目录

[一、ELF头 2](#_Toc61276039)

[二、符号信息 14](#_Toc61276040)

# 一、ELF头

源代码如下：

|  |
| --- |
| /\*  \* SimpleSection.c  \*  \* Linux:  \* gcc -c SimpleSection.c  \*  \* Windows  \* cl SimpleSection.c /c /Za  \*/  int printf(const char \*format, ...);  int global\_init\_var = 84;  int global\_uninit\_var;  void func1(int i) {  printf("%d\n", i);  }  int main(void) {  static int static\_var = 85;  static int static\_var2;  int a = 1;  int b;  func1(static\_var + static\_var2 + a + b);  return a;  } |

编译：gcc -c SimpleSection.c

查看section信息：objdump -h SimpleSection.o

（-h打印出段的信息）

|  |
| --- |
| (main ?:3 ✗) compile\_load\_study objdump -h SimpleSection.o  SimpleSection.o： 文件格式 elf64-x86-64  节：  Idx Name Size VMA LMA File off Algn  0 .text 00000057 0000000000000000 0000000000000000 00000040 2\*\*0  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE  1 .data 00000008 0000000000000000 0000000000000000 00000098 2\*\*2  CONTENTS, ALLOC, LOAD, DATA  2 .bss 00000004 0000000000000000 0000000000000000 000000a0 2\*\*2  ALLOC  3 .rodata 00000004 0000000000000000 0000000000000000 000000a0 2\*\*0  CONTENTS, ALLOC, LOAD, READONLY, DATA  4 .comment 0000002a 0000000000000000 0000000000000000 000000a4 2\*\*0  CONTENTS, READONLY  5 .note.GNU-stack 00000000 0000000000000000 0000000000000000 000000ce 2\*\*0  CONTENTS, READONLY  6 .eh\_frame 00000058 0000000000000000 0000000000000000 000000d0 2\*\*3  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA |

标记值：

CONTENTS表示该段在文件中存在

将上面的段图示：

|  |
| --- |
|  |

查看段的内容和反编译的指令：objdump -s -d SimpleSection.o

（-s 将所有段的内容以16进制打印出来，-d 打印出相应代码的反汇编）

|  |
| --- |
| SimpleSection.o： 文件格式 elf64-x86-64  Contents of section .text:  0000 554889e5 4883ec10 897dfc8b 45fc89c6 UH..H....}..E...  0010 488d3d00 000000b8 00000000 e8000000 H.=.............  0020 0090c9c3 554889e5 4883ec10 c745f801 ....UH..H....E..  0030 0000008b 15000000 008b0500 00000001 ................  0040 c28b45f8 01c28b45 fc01d089 c7e80000 ..E....E........  0050 00008b45 f8c9c3 ...E...  Contents of section .data:  0000 54000000 55000000 T...U...  Contents of section .rodata:  0000 25640a00 %d..  Contents of section .comment:  0000 00474343 3a202855 62756e74 7520372e .GCC: (Ubuntu 7.  0010 352e302d 33756275 6e747531 7e31382e 5.0-3ubuntu1~18.  0020 30342920 372e352e 3000 04) 7.5.0.  Contents of section .eh\_frame:  0000 14000000 00000000 017a5200 01781001 .........zR..x..  0010 1b0c0708 90010000 1c000000 1c000000 ................  0020 00000000 24000000 00410e10 8602430d ....$....A....C.  0030 065f0c07 08000000 1c000000 3c000000 .\_..........<...  0040 00000000 33000000 00410e10 8602430d ....3....A....C.  0050 066e0c07 08000000 .n......  Disassembly of section .text:  0000000000000000 <func1>:  0: 55 push %rbp  1: 48 89 e5 mov %rsp,%rbp  4: 48 83 ec 10 sub $0x10,%rsp  8: 89 7d fc mov %edi,-0x4(%rbp)  b: 8b 45 fc mov -0x4(%rbp),%eax  e: 89 c6 mov %eax,%esi  10: 48 8d 3d 00 00 00 00 lea 0x0(%rip),%rdi # 17 <func1+0x17>  17: b8 00 00 00 00 mov $0x0,%eax  1c: e8 00 00 00 00 callq 21 <func1+0x21>  21: 90 nop  22: c9 leaveq  23: c3 retq  0000000000000024 <main>:  24: 55 push %rbp  25: 48 89 e5 mov %rsp,%rbp  28: 48 83 ec 10 sub $0x10,%rsp  2c: c7 45 f8 01 00 00 00 movl $0x1,-0x8(%rbp)  33: 8b 15 00 00 00 00 mov 0x0(%rip),%edx # 39 <main+0x15>  39: 8b 05 00 00 00 00 mov 0x0(%rip),%eax # 3f <main+0x1b>  3f: 01 c2 add %eax,%edx  41: 8b 45 f8 mov -0x8(%rbp),%eax  44: 01 c2 add %eax,%edx  46: 8b 45 fc mov -0x4(%rbp),%eax  49: 01 d0 add %edx,%eax  4b: 89 c7 mov %eax,%edi  4d: e8 00 00 00 00 callq 52 <main+0x2e>  52: 8b 45 f8 mov -0x8(%rbp),%eax  55: c9 leaveq  56: c3 retq |

.data段保存的是那些已经初始化了的全局变量，全局静态变量和局部静态变量。

在printf用到的字符串常亮”%d\n”，它是一种只读数据，它被放到了”.rodata”段中。

“.rodata”段存放的是只读数据，一般是程序里面的只读变量（如const修饰的全局变量）和字符串常量。

.bss段存放的是未初始化的全局静态变量和局部静态变量

.comment段存放的是未初始化的全局变量

为了验证变量所在的地方，以下给出了一个测试案例：

|  |
| --- |
| // 全局静态变量 - 初始化  static int g\_s\_init\_var = 12; // 存放在.data段  // 全局静态变量 - 未初始化  static int g\_s\_uninit\_var; // 存放在.bss段  // 全局非静态变量 - 初始化  int g\_init\_var = 13; // 存放在.data段  // 全局非静态变量 - 未初始化  int g\_uninit\_var; // 存放在.COMMON符号段  // 全局静态常量  static const int g\_s\_c\_init\_var = 122; // 存放在.rodata段  // 全局常量  const int g\_c\_init\_var = 123; // 存放在.rodata段  int main(void) {  // 非全局静态变量 - 初始化  static int s\_init\_var = 14; // 存放在.data段  // 非全局静态变量 - 未初始化  static int s\_uninit\_var; // 存放在.bss段  // 非全局常量  const int c\_init\_var = 144; // 存放在栈段  // 非全局非静态变量 - 初始化  int init\_var = 15; // 存放在栈段  // 非全局非静态变量 - 未初始化  int uninit\_var; // 存放在栈段  return 0;  } |

根据上面的案例，使用gcc -c 只编译，之后使用objdump -x查看如下：

|  |
| --- |
| SimpleVariable.o： 文件格式 elf64-x86-64  SimpleVariable.o  体系结构：i386:x86-64， 标志 0x00000011：  HAS\_RELOC, HAS\_SYMS  起始地址 0x0000000000000000  节：  Idx Name Size VMA LMA File off Algn  0 .text 00000019 0000000000000000 0000000000000000 00000040 2\*\*0  CONTENTS, ALLOC, LOAD, READONLY, CODE  1 .data 0000000c 0000000000000000 0000000000000000 0000005c 2\*\*2  CONTENTS, ALLOC, LOAD, DATA  2 .bss 00000008 0000000000000000 0000000000000000 00000068 2\*\*2  ALLOC  3 .rodata 00000008 0000000000000000 0000000000000000 00000068 2\*\*2  CONTENTS, ALLOC, LOAD, READONLY, DATA  4 .comment 0000002a 0000000000000000 0000000000000000 00000070 2\*\*0  CONTENTS, READONLY  5 .note.GNU-stack 00000000 0000000000000000 0000000000000000 0000009a 2\*\*0  CONTENTS, READONLY  6 .eh\_frame 00000038 0000000000000000 0000000000000000 000000a0 2\*\*3  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA  SYMBOL TABLE:  0000000000000000 l df \*ABS\* 0000000000000000 SimpleVariable.c  0000000000000000 l d .text 0000000000000000 .text  0000000000000000 l d .data 0000000000000000 .data  0000000000000000 l d .bss 0000000000000000 .bss  0000000000000000 l O .data 0000000000000004 g\_s\_init\_var  0000000000000000 l O .bss 0000000000000004 g\_s\_uninit\_var  0000000000000000 l d .rodata 0000000000000000 .rodata  0000000000000000 l O .rodata 0000000000000004 g\_s\_c\_init\_var  0000000000000004 l O .bss 0000000000000004 s\_uninit\_var.1802  0000000000000008 l O .data 0000000000000004 s\_init\_var.1801  0000000000000000 l d .note.GNU-stack 0000000000000000 .note.GNU-stack  0000000000000000 l d .eh\_frame 0000000000000000 .eh\_frame  0000000000000000 l d .comment 0000000000000000 .comment  0000000000000004 g O .data 0000000000000004 g\_init\_var  0000000000000004 O \*COM\* 0000000000000004 g\_uninit\_var  0000000000000004 g O .rodata 0000000000000004 g\_c\_init\_var  0000000000000000 g F .text 0000000000000019 main  RELOCATION RECORDS FOR [.eh\_frame]:  OFFSET TYPE VALUE  0000000000000020 R\_X86\_64\_PC32 .text |

有时候希望变量或某部分代码能够放到指定的段中去，以实现某些指定的功能。在全局变量或函数之上加上”\_\_attribute\_\_((section(“name”)))”属性就可以把相应的变量或函数放到以”name”作为段名的段中了。

ELF结构

|  |
| --- |
|  |

ELF文件头

可以使用readelf来查看ELF头部，得到结果如下：

|  |  |
| --- | --- |
| (main ?:6 ✗) compile\_load\_study readelf -h SimpleSection.o  ELF 头：  Magic： 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00  类别: ELF64  数据: 2 补码，小端序 (little endian)  版本: 1 (current)  OS/ABI: UNIX - System V  ABI 版本: 0  类型: REL (可重定位文件)  系统架构: Advanced Micro Devices X86-64  版本: 0x1  入口点地址： 0x0  程序头起点： 0 (bytes into file)  Start of section headers: 1104 (bytes into file)  标志： 0x0  本头的大小： 64 (字节)  程序头大小： 0 (字节)  Number of program headers: 0  节头大小： 64 (字节)  节头数量： 13  字符串表索引节头： 12    #define EI\_NIDENT (16)  typedef struct  {  unsigned char e\_ident[EI\_NIDENT]; /\* Magic number and other info \*/  Elf32\_Half e\_type; /\* Object file type \*/  Elf32\_Half e\_machine; /\* Architecture \*/  Elf32\_Word e\_version; /\* Object file version \*/  Elf32\_Addr e\_entry; /\* Entry point virtual address \*/  Elf32\_Off e\_phoff; /\* Program header table file offset \*/  Elf32\_Off e\_shoff; /\* Section header table file offset \*/  Elf32\_Word e\_flags; /\* Processor-specific flags \*/  Elf32\_Half e\_ehsize; /\* ELF header size in bytes \*/  Elf32\_Half e\_phentsize; /\* Program header table entry size \*/  Elf32\_Half e\_phnum; /\* Program header table entry count \*/  Elf32\_Half e\_shentsize; /\* Section header table entry size \*/  Elf32\_Half e\_shnum; /\* Section header table entry count \*/  Elf32\_Half e\_shstrndx; /\* Section header string table index \*/  } Elf32\_Ehdr;  转换为：  typedef struct  {  unsigned char e\_ident[EI\_NIDENT]; /\* Magic number and other info \*/  uint16\_t e\_type; /\* Object file type \*/  uint16\_t e\_machine; /\* Architecture \*/  int32\_t e\_version; /\* Object file version \*/  uint32\_t e\_entry; /\* Entry point virtual address \*/  uint32\_t e\_phoff; /\* Program header table file offset \*/  uint32\_t e\_shoff; /\* Section header table file offset \*/  int32\_t e\_flags; /\* Processor-specific flags \*/  uint16\_t e\_ehsize; /\* ELF header size in bytes \*/  uint16\_t e\_phentsize; /\* Program header table entry size \*/  uint16\_t e\_phnum; /\* Program header table entry count \*/  uint16\_t e\_shentsize; /\* Section header table entry size \*/  uint16\_t e\_shnum; /\* Section header table entry count \*/  uint16\_t e\_shstrndx; /\* Section header string table index \*/  } Elf32\_Ehdr;   1. e\_ident      1. e\_type文件类型      1. e\_machine机器类型      1. Section Header Table段表   ELF文件中有很多各种各样的段，这个段表（Section Header Table）就是保存这些段的基本属性的结构。段表在ELF文件中的位置由ELF文件头的“e\_shoff”成员决定  objdump -h”命令只是把ELF文件中关键的段显示了出来，而省略了其他的辅助性的段，比如：符号表、字符串表、段名字符串表、重定位表等。我们可以使用readelf工具来查看ELF文件的段，它显示出来的结果才是真正的段表结构。   |  | | --- | | There are 13 section headers, starting at offset 0x450:  节头：  [号] 名称 类型 地址 偏移量  大小 全体大小 旗标 链接 信息 对齐  [ 0] NULL 0000000000000000 00000000  0000000000000000 0000000000000000 0 0 0  [ 1] .text PROGBITS 0000000000000000 00000040  0000000000000057 0000000000000000 AX 0 0 1  [ 2] .rela.text RELA 0000000000000000 00000340  0000000000000078 0000000000000018 I 10 1 8  [ 3] .data PROGBITS 0000000000000000 00000098  0000000000000008 0000000000000000 WA 0 0 4  [ 4] .bss NOBITS 0000000000000000 000000a0  0000000000000004 0000000000000000 WA 0 0 4  [ 5] .rodata PROGBITS 0000000000000000 000000a0  0000000000000004 0000000000000000 A 0 0 1  [ 6] .comment PROGBITS 0000000000000000 000000a4  000000000000002a 0000000000000001 MS 0 0 1  [ 7] .note.GNU-stack PROGBITS 0000000000000000 000000ce  0000000000000000 0000000000000000 0 0 1  [ 8] .eh\_frame PROGBITS 0000000000000000 000000d0  0000000000000058 0000000000000000 A 0 0 8  [ 9] .rela.eh\_frame RELA 0000000000000000 000003b8  0000000000000030 0000000000000018 I 10 8 8  [10] .symtab SYMTAB 0000000000000000 00000128  0000000000000198 0000000000000018 11 11 8  [11] .strtab STRTAB 0000000000000000 000002c0  000000000000007c 0000000000000000 0 0 1  [12] .shstrtab STRTAB 0000000000000000 000003e8  0000000000000061 0000000000000000 0 0 1  Key to Flags:  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),  L (link order), O (extra OS processing required), G (group), T (TLS),  C (compressed), x (unknown), o (OS specific), E (exclude),  l (large), p (processor specific) |   typedef struct  {  Elf32\_Word sh\_name; /\* Section name (string tbl index) \*/  Elf32\_Word sh\_type; /\* Section type \*/  Elf32\_Word sh\_flags; /\* Section flags \*/  Elf32\_Addr sh\_addr; /\* Section virtual addr at execution \*/  Elf32\_Off sh\_offset; /\* Section file offset \*/  Elf32\_Word sh\_size; /\* Section size in bytes \*/  Elf32\_Word sh\_link; /\* Link to another section \*/  Elf32\_Word sh\_info; /\* Additional section information \*/  Elf32\_Word sh\_addralign; /\* Section alignment \*/  Elf32\_Word sh\_entsize; /\* Entry size if section holds table \*/  } Elf32\_Shdr;     * sh\_type段的类型        * sh\_flag段的标志位   段的标志位表示该段在进程虚拟地址空间中的属性，比如是否可写，是否可执行等     * sh\_link、sh\_info段的链接信息   段的类型是与链接相关的（不论是动态链接或静态链接），比如重定位表、符号表等     * 重定位段   SimpleSection.o中有一个叫做“.rel.text”的段，它的类型（sh\_type）为“SHT\_REL”，也就是说它是一个重定位表（Relocation Table）。正如我们最开始所说的，链接器在处理目标文件时，须要对目标文件中某些部位进行重定位，即代码段和数据段中那些对绝对地址的引用的位置。这些重定位的信息都记录在ELF文件的重定位表里面，对于每个须要重定位的代码段或数据段，都会有一个相应的重定位表。比如SimpleSection.o中的“.rel.text”就是针对“.text”段的重定位表，因为“.text”段中至少有一个绝对地址的引用，那就是对“printf”函数的调用；而“.data”段则没有对绝对地址的引用，它只包含了几个常量，所以SimpleSection.o中没有针对“.data”段的重定位表“.rel.data”。   1. 字符串表   ELF文件中用到了很多字符串，比如段名、变量名等。因为字符串的长度往往是不定的，所以用固定的结构来表示它比较困难。一种很常见的做法是把字符串集中起来存放到一个表，然后使用字符串在表中的偏移来引用字符串 |

# 二、符号信息

* nm使用

使用nm查看SimpleSection.o的符号信息

|  |
| --- |
| (main ✓) compile\_load\_study nm SimpleSection.o  0000000000000000 T func1  0000000000000000 D global\_init\_var  U \_GLOBAL\_OFFSET\_TABLE\_  0000000000000004 C global\_uninit\_var  0000000000000024 T main  U printf  0000000000000004 d static\_var.1802  0000000000000000 b static\_var2.1803 |

* readelf查看符号信息

|  |
| --- |
| (main U:1 ?:2 ✗) compile\_load\_study readelf -s SimpleSection.o  Symbol table '.symtab' contains 17 entries:  Num: Value Size Type Bind Vis Ndx Name  0: 0000000000000000 0 NOTYPE LOCAL DEFAULT UND  1: 0000000000000000 0 FILE LOCAL DEFAULT ABS SimpleSection.c  2: 0000000000000000 0 SECTION LOCAL DEFAULT 1  3: 0000000000000000 0 SECTION LOCAL DEFAULT 3  4: 0000000000000000 0 SECTION LOCAL DEFAULT 4  5: 0000000000000000 0 SECTION LOCAL DEFAULT 5  6: 0000000000000004 4 OBJECT LOCAL DEFAULT 3 static\_var.1802  7: 0000000000000000 4 OBJECT LOCAL DEFAULT 4 static\_var2.1803  8: 0000000000000000 0 SECTION LOCAL DEFAULT 7  9: 0000000000000000 0 SECTION LOCAL DEFAULT 8  10: 0000000000000000 0 SECTION LOCAL DEFAULT 6  11: 0000000000000000 4 OBJECT GLOBAL DEFAULT 3 global\_init\_var  12: 0000000000000004 4 OBJECT GLOBAL DEFAULT COM global\_uninit\_var  13: 0000000000000000 36 FUNC GLOBAL DEFAULT 1 func1  14: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND \_GLOBAL\_OFFSET\_TABLE\_  15: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND printf  16: 0000000000000024 51 FUNC GLOBAL DEFAULT 1 main |

符号表的结构很简单，它是一个Elf32\_Sym结构（32位ELF文件）的数组，每个Elf32\_Sym结构对应一个符号。这个数组的第一个元素，也就是下标0的元素为无效的“未定义”符号。Elf32\_Sym的结构定义如下：

typedef struct {

Elf32\_Word st\_name;

Elf32\_Addr st\_value;

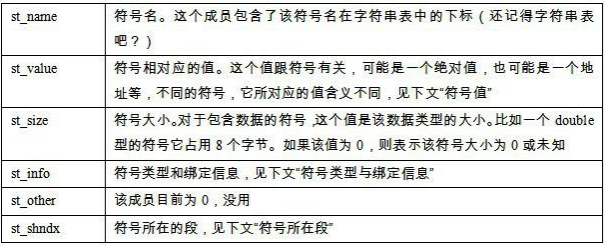
Elf32\_Word st\_size;

unsigned char st\_info;

unsigned char st\_other;

Elf32\_Half st\_shndx;

} Elf32\_Sym;



1. 符号类型和绑定信息（st\_info） 该成员低4位表示符号的类型（Symbol Type），高4位表示符号绑定信息（Symbol Binding）





1. 符号所在段（st\_shndx）如果符号定义在本目标文件中，那么这个成员表示符号所在的段在段表中的下标；但是如果符号不是定义在本目标文件中，或者对于有些特殊符号，sh\_shndx的值有些特殊



1. 符号值（st\_value） 我们前面已经介绍过，每个符号都有一个对应的值，如果这个符号是一个函数或变量的定义，那么符号的值就是这个函数或变量的地址, 按下面这几种情况区别对待:
2. 在目标文件中，如果是符号的定义并且该符号不是“COMMON块”类型的（即st\_shndx不为SHN\_COMMON），则st\_value表示该符号在段中的偏移。即符号所对应的函数或变量位于由st\_shndx指定的段，偏移st\_value的位置。这也是目标文件中定义全局变量的符号的最常见情况，比如SimpleSection.o中的“func1”、“main”和“global\_init\_var”。
3. 如果符号是“COMMON块”类型的（即st\_shndx为SHN\_COMMON），则st\_value表示该符号的对齐属性。比如SimpleSection.o中的“global\_uninit\_var”
4. 可执行文件中，st\_value表示符号的虚拟地址。这个虚拟地址对于动态链接器来说十分有用