现代Linux采用ELF(Executable and Linking Format)做为其可连接和可执行文件的格式，因此ELF格式也向我们透出了一点Linux核内的情景，就像戏台维幕留下的一条未拉严的缝。 PC世界32仍是主流，但64位的脚步却已如此的逼近。如果你对Windows比较熟悉，本文还将时时把你带回到PE中，在它们的相似之处稍做比较。ELF文件以“ELF头”开始，后面可选择的跟随着程序头和节头。地理学用等高线与等温线分别展示同一地区的地势和气候，程序头和节头则分别从加载与连接角度来描述EFL文件的组织方式。

## ELF头

---

ELF头也叫ELF文件头，它位于文件中最开始的地方。

```
/usr/src/linux/include/linux/elf.h
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 */
  Elf32_Off     e_phoff;
```

```
    Elf32_Off    e_shoff;
    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;

#define EI_NIDENT  16
```

ELF 头中每个字段的含意如下:

**Elf32_Ehdr->e_ident[]** (Magic)

这个字段是 ELF 头结构中的第一个字段，在 elf.h 中 EI_NIDENT 被定义为 16，因此它占用 16 个字节。e_ident 的前四个字节顺次应该是 0x7f、 0x45、 0x4c、 0x46，也就是"\177ELF"。这是 ELF 文件的标志，任何一个 ELF 文件这四个字节都完全相同。

| 16 进制 | 8 进制 | 字母 |
| --- | --- | --- |
| 0x7f | 0177 | |
| 0x45 | | E |
| 0x4c | | L |

0x46　　F

第 5 个字节标志了 ELF 格式是 32 位还是 64 位，32 位是 1，64 位是 2。

第 6 个字节，在 0x86 系统上是 1，表明数据存储方式为低字节优先。

第 10 个字节，指明了在 e_ident 中从第几个字节开始后面的字节未使用。

**Elf32_Ehdr->e_type**　(Type)

ELF 文件的类型，1 表示此文件是重定位文件，2 表示可执行文件，3 表示此文件是一个动态连接库。

**Elf32_Ehdr->e_machine**　(Machine)

CPU 类型，它指出了此文件使用何种指令集。如果是 Intel 0x386 CPU 此值为 3，如果是 AMD 64 CPU 此值为 62 也就是 16 进制的 0x3E。

**Elf32_Ehdr->e_version**　(Version)

ELF 文件版本，为 1。

**Elf32_Ehdr->e_entry**　(Entry point address)

可执行文件的入口虚拟地址。此字段指出了该文件中第一条可执行机器指令在进程被正确加载后的内存地址！**(注：入口地址并不是可执**

行文件的第一个函数 -- main 函数的地址)。

Elf32_Ehdr->e_phoff (Start of program headers)

程序头在 ELF 文件中的偏移量。如果程序头不存在此值为 0。

Elf32_Ehdr->e_shoff (Start of section headers)

节头在 ELF 文件中的偏移量。如果节头不存在此值为 0。

Elf32_Ehdr->e_ehsize (Size of ELF header)

它描述了“ELF 头”自身占用的字节数。

Elf32_Ehdr->e_phentsize (Size of program headers)

程序头中的每一个结构占用的字节数。**程序头也叫程序头表**，可以被看做一个在文件中连续存储的结构数组，数组中每一项是一个结构，此字段给出了这个结构占用的字节大小。e_phoff 指出程序头在 ELF 文件中的起始偏移。

Elf32_Ehdr->e_phnum (Number of program headers)

此字段给出了程序头中保存了多少个结构。如果程序头中有 3 个结构则程序头(程序头表)在文件中占用了(3×e_phentsize)个字节的大小。

Elf32_Ehdr->e_shentsize (Size of section headers)

节头中每个结构占用的字节大小。节头与程序头类似也是一个结构数组，关于这两个结构的定义将分别在讲述程序头和节头的时候给出。

Elf32_Ehdr->e_shnum (Number of section headers)

节头中保存了多少个结构。

Elf32_Ehdr->e_shstrndx (Section header string table index)

这是一个整数索引值。节头可以看作是一个结构数组，用这个索引值做为此数组的下标，它在节头中指定的一个结构进一步给出了一个“字符串表”的信息，而这个字符串表保存着节头中描述的每一个节的名称，包括字符串表自己也是其中的一个节。

至此为止我们已经讲述了“ELF 头”，在此过程中提前提到的一些将来才用的概念，不必急于了解。现在读者可自己编写一个小程序来验证刚学到的知识，这有助于进一步的学习。elf.h 文件一般会存在于/usr/include 目录下，直接 include 它就可以。但我们能够验证的知识有限，当更多知识联系在一起的时候我们的理解正误才可以得到更好的验证。接下来我们再学习程序头。

```
# readelf -h a.out
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:          3228 (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
```

## 程序头(程序头表) -- Program Header

---

程序头有时也叫程序头表，它保存了一个结构数组(结构 Elf32_Phdr 的数组)。程序头是从加载执行的角度看待 ELF 文件的结果，从它的角度 ELF 文件被分成许多个段。每个段保存着用于不同目的的数据，有的段保存着机器指令，有的段保存着已经初始化的变量;有的段会做为进程映像的一部分被操作系统读入内存，有的段则只存在于文件中。

后面还会讲到 ELF 的节头，节头把 ELF 文件分成了许多节。ELF 文件的一部分常常是既在某一段中又在某一节中。Linux 和 Windows 的进程空间都采用的是平坦模式，没有 x86 的段概念，这里 ELF 中提到的段仅是文件的分段与 x86 的段没有任何联系。

/usr/src/linux/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;
```

### Elf32_Phdr->p_type

段的类型，它能告诉我们这个段里存放着什么用途的数据。此字段的值是在 elf.h 中定义了一些常量。例如 1(PT_LOAD)表示是可加载的段，这样的段将被读入程序的进程空间成为内存映像的一部分。段的种类再不断增加，例如 7(PT_TLS)在以前就没有定义，它表示用于线程局部存储。

### Elf32_Phdr->p_flags

段的属性。它用每一个二进制位表示一种属，相应位为 1 表示含有相应的属性，为 0 表示不含那种属性。其中最低位是可执行位，次低位是可写位，第三低位是可读位。如果这个字段的最低三位同时为 1 那就表示这个段中的数据加载以后既可读也可写而且可执行的。同样在 elf.h 文件中也定义了一此常量(PF_X、 PF_W、PF_R)来测试这个字段的属性，做为一个好习惯应该尽量使用这此常量。

### Elf32_Phdr->p_offset

该段在文件中的偏移。这个偏移是相对于整个文件的。

### Elf32_Phdr->p_vaddr

该段加载后在进程空间中占用的内存起始地址。

### Elf32_Phdr->p_paddr

该段的物理地地址。这个字段被忽略，因为在多数现代操作系统下物理地址是进程无法触及的。

### Elf32_Phdr->p_filesz

该段在文件中占用的字节大小。有些段可能在文件中不存在但却占用一定的内存空间，此时这个字段为 0。

### Elf32_Phdr->p_memsz

该段在内存中占用的字节大小。有些段可能仅存在于文件中而不被加载到内存，此时这个字段为 0。

### Elf32_Phdr->p_align

对齐。现代操作系统都使用虚拟内存为进程序提供更大的空间，分页技术功不可没，页就成了最小的内存分配单位，不足一页的按一页算。所以加载程序数据一般也从一页的起始地址开始，这就属于对齐。

尽管我给出了描述每个段信息的程序头结构，但我并不打算介绍任何一个具体类型的段所存储的内容，大多数情况下它们和节中保存的内容是一致的。我们只关心可以加载的段，但上面给出的信息应该足够了。好啦，你现在就是操作系统，你已经知道了组成程序的指令和数据都存放在文件的各个段中，通过程序头你知道它们在文件中的偏移和它们在文件中的大小，你就可以把这个段读到它的进程空间中以 p_vaddr 开始

的地址处。水平所限，我所能表达的必然不是精确的，为了更好理解程序头与进程加载，我设计了一个小实验并给出 C 语言代码 -- 代码可以精确的说明一切!

## 覆盖 ELF 可执行文件入口指令的实验

---

现在掌握了 ELF 头和程序头，从加载执行程序的角度可以说已对 ELF 文件有了初步的了解。为更好理解它，做个试验吧!

回忆一下程序头表把 ELF 文件分成了许多段，并告诉操作系统怎样把这些段读到内存里去。当操作系统已按程序头表的指示把 ELF 文件各个段的数据读入到内存中相应的地方以后，就可以说操作系统已建立了完整且正确的进程映像(如果不考虑依赖)，下一步就是要执行程序了。ELF 头的 e_entry 给出了第一条机器指令在内存中的地址，操作系统只要在某个时候将指令流引向那里就可以了。

这个猜测对不对呢，下面的这个实验将从某种角度来证明它。首先准备好一段代码 -- exit_print()，把这段代码写到 ELF 文件 -- hello 中，代码写入的位置恰恰是 ELF 文件的第一条机器指令在文件中的位置。这样当系统把这个修改过的可执行程序 -- hello 加载到内存时，它原来入口处的指令已经换成了我们准备的这段代码，程序的行为被完全改变。可是 ELF 头的 e_entry 给出的是内存地址而不文件偏移，所以这需要我们自己找到这个文件偏移。怎么找？运用刚刚掌握的知识。程序头

不是给出了文件中每一段对应的内存起始地址吗，还有每一段在内存中占了多少字节。只要遍历程序头中的每一个结构，看看哪个段的起始内存地址小于等于 e_entry 并且该地址加上该段内存大小又大于 e_entry，那么这个段就是程序第一条指令所在的段。第一条指令在段中偏移就是 e_entry 减去该段的 p_vaddr 所得的值:

第一条指令在整个文件中偏移 = 该段的 p_offset + (e_entry - 该段的 p_vaddr)

下面就是我准备的那段代码，它是一个 C 函数 exit_print()。对于这段代码有三点需要说明:

1) 这个函数中不能调用常用的库函数，因为若从 so 中取函数我们现在无法解决动态引入;如果采用静态连接，被调用函数有可能再调用其它函数，而被调用函在内存映像的地址、大小都不易掌握。

2) 这个段代码最好是位置无关代码，这样能减少这个实验的代码量，而使用全局或静态变量将使我们花更大代价来实现位置无关，所以这个函数不使用它们。

3) 这个函数只能在 IA32 机器上运行，若想在其它环境下做此实验必须修改它的一段汇编代码。

另外我们没有判断 ELF 文件是否为可执行文件。为了确信这段代码被运行，它将在控制台输出“Hello zxl ”之后就结束整个程序。

鉴于上面的两点说明，我们不能使用 printf 和 malloc 输出字符串和为它分配内存，也没有把完整的字符串做为变量存储，而是用了堆栈

中的局部变量,这将导致栈中内存分配。把字符串放到 strHello 中用了四条C语句。注意,前三条中每条语句放入的四个字符的顺序是颠倒的,这是 x86 低字节优先存储造成的。最后一条 C 语句放入一个回车符'\n',字符串没有以 0 结尾。

```
void exit_print()
{
   char strHello[20];
   *((unsigned long*)&strHello[0])='lleH';
   *((unsigned long*)&strHello[4])='      o';
   *((unsigned long*)&strHello[8])='     zxl';
   strHello[12]='\n';

__asm__ volatile ("int $0x80; movl $0,%%ebx; movl $1,%%eax; int $0x80"\
      :                \
      : "a"((long) 4),        \
      "b" ((long)1),          \
      "c" ((long)(&strHello[0])),\
      "d" ((long)13));
}
```

exit_print 用到了一些汇编语法,不防在这里先复习下汇编,如果你

不喜欢看汇编，可以直接阅读后面给出的完整 C 代码，我可以保证它实现上面想要的功能。gcc 内部汇编以“__asm__”开始，关键字 volatile 告诉 gcc 不要优化。汇编体以一对小括号包围并以分号结束：输入部分把寄存器 EAX 置为 4，这是 write 系统调用的功能号;EBX 置为 1，这 write 系统调用使用的文件句柄，1 代表标准输出设备;寄存器 ECX 置为字符串的起始地址;寄存器 EDX 置为 13，这代表字符串的长度是 13 个字节;我们不关心系统返回值因此输出部分没有内容;接下来 int $0x80 把刚才的设置到寄存器的参数传给内核完成打印功能！后面在把寄存器 EBX 置 0、EAX 置 1 后又是一次系统调用，它将结束当前进程并把 EBX 中的 0 返回给父进程。函数 exit_print 说明完毕！

---

```
#include <stdio.h>

int main()
{
   printf("hello\n");
}

# gcc hello.c -o hello
# ./hello
hello
```

下面给出这个试验程序的完整代码，它被存为 mod_entry.c 文件，exit_print 函数也在其中。下面代码将替换 ELF 文件 hello 的入口地址(将 hello 文件放置在和 mod_entry.c 相同的目录下)。

```
//文件名 :mod_entry.c
//功能 : 覆盖 ELF 可执行文件指令入口

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <elf.h>

void exit_print()
{
   char strHello[20];
   *((unsigned long*)&strHello[0])='lleH';
   *((unsigned long*)&strHello[4])='      o';
   *((unsigned long*)&strHello[8])='     lxz';
   strHello[12]='\n';

   __asm__ volatile ("int $0x80; movl $0,%%ebx; movl $1,%%eax;
int $0x80"\
```

```
        :\
        : "a"((long) 4),\
        "b" ((long)1),\
        "c" ((long)(&strHello[0])),\
        "d" ((long)13));

}

/*
   AMD 64 下的调用 write 系统调用可能有如下形式,其中__syscall 是中断调用指令, __NR_write 是系统功能号:
   __asm__ volatile (__syscall \
   :                \
   : "a" (__NR_write),"D" ((long)(1)),"S" ((long)(&strHello[0])),"d" ((long)(13)) : "r11","rcx","memory" );
*/

//简单判断是否是 ELF 文件
int IsElf(Elf32_Ehdr *pEhdr)
{
   if( pEhdr->e_ident[EI_MAG0] !=0x7f
```

```
    || pEhdr->e_ident[EI_MAG1] !='E'
    || pEhdr->e_ident[EI_MAG2] !='L'
    || pEhdr->e_ident[EI_MAG3] !='F'
    || pEhdr->e_machine      !=EM_386)//是否在 x86 上运行
        return 0;
    return 1;
}

//将文件 hFile，从 pos 处开始读取 count 个字节到缓冲区 buf 中
int ReadAt(int hFile, int pos, void *buf, int count)
{
    if(pos == lseek(hFile, pos, SEEK_SET))
        return read(hFile, buf, count);
    return -1;
}

//将缓冲区 buf 中的内容(count 个字节)，写入文件 hFile 中(从 pos
处开始写入)
int WriteAt(int hFile, int pos, void* buf, int count)
{
    if(pos == lseek(hFile, pos, SEEK_SET))
        return write(hFile, buf, count);
```

```
    return -1;
}

//找到程序第一条指令所在的段，并把该段的程序头结构读到 pPhdr 指向的结构中
//参数entry为第一条可执行机器指令在进程被正确加载后的内存地址(Elf32_Ehdr->e_entry)
int FileEntryIndex(int hFile, Elf32_Ehdr* pEhdr, Elf32_Phdr *pPhdr, unsigned long entry)
{
    int i;
    for(i = 0; i < pEhdr->e_phnum; i++) {
        if(sizeof(*pPhdr) !=
                ReadAt(hFile,
                    pEhdr->e_phoff + i*pEhdr->e_phentsize,
                    pPhdr,
                    sizeof(*pPhdr)))
            return 0;

        if(entry >= pPhdr->p_vaddr && entry < (pPhdr->p_vaddr + pPhdr->p_memsz))
            return 1;
```

```
    }
    return 0;
}

int main()
{
    int hFile;
    int offset, size;
    Elf32_Ehdr ehdr;
    Elf32_Phdr phdr;

    //以读写方式打开文件
    hFile = open("hello", O_RDWR, 0);
    if(hFile < 0)
        return -1;

    //读取 ELF 头
    if(sizeof(ehdr) != ReadAt(hFile, 0, &ehdr, sizeof(ehdr)))
        goto error;

    //判断是否是 ELF 文件
    if(!IsElf(&ehdr))
```

```
        goto error;

    //找到该文件第一条指令所在的段并读出这个段的程序头结构信息
    if(!FileEntryIndex(hFile, &ehdr, &phdr, ehdr.e_entry))
        goto error;

    //计算第一条指令在整个文件中的位置
    offset = ehdr.e_entry - phdr.p_vaddr;
    offset += phdr.p_offset;

    //计算 exit_print 函数体的字节数
    size=(int)(&IsElf) - (int)(&exit_print);

    //修改 ELF 文件第一条可执行机器指令在进程被正确加载后的内存地址
    if(size != WriteAt(hFile, offset, exit_print, size))
        goto error;

    printf("write Elf file success!\n");

error:
    close(hFile);
```

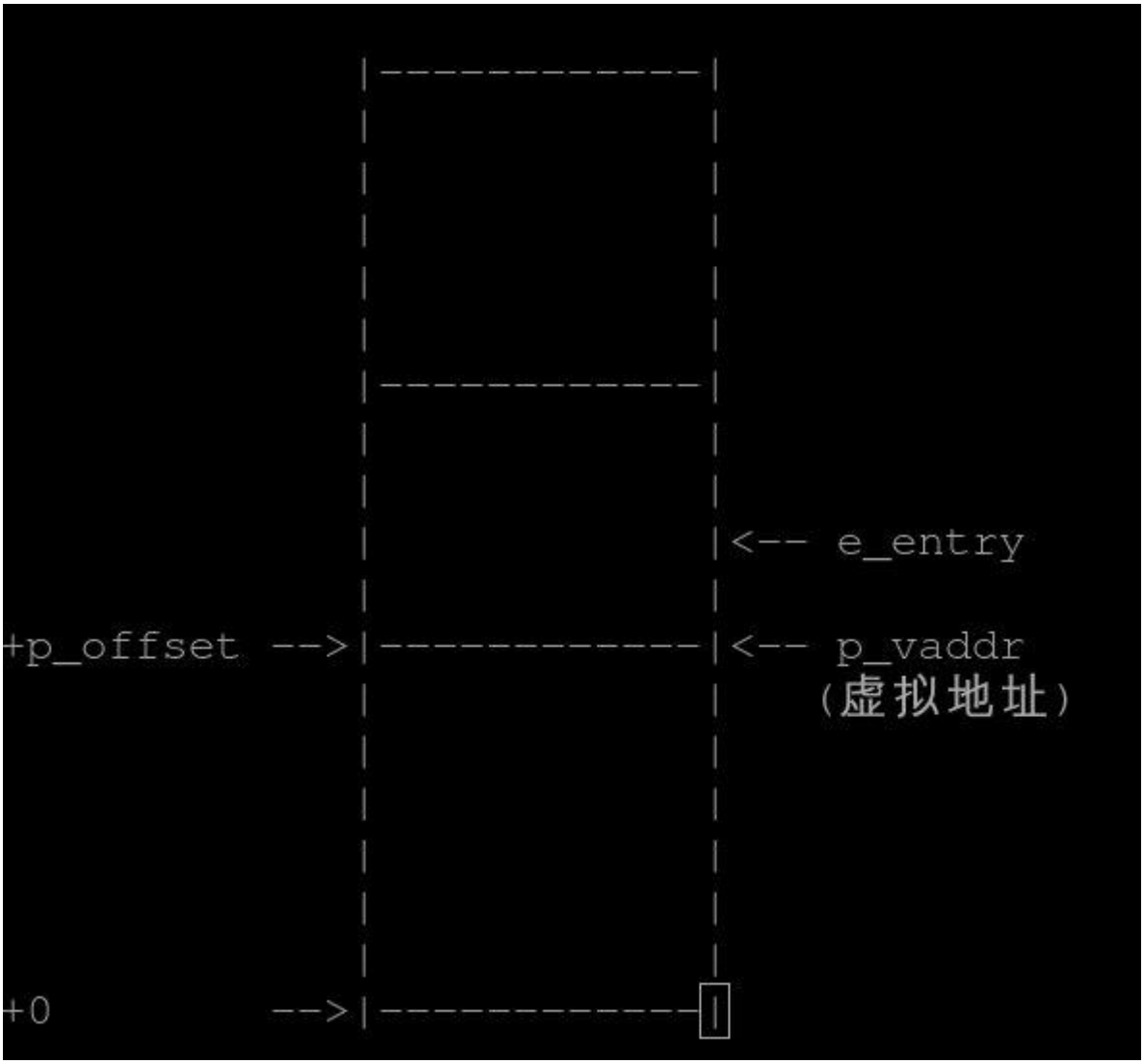
```
    return 0;
}
```

编译的时候 gcc 会有如下警告提示!

```
# gcc mod_entry.c -o mod_entry
mod_entry.c:20:37: warning: multi-character character constant
mod_entry.c:21:37: warning: multi-character character constant
mod_entry.c:22:37: warning: multi-character character constant
```

不用在意这个警告，它毫无防碍。这个程序非常简单，因为它忽略了许多本该注意的问题，比如被修改的 ELF 文件的那个段是否足够大可以容下我们的 exit_print 函数体？实事上我们的函数很小，它几乎总能使你的试验成功。

```
# ./mod_entry
write Elf file success!
# ./hello (此时 hello 文件的入口地址已经被修改了)
Hello  zxl
```




mod_entry 将函数 exit_print 的二进制机器指令复制到 hello 文件 e_entry 开始的位置。注意复制后，hello 的入口地址依然是 e_entry。

## 节头(节头表) -- Section Header

---

节头也叫节头表。ELF 头的 e_shoff 字段给出了节头在整个文件中的偏移(如果节头存在的话)，节头可看做一个在文件中连续存储的结构数组(Elf32_Shdr 结构的数组)，数组的长度由 ELF 头的 e_shnum 字段给出，

数组中每个结构的字节大小由 ELF 头的 e_shentsize 字段给出。把文件指针移到在 ELF 头中 e_shoff 给出的位置,然后读出的内容就是节头了。节头表是从连接角度看待 ELF 文件的结果，所以从节头的角度 ELF 文件分成了许多的节，每个节保存着用于不同目的的数据，这些数据可能被前面提到的程序头重复引用。关于节的内容非常琐碎，完成一次任务所的需的信息往往被分散到不同的节里。

相对而言，PE 中的资源表、引入表、导出表都集中给出了所有相关的信息，理解起来真是方便多了。由于节中数据的用途不同，节被分为不同的类型，每种类型的节都有自己组织数据的方式。

有的节存储着一些字符串，例如前面提过的**字符串表**就是一种类型的节;

有的节保存一张符号表，程序从动态连接库中引入的函数和变量都会出现在一个叫做”**动态符号表**“的节中;

**重定位表**则包含在重定位节中。

不管这些节是何种类型，在节头中都用相同的结构保存着与这些节有关的信息。先来看一看节头中用来保存这些信息的结构吧:

```
/usr/src/linux/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->sh_name

这个整数占用两个字节，它能告诉你这个节的名字。讲字符串表的时候会再来研究这个字段。

### Elf32_Shdr->sh_type

节的类型。或者说它能告诉你这个节里存放的是什么样的数据。随着 ELF 的发展，用于不同目的的节会不断增多，节的类型值是在 elf.h 中定义的一些常量。例如字符串表是 SHT_STRTAB，符号表是 SHT_SYMTAB 等。

### Elf32_Shdr->sh_flags

节的属性。这个字段在 32 位下占 4 个字节，64 位下占 8 个字节。与程序头中的 p_flags 字段一样，它用每一个二进制位表示一种属。其中

最低位如果为 1 表示此节在进程执行过程中可写，次低位为 1 表示此节的内容加载时要读到内存中去，第三低位为 1 表示这个节中的数据是可执行的机器指令。一些常量，如 SHF_INFO_LINK，帮助用来测试节的属性。

### Elf32_Shdr->sh_addr

如果此节的内容将出现在进程空间里，这个字段给出了该节在内存中起始地址。

### Elf32_Shdr->sh_offset

如果此节在文件中占用一定的字节，这个字段给出了该节在整个文件中的起始偏移量。

### Elf32_Shdr->sh_size

如果此节在文件中占用一定的字节，这个字段给出了该节在文件中的字节大小，如果此节在文件中不存在但却存在于内存中那么此字段给出了此节在内存中的字节大小。

### Elf32_Shdr->sh_link

如果另一个节与这个节相关联，这个字段给出了相关的节在节头中的索引。

### Elf32_Shdr->sh_info

这个字段如果用到再说。

### Elf32_Shdr->sh_addralign

地址对齐。这个数是 2 的整数次幂，对齐只能是 2 字节对齐、4 字节对齐、8 字节对齐等。如果这个数是 0 或 1 表示这个节不用对齐。

### Elf32_Shdr->sh_entsize

这个字段是一个代表字节大小的数，对某些节才有意义。例如对动态符号节来说这个字段就给出动态符号表中每个符号结构的字节大小。

节头的结构讲完了，很多一时用不到的知识被暂时抛弃。对于节的知识我们掌握了很少，但我希望至少能够知道每个节的名字。所以我们必须开始接触我们将要学习的几种类型节的第一类 -- 字符串表！

## 常量字符串表

开发中经常要用到字符串常量，C++编译器对字符串常量的处理是，在栈上分配空间，初始化，然后返回指针。例如，以下语句的每一次调用，都是要执行上述操作的：

?

```
1   const char *pstr=”Hello world”;
2   printf(“Good luck !”);
3   SendEvent(“PlayAnimation”);
```

如果一个字符串常量会反复用到，不断的拷贝操作其实是一种浪费。而在我们的系统中，无论是消息调用，对象查找，都大量涉及到字符串常量的使用。如果在程序初始化的时候就能把这些要经常用到的字符串都载入内存中，那么用的时候就仅仅是一次指针查找操作据 OK 了。

另外还考虑到，像界面中的静态文本，提示信息，游戏中的 NPC 对白，任务描述等等，如果能用一个外部表格索引起来，每一段文本对应一个唯一 ID，那么无论是策划维护，或翻译成其他语言，都非常方便。

所以前几天花了一个晚上的时间实现了一个常量字符串表。大致的使用流程是这样的，使用一个 string.def 文件来维护一个字符串列表。当然，实际工作中是用 Excel 之类的表格工具，并最终导出成如下格式：

```
?
1   table_name
2   {
3       str1 : Hello, world
4       str2 : Good luck !
5       str3 : PlayAnimation
6   }
```

然后用一个脚本把上述 string.def 转换成 C++头文件 stringID.h，里面记录了在程序中使用的字符串的 ID：

```
?
1   enum TABLE_NAME
2   {
3       CSTR_STR1=0x0001,
4       CSTR_STR2=0x0002,
5       CSTR_STR3=0x0003,
6   };
```

同时还会生成一个 string.tab 文件，以二进制的格式保存了每条字符串的文本，长度。在我的实现里，还额外记录了该字符串的 Hash 值。在程序运行的过程中，根据需要，从 string.tab 中将指定的部分载入内存，再通过 ID 获取字符串记录的指针：

```
?
1   xconst_string cstr=CSTR_STR1;
2   printf(cstr.c_str()); // 输出字符串
3   printf("size=%d",cstr.size()); // 字符串长度
4   printf("hash=%d",cstr.hash()); // Hash 值
```

xconst_string 对象可以根据字符串 ID，到表格中查找相关的记录并获取指针。此外，它只提供读取和比较的接口。

设计的关键是如何通过 ID 来获取字符串记录。比较直观的做法是用字典做索引，这种做法还有一个额外的好处是可以动态添加字符串表和字符串记录——尽管没啥意义，因为既然是常量字符串，自然是在编译期就可以决定的。而不好的地方是：1，字典需要额外的内存开销；2，初始化的时候要逐条记录插入到字典中；3，HashTable 的查找虽然快，但应该还有更快的方法。

于是我用了另外一种实现方法。在 string.tab 中，字符串记录是以二进制格式逐条存储的，每条记录的起始位置都能知道。此外表的数量也是一定的。所以我给每个表分配了一个 Table ID，并使得 String ID=Table ID+表中偏移。考虑到在实际应用中，表的数量不会很多，一般也就 10 来个，所以我创建一个 std::vector tables，并为每个表预留了一个占位符。于是，字符串查找最终就变成了两次间接寻址操作：
tables[table_id][offset]

其他一些细节：字符串表只有在需要的时候才应该被加载进来，用完之后就应该清除出去。加载的时间很容易知道，但什么时候清理，却很难界定。一种方法是通过在字符串表上加引用计数；第二种方法是将字

符串表按场景或关卡进行整理，在切换场景或关卡的时候更换字符串表。此外，还应该能够通过某些机制来检测字符串 ID 的有效性，这对于 DEBUG 必不可少。

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。比如表 3-12 这个字符串表。

表 3-12

| 偏移 | +0 | +1 | +2 | +3 | +4 | +5 | +6 | +7 | |
|---|---|---|---|---|---|---|---|---|---|
| +0 | \0 | h | e | l | l | o | w | o | |
| +10 | d | \0 | M | y | v | a | r | i | |
| +20 | l | e | \0 | | | | | | |

那么偏移与它们对应的字符串如表 3-13 所示。

表 3-13

| 偏移 | 字符串 |
|---|---|
| 0 | 空字符串 |
| 1 | hello world |
| 6 | world |
| 12 | Myvariable |

通过这种方法，在 ELF 文件中引用字符串只须给出一个数字下标即可，不用考虑字符串长度的问题。一般字符串表在 ELF 文件中也以段的形式保存，常见的段名为".strtab"或".shstrtab"。这两个字符串表分别为字符串表（String Table）和段表字符串表（Section Header String Table）。顾名思义，字符串表用来保存普通的字符串，比如符号的名字；段表字符串表用来保存段表中用到的字符串，最常见的就是段名（sh_name）。

接着我们再回头看这个 ELF 文件头中的"e_shstrndx"的含义，我们在前面提到过，"e_shstrndx"是 Elf32_Ehdr 的最后一个成员，它是"Section header string table index"的缩写。我们知道段表字符串表本身也是 ELF 文件中的一个普通的段，知道它的名字往往叫做".shstrtab"。那么这个"e_shstrndx"就表示".shstrtab"在段表中的下标，即段表字符串表在段表中的下标。前面的 SimpleSection.o 中，"e_shstrndx"的值为 8，我们再对照

"readelf -S"的输出结果，可以看到".shstrtab"这个段刚好位于段表中的下标为 8 的位置上。由此，我们可以得出结论，只有分析 ELF 文件头，就可以得到段表和段表字符串表的位置，从而解析整个 ELF 文件。

# ELF 简单解析（后续扩充）

## ELF Header:

```
Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00          (魔数--操作系统在加载可执行文件时，判断该魔数是否正确)
  Class:                             ELF32                (机器字节长度)
  Data:                              2's complement, little endian     (数据存储方式)
  Version:                           1 (current)            (版本)
  OS/ABI:                            UNIX - System V             (运行平台)
  ABI Version:                       0                    (ABI 版本)
  Type:                              EXEC (Executable file)         (ELF 重定位类型--可重定位.o,可执行 ELF,共享目标文件.so)
  Machine:                           Intel 80386            (硬件平台)
  Version:                           0x1                (硬件平台版本)
  Entry point address:               0x8048360                (入口地址)
  Start of program headers:          52 (bytes into file)         (程序头的入口)
  Start of section headers:          5192 (bytes into file)         (段表的位置)
  Flags:                             0x0                (ELF 标志位)
  Size of this header:               52 (bytes)                (程序头的长度)
  Size of program headers:           32 (bytes)                (段表的长度)
  Number of program headers:         8
  Size of section headers:           40 (bytes)
  Number of section headers:         36                    (段表描述符数量)
  Section header string table index: 33
```

其他：

ELF（Common Object File Format）

COFF（Common Object File Format）

PE COFF（Portable 指的是文件格式相同，不是代表能够在多个平台上运行）

.out 类似于 ELF，缺乏对共享库的支持

//////////////////////////////////////////////////////////////////////////////
///////

## Section Headers:

There are 36 section headers, starting at offset 0x1448:(注意：0x1448=5192 （也就是上面的start of section headers））

**Name:** 段名

**Type:** 段类型 含义

| | |
|---|---|
| NULL: | 无效段 |
| PROGBIT: | 程序段，代码段数据段 |
| SYMTAB: | 段内容为符号表 |
| STRTAB: | 字符串表 |
| RELA: | 重定位表 |
| HASH: | 希哈表 |
| DYNAMIC: | 动态链接信息 |
| NOTE: | 提示性信息 |
| NOBITS: | 表示该段在文件中没有内容，比如.bss段 |
| REL: | 重定位信息 |
| SHLIB: | 保留 |
| DNYSYM: | 动态链接到符号表 |

**Flg:** 表示该段在进程虚拟空间中到属性

| | |
|---|---|
| WRITE: | 可写 |
| ALLOC: | 在进程空间中须分配空间 |
| EXECINSTR: | 可被执行 |

**Addr:** 段虚拟地址

**Lk,Inf:** 如果段到类型与链接相关的，比如重定位表，符号表等。则**Lk,Inf**包含一定到意义。

| 类型: | sh_link | sh_info |
|---|---|---|
| 符号表 | 操作系统相关 | 0 |
| 重定位表 | 该段所使用的对应符号表在段表中到下标 | 该重定位表所作用的段在段 |

表中到下标

# 关于重定位表：

链接器在处理目标文件时，需要对目标文件中到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到位置。
比如,[.rel.plt]所示用到的符号表下标为 6，也就是用到的符号表是.dynsym.Inf 为 13,表示该重定位作用于.plt 段表。

# 关于字符串表：

ELF 文件中用到了许多字符串，比如段名，变量名等，长度不定，通常使用字符串表来存放，并使用字符串在表中的
偏移来引用字符传串。字符串表在 ELF 文件中也以段表到形式保存，常见段名为.strtab 或.shstrtab.前者字符串表（存放普通到字符串），后者段表字符串表（存放段表中用到的字符串），ELF header 到最后一项标明
段表字符串表在段表中的位置，本例子为 33.

```
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048134 000134 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048148 000148 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048168 000168 000024 00   A  0   0  4
  [ 4] .hash             HASH            0804818c 00018c 00002c 04   A  6   0  4
  [ 5] .gnu.hash         GNU_HASH        080481b8 0001b8 000020 04   A  6   0  4
  [ 6] .dynsym           DYNSYM          080481d8 0001d8 000060 10   A  7   1  4
  [ 7] .dynstr           STRTAB          08048238 000238 000054 00   A  0   0  1
  [ 8] .gnu.version      VERSYM          0804828c 00028c 00000c 02   A  6   0  2
  [ 9] .gnu.version_r    VERNEED         08048298 000298 000020 00   A  7   1  4
  [10] .rel.dyn          REL             080482b8 0002b8 000008 08   A  6   0  4
  [11] .rel.plt          REL             080482c0 0002c0 000020 08   A  6  13  4
  [12] .init             PROGBITS        080482e0 0002e0 000030 00  AX  0   0  4
  [13] .plt              PROGBITS        08048310 000310 000050 04  AX  0   0  4
  [14] .text             PROGBITS        08048360 000360 0001bc 00  AX  0   0 16
  [15] .fini             PROGBITS        0804851c 00051c 00001c 00  AX  0   0  4
  [16] .rodata           PROGBITS        08048538 000538 000011 00   A  0   0  4
  [17] .eh_frame         PROGBITS        0804854c 00054c 000004 00   A  0   0  4
  [18] .ctors            PROGBITS        08049f0c 000f0c 000008 00  WA  0   0  4
  [19] .dtors            PROGBITS        08049f14 000f14 000008 00  WA  0   0  4
  [20] .jcr              PROGBITS        08049f1c 000f1c 000004 00  WA  0   0  4
  [21] .dynamic          DYNAMIC         08049f20 000f20 0000d0 08  WA  7   0  4
  [22] .got              PROGBITS        08049ff0 000ff0 000004 04  WA  0   0  4
```

```
  [23] .got.plt          PROGBITS        08049ff4 000ff4 00001c 04  WA  0   0  4
  [24] .data             PROGBITS        0804a010 001010 000008 00  WA  0   0  4
  [25] .bss              NOBITS          0804a018 001018 000008 00  WA  0   0  4
  [26] .comment          PROGBITS        00000000 001018 000046 01  MS  0   0  1
  [27] .debug_aranges    PROGBITS        00000000 001060 000020 00      0   0  8
  [28] .debug_pubnames   PROGBITS        00000000 001080 000025 00      0   0  1
  [29] .debug_info       PROGBITS        00000000 0010a5 0000f2 00      0   0  1
  [30] .debug_abbrev     PROGBITS        00000000 001197 00005f 00      0   0  1
  [31] .debug_line       PROGBITS        00000000 0011f6 000082 00      0   0  1
  [32] .debug_str        PROGBITS        00000000 001278 000092 01  MS  0   0  1
  [33] .shstrtab         STRTAB          00000000 00130a 00013e 00      0   0  1
  [34] .symtab           SYMTAB          00000000 0019e8 000490 10     35  52  4
  [35] .strtab           STRTAB          00000000 001e78 000219 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)

There are no section groups in this file.
////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x00550 0x00550 R E 0x1000
  LOAD           0x000f0c 0x08049f0c 0x08049f0c 0x0010c 0x00114 RW  0x1000
  DYNAMIC        0x000f20 0x08049f20 0x08049f20 0x000d0 0x000d0 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x000f0c 0x08049f0c 0x08049f0c 0x000f4 0x000f4 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .hash .gnu.hash .dynsym .d
ynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini .
```

```
rodata .eh_frame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06
   07     .ctors .dtors .jcr .dynamic .got

Dynamic section at offset 0xf20 contains 21 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
 0x0000000c (INIT)                       0x80482e0
 0x0000000d (FINI)                       0x804851c
 0x00000004 (HASH)                       0x804818c
 0x6ffffef5 (GNU_HASH)                   0x80481b8
 0x00000005 (STRTAB)                     0x8048238
 0x00000006 (SYMTAB)                     0x80481d8
 0x0000000a (STRSZ)                      84 (bytes)
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000015 (DEBUG)                      0x0
 0x00000003 (PLTGOT)                     0x8049ff4
 0x00000002 (PLTRELSZ)                   32 (bytes)
 0x00000014 (PLTREL)                     REL
 0x00000017 (JMPREL)                     0x80482c0
 0x00000011 (REL)                        0x80482b8
 0x00000012 (RELSZ)                      8 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
 0x6ffffffe (VERNEED)                    0x8048298
 0x6fffffff (VERNEEDNUM)                 1
 0x6ffffff0 (VERSYM)                     0x804828c
 0x00000000 (NULL)                       0x0

Relocation section '.rel.dyn' at offset 0x2b8 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ff0  00000106 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x2c0 contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a000  00000107 R_386_JUMP_SLOT   00000000   __gmon_start__
0804a004  00000207 R_386_JUMP_SLOT   00000000   putchar
0804a008  00000307 R_386_JUMP_SLOT   00000000   __libc_start_main
0804a00c  00000407 R_386_JUMP_SLOT   00000000   printf

There are no unwind sections in this file.
```

```
Symbol table '.dynsym' contains 6 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     2: 00000000     0 FUNC    GLOBAL DEFAULT  UND putchar@GLIBC_2.0 (2)
     3: 00000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.0
(2)
     4: 00000000     0 FUNC    GLOBAL DEFAULT  UND printf@GLIBC_2.0 (2)
     5: 0804853c     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used

Symbol table '.symtab' contains 73 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 08048134     0 SECTION LOCAL  DEFAULT    1
     2: 08048148     0 SECTION LOCAL  DEFAULT    2
     3: 08048168     0 SECTION LOCAL  DEFAULT    3
     4: 0804818c     0 SECTION LOCAL  DEFAULT    4
     5: 080481b8     0 SECTION LOCAL  DEFAULT    5
     6: 080481d8     0 SECTION LOCAL  DEFAULT    6
     7: 08048238     0 SECTION LOCAL  DEFAULT    7
     8: 0804828c     0 SECTION LOCAL  DEFAULT    8
     9: 08048298     0 SECTION LOCAL  DEFAULT    9
    10: 080482b8     0 SECTION LOCAL  DEFAULT   10
    11: 080482c0     0 SECTION LOCAL  DEFAULT   11
    12: 080482e0     0 SECTION LOCAL  DEFAULT   12
    13: 08048310     0 SECTION LOCAL  DEFAULT   13
    14: 08048360     0 SECTION LOCAL  DEFAULT   14
    15: 0804851c     0 SECTION LOCAL  DEFAULT   15
    16: 08048538     0 SECTION LOCAL  DEFAULT   16
    17: 0804854c     0 SECTION LOCAL  DEFAULT   17
    18: 08049f0c     0 SECTION LOCAL  DEFAULT   18
    19: 08049f14     0 SECTION LOCAL  DEFAULT   19
    20: 08049f1c     0 SECTION LOCAL  DEFAULT   20
    21: 08049f20     0 SECTION LOCAL  DEFAULT   21
    22: 08049ff0     0 SECTION LOCAL  DEFAULT   22
    23: 08049ff4     0 SECTION LOCAL  DEFAULT   23
    24: 0804a010     0 SECTION LOCAL  DEFAULT   24
    25: 0804a018     0 SECTION LOCAL  DEFAULT   25
    26: 00000000     0 SECTION LOCAL  DEFAULT   26
    27: 00000000     0 SECTION LOCAL  DEFAULT   27
    28: 00000000     0 SECTION LOCAL  DEFAULT   28
    29: 00000000     0 SECTION LOCAL  DEFAULT   29
    30: 00000000     0 SECTION LOCAL  DEFAULT   30
    31: 00000000     0 SECTION LOCAL  DEFAULT   31
```

```
    32: 00000000     0 SECTION LOCAL  DEFAULT   32
    33: 00000000     0 FILE    LOCAL  DEFAULT  ABS init.c
    34: 00000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    35: 08049f0c     0 OBJECT  LOCAL  DEFAULT   18 __CTOR_LIST__
    36: 08049f14     0 OBJECT  LOCAL  DEFAULT   19 __DTOR_LIST__
    37: 08049f1c     0 OBJECT  LOCAL  DEFAULT   20 __JCR_LIST__
    38: 08048390     0 FUNC    LOCAL  DEFAULT   14 __do_global_dtors_aux
    39: 0804a018     1 OBJECT  LOCAL  DEFAULT   25 completed.6990
    40: 0804a01c     4 OBJECT  LOCAL  DEFAULT   25 dtor_idx.6992
    41: 080483f0     0 FUNC    LOCAL  DEFAULT   14 frame_dummy
    42: 00000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    43: 08049f10     0 OBJECT  LOCAL  DEFAULT   18 __CTOR_END__
    44: 0804854c     0 OBJECT  LOCAL  DEFAULT   17 __FRAME_END__
    45: 08049f1c     0 OBJECT  LOCAL  DEFAULT   20 __JCR_END__
    46: 080484f0     0 FUNC    LOCAL  DEFAULT   14 __do_global_ctors_aux
    47: 00000000     0 FILE    LOCAL  DEFAULT  ABS comline.c
    48: 08049ff4     0 OBJECT  LOCAL  HIDDEN   23 _GLOBAL_OFFSET_TABLE_
    49: 08049f0c     0 NOTYPE  LOCAL  HIDDEN   18 __init_array_end
    50: 08049f0c     0 NOTYPE  LOCAL  HIDDEN   18 __init_array_start
    51: 08049f20     0 OBJECT  LOCAL  HIDDEN   21 _DYNAMIC
    52: 0804a010     0 NOTYPE  WEAK   DEFAULT   24 data_start
    53: 08048480     5 FUNC    GLOBAL DEFAULT   14 __libc_csu_fini
    54: 08048360     0 FUNC    GLOBAL DEFAULT   14 _start
    55: 00000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    56: 00000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
    57: 08048538     4 OBJECT  GLOBAL DEFAULT   16 _fp_hw
    58: 0804851c     0 FUNC    GLOBAL DEFAULT   15 _fini
    59: 00000000     0 FUNC    GLOBAL DEFAULT  UND putchar@@GLIBC_2.0
    60: 00000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
    61: 0804853c     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
    62: 0804a010     0 NOTYPE  GLOBAL DEFAULT   24 __data_start
    63: 0804a014     0 OBJECT  GLOBAL HIDDEN   24 __dso_handle
    64: 08049f18     0 OBJECT  GLOBAL HIDDEN   19 __DTOR_END__
    65: 08048490    90 FUNC    GLOBAL DEFAULT   14 __libc_csu_init
    66: 00000000     0 FUNC    GLOBAL DEFAULT  UND printf@@GLIBC_2.0
    67: 0804a018     0 NOTYPE  GLOBAL DEFAULT  ABS __bss_start
    68: 0804a020     0 NOTYPE  GLOBAL DEFAULT  ABS _end
    69: 0804a018     0 NOTYPE  GLOBAL DEFAULT  ABS _edata
    70: 080484ea     0 FUNC    GLOBAL HIDDEN   14 __i686.get_pc_thunk.bx
    71: 08048414   106 FUNC    GLOBAL DEFAULT   14 main
    72: 080482e0     0 FUNC    GLOBAL DEFAULT   12 _init

Histogram for bucket list length (total of 3 buckets):
 Length  Number     % of total  Coverage
```

```
      0  0          (  0.0%)
      1  1          ( 33.3%)     20.0%
      2  2          ( 66.7%)    100.0%

Histogram for `.gnu.hash' bucket list length (total of 2 buckets):
 Length  Number     % of total  Coverage
      0  1          ( 50.0%)
      1  1          ( 50.0%)    100.0%

Version symbols section '.gnu.version' contains 6 entries:
 Addr: 000000000804828c  Offset: 0x00028c  Link: 6 (.dynsym)
  000:   0 (*local*)       0 (*local*)       2 (GLIBC_2.0)     2 (GLIBC_2.0)
  004:   2 (GLIBC_2.0)     1 (*global*)

Version needs section '.gnu.version_r' contains 1 entries:
 Addr: 0x0000000008048298  Offset: 0x000298  Link: 7 (.dynstr)
  000000: Version: 1  File: libc.so.6  Cnt: 1
  0x0010:   Name: GLIBC_2.0  Flags: none  Version: 2

Notes at offset 0x00000148 with length 0x00000020:
  Owner        Data size    Description
  GNU        0x00000010    NT_GNU_ABI_TAG (ABI version tag)

Notes at offset 0x00000168 with length 0x00000024:
  Owner        Data size    Description
  GNU        0x00000014    NT_GNU_BUILD_ID (unique build ID bitstring)
```