ELF 文件格式学习

Daniel Wood 20110301

写在前面的话：

在阅读 TCP/IP 相关书籍的时候遇到 ELF 文件格式(ELF 和 TCP/IP 无直接关系)，所以查阅了网上很多这方面的资料，现在整理成篇，方便自己以后 review。

如果你是 ELF 的高手，那么你可以跳过了，如果你和我一样，对 ELF 文件格式一窍不通，或者你还未曾听说过这种格式（虽然你在平时工作中已经接触了很多这种格式的文件），那么你也许可以看看这篇基础的文章。如果你确实无聊的话，可以动手实践一下。

参考文献：

[1] Executable and Linkable Format (ELF).pdf

[2] 结合实例解读 ELF 文件-阅读笔记 bkbll(bkbll@cnhonker.net, kbll@tom.com)

[3] About ELF

[4]

[5]

自 UNIX 系统实验室(USL)开发和发布了 Executable and linking Format(ELF)这样的二进制格式以后，在*nix 系统上 ELF 就取代了 out 可执行文件格式，成为了主要的目标文件格式。

注：这里的目标文件是指（可暂时理解为）gcc 用–c，-o，-shared 所产生的.o，可执行(默认是 a.out)，.so 文件。

1. ELF 文件类型

目标文件（也就是 ELF 文件）格式主要三种：

- 可重定向文件（Relocatable file）：文件保存着代码和适当的数据，用来和其他的目标文件一起来创建一个可执行文件或者是一个共享目标文件。由编译器和汇编器生成，将由链接器处理。

- 可执行文件（Executable File）：文件保存着一个用来执行的程序；该文件指出了 exec(BA_OS)如何来创建程序进程映象。所有重定向和符号都解析完成了，如果存在共享库的链接，那么将在运行时解析。

- 共享目标文件（Shared object file）：就是所谓的共享库。文件保存着代码和合适的数据，用来被下面的两个链接器链接。第一个是连接编辑器[请参看 ld(SD_CMD)]，可以和其他的可重定向和共享目标文件来创建其他的目标文件。第二个是动态链接器，联合一个可执行文件和其他的共享目标文件来创建一个进程映象。包含链接时所需的符号信息和运行时所需的代码。

下面用实例来了解这三种 ELF 文件。

代码 hello.c

```
#include <stdio.h>

int main()

{

        printf("Hello World\n");

        return 0;

}
```

用 gcc 编译 hello.c 文件

编译环境：

```
$ uname -a

Linux ubuntu804 2.6.24-26-generic #1 SMP Tue Dec 1 18:37:31 UTC 2009 i686 GNU/Linux

$ gcc --version

gcc (GCC) 4.2.4 (Ubuntu 4.2.4-1ubuntu4)

Copyright (C) 2007 Free Software Foundation, Inc.

This is free software; see the source for copying conditions.  There is NO
```

```
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

1）编译重定向文件：

```
$ gcc -c hello.c

$file hello.o

hello.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
```

2）编译可执行文件：

```
$ gcc -o hello hello.o

$ file hello

hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.6.8, dynamically linked (uses shared libs), not stripped
```

可以用 ldd 命令查看 hello 这个可执行文件动态链接的共享库。

```
$ ldd hello

        libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7d99000)

        /lib/ld-linux.so.2 (0xb7ef8000)
```

3）编译共享目标文件：

```
$ gcc -shared hello.c  -o hello.so

$ file hello.so

hello.so: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), not stripped
```

注：这里只是为了说明共享目标文件的 ELF 格式，这并不是平常的共享库，因为里面没有函数接口等东东。

2. ELF 文件格式

目标文件既要参与程序链接又要参与程序执行。出于方便性和效率考虑，目标文件格式提供了两种并行视图，分别反映了这些活动的不同需求。编译器，链接器把它看作是 sections 的集合，loader 把它看作是 segments 的集合。

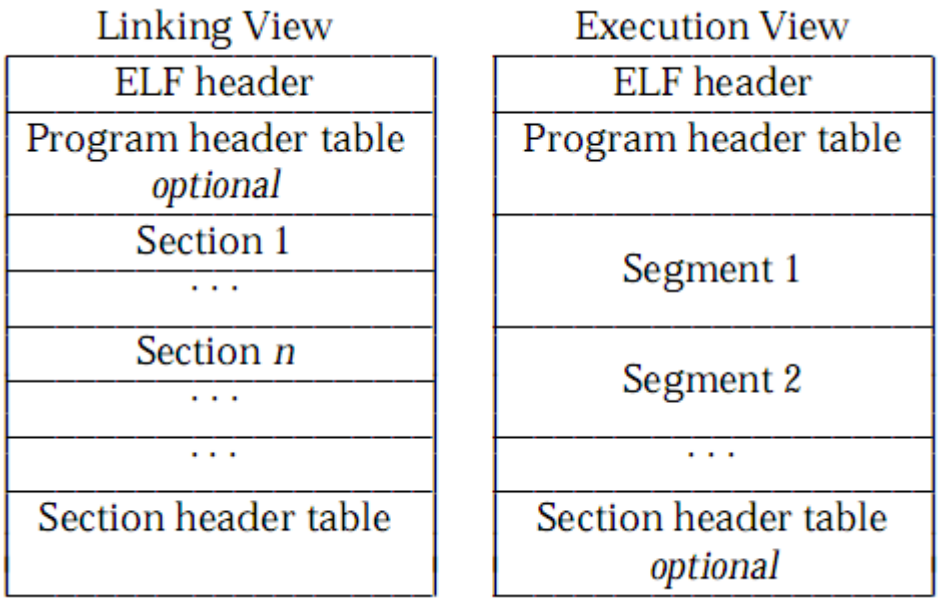

表 1：Object File Format

From：Executable and Linkable Format (ELF).pdf

一般的 ELF 文件包括三个索引表：ELF header，Program header table，Section header table。

ELF header：在文件的开始，保存了路线图(road map)，描述了该文件的组织情况。

Program header table：告诉系统如何创建进程映像。用来构造进程映像的目标文件必须具有程序头部表，可重定位文件不需要这个表。

Section header table：包含了描述文件节区的信息，每个节区在表中都有一项，每一项给出诸如节区名称、节区大小这类信息。用于链接的目标文件必须包含节区头部表，其他目标文件可以有，也可以没有这个表。

在 32 位的机器上，ELF 中各数据类型所占字节数，各类型定义可查看 include\linux\elf.h：

Name	Size	Alignment	Purpose
Elf32_Addr	4	4	Unsigned program address
Elf32_Half	2	2	Unsigned medium integer
Elf32_Off	4	4	Unsigned file offset
Elf32_Sword	4	4	Signed large integer
Elf32_Word	4	4	Unsigned large integer
unsigned char	1	1	Unsigned small integer

表 2：32-Bit Data Type

From：Executable and Linkable Format (ELF).pdf

2.1 ELF header

文件的最开始几个字节给出如何解释文件的提示信息。这些信息独立于处理器，也独立于文件中的其余内容。ELF header 是一个 elf32_hdr 结构体，定义在 include\linux\elf.h

```
#define EI_NIDENT       16

typedef struct elf32_hdr{
  unsigned char   e_ident[EI_NIDENT];
  Elf32_Half      e_type;
  Elf32_Half      e_machine;
  Elf32_Word      e_version;
  Elf32_Addr      e_entry;  /* Entry point */
  Elf64_Off e_phoff;                /* Program header table file offset */
  Elf64_Off e_shoff;                /* Section header table file offset */
  Elf32_Word      e_flags;
  Elf32_Half      e_ehsize;
  Elf32_Half      e_phentsize;
  Elf32_Half      e_phnum;
  Elf32_Half      e_shentsize;
  Elf32_Half      e_shnum;
```

```
  Elf32_Half      e_shstrndx;
} Elf32_Ehdr;
```

对照表 2，Elf32_Half 占 2 个字节，unsigned char 占 1 个字节，所以数组 e_ident 占 16 个字节，其他的 Elf32_Word，Elf32_Off，Elf32_Addr 都占 4 个字节，所以得出 Elf32_Ehdr 这个结构体总共占了 52 个字节（16+4*5+2*8=52）。所以 ELF 文件开头的 52 个字节属于 ELF header，知道这点我们用 hexdump，objdump，readelf 等工具查看 ELF 文件的时候有用。

用 readelf 命令/工具去读上面产生的 hello.o 这个可重定向文件。

这里我们只用 readelf 命令的-h 选项查看它们的 ELF header，以便对它有个感性认识^-^

```
$ readelf -h hello.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          232 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
```

```
Size of section headers:           40 (bytes)
Number of section headers:         11
Section header string table index: 8
```

这是重定向文件 hello.o 的 ELF header 里面的内容，从这里可以狠清楚地看出 Elf32_Ehdr 结构体中的内容。

我们可以通过 hexdump 命令/工具去查看 ELF 文件，它显示十六进制的内容。

```
$ hexdump -C -n 52 hello.o
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  01 00 03 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
00000020  e8 00 00 00 00 00 00 00  34 00 00 00 00 00 28 00  |........4.....(.|
00000030  0b 00 08 00                                       |....|
00000034
```

下面结合 readelf 和 hexdump 两个命令对 ELF header 的查看结果来分析 Elf32_Ehdr 结构体中各项的意义。

首先是第一项 Elf32_Ehdr 结构体中的数组：unsigned char e_ident[EI_NIDENT];

数组 e_ident 的下标是通过一组宏定义去索引的。

linux 代码版本是 2.6.29

```
#define EI_MAG0         0       /* e_ident[] indexes */
#define EI_MAG1         1
#define EI_MAG2         2
#define EI_MAG3         3
#define EI_CLASS        4
#define EI_DATA         5
#define EI_VERSION      6
#define EI_OSABI        7
#define EI_PAD          8
```

数组中各个成员代表不同的意义：

Name	Value	Purpose
EI_MAG0	0	File identification
EI_MAG1	1	File identification
EI_MAG2	2	File identification
EI_MAG3	3	File identification
EI_CLASS	4	File class
EI_DATA	5	Data encoding
EI_VERSION	6	File version
EI_PAD	7	Start of padding bytes
EI_NIDENT	16	Size of e_ident[]

表 3：e_ident[ ] Identification Indexes

From：Executable and Linkable Format (ELF).pdf

- e_ident 数组

前四项：e_ident[EI_MAG0].. e_ident[EI_MAG3]

e_ident 数组总的前四项是一个 Magic Number，代表这是一个 ELF 文件。任何一个 ELF 文件的这四项值是固定的：

Name	Value	Position
ELFMAG0	0x7f	e_ident[EI_MAG0]
ELFMAG1	'E'	e_ident[EI_MAG1]
ELFMAG2	'L'	e_ident[EI_MAG2]
ELFMAG3	'F'	e_ident[EI_MAG3]

表 4：Magic Number

From：Executable and Linkable Format (ELF).pdf

相对应的就是 hexdump -C -n 52 hello.o 命令中得到的前面四个字节：7f 45 4c 46

ps：十六进制 0x45=十进制 69='E' 十六进制 0x4c=十进制 76='L' 十六进制 0x46=十进制 70='F'

第五项：e_ident[EI_CLASS]：1 字节：01：32 位目标文件

标识文件的类型，其值如下表

Name	Value	Meaning
ELFCLASSNONE	0	Invalid class
ELFCLASS32	1	32-bit objects
ELFCLASS64	2	64-bit objects

表 5：value of e_ident[EI_CLASS]

From：Executable and Linkable Format (ELF).pdf

第六项：e_ident[EI_DATA]：1 字节：01：高位在前。

标识数据编码方式，不具体展开。

第七项：e_ident[EI_VERSION]：1 字节：01：current version

表明 ELF header 的版本号。

- e_type：2 字节：01 00：可重定向文件。

  目标文件的类型，即可重定向，可执行，共享目标文件等。

Name	Value	Meaning
ET_NONE	0	No file type
ET_REL	1	Relocatable file
ET_EXEC	2	Executable file
ET_DYN	3	Shared object file
ET_CORE	4	Core file
ET_LOPROC	0xff00	Processor-specific
ET_HIPROC	0xffff	Processor-specific

表 6：ELF file type

From：Executable and Linkable Format (ELF).pdf

- e_machine：2 字节：03 00：Intel 80386

  目标体系结构类型，不展开。

- e_version：4 字节：01 00 00 00：current version。ELF 文件版本。

- e_entry：4 字节：00 00 00 00：无程序入口。程序入口的虚拟地址。

- e_phoff：4 字节：00 00 00 00：偏移量为 0。Program Header Table 的偏移量。
- e_shoff：4 字节：e8 00 00 00 ：Section header table 偏移量为 0xe8=232。

  Section Header Table 的偏移量。
- e_flags：4 字节：00 00 00 00。未指定。保存与文件相关的， 特定于处理器的标志。 标志名称采用 EF_machine_flag 的格式
- e_ehsize：2 字节：34 00：ELF header 表头大小为 0x34=52。ELF header 的大小。
- e_phentsize：2 字节：00 00：Program Header 大小为 0，因为没有。每个 Program Header 大小。
- e_phnum：2 字节：00 00：Program Header 总数为 0。Program Header 的总数。
- e_shentsize：2 字节：28 00：Section Header 大小为 0x28=40。每个 Section Header 大小。
- e_shnum：2 字节：0b 00：Section Header 的总数为 0x0b=11。Section Header 的总数。
- e_shstrndx：2 字节：08 00：索引值为 8。Section header string table index，Section 的字符串表在 section header table 的索引值，这个值很重要。

注意看它的 Start of program headers 以及 Size of program headers 都是 0，说明这个 hello.o 里面没有 program header table，这也论证了上节对 ELF 结构的说明，并不是所有的 ELF 文件都需要有 program header table 或 section headers table 的。

下面是 hello 可执行文件的 ELF header，这个输出狠重要，下面都围绕这个输出讲解。

```
$ readelf -h hello
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
```

```
Version:                           1 (current)
OS/ABI:                            UNIX - System V
ABI Version:                       0
Type:                              EXEC (Executable file)
Machine:                           Intel 80386
Version:                           0x1
Entry point address:               0x80482f0
Start of program headers:          52 (bytes into file)
Start of section headers:          3220 (bytes into file)
Flags:                             0x0
Size of this header:               52 (bytes)
Size of program headers:           32 (bytes)
Number of program headers:         7
Size of section headers:           40 (bytes)
Number of section headers:         36
Section header string table index: 33
```

2.2 Section

Section，中文翻译是节区，在这里我就用 section。在一个 ELF 文件中有一个 section header table，通过它我们可以定位到所有的 section，而 ELF header 中的 e_shoff 变量就是保存 section header table 入口对文件头的偏移量。而每个 section 都会对应一个 section header，所以只要在 section header table 中找到每个 section header，就可以通过 section header 找到你想要的 section。

事实上 section header table 是一个 section header 结构体 Elf32_Shdr 类型的数组，我们可以从 ELF header 结构体的 e_shoff 变量中得到数组首地址，这个地址是相对 ELF 文件开头来说的。如果我们知道某个 section 在 section header table 的索引，那么我们就可以直接计算得到该 section header 在文件中的位置，然后通过 section header 中的内容，进一步得到 section 的位置，从而读取 section 其中的内容。

下面以可执行文件 hello 为例，以保存字符串表的 section 为例来讲解读取某个 section 的过程。选择保存字符串表的 section 因为我们从 ELF header 中就可以得知它在 section header table 的索引值为 33。

注：一般 ELF 中有两个字符串表 section 一个名为“.strtab”，一个名为“.shstrtab”，分别为字符串表和段表字符串表。我们分析的是段表字符串表。

先给出 sections header 结构体的定义：

include\linux\elf.h

```
typedef struct {
  Elf32_Word    sh_name;
  Elf32_Word    sh_type;
  Elf32_Word    sh_flags;
  Elf32_Addr    sh_addr;
  Elf32_Off     sh_offset;
  Elf32_Word    sh_size;
  Elf32_Word    sh_link;
  Elf32_Word    sh_info;
  Elf32_Word    sh_addralign;
  Elf32_Word    sh_entsize;
} Elf32_Shdr;
```

Elf32_Shdr 结构体中的成员见名知意，就不具体讲解了。

同样的我们可以计算出这个结构体的大小是 40 字节（4*10），这个值和上面 readelf 读出来的 Size of section headers 值相同。从 readelf –h hello 中，我们还获得有关 section 的信息就是，有 36 个 section header，字符串表在 section header table 中的索引值为 33。

下面用命令先去看看 hello 中所有的 section header。

```
$ readelf -S hello
There are 36 section headers, starting at offset 0xc94:

Section Headers:
```

```
[Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
[ 0]                   NULL            00000000 000000 000000 00      0   0  0
[ 1] .interp           PROGBITS        08048114 000114 000013 00   A  0   0  1
[ 2] .note.ABI-tag     NOTE            08048128 000128 000020 00   A  0   0  4
[ 3] .hash             HASH            08048148 000148 000028 04   A  5   0  4
[ 4] .gnu.hash         GNU_HASH        08048170 000170 000020 04   A  5   0  4
[ 5] .dynsym           DYNSYM          08048190 000190 000050 10   A  6   1  4
[ 6] .dynstr           STRTAB          080481e0 0001e0 00004a 00   A  0   0  1
[ 7] .gnu.version      VERSYM          0804822a 00022a 00000a 02   A  5   0  2
[ 8] .gnu.version_r    VERNEED         08048234 000234 000020 00   A  6   1  4
[ 9] .rel.dyn          REL             08048254 000254 000008 08   A  5   0  4
[10] .rel.plt          REL             0804825c 00025c 000018 08   A  5  12  4
[11] .init             PROGBITS        08048274 000274 000030 00  AX  0   0  4
[12] .plt              PROGBITS        080482a4 0002a4 000040 04  AX  0   0  4
[13] .text             PROGBITS        080482f0 0002f0 00014c 00  AX  0   0 16
[14] .fini             PROGBITS        0804843c 00043c 00001c 00  AX  0   0  4
[15] .rodata           PROGBITS        08048458 000458 000014 00   A  0   0  4
[16] .eh_frame         PROGBITS        0804846c 00046c 000004 00   A  0   0  4
[17] .ctors            PROGBITS        08049470 000470 000008 00  WA  0   0  4
[18] .dtors            PROGBITS        08049478 000478 000008 00  WA  0   0  4
[19] .jcr              PROGBITS        08049480 000480 000004 00  WA  0   0  4
[20] .dynamic          DYNAMIC         08049484 000484 0000d0 08  WA  6   0  4
[21] .got              PROGBITS        08049554 000554 000004 04  WA  0   0  4
[22] .got.plt          PROGBITS        08049558 000558 000018 04  WA  0   0  4
[23] .data             PROGBITS        08049570 000570 00000c 00  WA  0   0  4
[24] .bss              NOBITS          0804957c 00057c 000004 00  WA  0   0  4
[25] .comment          PROGBITS        00000000 00057c 000126 00      0   0  1
```

```
[26] .debug_aranges    PROGBITS        00000000 0006a8 000050 00      0   0  8
[27] .debug_pubnames   PROGBITS        00000000 0006f8 000025 00      0   0  1
[28] .debug_info       PROGBITS        00000000 00071d 0001a7 00      0   0  1
[29] .debug_abbrev     PROGBITS        00000000 0008c4 00006f 00      0   0  1
[30] .debug_line       PROGBITS        00000000 000933 000129 00      0   0  1
[31] .debug_str        PROGBITS        00000000 000a5c 0000bb 01  MS  0   0  1
[32] .debug_ranges     PROGBITS        00000000 000b18 000040 00      0   0  8
[33] .shstrtab         STRTAB          00000000 000b58 000139 00      0   0  1
[34] .symtab           SYMTAB          00000000 001234 0004a0 10     35  55  4
[35] .strtab           STRTAB          00000000 0016d4 000206 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

可以从中得出索引值为 33 的 section header 是.shstrtab。也就是我们要查看的字符串表 section。

下面我们偷懒用 readelf 命令去查看.shstrtab 这个 section 中的内容，当然另一种方法（目前我能想到的）就是通过 hexdump 去查看，不过那样的话需要一点计算。

```
$ readelf -x 33  hello
Hex dump of section '.shstrtab':
  0x00000000 002e7379 6d746162 002e7374 72746162 ..symtab..strtab
  0x00000010 002e7368 73747274 6162002e 696e7465 ..shstrtab..inte
  0x00000020 7270002e 6e6f7465 2e414249 2d746167 rp..note.ABI-tag
  0x00000030 002e676e 752e6861 7368002e 64796e73 ..gnu.hash..dyns
  0x00000040 796d002e 64796e73 7472002e 676e752e ym..dynstr..gnu.
  0x00000050 76657273 696f6e00 2e676e75 2e766572 version..gnu.ver
  0x00000060 73696f6e 5f72002e 72656c2e 64796e00 sion_r..rel.dyn.
```

```
0x00000070 2e72656c 2e706c74 002e696e 6974002e .rel.plt..init..
0x00000080 74657874 002e6669 6e69002e 726f6461 text..fini..roda
0x00000090 7461002e 65685f66 72616d65 002e6374 ta..eh_frame..ct
0x000000a0 6f727300 2e64746f 7273002e 6a637200 ors..dtors..jcr.
0x000000b0 2e64796e 616d6963 002e676f 74002e67 .dynamic..got..g
0x000000c0 6f742e70 6c74002e 64617461 002e6273 ot.plt..data..bs
0x000000d0 73002e63 6f6d6d65 6e74002e 64656275 s..comment..debu
0x000000e0 675f6172 616e6765 73002e64 65627567 g_aranges..debug
0x000000f0 5f707562 6e616d65 73002e64 65627567 _pubnames..debug
0x00000100 5f696e66 6f002e64 65627567 5f616262 _info..debug_abb
0x00000110 72657600 2e646562 75675f6c 696e6500 rev..debug_line.
0x00000120 2e646562 75675f73 7472002e 64656275 .debug_str..debu
0x00000130 675f7261 6e676573 00              g_ranges.
```

下面用 hexdump 的方法去读取.shstrtab 这个 section 中的内容。

首先你从 ELF header 的 e_shoff 变量中得到的是 section header table 相对文件头的偏移量为 3220 字节（以 hello 为例）。每个 section header 的大小是 40 字节，.shstrtab 在 section header table 中的索引值为 33，所以我们首先可以得到.shstrtab 这个 section header 的偏移量：3220+40*33=4540。

我们先得到.shstrtab 的 section header 中的内容：

```
$ hexdump -s 4540 -n 40 -C hello
000011bc  11 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00  |................|
000011cc  58 0b 00 00 39 01 00 00  00 00 00 00 00 00 00 00  |X...9...........|
000011dc  01 00 00 00 00 00 00 00                          |........|
000011e4
```

对照上面的十六进制值和 section header 结构体 Elf32_Shdr，我们需要得到 sh_offset 这个变量的值，即 section 的第一个字节与文件头之间的偏移。这个变量是 section header 的 17-20 字节，所以我们得到 58 0b 00 00。那么这个 section 的首

地址是 0xb58=2904。我们还可以得到这个 section 的大小，在 sh_offset 后四个字节中，保存在变量 sh_size 中为 39 01 00 00：0x139=313。所以我们可以得到：

```
$ hexdump -s 2904 -n 313 -C hello
00000b58  00 2e 73 79 6d 74 61 62  00 2e 73 74 72 74 61 62  |..symtab..strtab|
00000b68  00 2e 73 68 73 74 72 74  61 62 00 2e 69 6e 74 65  |..shstrtab..inte|
00000b78  72 70 00 2e 6e 6f 74 65  2e 41 42 49 2d 74 61 67  |rp..note.ABI-tag|
00000b88  00 2e 67 6e 75 2e 68 61  73 68 00 2e 64 79 6e 73  |..gnu.hash..dyns|
00000b98  79 6d 00 2e 64 79 6e 73  74 72 00 2e 67 6e 75 2e  |ym..dynstr..gnu.|
00000ba8  76 65 72 73 69 6f 6e 00  2e 67 6e 75 2e 76 65 72  |version..gnu.ver|
00000bb8  73 69 6f 6e 5f 72 00 2e  72 65 6c 2e 64 79 6e 00  |sion_r..rel.dyn.|
00000bc8  2e 72 65 6c 2e 70 6c 74  00 2e 69 6e 69 74 00 2e  |.rel.plt..init..|
00000bd8  74 65 78 74 00 2e 66 69  6e 69 00 2e 72 6f 64 61  |text..fini..roda|
00000be8  74 61 00 2e 65 68 5f 66  72 61 6d 65 00 2e 63 74  |ta..eh_frame..ct|
00000bf8  6f 72 73 00 2e 64 74 6f  72 73 00 2e 6a 63 72 00  |ors..dtors..jcr.|
00000c08  2e 64 79 6e 61 6d 69 63  00 2e 67 6f 74 00 2e 67  |.dynamic..got..g|
00000c18  6f 74 2e 70 6c 74 00 2e  64 61 74 61 00 2e 62 73  |ot.plt..data..bs|
00000c28  73 00 2e 63 6f 6d 6d 65  6e 74 00 2e 64 65 62 75  |s..comment..debu|
00000c38  67 5f 61 72 61 6e 67 65  73 00 2e 64 65 62 75 67  |g_aranges..debug|
00000c48  5f 70 75 62 6e 61 6d 65  73 00 2e 64 65 62 75 67  |_pubnames..debug|
00000c58  5f 69 6e 66 6f 00 2e 64  65 62 75 67 5f 61 62 62  |_info..debug_abb|
00000c68  72 65 76 00 2e 64 65 62  75 67 5f 6c 69 6e 65 00  |rev..debug_line.|
00000c78  2e 64 65 62 75 67 5f 73  74 72 00 2e 64 65 62 75  |.debug_str..debu|
00000c88  67 5f 72 61 6e 67 65 73  00                       |g_ranges.|
00000c91
```

结论：通过 hexdump 和 readelf 得到的.shstrtab 的 section 结果相同。

2.3 Program

可执行文件或者共享目标文件的程序头部是一个结构数组（结构体为 Elf32_Phdr）， 每个结构描述了一个段或者系统准备程序执行所必需的其它信息。这里的段是指 segment，有些 segment 中保存着机器指令，有些保存着已初始化的变量，有些则作为进程镜像的一部分被操作系统读入内存。

下面给出 Program header 的结构体定义。

include\linux\elf.h

```
typedef struct elf32_phdr{
  Elf32_Word    p_type;
  Elf32_Off     p_offset;
  Elf32_Addr    p_vaddr;
  Elf32_Addr    p_paddr;
  Elf32_Word    p_filesz;
  Elf32_Word    p_memsz;
  Elf32_Word    p_flags;
  Elf32_Word    p_align;
} Elf32_Phdr;
```

我们从 ELF 中可以获得关于 program 的信息就是 Program Header Table 的偏移量 e_phoff: 52 (bytes into file)，Program Header 大小 e_phentsize：32 (bytes)，Program Header 总数 e_phnum：7。

```
$ readelf -l hello

Elf file type is EXEC (Executable file)
Entry point 0x80482f0
There are 7 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
```

```
  PHDR           0x000034 0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4
  INTERP         0x000114 0x08048114 0x08048114 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x00470 0x00470 R E 0x1000
  LOAD           0x000470 0x08049470 0x08049470 0x0010c 0x00110 RW  0x1000
  DYNAMIC        0x000484 0x08049484 0x08049484 0x000d0 0x000d0 RW  0x4
  NOTE           0x000128 0x08048128 0x08048128 0x00020 0x00020 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4


 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-
tag .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .t
ext .fini .rodata .eh_frame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag
   06
```