obj 文件的重定位

```
[wzhou@dcmp10 ~]$ cat hello.c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char** argv)
{
       getuid();
       getchar();
       return 0;
}
```

gcc -c hello.c -o hello.o
生成 obj 文件。

```
[wzhou@dcmp10 ~]$ readelf -S hello.o
There are 10 section headers, starting at offset 0xe0:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00002d 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000330 000010 08      8   1  4
  [ 3] .data             PROGBITS        00000000 000064 000000 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 000064 000000 00  WA  0   0  4
```

```
  [ 5] .note.GNU-stack   PROGBITS       00000000 000064 000000 00     0   0  1
  [ 6] .comment         PROGBITS       00000000 000064 000031 00     0   0  1
  [ 7] .shstrtab        STRTAB         00000000 000095 000049 00     0   0  1
  [ 8] .symtab          SYMTAB         00000000 000270 0000a0 10     9   7  4
  [ 9] .strtab          STRTAB         00000000 000310 00001d 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)
[wzhou@dcmp10 ~]$
```

dump .rel.text section

```
[wzhou@dcmp10 ~]$ hexdump -C -s 0x330 -n 16 hello.o
00000330  1d 00 00 00 02 08 00 00  22 00 00 00 02 09 00 00  |........".......|
```

两个 entries。

```
typedef struct {
Elf32_Addr r_offset;
Elf32_Word r_info ;
} Elf32_Rel;
```

1. r_offset = 0x1d，r_info = （sym, type）= (08, 02) = (08, R_386_PC32)
2. r_offset = 0x22，r_info = （sym, type）= (09, 02) = (09, R_386_PC32)

其中 sym 为指向 symbol table 的 index。

dump .symtab section

```
[wzhou@dcmp10 ~]$ hexdump -C -s 0x270 -n 160 hello.o
00000270  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000280  01 00 00 00 00 00 00 00  00 00 00 00 04 00 f1 ff  |................|
00000290  00 00 00 00 00 00 00 00  00 00 00 00 03 00 01 00  |................|
000002a0  00 00 00 00 00 00 00 00  00 00 00 00 03 00 03 00  |................|
000002b0  00 00 00 00 00 00 00 00  00 00 00 00 03 00 04 00  |................|
000002c0  00 00 00 00 00 00 00 00  00 00 00 00 03 00 05 00  |................|
000002d0  00 00 00 00 00 00 00 00  00 00 00 00 03 00 06 00  |................|
000002e0  09 00 00 00 00 00 00 00  2d 00 00 00 12 00 01 00  |........-.......|
000002f0  0e 00 00 00 00 00 00 00  00 00 00 00 10 00 00 00  |................|   sym 为 8 的重定位项
00000300  15 00 00 00 00 00 00 00  00 00 00 00 10 00 00 00  |................|   sym 为 9 的重定位项
```

第一重定位项的名字在.strtab section 的偏移 0x0e
第二重定位项的名字在.strtab section 的偏移 0x15

dump .strtab section 的内容

```
[wzhou@dcmp10 ~]$ hexdump -C -s 0x310 -n 29 hello.o
00000310  00 68 65 6c 6c 6f 2e 63  00 6d 61 69 6e 00 67 65  |.hello.c.main.ge|
00000320  74 75 69 64 00 67 65 74  63 68 61 72 00           |tuid.getchar.|
```

即第一重定位项的名字为“getuid”
即第二重定位项的名字为“getchar”

1. r_offset = 0x1d，r_info = ("getuid", R_386_PC32)
2. r_offset = 0x22，r_info = ("getchar", R_386_PC32)
其中 r_offset 是.text section 中的偏移。

```
[wzhou@dcmp10 ~]$ objdump -d hello.o

hello.o:    file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:   55                   push   %ebp
   1:   89 e5                mov    %esp,%ebp
   3:   83 ec 08             sub    $0x8,%esp
   6:   83 e4 f0             and    $0xfffffff0,%esp
   9:   b8 00 00 00 00         mov    $0x0,%eax
   e:   83 c0 0f             add    $0xf,%eax
  11:   83 c0 0f             add    $0xf,%eax
  14:   c1 e8 04             shr    $0x4,%eax
  17:   c1 e0 04             shl    $0x4,%eax
  1a:   29 c4                sub    %eax,%esp
  1c:   e8 ①fc ff ff ff         call   1d <main+0x1d>
  21:   e8 ②fc ff ff ff         call   22 <main+0x22>
  26:   b8 00 00 00 00         mov    $0x0,%eax
  2b:   c9                   leave
```

```
 2c:   c3                      ret
```

上面①是偏移 0x1d，而②是偏移 0x22。正好是调用两个函数的地方。
也就是在源代码中下面两行需要重定位。

```
int main(int argc, char** argv)
{
      ①getuid();
      ②getchar();
      return 0;
}
```

linker 会根据重定位项来 fix 该地址。

用 readelf 可以验证之。

```
[wzhou@dcmp10 ~]$ readelf -r hello.o

Relocation section '.rel.text' at offset 0x330 contains 2 entries:
 Offset     Info    Type          Sym.Value  Sym. Name
0000001d  00000802 R_386_PC32       00000000   getuid
00000022  00000902 R_386_PC32       00000000   getchar
```

obj 文件（在 ELF Specification 中被称为 relocatable file，我翻译成可重定位文件）的重定位与 executable and shared object file 中的重定位是不同的。

obj 文件的重定位是静态期（也即编译期的重定位），而 executable and shared object file 的重定位是运行期（即在其被载入时）的。
比如下面简单的例子：

```
[wzhou@dcmp10 ~]$ cat test1.c
```

```
int func1(void);

int main()
{
       func1();

       return  0;
}
```

只编译而不链接（这时链接也不会成功，func1 函数是什么还不知道呢）

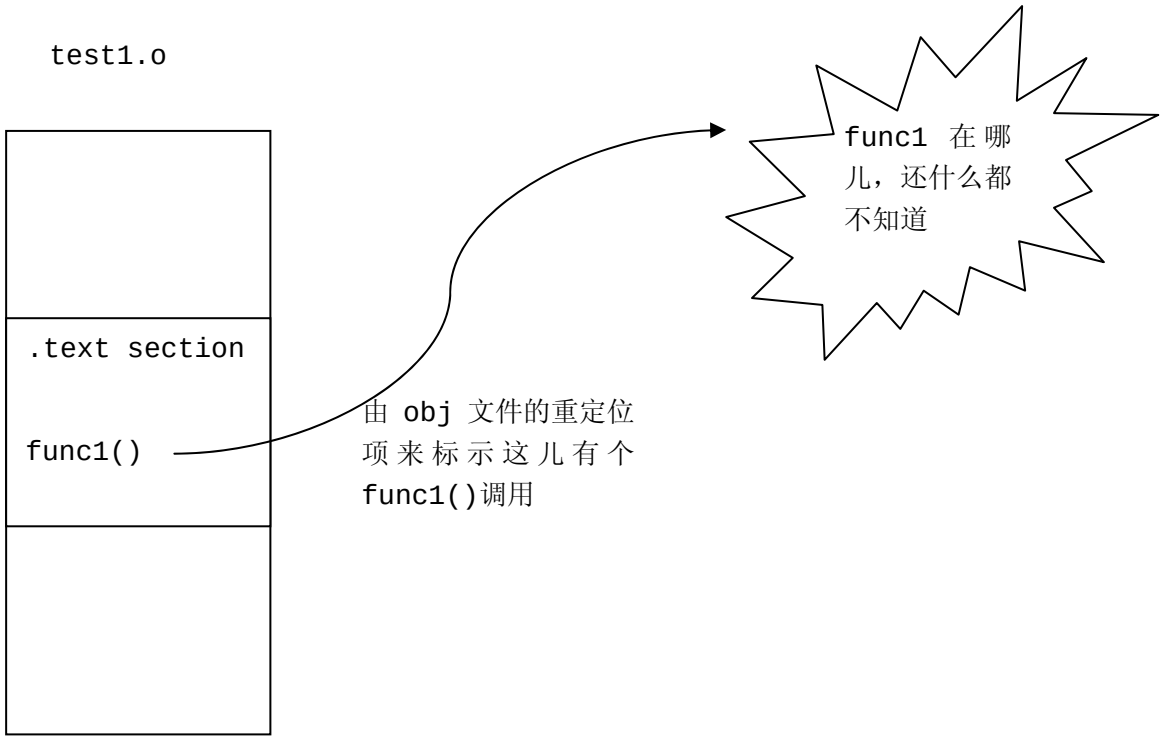
```
gcc -c test1.c -o test1.o
```

```
[wzhou@dcmp10 ~]$ readelf -r test1.o
察看重定位表，只有一项
Relocation section '.rel.text' at offset 0x30c contains 1 entries:
 Offset     Info    Type           Sym.Value  Sym. Name
0000001d  00000802 R_386_PC32       00000000   func1
[wzhou@dcmp10 ~]$
[wzhou@dcmp10 ~]$ objdump -d test1.o

test1.o:    file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:   55                   push   %ebp
```

```
  1:   89 e5                mov    %esp,%ebp
  3:   83 ec 08              sub    $0x8,%esp
  6:   83 e4 f0              and    $0xfffffff0,%esp
  9:   b8 00 00 00 00         mov    $0x0,%eax
  e:   83 c0 0f              add    $0xf,%eax
 11:   83 c0 0f              add    $0xf,%eax
 14:   c1 e8 04              shr    $0x4,%eax
 17:   c1 e0 04              shl    $0x4,%eax
 1a:   29 c4                sub    %eax,%esp
 1c:   e8 [1]fc ff ff ff         call   1d <main+0x1d>
 21:   b8 00 00 00 00         mov    $0x0,%eax
 26:   c9                   leave
 27:   c3                   ret
[wzhou@dcmp10 ~]$
```

也就是调用 func1()在编译期没办法定下来，所以只能标示需要重定位。

[1] 这里也就是重定位项指出的.text section 中的偏移为 0x1d 的需要重定位的地方


test1.o
.text section
func1()
由 obj 文件的重定位项来标示这儿有个func1()调用
func1 在哪儿，还什么都不知道

```
[wzhou@dcmp10 ~]$ cat test2.c        func1()定义在test2.c中
int func1(void)
{
       return  2;
}
[wzhou@dcmp10 ~]$ gcc -c test2.c -o test2.o       编译但不链接test2.c
```

```
[wzhou@dcmp10 ~]$ readelf -r test2.o

There are no relocations in this file.
```

该obj文件没有未定的东东，自然没有重定位项。

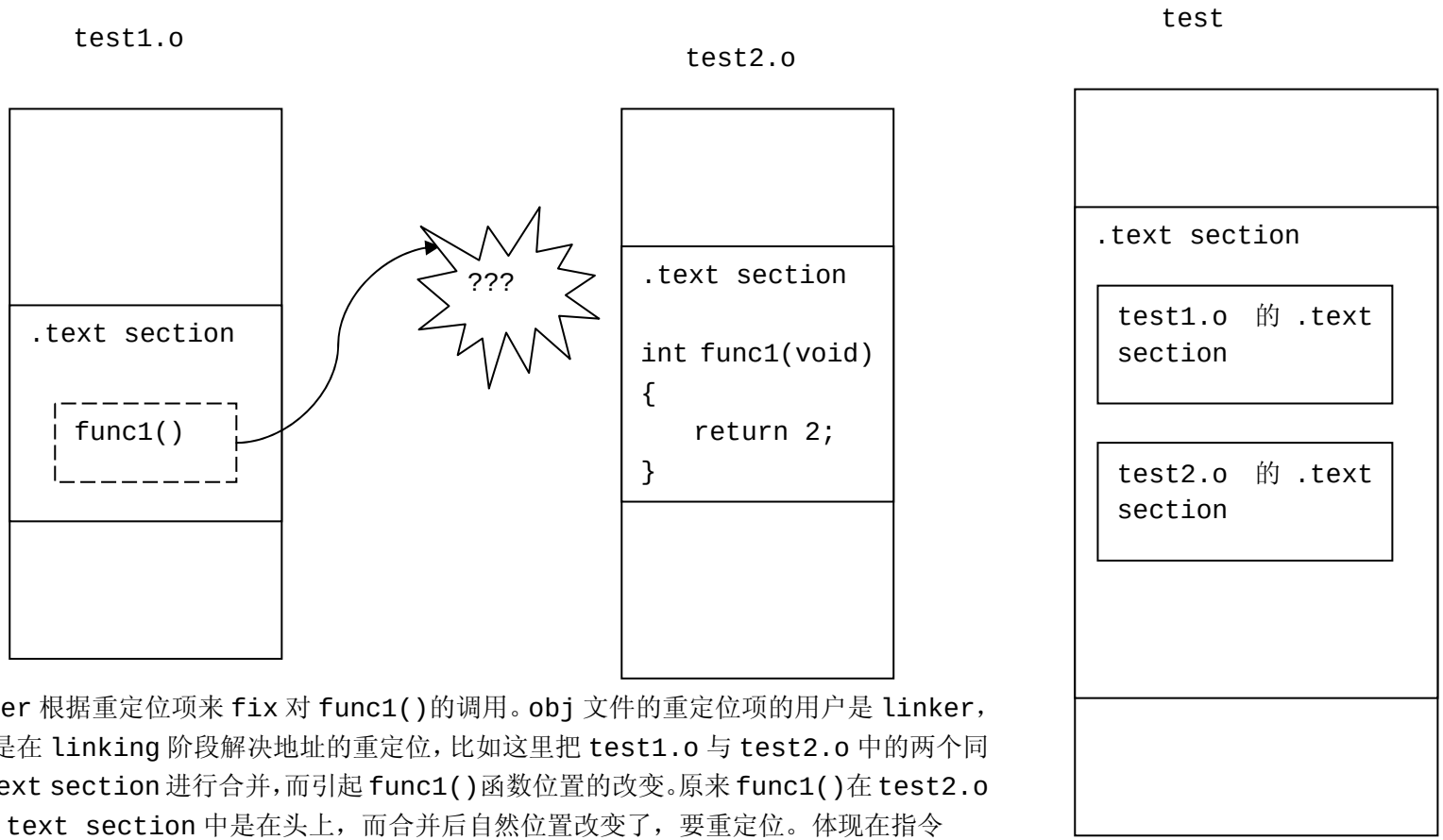
```
[wzhou@dcmp10 ~]$ gcc -o test test1.o test2.o
[wzhou@dcmp10 ~]$ readelf -r test

Relocation section '.rel.dyn' at offset 0x244 contains 1 entries:
 Offset     Info    Type          Sym.Value  Sym. Name
08049528  00000406 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x24c contains 1 entries:
 Offset     Info    Type          Sym.Value  Sym. Name
08049538  00000107 R_386_JUMP_SLOT   00000000   __libc_start_main
```

在生成的test中没有原来标示func1()调用的重定位项。

```
[wzhou@dcmp10 ~]$ objdump -d test
```

```
08048334 <main>:
 8048334:      55                   push   %ebp
 8048335:      89 e5                mov    %esp,%ebp
 8048337:      83 ec 08             sub    $0x8,%esp
 804833a:      83 e4 f0             and    $0xfffffff0,%esp
 804833d:      b8 00 00 00 00         mov    $0x0,%eax
 8048342:      83 c0 0f             add    $0xf,%eax
 8048345:      83 c0 0f             add    $0xf,%eax
 8048348:      c1 e8 04             shr    $0x4,%eax
 804834b:      c1 e0 04             shl    $0x4,%eax
 804834e:      29 c4                sub    %eax,%esp
 8048350:      e8 07 00 00 00         call   804835c <func1>  这里是一个 near call，即相对于 0x8048355 + 0x07 的调用
 8048355:      b8 00 00 00 00         mov    $0x0,%eax
 804835a:      c9                   leave
 804835b:      c3                   ret

0804835c <func1>:
 804835c:      55                   push   %ebp
 804835d:      89 e5                mov    %esp,%ebp
 804835f:      b8 02 00 00 00         mov    $0x2,%eax
 8048364:      c9                   leave
 8048365:      c3                   ret
 8048366:      90                   nop
 8048367:      90                   nop
```

在最终的可执行文件中对 `func1()`的调用完全定下来了，没有需要“重定位的”。



linker 根据重定位项来 fix 对 func1()的调用。obj 文件的重定位项的用户是 linker，并且是在 linking 阶段解决地址的重定位，比如这里把 test1.o 与 test2.o 中的两个同名.text section 进行合并，而引起 func1()函数位置的改变。原来 func1()在 test2.o 中的.text section 中是在头上，而合并后自然位置改变了，要重定位。体现在指令 call near +0x7

上面的例子当然是只在编译期就能“定位”的，若没法在 link 阶段定位，比如对其他共享库的函数的调用，那自然该重定位项要保留，以被 ELF loader 在载入该可执行文件时来重定位，但这种重定位与 obj 文件的重定位又有不同了。

Walter Zhou
mailto:z-l-dragon@hotmail.com