起到了 I_{s+1}^* 的作用。

首先，我们讨论函数，因为函数的情况较为简单。对于外部的函数，我们在 `source1.c` 中仅仅声明和引用。实际上，函数声明只分为两种情况：第一种情况，它被 `static` 修饰，也就是只对当前源文件可见的静态函数；除此以外，函数都是外部符号。因此，关键字 `extern` 对函数声明是可选的：

```
source1.c
1 extern void func4();
2         void func5();
```

这两种声明都会产生同样的符号。由于函数 `func4()` 和 `func5()` 并没有定义，因此它们在.text 节中并不占据任何 Byte。如果它们没有在当前的源文件中被引用的话，编译器应当优化这两个无用的声明。如果这两个函数有被引用，那么这两个函数会出现在符号表和重定位表中，等待链接时被重定位到它们的定义。在符号表中，有关 `func4()` 与 `func5()` 的表项是这样的：

<https://github.com/yangminz>
<https://space.bilibili.com/4564101>
<https://www.zhihu.com/people/zhao-yang-min>

计算机系统

```
func5: 29 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
        st_name    BT .. st_shndx   st_value      st_size  
func4: 21 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
        st_name    BT .. st_shndx   st_value      st_size
```

由于外部函数是仅声明的，因此它并不归属于当前 ELF 中的任意一个 Section。于是它的 Section 索引 Elf64_Sym.st_shndx 被置为 0x0000，也就是 SHT 的 [0]，因此被视为无定义 (SHN_UNDEF)。同时，它的 Type 则是无类型的——STT_NOTYPE。但是，需要特别注意的是外部函数的 Bind 依旧是全局符号 STB_GLOBAL。

变量的情况要比函数复杂一些，这是由于省略 `extern` 导致的。对于变量，被 `static` 修饰的仍然是静态符号，但不被 `static` 所修饰的，却未必是外部符号。未被初始化、未被 `extern` 修饰的变量，被称为临时性定义 (Tentative Definition)。考虑以下几种情况：

```
source1.c  
1 extern int data4;  
2         int data5;  
3         int data6 = 0;  
4 static int data7;  
5 static int data8 = 0;  
  
它们的符号表项分别为：  
  
data4: 21 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
        st_name    BT .. st_shndx   st_value      st_size  
data5: 13 00 00 00 11 00 f2 ff 04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00  
        st_name    BT .. st_shndx   st_value      st_size  
data6: 19 00 00 00 11 00 04 00 00 00 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00  
        st_name    BT .. st_shndx   st_value      st_size  
data7: 07 00 00 00 01 00 04 00 04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00  
        st_name    BT .. st_shndx   st_value      st_size  
data8: 0d 00 00 00 00 01 00 04 00 08 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00  
        st_name    BT .. st_shndx   st_value      st_size
```

`data4` 是作为外部符号而声明的，它显式地被 `extern` 所修饰，因此与外部的函数一样，Bind 为全局符号 `STB_GLOBAL`，Type 为无类型 `STT_NOTYPE`，Section 为未定义 `SHN_UNDEF`。

我们特别注意 `data5`，它既未被 `extern` 所修饰，也没有初始赋值，因此是 Tentative Definition。然而 `data5` 和 `data4` 一样，都可以被用来声明外部变量。尽管如此，这两行语句的本质是不一样的：`extern int data4;` 是严格的声明语句，它说明符号 `data4` 的内存地址并不在当前源文件产生的 ELF 中；而 `int data5;` 只是缺省赋值的初始化，它的地址存在于当前的 ELF 文件的 COMMON 节。这两种声明都是弱符号的声明，在符号解析中会被进一步处理。

因此 `data5` 的 Bind 为 `STB_GLOBAL`，而我们已知它的 Type 为 `STT_OBJECT`，更重要的，`data5` 的 Section 是 COMMON 节 `SHN_COMMON` (0xffff2)。正如 Tentative Definition 的语义一样，COMMON 是 ELF 中临时的节，并不在 EOF 中存在。被存放在 COMMON 中的变量，会在链接时决定它的最终去向。如果链接完成后找到其定义，则其地址在 EOF 的 `.data` 段，如果没有找到定义，则按照全 0 默认初始化，地址在 `.bss` 段。在 COMMON 段中，`Elf64_Sym.st_value` 指该符号的对齐要求。

由于 `data5` 的默认值是 0，因此我们考虑第三种情况，也就是 `data6`。`data6` 与 `data5` 的区别在于，`data6` 被显式地初始化，但被初始化为 0。ELF 将所有被初始化为 0 的变量集中管理到 `.bss` 节（也就是该 ELF 的 [4] 节），因为没有必要在 `.data` 节中写入大量的 0。BSS 是 Block Started by Symbol 的缩写，`.bss` 节管理初始化为 0 的全局变量，未初始化的静态变量 (`data7`)，以及初始化为 0 的静态变量 (`data8`)。

Weak Bind 符号 除此以外,在 ELF 的格式中还存在着一种 *Weak Bind* 的符号,也就是 `Elf64_Sym.st_info` 中的 Bind 为 `STB_WEAK`。这种类型的符号需要我们显式地修饰: `_attribute__((weak))`。Weak Bind 的符号也可以定义符号,但如果链接时发现了 `Defined` 符号,那么弱定义会被新的定义覆盖。这样的弱定义像是一种占位符 (Place Holder)。例如我们通过函数创建链表: `uint64_t linkedlist_allocate()`。按理来说,函数是外部定义,被链接到当前 ELF 的。可是我们并不确定其他的 ELF 中是否真的存在该函数的定义,因此我们可以给这个函数一个弱定义:

```
1 __attribute__((weak)) uint64_t linkedlist_allocate()
2 {
3     return 0xffffffffffffffff;
4 }
```

这样,当外部 ELF 不存在 `linkedlist_allocate()` 的定义时,我们仍能将 ELF 链接到合法的 EOF。但工程历史和经验告诉我们,Weak Bind 符号容易带来很多隐患,并不可靠,因此不被推荐使用。

到此为止,我们已经基本描述了 ELF 文件的主要结构。目前,我们获得任一符号的引用,便可以在生成的 ELF 文件中找到它的地址与数值了。这个映射的过程大致如图11所描述。

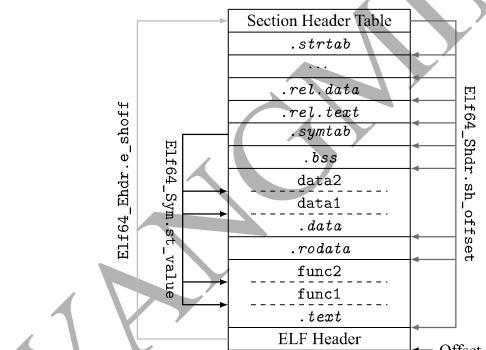


图 11: ELF 文件符号引用

1.2 符号解析

符号解析是我们做静态链接的第一步。当所有的 ELF 文件被读入内存以后,我们面临的一个问题:不同的文件可能声明了同名的符号。符号解析需要解决这样的声明冲突。例如,不同的文件中声明了全局符号 `int a`,我们需要在所有的 ELF 文件中选择一个作为它的定义。

对于所有的全局符号,我们考虑三种类型:有定义(Defined)、临时定义(Tentative)、未定义(Undefined),这我们在符号表一节中已经简单讨论过了。现在,我们更严谨地描述与总结这三种类型:

- 有定义的符号是指该符号在 ELF 文件中有确定的位置 (`.text` 节, `.rodata` 节, `.data` 节, `.bss` 节),用来储存它的数值;
- 临时定义是指该符号没有确定的储存空间,通常也就是 COMMON 节中的全局变量,因为我们在编译阶段不能确定该变量最终属于 `.data` 节或是 `.bss` 节,因此它是临时定义;

- 最后，未定义的符号也就是没有储存空间的符号，也就是 `Elf64_Sym.shndx` 为 `SHN_UNDEF`。

再加上 Weak Bind 的符号，这四种符号之间的优先级如下：`Undefined < Weak Bind < Tentative < Defined`。其中 `Defined` 是我们通常所说的强符号（Strong Symbol），在所有的 ELF 文件的所有同名符号中，`Defined` 符号只能存在一个；`Tentative`、`Weak Bind`、`Undefined` 是弱符号（Weak Symbol），它们可以被更高优先级的定义所覆盖。我们列一张表格，囊括大部分的情况：

Bind	Type	Section	Declaration & Definition	Symbol
STB_GLOBAL	STT_NOTYPE	SHN_UNDEF	<extern> void f();	Undefined
STB_GLOBAL	STT_FUNC	.text	<extern> void f() {}	Defined
STB_WEAK	STT_NOTYPE	SHN_UNDEF	__attribute__((weak)) <extern> void f();	Undefined
STB_WEAK	STT_FUNC	.text	__attribute__((weak)) <extern> void f() {}	Weak Bind
STB_GLOBAL	STT_NOTYPE	SHN_UNDEF	extern int d;	Undefined
STB_GLOBAL	STT_OBJECT	SHN_COMMON	int d;	Tentative
STB_GLOBAL	STT_OBJECT	.data	int d = 1;	Defined
STB_GLOBAL	STT_OBJECT	.bss	int d = 0;	Defined
STB_WEAK	STT_NOTYPE	SHN_UNDEF	__attribute__((weak)) extern int d;	Undefined
STB_WEAK	STT_OBJECT	.bss	__attribute__((weak)) int d <= 0;	Weak Bind
STB_WEAK	STT_OBJECT	.data	__attribute__((weak)) int d = 1;	Weak Bind

特别是 `__attribute__((weak)) int d;`，我们注意到它本是 COMMON 中的符号，然而被 Weak Bind 修饰以后，链接器 `ld` 便取默认的定义值 0，因此被储存在 `.bss` 节，作为弱定义的符号存在。

在编译阶段，每一个源文件中，一个符号名只对应一个符号。例如下面的情况是非法的，会引起巨大的歧义和混淆，即便在编译阶段也无法被处理：

```

1 int a = 0xffffffff;
2 int a()
3 {
4     a += a();
5     return a;
6 }

```

但是，不同的源文件之间却可能分别定义 `int a` 与 `int a()`。对于同名的符号（函数或变量），我们主要考虑 `Elf64_Sym.st_info` 中的 Type、`Elf64_Sym.st_size`。Type 与 Size 相同的情况即上述的解析，我们比较符号的在 `Undefined < Weak Bind < Tentative < Defined` 中的优先程度，选择写入 EOF 的符号。主要采用下述三条规则，从 `.o` 文件不同，但 `Elf64_Sym.st_name` 相同的符号中，选择写入 EOF：

1. 只允许存在一个符号为 `Defined`，也即强符号，并且选择该强符号；
2. 多个弱符号（`Undefined`、`Tentative` 以及相应 `Weak Bind`）与一个强符号共同存在时，选择强符号；
3. 只有多个弱符号时，选择优先程度最高的；优先程度相同时，任意选择一个弱符号。

对于不同 Type 或 Size，情况更加复杂。例如 Type 不同，也即一个符号是 `STT_FUNC`，另一个符号为 `STT_OBJECT`。或者同为 Type 相同，但 size 不同，例如定义为不同长度的 `int` 数组。如果链接器能够处理，则被称为 Complex Resolution，`ld` 会给出 Warning；如果无法处理，这被称为 Fatal Resolution，`ld` 会失败。

1.3 Section 合并

实现符号解析以后，我们还需要将相同类型的 Section 进行重定位与合并。这一步处理所有被解析过的符号，根据它们所属的 Section，将符号放在同一 Section 中。至此，我们便可以确定该 Section 在 EOF 中

的大小了，所有被合并的 Section 被称为段（Segment），Segment 的概念会被延续到 EOF 的运行时，也就是进程。同时，在合并 Section 时，我们也可以计算出 Segment 的运行时起始地址（Run-time Address），因为.text 在 64 位系统中作为只读内存，是从虚拟地址 0x00400000 开始的。这部分内容更侧重实现，因此我们在 1.5 中讨论。

1.4 外部引用的重定位

完成符号处理以及 Section 的合并以后，我们已经获得了囊括全部信息的 EOF 的雏形了。在此基础上，我们需要对符号与引用进行重定位（Relocation）。重定位的过程，与我们在本章一开始所讨论的 I_{s+1}^* 的逻辑是非常相似的。举一个例子，我们在当前 ELF 文件中遇到无法处理的引用，可能一个定义在其他 ELF 文件中的函数 void `undef_func()`，一个定义在其他 ELF 文件中的数组 `int undef_array[2]`，却被当前的 ELF 文件代码所引用：

```
referencing.c
1 extern void undef_func();
2 extern int undef_array[2];
3 void main()
4 {
5     undef_func();           // reference <1>
6     undef_array[0] = 1;    // reference <2>
7     undef_array[1] = 2;    // reference <3>
8 }
```

此时，正如我们在 URM 程序中增加到 I_{s+1}^* 的跳转一样，我们在当前 ELF 文件中增加一些占位项，用来记录 `undef_func` 与 `undef_array` 在当前 ELF 中被引用的位置。这些记录位置的项，就是重定位项。因为所有 ELF 的符号只有在链接阶段才彼此可见，因此当链接器解析到 `undef_*` 的定义时，链接器可以通过重定位项，将 `undef_*` 的引用指向它们真正的定义。

在 /usr/include/elf.h 中，重定位项由结构体 `Elf64_Rela` 描述：

```
/usr/include/elf.h
1 typedef struct
2 {
3     Elf64_Addr   r_offset; /* Address */
4     Elf64_Xword  r_info;  /* Relocation type and symbol index */
5     Elf64_Sxword r_addend; /* Addend */
6 } Elf64_Rela;
```

其中 `Elf64_Rela.r_offset` 记录了引用的位置，是相对于当前 Section 的 Byte 偏移。`r_info` 是一个 8-Byte 的结构域，其中低位的 4-Byte 为重定位的类型，它提示链接器应当如何处理这一重定位项，高位的 4-Byte 为被引用的符号在符号表 `.symtab` 中的序号。最后，`r_addend` 是重定位计算中的偏置值。我们观察上文中的例子，给出 `undef_func` 和 `undef_array` 三处被引用位置的重定位项：

```
Ref<1>: 0a_00_00_00_00_00_00_04_00_00_00_0a_00_00_00_fc_ff_ff_ff_ff_ff_ff
          r_offset      type        sym      r_addend
Ref<2>: 10_00_00_00_00_00_00_02_00_00_00_0b_00_00_00_f8_ff_ff_ff_ff_ff
          r_offset      type        sym      r_addend
Ref<3>: 1a_00_00_00_00_00_00_02_00_00_00_0b_00_00_00_fc_ff_ff_ff_ff_ff
          r_offset      type        sym      r_addend
```

由于这三处位置都在引用者 `main()` 中，因此属于 `.text` 范围，所以重定位项在 `.rel.text` 中。我们观察一下引用者：

```
0: 55          push  %rbp
1: 48 89 e5    mov    %rsp,%rbp
4: b8 00 00 00 00  mov    $0x0,%eax
9: e8 00 00 00 00  callq  e <main+0xe>
e: c7 05 00 00 00 01  movl   $0x1,0x0(%rip) # 18 <main+0x18>
15: 00 00 00
18: c7 05 00 00 00 02  movl   $0x2,0x0(%rip) # 22 <main+0x22>
1f: 00 00 00
22: 90          nop
23: 5d          pop    %rbp
24: c3          retq
```

三处引用对 `.text` 节起始地址的 Byte 偏移，都由各自的 `Elf64_Rela.r_offset` 给出。第一个位置引用了符号表中索引为 [0xa] 的符号，也就是 `undef_function`，第二处和第三处引用的符号在符号表中的索引都是 [0xb]，也就是 `undef_array`。

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
10:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	<code>undef_func</code>
11:	0000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	<code>undef_array</code>

下面，我们来考虑链接器应当如何解析这些位置对符号的引用。根据 `Elf64_Rela.r_info` 中 `Type` 的不同，链接器采用不同的策略进行重定位。这些策略其实都在完成一件事：当程序访问引用对象时，根据引用对象当前的位置 (`<main+0xe>`)，计算出被引用者 (`undef_array`) 的位置。

1.4.1 PC 相对寻址

程序计数器的相对寻址 (PC Relative Address) 主要考虑 `%rip` 指针，这对于 `.rel.text` 中的重定位项是非常有用的。因为 `.rel.text` 中的引用总发生在 `.text` 节内，因此总能根据 `%rip` 指针的值计算被引用者的位置。引用位置 `<1>` 所采用的 `R_X86_64_PLT32`，以及 `<2>` 和 `<3>` 所采用的 `R_X86_64_PC32`，都是对 `%rip` 指针的相对地址。

利用相对地址完成重定位依赖于一个程序加载¹的一个性质：我们考虑考虑磁盘上的 EOF 文件与运行时的进程内存。在程序文件加载时，EOF 文件的 `.text`, `.rodata`, `.data` 等段 (Segment) 中任意两个位置 `[i]` 与 `[j]`，在程序加载到内存以后，这两个位置的相对地址 `i - j` 保持不变。因此，在 `.text` 段的 `%rip` 指针可以通过相对地址找到被引用者 `undef_array` 的位置。

`R_X86_64_PC32` 以 `undef_array` 为例。当链接器获得 `undef_array` 的定义以后 (符号处理阶段完成)，位置 `<2>` 与 `<3>` 的重定位项 `Elf64_Rela.r_addr` 将指向 `undef_array` 在 EOF 中的符号表位置。然后，根据图11中的映射，我们已经能够确定 `undef_array` 在 EOF 中的位置，记为 `[S]`。

同样，在符号处理和重定位 Section 中，我们也维护与更新 `Elf64_Rela.r_offset`，这样就可以找到 EOF 中 `<2>` 与 `<3>` 的地址，记为 `[P]`。对于 `.rel.text` 中的重定位项，我们取 EOF 中 `.text` 的起始地址 `Elf64_Shdr.sh_offset`，得到 `P = Elf64_Shdr.sh_offset + Elf64_Rela.r_offset`。

这样，从 `[P]` 到 `[S]` 的相对地址就是 `S - P`。考虑到 `%rip` 指针在引用时的位置，我们还需额外增加 `r_addend`，记为 `A`，用来矫正位置。这样，我们将相对地址 `S + A - P` 写入 `<2>` 与 `<3>` 的占位符，将

¹我们在第??章中再讨论加载 (Loading)。

它们的 4-Byte 00 00 00 00 修改为 S - P + A。我们举一个例子，在另一个文件中实现 `undef_func` 与 `undef_array`:

```
referenced.c

1 void      undef_func() {}
2 int       undef_array[2] = {-1, -2};
```

然后将两个生成的 ELF 文件连接起来:

```
gcc -c referencing.c -o referencing.o
gcc -c referenced.c -o referenced.o
ld --entry=main referencing.o referenced.o -o relocated.o
```

生成 EOF 文件以后，这一地址会被修正为 `undef_array` 的运行时的 PC 相对地址 (S + A - P):

```
4000f6: c7 05 00 0f 20 00 01    movl   $0x1,0x200f00(%rip)    # 601000 <undef_array>
4000fd: 00 00 00
400100: c7 05 fa 0e 20 00 02    movl   $0x2,0x200efa(%rip)    # 601004 <undef_array+0x4>
400107: 00 00 00
```

其中引用 <2> 在 EOF 中的 `r_offset` 为 0x4000f8。执行该指令时，寄存器`%rip` 指向下一条指令，为 0x400100。加上相对地址 0x200f00，为 `undef_array[0]` 的地址: 0x400100 + 0x200f00 = 0x601000。同理，执行第二条指令时（引用 <3>），`%rip` 指向下一条指令，为 0x40010a，加上相对地址 0x200efa，为 `undef_array[1]` 的地址: 0x40010a + 0x200efa = 0x601004，距离 0x601000 刚好相差 0x4，为一个 `int` 类型的长度。

R_X86_64_PLT32 **R_X86_64_PLT32** 与 **R_X86_64_PC32** 的计算过程相似，不过它不再计算 [S]，而计算程序链接表 (PLT) 的位置 [L]，相对地址计算为 L + A - P。其中有关 PLT 我们在 1.6 节中再讨论。

1.4.2 绝对寻址

R_X86_64_32 是绝对寻址的策略，它的重定位方法就是将被引用者的地址 [S] 写入位置 [P]，考虑`%rip` 执行指令，实际写入的绝对地址是 S + A。

1.5 文本静态链接的实现

1.5.1 可执行可链接文本格式

回顾我们在第??章中所作的工作，我们其实粗糙地实现了一个汇编代码的解释器。其实，真正的汇编指令也可以看作被 CPU 解释执行的，只不过是以字节为单位。在字符串汇编指令的基础上，我们已经可以解释执行汇编指令的字符串了。而为了更进一步实现链接，正如先前两节（1.2, 1.4）所讨论的，我们需要额外提供符号表与重定位表。为了确定这两张表，我们需要设计文件格式，使得 `.elf.txt` 文本能够像 `.o` 文件一样，定位到所有的 Section。

下面，我们以语法的形式给出 `.elf.txt` 文本的结构。按照惯例，加粗的是被识别的终结符，`\n` 是 ASCII 中的换行符，`.elf.txt` (文本)、`SHT` (节头表)、`SHTentry` (节头表的表项)、`Section` (节) 是可以被展开与归约的表达式:

```

.elf.txt → Numbertxt \n Numbersht \n SHT Section
SHT → SHTentry
SHTentry → ε | sh_name,sh_addr,sh_offset,sh_size\n SHTentry
Section → ε | .symtab Section | .text Section |
           .rodata Section | .data Section |
           .rel.text Section | .rel.data Section

```

其中 *SHTentry* 被展开的次数为 *Number_{sht}*, 也即 SHT 一共有 *Number_{sht}* 行表项。接下来, *Section* 可以被展开为以下几种 Section: *.text*, *.rodata*, *.data*, *.symtab*, *.rel.text*, *.rel.data*, 这是按照它们在 SHT 中的 *sh_name* 所确定的。它们对第一行 ([0]) 的偏移以及行数由 *sh_offset* 以及 *sh_size* 所确定。例如:

```

0 25
1 4
2 .text,0x0,6,10
3 .data,0x0,16,3
4 .symtab,0x0,19,4
5 .rel.text,0x0,23,2

```

从文本中被解析的 SHT 被存放在结构体中:

```

/src/headers/linker.h

1 typedef struct
2 {
3     char      sh_name[MAX_CHAR_SECTION_NAME];
4     uint64_t  sh_addr;
5     uint64_t  sh_offset;
6     uint64_t  sh_size;
7 } sh_entry_t;

```

对于 *.text*, 它的语法很简单。其中表达式 *Instruction* 按照我们在第??章中所描述的语法进行展开, 展开的次数为 SHT 表项中的 *sh_size*:

```

.text → Instruction
Instruction → ε | Op F \n Instruction

```

例如:

```

6 push    %rbp
7 mov     %rsp,%rbp
8 sub     $0x10,%rsp
9 mov     $0x2,%esi
10 lea    0x000000000000000(%rip),%rdi
11 callq  0x0000000000000000
12 mov     %rax,-0x8(%rbp)
13 mov     -0x8(%rbp),%rax
14 leaveq 
15 retq

```

<https://github.com/yangminz>
<https://space.bilibili.com/4564101>
<https://www.zhihu.com/people/zhao-yang-min> 计算机系统

.rodata 与 .data 一样，为了方便，我们将所有数值对齐扩展为 64 Bits，每一行记一个 16 进制数：

```
.rodata → NumberLine  
.data → NumberLine  
NumberLine → ε | HexNumber \n NumberLine
```

```
16 0x0000000012340000  
17 0x000000000000abcd  
18 0x0000000f00000000
```

.symtab 的语法与其他表类 Section 是相似的：

```
.symtab → STentry  
SHTentry → st_name,bind,type,st_shndx,st_value,st_size \n STentry  
19 array,STB_GLOBAL,STT_OBJECT,.data,0,2  
20 column,STB_GLOBAL,STT_OBJECT,.data,2,1  
21 main,STB_GLOBAL,STT_FUNC,.text,0,10  
22 sum,STB_GLOBAL,STT_NOTYPE,SHN_UNDEF,0,0
```

描述 .symtab 的结构体与枚举类型如下：

```
/src/headers/linker.h  
1 typedef enum  
2 {  
3     STB_LOCAL,  
4     STB_GLOBAL,  
5     STB_WEAK  
6 } st_bind_t;  
7  
8 typedef enum  
9 {  
10    STT_NOTYPE,  
11    STT_OBJECT,  
12    STT_FUNC  
13 } st_type_t;  
14  
15 typedef struct  
16 {  
17     char      st_name[MAX_CHAR_SECTION_NAME];  
18     st_bind_t bind;  
19     st_type_t type;  
20     char      st_shndx[MAX_CHAR_SECTION_NAME];  
21     uint64_t   st_value;  
22     uint64_t   st_size;  
23 } st_entry_t;
```

关于重定位，我们给出 .rel.text 以及 .rel.data 的语法：

```
.rel.text → RelocEntry  
.rel.data → RelocEntry  
RelocEntry → offset,column,type,symbol,addend \n RelocEntry  
23 4,7,R_X86_64_PC32,0,-4  
24 5,7,R_X86_64_PLT32,3,-4
```

它们所对应的结构体与枚举类型如下。其中 `rl_entry_t.r_row` 与 `r_col` 标志了引用的位置 [P]，为了简单起见，引用的占位符总为 8-Byte 的“0x0000000000000000”。在 `Elf64_Rela` 中，标记位置的结构域是以 Byte 为单位的 `r_offset`。对于我们的 `.elf.txt` 文本而言，我们用文本行索引 (`r_row`) 与行内列索引 (`r_col`) 更为自然：

```
/src/headers/linker.h  
1 typedef enum  
2 {  
3     R_X86_64_32,  
4     R_X86_64_PC32,  
5     R_X86_64_PLT32,  
6 } reltype_t;  
7  
8 typedef struct  
9 {  
10    uint64_t    r_row;  
11    uint64_t    r_col;  
12    reltype_t   type;  
13    uint32_t    sym;  
14    int64_t    r_addend;  
15 } rl_entry_t;
```

最后，这些文本行的总行数应当为 `Numbertxt`。至此，我们已经描述了可执行可连接文本的结构，它的实现在以下函数中，我们还可以增加过滤注释等功能，提高 `.elf.txt` 的可读性：

```
/src/linker/parseElf.c  
1 // interface exposed to shared object library staticLinker.so  
2 void parse_elf (const char *filename, elf_t *elfp);  
从磁盘中被读入的 .elf.txt 被结构体 elf_t 所描述：
```

```
/src/headers/linker.h  
1 typedef struct  
2 {  
3     // buffer for whole text  
4     uint64_t    line_count;  
5     char        buffer[MAX_ELF_FILE_LENGTH][MAX_ELF_FILE_WIDTH];  
6     // Section Header Table  
7     uint64_t    sht_count;  
8     sh_entry_t  *sht;  
9     // Symbol Table
```

```
10     uint64_t      symt_count;
11     st_entry_t    *symt;
12     // Relocation Table
13     uint64_t      reltext_count;
14     rl_entry_t   *reltext;
15     uint64_t      reldata_count;
16     rl_entry_t   *reldata;
17 } elf_t;
```

1.5.2 符号解析

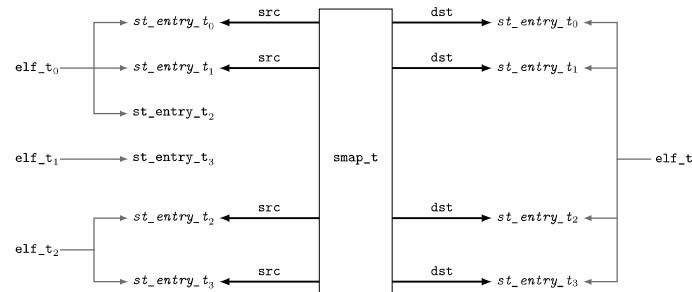
从磁盘中读取 .elf.txt 文件至结构体 `elf_t` 以后，我们开始进行静态链接。为此，我们设计这样的接口：读入一个 `elf_t` 的指针数组，这个数组的每一个元素指向一个 .elf.txt 文件；对于输出的 EOF，我们暂时也用 `elf_t` 结构体来表示。在头文件 /src/headers/linker.h 中，暴露函数 `link_elf`，负责静态链接所有读入的 ELF 文件：

```
/src/headers/linker.h
1 void link_elf(
2     elf_t **srcs, int num_srcs,
3     elf_t *dst);
```

`link_elf` 的第一步工作是进行符号解析。我们需要扫描所有的 ELF 文件，对每一个文件扫描它的符号表，然后将这些符号储存在一个内部的符号表中。这个内部的符号表可以被轻易地转换为 EOF 最终的符号表。从信息上看，内部符号表需要比 EOF 的 `.syms` 储存更多的信息，方便我们合并 Section 和对符号引用的重定位。为此，我们设计数据结构如下：

```
/src/linker/staticlink.c
1 typedef struct
2 {
3     elf_t      *elf; // src elf file
4     st_entry_t *src; // src symbol
5     st_entry_t *dst; // dst symbol
6 } smap_t;
```

这个数据结构建立了输入 ELF 文件以及它们的符号，与输出 EOF 文件之间的映射关系：



<https://github.com/yangminz>
<https://space.bilibili.com/4564101>
<https://www.zhihu.com/people/zhao-yang-min> 计算机系统

图 12: smap_t 在 ELF 文件与 EOF 文件之间的符号映射

smap_t 指向最终写入 EOF 中的一个符号，它可能是全局符号，也可能是静态符号（我们不考虑 Weak Bind）。其中指针 smap_t.src 指向输入的 ELF 文件中的符号表表项，指针 smap_t.elf 指向该 ELF 文件，指针 smap_t.dst 指向输出的 EOF 文件中的符号表表项。

函数 symbol_processing 以及 simple_resolution 用来完成符号解析。遍历每一个输入的 ELF 文件，以及该 ELF 文件的符号表 elf_t.symt，对符号表表项与当前缓存在 smap_t 中的符号进行对比，一旦发现同名的全局符号，则进行对比两者的优先程度，进行符号解析：

```
/src/linker/staticlink.c

1 static void symbol_processing(
2     elf_t **srcs, int num_srcs, elf_t *dst,
3     smap_t *smap_table, int *smap_count);
4 static void simple_resolution(
5     st_entry_t *sym, elf_t *sym_elf, smap_t *candidate);
```

1.5.3 Section 合并

ELF 中的 Section 合并为 EOF 中的 Segment，我们首先需要计算 EOF 全文本的行数 line_count，SHT 的行数 sht_count，以及 SHT 本身，并且将这几个结果写入文本缓存 buffer。

计算 SHT 的方法很简单。考虑到 smap_t 缓存了所有最终写入 EOF 的符号，因此遍历 smap_t 数组，累加每一个 Section 中的符号行数 smap_t.src->st_size，即可得到 sh_entry_t.sh_size，以及下一个 Section 的起始地址 sh_entry_t.sh_offset。因为 EOF 中不再有 COMMON 段，因此我们只需要考虑四种 Section: .text, .rodata, .data, .symtab。考虑到 .text Segment 的运行时起始地址是 0x00400000，因此其余 Segment 的运行时地址也可以计算得到了。这些计算通过函数 compute_section_header 实现：

```
/src/linker/staticlink.c

1 static void compute_section_header(
2     elf_t *dst,
3     smap_t *smap_table, int *smap_count);
```

接着，按照 EOF 的 SHT 中 Segment 的顺序，将所有 ELF 的各个 Section 合并为 Segment。遍历 ELF 文件，遍历 ELF 文件的符号表，对比 smap_t 中缓存的映射，将会被写入 EOF 的符号合并到 EOF 的 buffer 中。这部分计算通过函数 merge_section 实现：

```
/src/linker/staticlink.c

1 static void merge_section(
2     elf_t **srcs, int num_srcs, elf_t *dst,
3     smap_t *smap_table, int *smap_count);

至此，整个 EOF 的 elf_t.buffer 已经完成写入了，所剩下的工作就是完成符号引用的重定位 1.5.4。
```

1.5.4 符号引用重定位

最后，关于重定位，我们用以下函数实现：

```
/src/linker/staticlink.c

1 static void relocation_processing(elf_t **srcs, int num_srcs, elf_t *dst,
2     smap_t *smap_table, int *smap_count);
```

调用函数 `relocation_processing` 时，我们已经将各个 ELF 文件的 Section 合并为 Segment 了。与此同时，我们也可以确定到 EOF 各个 Segment 加载到内存以后的运行时地址。因此，每一个符号的运行时地址，我们也可以计算得到了。关于符号的运行时地址，我们通过以下函数计算：

```
/src/linker/staticlink.c  
1 static uint64_t get_symbol_runtime_address(elf_t *dst, st_entry_t *sym);
```

由于我们简化了数据模型，只有 `inst_t` 作为指令类型，`uint64_t` 作为数据类型，因此它们都是定长的，计算运行时地址非常简单：

符号 sym 所在 EOF 段	符号 sym 对 Segment 运行时起始地址的 Byte 偏移
.text	<code>sym->st_value * sizeof(inst_t)</code>
.rodata	<code>sym->st_value * sizeof(uint64_t)</code>
.data	<code>sym->st_value * sizeof(uint64_t)</code>

.text 段的运行时起始地址为 `0x00400000`，.rodata 则在.text 段后，并且以 `uint64_t` 对齐，.data 段也一样。关于各个段的运行时起始地址，其实可以在合并 Section 的时候完成计算。

关于符号引用的重定位，我们可以遍历输入的 ELF 文件，对当前 ELF 文件的两个重定位表：`.rel.text` 与 `.rel.data` 进行第二层遍历，这样就找到了每一个重定位项。对于重定位项，我们通过 `rl_entry_t.r_row` 和 `rl_entry_t.r_col` 计算到发生符号引用的位置；根据 `rl_entry_t.sym` 在 `smap_table` 中查找到所引用的符号 `sym`，计算到 `sym` 的运行时地址。最后，根据 `rl_entry_t.type`，按照 PC 相对寻址或者绝对寻址，将 `sym` 的运行时地址写入发生引用的位置，就完成了重定位。

至此，我们建立在文本上的静态链接器就已经实现了。

1.6 位置无关代码与动态链接

最后，我们来讨论一下动态链接以及位置无关代码，作为整个链接话题的结束。

我们对动态链接的需求是强烈的。按照静态链接，每一次静态链接完成以后，所有 ELF 文件中的 `.text` 节与 `.data` 节都会被完整地复制到 EOF 文件中。以常用的 C 语言库函数 `malloc` 为例，如果有 n 个 EOF 程序文件，它们的源码中都引用了 `malloc` 的定义，那么 `malloc` 的 `.text` 与 `.data` 就会被复制 n 次。这对于磁盘上的程序文件而言还算可以接受，因为我们总是假设磁盘有足够的储存空间。但是内存是更加稀缺的资源，如果这 n 个程序都在内存上运行，采用静态链接的话，那么内存上就有存在 `malloc` 的 n 份拷贝，这无疑是资源的浪费。

因此，更合理的策略是只在内存上保留一份 `malloc` 的代码，特别考虑到 `.text` 是只读的，多次拷贝更显得没有意义²。这些在内存中只有一份拷贝，被多个进程共享的数据（.text 段、.rodata 段），就是共享库（Shared Library）。在程序的运行时，共享库以动态链接（Dynamic Linking）的方式被程序引用，实现符号解析与重定位。

共享库作为.so 文件存在磁盘上（一个常见的 C 语言共享库是 `libc.so`），在运行时内存中，被加载到 Mmap 区，也即内存映射（Memory Mapping）。如图14，Mmap 在 Heap 与 Stack 之间，并且和 Heap 一样，向上增长。

² 不仅如此，从虚拟内存的角度考虑，只留一份备份可以提高页（Page）的命中率，减少页面的换入换出。具体见第22章。而对于 .data 段，考虑到两个进程可能同时修改一个全局变量，为了隔离进程，因此读写段不得不对每一个进程做一次拷贝。

动态链接与静态链接一样，需要处理符号解析与重定位的问题，这部分的流程与静态链接是相似的，我们就不予讨论了。除此以外，动态链接有它独有的问题。由于我们不能确定所使用共享库的数量，因此不可能对共享库进行 Segment 合并：例如进程 P_1 使用了共享库 L_1 ，进程 P_2 使用了共享库 L_1 与 L_2 ，那么，当 P_1 与 P_2 都在运行时， L_1 与 L_2 都需要驻留在 Mmap 中，并且各自独立占据一片物理内存的空间。但是，从 P_1 的角度看，它的虚拟内存中并不存在 L_2 ， P_1 也应该能够使用 L_2 部分的虚拟内存空间。

为了实现这一点，共享库需要支持在任意地址被加载，而不是从固定的起始地址开始。这样，每一个共享库的运行时起始地址，直到运行时，才能被操作系统确定。相反，如果共享库在编译时就假定了自己被加载的虚拟地址³，那么，当 P_1 想要使用 L_2 部分的内存空间时，就会发生地址冲突。因此，我们（编译器）需要设计文件的格式，使得 EOF 程序文件 P 对共享库 L 内的符号 S_L 引用，与 L 的运行时虚拟地址无关。

1.6.1 Global Offset Table

为了解决这些与动态链接有关的困难，我们采取一种被称为位置无关代码（Position-Independent Code, PIC）的技术。PIC 的实现依赖于一个简单的事实：EOF 文件的 Segment 是按照它们在磁盘文件中的结构，被复制到运行时内存中的。因此，在 .text, .rodata, .data 构成的连续虚拟内存中，对任意两个符号（函数或变量） S_1 与 S_2 ，它们在磁盘上的地址相差：

$$\Delta_{1,2} = S_1.\text{st_value} - S_2.\text{st_value}$$

那么，在进程的内存空间中， S_1 与 S_2 的虚拟地址依然相差 $\Delta_{1,2}$ 。这样，如果我们需要引用一个绝对地址，那么，我们只需要知道共享库的起始地址就可以了。

PIC 通过全局偏移表（Global Offset Table, GOT）得到实现。GOT 被储存在 .data 段中，它对 .text 段中任意一个 Byte 的距离，在磁盘上与内存中是不变的。GOT 的每一条表项大小为 8 Bytes，为被引用的共享库 L 中符号 S 的运行时绝对地址 A_S 。引用 L 的，既可以是 EOF 文件，也可以是另一个共享库，它们在磁盘上时，GOT 为空表，因为编译器无法确定 L 的运行时地址。当引用者被加载到内存以后，动态链接器根据引用者的重定位表与符号表找到 S 的绝对地址 A_S ，再将 A_S 写入 GOT。

我们考虑一个例子。假定用户程序在 .text 中引用了一个全局变量 `extern int a;`

```
test.c
1 extern int a;
2 void func()
3 {
4     int local_a = a;
5 }
```

如果将 test.c 编译为 ELF 文件：`gcc -c test.c -o test.o`，那么，引用 a 的代码会采用 PC 相对寻址的方法，并且产生重定位项，标记静态链接需要修正的位置：`mov 0x0(%rip),%eax`。

如果将 test.c 编译为共享库：`gcc -fPIC -shared -o test.so test.c`，那么，test.so 在引用另一个共享库中的全局变量 a。所以 test.so 需要通过 GOT 与动态链接器才能确定 a 的地址：

```
1 mov    0x200a3b(%rip),%rax
2 mov    (%rax),%rbx
```

其中 PC 相对寻址 `0x200a3b(%rip)` 找到 GOT 的表项，将 a 的运行时 A_a 从 GOT 中写入寄存器 `%rax`。然后将内存中该地址的值（`%rax`）拷贝给寄存器 `%rbx`，于是就得到了 a 在运行时的值，如图13。其中相对

³实际上，普通的 EOF 程序文件就是假定自己总是在 `0x00400000` 被加载的。

寻址 `0x200a3b` 正体现了 $\Delta_{1,2}$ 在磁盘上与内存中的不变性。并且，我们没有修改 `.text` 段（只读），而只修改了 `.data` 段（读写）。

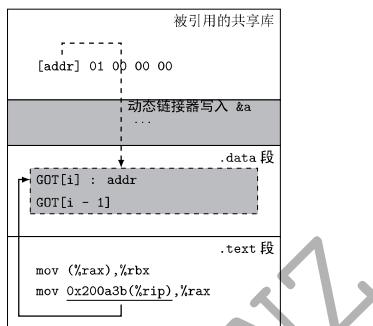


图 13: GOT 表引用绝对地址

EOF 文件引用共享库的方法也是一样的。

1.6.2 Procedure Linkage Table

GOT 解决了对共享库中全局变量的引用，但是我们还要处理对函数的调用。为此，我们需要函数连接表 (Procedure Linkage Table, PLT)。同样的，调用共享库中函数，不可以修改用户程序 `.text` 段中 `call` 指令的参数，因此我们需要采用 GOT 与 PLT 相结合的方式实现。

PLT 实现在 `.text` 段中，它的每一个表项大小为 16 Bytes，内容为若干指令。这样看来，PLT 的表项就像是 `goto` 的标签，`rip` 可跳转到该地址。

```
prog.c
1 #include <stdlib.h>
2 void main()
3 {
4     int *a = malloc(64 * sizeof(int));
5 }
```

以 `malloc` 为例。用户程序的 `.text` 段中，由于我们不知道 `malloc` 的运行时地址，因此在执行 `call` 指令时，我们跳转到 `malloc` 对应的 PLT 表项，`malloc@plt`。

注意，下面我们所标注的地址，都是运行时地址。EOF 程序文件的 `.text` 起始偏移为 `0x540`，加载到运行时地址 `0x00400000`。因此，运行时地址对于 EOF 文件偏移的 Byte 差值为 `0x00400000 - 0x540 = 0x003FFAC0`。

```
1      400117:    callq  3ffcc8 <malloc@plt>
PLT 表为:
```

```
1 <.plt>
2     3ffd0:    pushq  0x200aaa(%rip)
3     3ffd6:    jmpq   *0x200aac(%rip)      # jump to dynamic linker
4     3fffd:    nopl    0x0(%rax)
5
6 <malloc@plt>:
7     3ffe0:    jmpq   *0x200aaa(%rip)      # jump to GOT
8     3ffe6:    pushq  $0x0                 # prepare dynamic linker's argument
9     3ffb2:    jmpq   3ffd0 <.plt>
10
11 <.text>
```

malloc@plt 的第一条 jmpq 指令利用了位置无关性，取 GOT 表中缓存的 malloc 地址。执行 jmpq 指令时，%rip 的值为下一条指令的地址 0x3ffe6，此时 PC 相对地址计算为 $0x3ffe6 + 0x200aaa = 0x600a90$ 。这个地址是 GOT 表中的 [1] 项：

```
1 <_GLOBAL_OFFSET_TABLE_>:
2     600a78:    88 08 60 00 00 00 00 00      # address of .dynamic
3     ...
4     600a80:    00 00 00 00 00 00 00 00
5     600a90:    e6 ff 3f 00 00 00 00 00      # run-time address of malloc
6     ...
7
8 <.data>
```

可以看到，地址 600a90 上的 8-Byte 数据为 3ffe6，于是 jmpq 指令实际执行的是 jmpq 0x3ffe6，也就是回到返回地址，执行指令 pushq \$0x0。

这是因为动态链接采用了动态链接采用了延迟绑定（Lazy Binding）的技术，按照需要进行动态链接，提高性能。因为用户程序通常只使用共享库中的少数几个函数，而共享库本身提供了大量的函数。如果计算共享库中所有的函数地址，会消耗大量时间在不会被调用的函数上。因此，我们采用延迟绑定的技术，只在 malloc 第一次被调用时，计算它的运行时地址。显然，这个技术的思路与一些经典的设计模式是相同的，并且给出了汇编代码上的实现。

600a90 上的 8-Byte 数据应当为 malloc 的运行时地址，但 GOT 刚从磁盘中加载时，所填写的值是 jmpq 的返回地址。于是%rip 返回到 pushq \$0x0，跳转到 <.plt>，然后跳转到动态链接器的代码段。

当动态链接器取得 malloc 的运行时地址以后，例如 0x00007fffff41b6b0，动态链接器向 600a90 填写 0x00007fffff41b6b0。如果再次调用 malloc@plt，执行到 jmpq *0x200aaa(%rip) 时，600a90 上的 8-Byte 数据为 0x00007fffff41b6b0，这样就不必经过 <.plt> 项中的动态链接器，而可以直接跳转到 malloc：

```
1 <_GLOBAL_OFFSET_TABLE_>:
2     600a78:    88 08 60 00 00 00 00 00
3     ...
4     600a90:    b0 b6 41 ff ff 7f 00 00      # run-time address of malloc
5     ...
```

动态链接调用 malloc 的过程如图14所示：用户程序跳转到 malloc@plt，取 GOT 的缓存。在第一次调用 malloc@plt 时，跳转到动态链接器，动态链接器向 GOT 填写 malloc 的运行时地址，方便接下来的调用。并且，动态链接器将%rip 指针指向 malloc 的入口。

malloc 所在的共享库储存全局变量，用双向循环链表管理堆上的内存，这些全局变量是会被读写的，因此每个进程管理一份私有的备份。用户程序请求 malloc 分配内存时，malloc 会检查链表，如果发现有

足够大的空闲节点，`malloc`会将空闲节点分裂为合适的大小，返回给用户程序。如果整个堆上都不存在足够的空闲块，`malloc`会执行系统调用 `brk`。

系统调用 `brk` 会触发一次中断，从用户态的程序陷入到内核态，由内核托管 CPU 与内存的全部资源。注意，这一中断的过程，其实与动态链接的过程是相似的，其实我们可以将操作系统看作一个安全高效的共享库。操作系统根据收到的中断号（`brk`）分配一个物理页给当前进程，并且更新页表映射，使运行时虚拟内存上的堆向上增长。等到中断返回用户态，`malloc`根据新得到的空闲页分配内存，返回给用户进程。

这样，用户程序就得到了堆上的一片内存。这就是 `int *a = malloc(64 * sizeof(int));` 的全部故事。

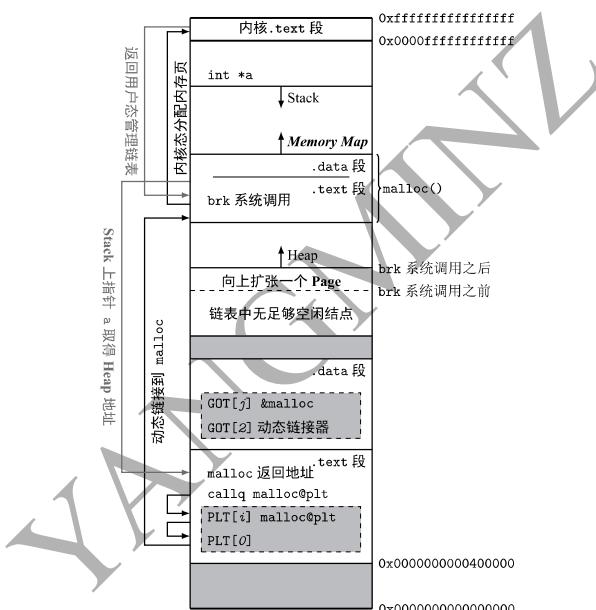


图 14: `int *a = malloc(64 * sizeof(int))` 动态链接

我们之所以大费周章讨论链接，是因为链接器距离加载与运行很近。了解链接，更了解程序在磁盘和在运行时的模型，才方便我们理解进程的概念。不仅如此，动态链接与共享库的概念，将帮助我们理解操作系统的内核。这样，在我们讨论第22章的时候，我们心中才有更清晰的模型。所以，学习链接不是目的，而是手段。关于链接器，CMU的教科书 [[bryant2003computer](#)] 介绍了链接器的主要工作。除此以外，Sun公司的技术文档 [[solaris2004](#)] 有更详细和全面的介绍。