32位elf格式中的10种重定位类型 精

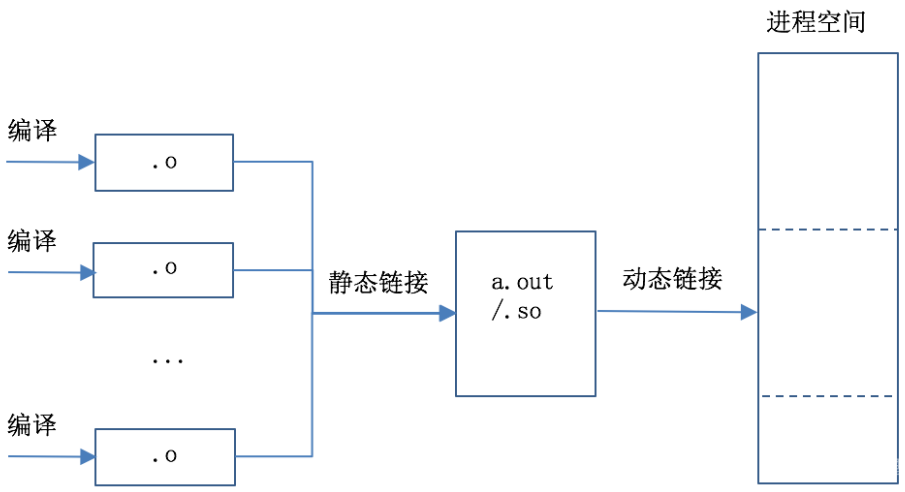
xinpoo 1天前 464

1

我之前在另外一个论坛发过一篇这样的主题，但是当时还剩下一些疑问没有想清楚，最近利用业余时间再次学习了elf格式，针对10种重定位类型重新做了总结，希望分享出来，可以带给初学者一点帮助。

首先需要知道的是，一个程序从源码到被执行，当中经历了3个过程：

- 编译：将.c文件编译成.o文件，不关心.o文件之间的联系；
- 静态链接：将所有.o文件合并成一个.so或a.out文件，处理所有.o文件节区在目标文件中的布局；
- 动态链接：将.so或a.out文件加载到内存，处理加载文件在的内存中的布局。

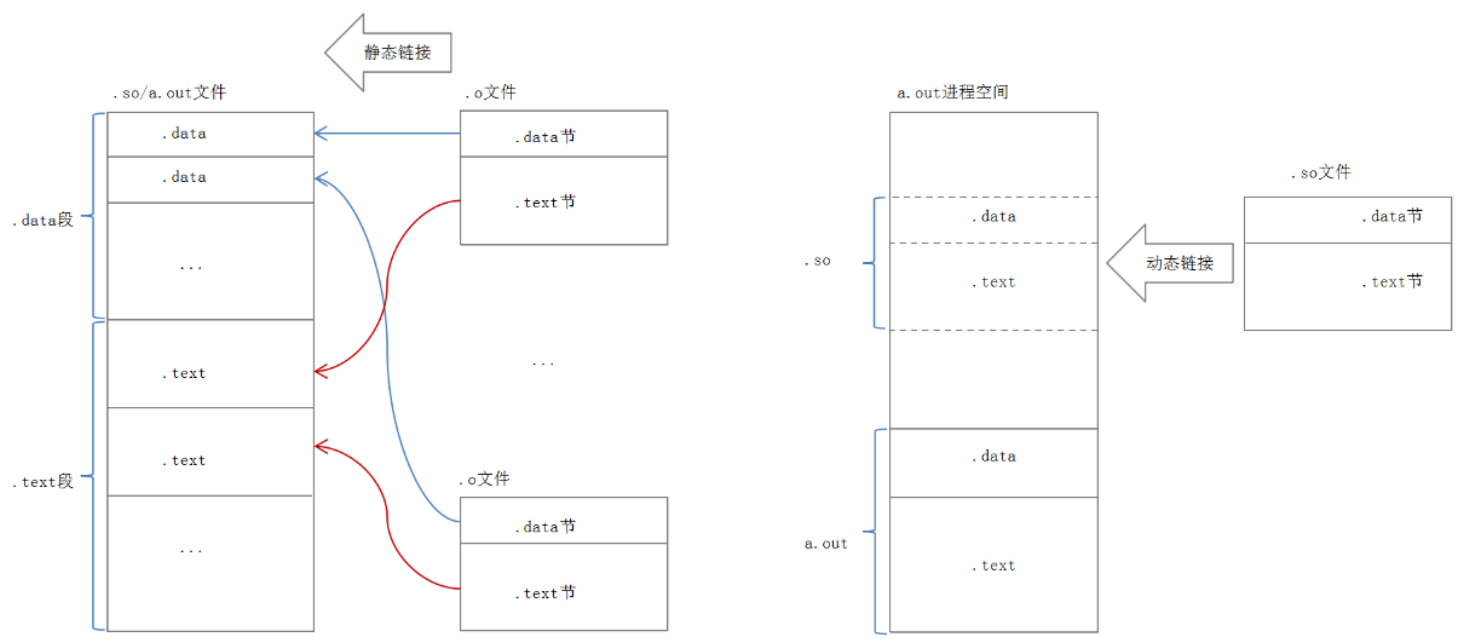


c程序中引用全局变量的语句，经过编译得到的机器码会包含一个地址值部分，机器码执行时，该值必须为变量在内存中的绝对地址，调用函数的语句，经过编译得到的机器码也包含一个地址值部分，机器码执行时，该值必须为内存中函数地址与下一条指令地址的偏移。但是在编译、静态链接，甚至动态链接之后，该地址值部分可能暂时无法满足最终要求，从而必须相应设置一个重定项，要求后续过程对该值进行修改，重定项一方面标记了地址值的位置，另一方面提供了计算正确地址值的方法和计算参数。

对于局部变量的使用，由于程序执行时，esp寄存器保存的一定是栈顶的内存地址，那么从逻辑上讲，编译阶段就可以确定所有局部变量运行时的内存地址，所以不需要设置重定项。

另外，elf格式中设计了10种不同的重定位类型，是由于不同场合，对地址值进行重新计算的方法和参数不同：

- 引用变量的指令中，需要使用变量的绝对地址，而函数调用指令，需要使用函数与下一条指令地址的相对地址。
- -fPIC编译选项，可以决定物理内存中的同一份.so镜像，是否可以被多个进程共享。
- 静态ld是将.o文件按节区"撕开"，将各个.o文件中相同类型的节，合并为.so或a.out文件中的一个段，而动态ld则是维持.so文件"原状"，合并到进程的虚拟内存空间。



具体来讲，以下表格包含了生成各种类型重定项的情况：

gcc / sym	-fPIC					
	.o	.so	.o		.so	a.out
extern int g;	R_386_32 (g->.text)	R_386_32 (g->.text)	R_386_GOT32 (g->.text)	R_386_PC32 (__x86.get_pc_thunk.cx->.text) R_386_GOTPC(_GLOBAL_OFFSET_TABLE_->.text)	R_386_GLOB_DAT (g->.got)	R_386_COPY (g_v->.bss)
int g;	R_386_32 (g->.text)	R_386_32 (g->.text)	R_386_GOT32 (g->.text)		R_386_GLOB_DAT (g->.got)	
int g = 0x01020304;	R_386_32 (g->.text)	R_386_32 (g->.text)	R_386_GOT32 (g->.text)		R_386_GLOB_DAT (g->.got)	
static int g;	R_386_32 (.bss->.text)	R_386_RELATIVE (->.text)	R_386_GOTOFF (.bss->.text)			
static int g = 0x01020304;	R_386_32 (.data->.text)	R_386_RELATIVE (->.text)	R_386_GOTOFF (.data->.text)			
extern void f();	R_386_PC32 (f->.text)	R_386_PC32 (f->.text)	R_386_PLT32 (f->.text)	R_386_PC32 (__x86.get_pc_thunk.bx->.text)	R_386_JMP_SLOT (f->.got.plt)	
void f(){}	R_386_PC32 (f->.text)	R_386_PC32 (f->.text)	R_386_PLT32 (f->.text)	R_386_GOTPC(_GLOBAL_OFFSET_TABLE_->.text)	R_386_JMP_SLOT (f->.got.plt)	
static void f(){}						

- 全局变量，在不加-fPIC编译生成的.o文件中，每个引用处对应一个R_386_32重定位项，非static全局变量，在不加-fPIC编译生成的.so文件中，每个引用处对应一个R_386_32重定位项；
- static全局变量，在不加-fPIC编译生成的.so文件中，每个引用处对应一个R_386_RELATIVE重定位项；
- 非static全局变量，在加-fPIC编译生成的.o文件中，每个引用处对应一个R_386_GOT32重定位项；
- static全局变量，在加-fPIC编译生成的.o文件中，每个引用处对应一个R_386_GOTOFF重定位项；
- 非static全局变量，在加-fPIC编译生成的.so文件中，每个引用处对应一个R_386_GOLB_DAT重定位项；
- a.out中利用extern引用.so中的变量，每个引用处对应一个R_386_COPY重定位项；
- 非static函数，在不加-fPIC编译生成的.o和.so文件中，每个调用处对应一个R_386_PC32重定位项；
- 非static函数，在加-fPIC编译生成的.o文件中，每个调用处对应一个R_386_PLT32重定位项；
- 非static函数，在加-fPIC编译生成的.so文件中，每个调用处对应一个R_386_JMP_SLOT重定位项；
- 全局变量，在加-fPIC编译生成的.o文件中，会额外生成R_386_PC32和R_386_GOTPC重定位项，非static函数，在加-fPIC编译生成的.o文件中，也会额外 生成R_386_PC32和R_386_GOTPC重定位项。

1. R_386_32

   公式：S+A

   S：重定项中VALUE成员所指符号的内存地址

   A：被重定位处原值，表示"引用符号的内存地址"与S的偏移

```
1  // g.c
2  extern int g1;
3  int g2;
4  int g3 = 0x03030303;
5  static int g4;
6  static int g5 = 0x05050505;
7
8  void fun(int a[5])
9  {
10     a[0] = g1;
11     a[1] = g2;
12     a[2] = g3;
13     a[3] = g4;
14     a[4] = g5;
15 }
```

   将g.c编译成g.o文件，观察包含的重定项信息：

```
[elf@ubuntu:rel]$ gcc -c g.c -g -Wall
[elf@ubuntu:rel]$ objdump -S g.o
Disassembly of section .text:
void fun(int a[5])
{
   0:    55                      push   %ebp
   1:    89 e5                   mov    %esp,%ebp
    a[0] = g1;
   3:    8b 15 00 00 00 00       mov    0x0,%edx
   9:    8b 45 08                mov    0x8(%ebp),%eax
   c:    89 10                   mov    %edx,(%eax)
    a[1] = g2;
   e:    8b 45 08                mov    0x8(%ebp),%eax
  11:    8d 50 04                lea    0x4(%eax),%edx
  14:    a1 00 00 00 00          mov    0x0,%eax
  19:    89 02                   mov    %eax,(%edx)
    a[2] = g3;
  1b:    8b 45 08                mov    0x8(%ebp),%eax
  1e:    8d 50 08                lea    0x8(%eax),%edx
  21:    a1 00 00 00 00          mov    0x0,%eax
  26:    89 02                   mov    %eax,(%edx)
    a[3] = g4;
  28:    8b 45 08                mov    0x8(%ebp),%eax
  2b:    8d 50 0c                lea    0xc(%eax),%edx
  2e:    a1 00 00 00 00          mov    0x0,%eax
  33:    89 02                   mov    %eax,(%edx)
    a[4] = g5;
  35:    8b 45 08                mov    0x8(%ebp),%eax
  38:    8d 50 10                lea    0x10(%eax),%edx
  3b:    a1 04 00 00 00          mov    0x4,%eax
  40:    89 02                   mov    %eax,(%edx)
}
  42:    5d                      pop    %ebp
  43:    c3                      ret

[elf@ubuntu:rel]$ objdump -r g.o
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000005 R_386_32          g1
00000015 R_386_32          g2
00000022 R_386_32          g3
0000002f R_386_32          .bss
0000003c R_386_32          .data

[elf@ubuntu:rel]$ objdump -sj .data g.o
Contents of section .data:
 0000 03030303 05050505                    ........
```

- "00000005 R_386_32 g1"：编译器连g1在哪个.o文件都不知道，当然更不知道g1运行时的地址，所以在g.o文件中设置一个重定项，要求后续过程根据"S(g1内存地址)+A(0)"，修改g.o镜像中0x05偏移处的值；
- "0000002f R_386_32 .bss"：g4在g.o文件.bss节的0偏移处（由于加载时必然知道.bss的内容为全0，就是说elf文件只需要记录.bss的位置和大小，不需要安排空间记录.bss内容，而且就算文件中为.bss节安排了空间，也无法区分g4在.bss节的什么位置，所以g4在.bss节中的偏移，要通过查看.bss节起始位置和g4符号的位置来验证），要求后续过程根据"S(g.o镜像中.bss的内存地址)+A(0)"，修改g.o镜像中0x2f偏移处的值；
- "0000003c R_386_32 .data"：g5在g.o文件.data节的0x04偏移处，要求后续过程根据"S(g.o镜像中.data的内存地址)+A(0x04)"，修改g.o镜像中0x3c偏移处的值；
- "00000015 R_386_32 g2"：g2在g.o文件的.bss节，要求后续过程根据"S(g2内存地址)+A(0)"，修改g.o镜像中0x15偏移处的值；
- "00000022 R_386_32 g3"：g3在g.o文件的.data节，要求后续过程根据"S(g3内存地址)+A(0)"，修改g.o镜像中0x22偏移处的值。

g1与g4/g5重定项区别：当前没有g1位置的任何线索，所以希望延迟到加载时，通过搜索动态符号表确定g1的内存地址，而g4/g5在g.o的.bss/.data节中，并且有static属性，不可能被外部引用，加载到内存必然还在g.o镜像的.bss/.data节中，所以编译器使用.bss/.data作为重定位计算参数，可以避免后续过程搜索动态符号表，提高重定位效率；

g2/g3与g4/g5重定项区别：g2/g3虽然和g4/g5一样，也在g.o的.bss/.data节中，但g2/g3可以被外部引用，在一种特殊情况下，g2/g3会被安排到其它地方，如果仍然使用在g.o镜像中.bss/.data的地址进行重定位，就会导致进程运行的逻辑错误，稍后介绍R_386_COPY类型时，会详细说明。

2. R_386_RELATIVE

公式：B+A

B：.so文件加载到内存中的基地址

A：被重定位处原值，表示引用符号在.so文件中的偏移

将上述g.o文件，链接成libg.so文件，重定位信息如下：

```
[elf@ubuntu:rel]$ gcc -shared g.o -o libg.so
[elf@ubuntu:rel]$ objdump -S libg.so
Disassembly of section .text:
void fun(int a[5])
{
 55b:    55                      push   %ebp
 55c:    89 e5                   mov    %esp,%ebp
   a[0] = g1;
 55e:    8b 15 00 00 00 00       mov    0x0,%edx
 564:    8b 45 08                mov    0x8(%ebp),%eax
 567:    89 10                   mov    %edx,(%eax)
   a[1] = g2;
 569:    8b 45 08                mov    0x8(%ebp),%eax
 56c:    8d 50 04                lea    0x4(%eax),%edx
 56f:    a1 00 00 00 00          mov    0x0,%eax
 574:    89 02                   mov    %eax,(%edx)
   a[2] = g3;
 576:    8b 45 08                mov    0x8(%ebp),%eax
 579:    8d 50 08                lea    0x8(%eax),%edx
 57c:    a1 00 00 00 00          mov    0x0,%eax
 581:    89 02                   mov    %eax,(%edx)
   a[3] = g4;
 583:    8b 45 08                mov    0x8(%ebp),%eax
 586:    8d 50 0c                lea    0xc(%eax),%edx
 589:    a1 24 20 00 00          mov    0x2024,%eax
 58e:    89 02                   mov    %eax,(%edx)
   a[4] = g5;
 590:    8b 45 08                mov    0x8(%ebp),%eax
 593:    8d 50 10                lea    0x10(%eax),%edx
 596:    a1 1c 20 00 00          mov    0x201c,%eax
 59b:    89 02                   mov    %eax,(%edx)
}
 59d:    5d                      pop    %ebp
 59e:    c3                      ret

[elf@ubuntu:rel]$ objdump -R libg.so
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
0000058a R_386_RELATIVE    *ABS*
00000597 R_386_RELATIVE    *ABS*
..
00000560 R_386_32          g1
00000570 R_386_32          g2
0000057d R_386_32          g3
..

[elf@ubuntu:rel]$ readelf -S libg.so
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [..]
  [21] .data             PROGBITS        00002014 001014 00000c 00  WA  0   0  4
  [22] .bss              NOBITS          00002020 001020 00000c 00  WA  0   0  4
  [..]

[elf@ubuntu:rel]$ objdump -sj .data libg.so
Contents of section .data:
 2014 14200000 03030303 05050505                    . .........
```

- "00000560 R_386_32 g1"：任然没有g1位置的任何线索，所以重定项保持原有的计算方法和参数；
- "00000570 R_386_32 g2"：不确定是否需要放弃.bss中的位置，所以仍然使用g2的内存地址进行重定位计算；
- "0000057d R_386_32 g3"：不确定是否需要放弃.data中的位置，所以仍然使用g3的内存地址进行重定位计算；
- "0000058a R_386_RELATIVE *ABS*"：.so文件.bss段的第一项用于保存.bss本身的位置，g.o的.bss节被安排在了libg.so的0x2024处，所以静态ld根据g.o中的R_386_32重定项，进一步精确了g4在libg.so的0x2024偏移处，但g4的内存地址，还需要加上libg.so的加载地址，所以重定位类型转换为R_386_RELATIVE；
- "00000597 R_386_RELATIVE *ABS*"：.so文件.data段的第一项用于保存.data本身的位置，g.o的.bss节被安排在了libg.so的0x2018处，所以静态ld根据g.o中的R_386_32重定项，进一步精确了g4在libg.so的0x201c偏移处，但g5的内存地址，还需要加上libg.so的加载地址，所以重定位类型转换为R_386_RELATIVE。

3. R_386_COPY
   公式：无

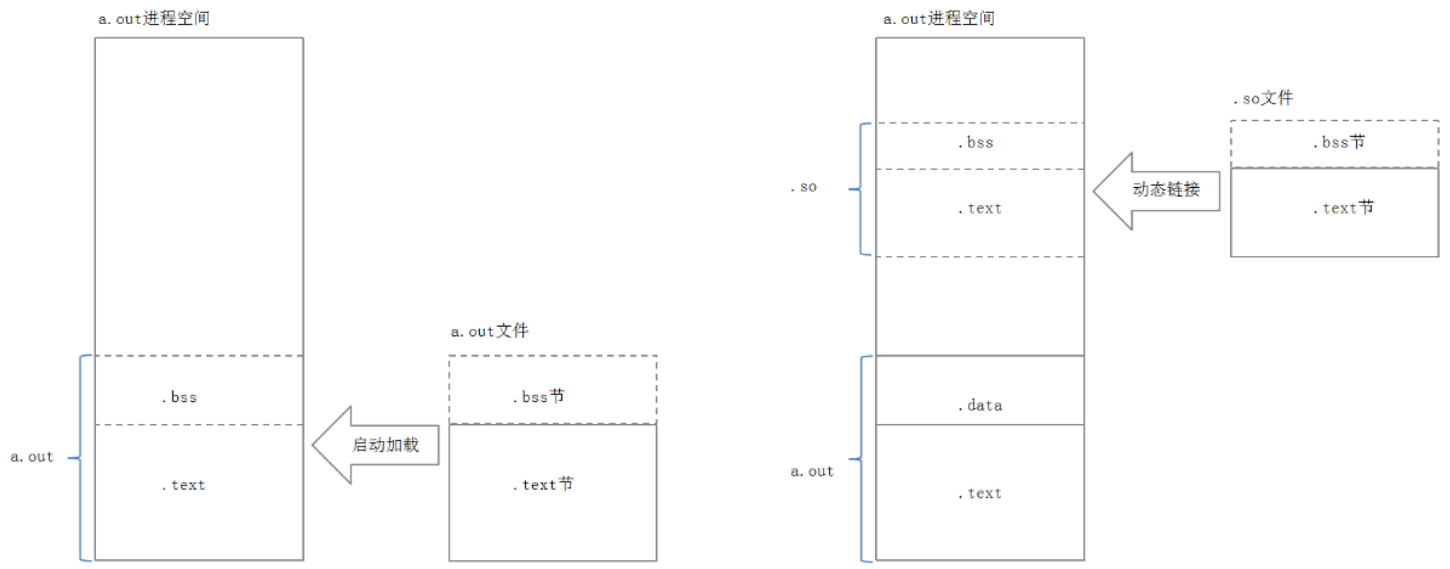
```c
// g.c
int g = 1;

// main.c
extern int g;

void fun(int *a)
{
    *a = g;
}

int main()
{
    return 0;
}
```

将g.c编译为libg.so，main.c编译为a.out，由于a.out引用了libg.so中的全局变量g，从而可以出现说明R_386_32类型时提到的特殊情况：

```
[elf@ubuntu:rel]$ gcc -c g.c main.c -g -Wall
[elf@ubuntu:rel]$ gcc -shared g.o -o libg.so
[elf@ubuntu:rel]$ gcc main.o -lg -L.
[elf@ubuntu:rel]$ objdump -S a.out
Disassembly of section .text:
void fun(int *a)
{
 804852d:    55                      push   %ebp
 804852e:    89 e5                   mov    %esp,%ebp
   *a = g;
 8048530:    8b 15 1c a0 04 08       mov    0x804a01c,%edx
 8048536:    8b 45 08                mov    0x8(%ebp),%eax
 8048539:    89 10                   mov    %edx,(%eax)
}
 804853b:    5d                      pop    %ebp
 804853c:    c3                      ret

[elf@ubuntu:rel]$ objdump -R a.out
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
..
0804a01c R_386_COPY        g
..

[elf@ubuntu:rel]$ readelf -S a.out
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [..]
  [25] .bss              NOBITS          0804a01c 00101c 000008 00  WA  0   0  4
  [..]

[elf@ubuntu:rel]$ nm a.out | grep B | grep g
0804a01c B g
[elf@ubuntu:rel]$ nm libg.so | grep D | grep g
00002018 D g
```

a.out中使用了libg.so中的全局变量g，这样就必须等到执行阶段，确定了libg.so在进程空间的位置后，才能知道g的绝对地址，不细想的话，可能会认为通过设置一个R_386_32或R_386_RELATIVE重定项，就能解决问题了。

但遗憾的是，a.out的.text段，不可以有重定项：



上图希望展示的是，在一个进程的创建过程中，a.out是最先映射到该进程的虚拟空间，然后才会映射所依赖的.so。换句话说，在a.out加载的时候，仍然不知道g的地址，而如果等加载libg.so时再处理重定项，虽然知道g的地址了，但a.out的.text段所在内存页，这时已经被设置为只读，也无法进行重定位。

所以，针对这种情况，静态ld会将g转移到a.out的.bss段。由于a.out的加载地址，是在静态链接阶段就确定的（通过链接脚本设置，32位系统默认设置为0x8048000），从而静态ld也可以知道g的运行时地址，那么就不需要重定项了，但同时又带来2个新的问题：

a. 毕竟libg.so中的g才是是原生的，怎么保证遵循libg.so中g的初始值？

其实这就是设计R_386_COPY类型的用意，它表示让动态ld加载libg.so时知道g的初始值后，将值复制到内存中a.out的.bss段。

但是如果再仔细想想，其实静态链接阶段，就有机会从libg.so中读取g的初始值，并且如果不将g安排在a.out的.bss段，而是安排在.data段，存储空间也具备了，按道理就不需要R_386_COPY类型了。个人猜测，可能是设计者本着.data只存储显式赋初值的变量的原则，而没有这样实现。

b. g既然已经转移到新地地方了，怎么保证lig.so和a.out的.text段使用同一处的g？

分析R_386_32类型时，已经看到g2/g3和g4/g5一样，分别在g.o的.bss/.data节，重定项中却仍然使用g2/g3作为计算参数，其实就是为了在这种情况下，放弃使用本身.bss/.data段中的g，而使用a.out中的g。

4. R_386_PC32

公式：S+A-P

S：重定项中VALUE成员所指符号的内存地址

A：被重定位处原值，表示"被重定位处"与"下一条指令"的偏移

P：被重定位处的内存地址

```c
// f.c
extern void f1();
void f2() {}
static void f3() {}

void fun()
{
    f1();
    f2();
    f3();
}
```

将f.c编译成f.o文件，观察包含的重定项信息：

```
[elf@ubuntu:rel]$ gcc -c f.c -g -Wall
[elf@ubuntu:rel]$ objdump -S f.o
Disassembly of section .text:
void f2() {}
   0:    55                      push   %ebp
   1:    89 e5                   mov    %esp,%ebp
   3:    5d                      pop    %ebp
   4:    c3                      ret

static void f3() {}
   5:    55                      push   %ebp
   6:    89 e5                   mov    %esp,%ebp
   8:    5d                      pop    %ebp
   9:    c3                      ret

void fun()
{
   a:    55                      push   %ebp
   b:    89 e5                   mov    %esp,%ebp
   d:    83 ec 08                sub    $0x8,%esp
    f1();
  10:    e8 fc ff ff ff          call   11 <fun+0x7>
    f2();
  15:    e8 fc ff ff ff          call   16 <fun+0xc>
    f3();
  1a:    e8 e6 ff ff ff          call   5 <f3>
}
  1f:    c9                      leave
  20:    c3                      ret

[elf@ubuntu:rel]$ objdump -r f.o
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000011 R_386_PC32        f1
00000016 R_386_PC32        f2
```

- "00000011 R_386_PC32 f1"：编译器连f1指令块在哪个.o文件都不知道，当然更不知道f1运行时的地址，所以在g.o文件中设置一个重定项，要求后续过程根据"S(f1内存地址)+A(-4)-P(被重定位处内存地址)"，即"S(f1内存地址)-(p+4)(下一条指令内存地址)"，修改g.o文件中0x11偏移处的值；
- "00000016 R_386_PC32 f2"：f2类似分析R_386_32类型时的g2/g3，虽然在g.o文件，但有可能被外部调用，所以和f1一样，编译器在g.o文件中设置一个重定项，要求后续过程根据"S(f2内存地址)+A(-4)-P(被重定位处内存地址)"，即"S(f2内存地址)-(p+4)(下一条指令内存地址)"，修改g.o文件中的0x16偏移处的值。

由于调用函数的指令中，要求的是相对地址，并且编译阶段就能确定f3()与fun()的偏移，即"f3加载地址(B+0x05)-下一条指令内存地址(B+1f)=0xe6ffffff"，加载到内存也不会发生改变，所以0x1b处不需要被重定位。

将上述f.o文件，链接为libf.so，静态ld无法对R_386_PC32重定项做进一步处理，这样，加载时动态ld会通过搜索动态符号表，确定libf.so镜像中0x53c/0x541处的地址值，保证运行时能调用到到f1()/f2()函数：

```
[elf@ubuntu:rel]$ gcc -shared f.o -o libf.so
[elf@ubuntu:rel]$ objdump -S libf.so
Disassembly of section .text:
void f2() {}
 52b:    55                      push   %ebp
 52c:    89 e5                   mov    %esp,%ebp
 52e:    5d                      pop    %ebp
 52f:    c3                      ret

static void f3() {}
 530:    55                      push   %ebp
 531:    89 e5                   mov    %esp,%ebp
 533:    5d                      pop    %ebp
 534:    c3                      ret

void fun()
{
 535:    55                      push   %ebp
 536:    89 e5                   mov    %esp,%ebp
 538:    83 ec 08                sub    $0x8,%esp
    f1();
 53b:    e8 fc ff ff ff          call   53c <fun+0x7>
    f2();
 540:    e8 fc ff ff ff          call   541 <fun+0xc>
    f3();
 545:    e8 e6 ff ff ff          call   530 <f3>
}
 54a:    c9                      leave
 54b:    c3                      ret

[elf@ubuntu:rel]$ objdump -R libf.so
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
..
0000053c R_386_PC32        f1
00000541 R_386_PC32        f2
..
```

5. R_386_GOTPC

公式：GOT+A-P

GOT：运行时，.got段的结束地址

A：被重定位处原值，表示"被重定位处"在机器码中的偏移

P：被重定位处的内存地址

```c
// g.c
extern int g1;

void fun(int a[1])
{
    a[0] = g1;
}
```

由于程序执行时，eip寄存器保存的一定是当前指令的内存地址，虽然eip寄存器不直接提供给软件使用，但是有间接的方法可以获取，那么从逻辑上讲，所有跟代码区有固定偏移的内容，编译和静态链接阶段，就可以确定它们的内存地址。利用这个特点，可以将代码区中的被重定位处转移出去，加-fPIC选项将g.c编译为g.o，并链接为libg.so，可以验证这一点：

```
[elf@ubuntu:rel]$ gcc -c g.c -fPIC -g -Wall
[elf@ubuntu:rel]$ objdump -S g.o
Disassembly of section .text:
void fun(int a[1])
{
   0:    55                      push   %ebp
   1:    89 e5                   mov    %esp,%ebp
   3:    e8 fc ff ff ff          call   4 <fun+0x4>
   8:    81 c1 02 00 00 00       add    $0x2,%ecx
    a[0] = g1;
   e:    8b 81 00 00 00 00       mov    0x0(%ecx),%eax
  14:    8b 10                   mov    (%eax),%edx
  16:    8b 45 08                mov    0x8(%ebp),%eax
  19:    89 10                   mov    %edx,(%eax)
}
  1b:    5d                      pop    %ebp
  1c:    c3                      ret

Disassembly of section .text.__x86.get_pc_thunk.cx:
00000000 <__x86.get_pc_thunk.cx>:
   0:    8b 0c 24                mov    (%esp),%ecx
   3:    c3                      ret

[elf@ubuntu:rel]$ objdump -r g.o
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000004 R_386_PC32        __x86.get_pc_thunk.cx
0000000a R_386_GOTPC       _GLOBAL_OFFSET_TABLE_
00000010 R_386_GOT32       g1

[elf@ubuntu:rel]$ readelf -S g.o
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [..]

[elf@ubuntu:rel]$ gcc -shared g.o -o libg.so
[elf@ubuntu:rel]$ objdump -S libg.so
Disassembly of section .text:
void fun(int a[1])
{
 50b:    55                      push   %ebp
 50c:    89 e5                   mov    %esp,%ebp
 50e:    e8 15 00 00 00          call   528 <__x86.get_pc_thunk.cx>
 513:    81 c1 ed 1a 00 00       add    $0x1aed,%ecx
    a[0] = g1;
 519:    8b 81 ec ff ff ff       mov    -0x14(%ecx),%eax
 51f:    8b 10                   mov    (%eax),%edx
 521:    8b 45 08                mov    0x8(%ebp),%eax
 524:    89 10                   mov    %edx,(%eax)
}
 526:    5d                      pop    %ebp
 527:    c3                      ret

00000528 <__x86.get_pc_thunk.cx>:
 528:    8b 0c 24                mov    (%esp),%ecx
 52b:    c3                      ret

[elf@ubuntu:rel]$ objdump -R libg.so
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
..
00001fec R_386_GLOB_DAT    g1
..

[elf@ubuntu:rel]$ readelf -S libg.so
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [..]
  [19] .got              PROGBITS        00001fe8 000fe8 000018 04  WA  0   0  4
  [..]

[elf@ubuntu:rel]$ objdump -sj .got libg.so
Contents of section .got:
 1fe8 00000000 00000000 00000000 00000000  ................
 1ff8 00000000 00000000                    ........
```

g.o中包含3个重定项：

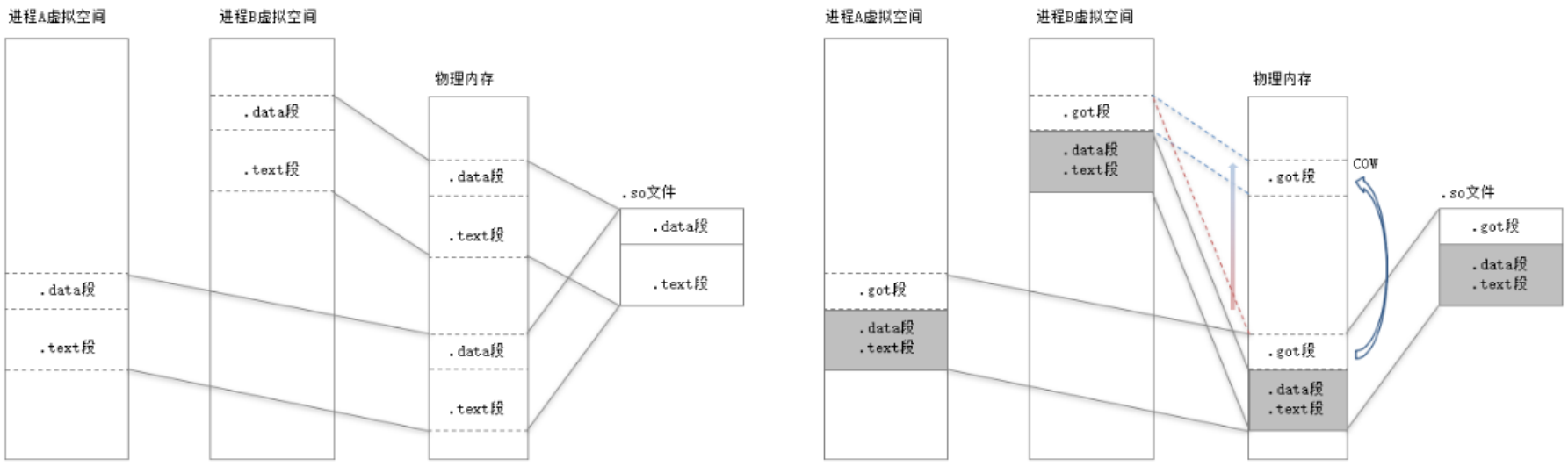
- "00000004 R_386_PC32 __x86.get_pc_thunk.cx"：R_386_PC32重定位类型已经介绍过了，这条重定项可以保证，运行时当前指令可以调用到编译器自动生成的__x86.get_pc_thunk.cx()函数，由于call指令会将下一条指令内存地址B+0x513压栈，这样从该函数经过一次再回到B+0x513处的指令时，当前指令的内存地址B+0x513就存到了ecx寄存器；
- "0000000a R_386_GOTPC _GLOBAL_OFFSET_TABLE_"：要求静态ld根据"GOT(B+.got结束位置在libg.so中的偏移)+A(2)-P(B+0x515)"，即libg.so文件中.got结束位置相对0x515处机器码的偏移，修改g.o文件中0x0a偏移处的值，这样运行时加上ecx寄存器中当前指令的内存地址后，就是.got段结束位置的内存地址；
- "00000010 R_386_GOT32 g1"：要求静态ld在目标文件中生成.got表，并在.got表中安排4字节存储g1地址，这样代码区就可以从.got表中获取g1地址，而.got表运行时的结束地址，以及存储g1地址的位置在.got表中的偏移，静态链接阶段都是知道的，从而就不需要对代码区进行重定位。

静态ld在libg.so中设置了.got段，并将代码区中的重定位处，转移到.got段中：

- "00001fec R_386_GLOB_DAT g1"：0x1fec处用于保存运行g1的内存地址，但当前没有g1位置的任何线索，所以留下重定项，要求后续过程进行修改。

对于R_386_32、R_386_RELATIVE类型的重定项，由于被重定位处在代码区，而重定项计算参数的地址，在不同进程中是不同的，所以不同进程对.so代码区的修改要求就不同，这样就不能共享同一份物理内存中的.so镜像。

R_386_GLOB_DAT的优势就在于，它将"散落"在代码区的被重定位处，集中转移到.got表中，从而大大减小了不可共享区域，如下图所示，进程B希望加载.so文件时，发现内存中已经存在该.so的镜像了，就直接映射到自己的虚拟空间，动态ld在处理重定项时，仅需要修改小小的.got段，并通过COW（写时复制）机制，创建了一个.got副本，从而也可以保证与其它进程互不干扰。



6. R_386_GOT32

公式：G

G：引用符号的地址指针，相对于GOT的偏移

```
// g.c
extern int g1;
int g2;
int g3 = 0x03030303;

void fun(int a[3])
{
    a[0] = g1;
    a[1] = g2;
    a[2] = g3;
}
```

将g.c编译成g.o文件，观察包含的重定项信息：

```
[elf@ubuntu:rel]$ gcc -c g.c -fPIC -g -Wall
[elf@ubuntu:rel]$ objdump -S g.o
Disassembly of section .text:
void fun(int a[3])
{
   0:    55                      push   %ebp
   1:    89 e5                   mov    %esp,%ebp
   3:    e8 fc ff ff ff          call   4 <fun+0x4>
   8:    81 c1 02 00 00 00       add    $0x2,%ecx
    a[0] = g1;
   e:    8b 81 00 00 00 00       mov    0x0(%ecx),%eax
  14:    8b 10                   mov    (%eax),%edx
  16:    8b 45 08                mov    0x8(%ebp),%eax
  19:    89 10                   mov    %edx,(%eax)
    a[1] = g2;
  1b:    8b 45 08                mov    0x8(%ebp),%eax
  1e:    8d 50 04                lea    0x4(%eax),%edx
  21:    8b 81 00 00 00 00       mov    0x0(%ecx),%eax
  27:    8b 00                   mov    (%eax),%eax
  29:    89 02                   mov    %eax,(%edx)
    a[2] = g3;
  2b:    8b 45 08                mov    0x8(%ebp),%eax
  2e:    8d 50 08                lea    0x8(%eax),%edx
  31:    8b 81 00 00 00 00       mov    0x0(%ecx),%eax
  37:    8b 00                   mov    (%eax),%eax
  39:    89 02                   mov    %eax,(%edx)
}
  3b:    5d                      pop    %ebp
  3c:    c3                      ret

Disassembly of section .text.__x86.get_pc_thunk.cx:
00000000 <__x86.get_pc_thunk.cx>:
   0:    8b 0c 24                mov    (%esp),%ecx
   3:    c3                      ret

[elf@ubuntu:rel]$ objdump -r g.o
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000004 R_386_PC32        __x86.get_pc_thunk.cx
0000000a R_386_GOTPC       _GLOBAL_OFFSET_TABLE_
00000010 R_386_GOT32       g1
00000023 R_386_GOT32       g2
00000033 R_386_GOT32       g3
```

- "00000010 R_386_GOT32 g1"：要求静态ld根据"G(g1地址指针相对GOT的偏移)"，修改g.o镜像0x10偏移处的值；
- "00000023 R_386_GOT32 g2"：要求静态ld根据"G(g2地址指针相对GOT的偏移)"，修改g.o镜像0x23偏移处的值；
- "00000033 R_386_GOT32 g3"：要求静态ld根据"G(g3地址指针相对GOT的偏移)"，修改g.o镜像0x33偏移处的值；

7. R_386_GOLB_DAT

   公式：S

   S：重定项中VALUE成员所指符号的内存地址

   将上述g.o链接为libg.so文件，发现被重定位处都被集中转移到.got段中：

```
[elf@ubuntu:rel]$ gcc -shared g.o -o libg.so
[elf@ubuntu:rel]$ objdump -S libg.so
Disassembly of section .text:
void fun(int a[3])
{
 54b:    55                      push   %ebp
 54c:    89 e5                   mov    %esp,%ebp
 54e:    e8 35 00 00 00          call   588 <__x86.get_pc_thunk.cx>
 553:    81 c1 ad 1a 00 00       add    $0x1aad,%ecx
    a[0] = g1;
 559:    8b 81 e4 ff ff ff       mov    -0x1c(%ecx),%eax
 55f:    8b 10                   mov    (%eax),%edx
 561:    8b 45 08                mov    0x8(%ebp),%eax
 564:    89 10                   mov    %edx,(%eax)
    a[1] = g2;
 566:    8b 45 08                mov    0x8(%ebp),%eax
 569:    8d 50 04                lea    0x4(%eax),%edx
 56c:    8b 81 f4 ff ff ff       mov    -0xc(%ecx),%eax
 572:    8b 00                   mov    (%eax),%eax
 574:    89 02                   mov    %eax,(%edx)
    a[2] = g3;
 576:    8b 45 08                mov    0x8(%ebp),%eax
 579:    8d 50 08                lea    0x8(%eax),%edx
 57c:    8b 81 e8 ff ff ff       mov    -0x18(%ecx),%eax
 582:    8b 00                   mov    (%eax),%eax
 584:    89 02                   mov    %eax,(%edx)
}
 586:    5d                      pop    %ebp
 587:    c3                      ret

00000588 <__x86.get_pc_thunk.cx>:
 588:    8b 0c 24                mov    (%esp),%ecx
 58b:    c3                      ret

[elf@ubuntu:rel]$ objdump -R libg.so
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
..
00001fe4 R_386_GLOB_DAT    g1
00001fe8 R_386_GLOB_DAT    g3
..
00001ff4 R_386_GLOB_DAT    g2
..

[elf@ubuntu:rel]$ readelf -S libg.so
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [..]
  [19] .got              PROGBITS        00001fe0 000fe0 000020 04  WA  0   0  4
  [..]

[elf@ubuntu:rel]$ objdump -sj .got libg.so
Contents of section .got:
 1fe0 00000000 00000000 00000000 00000000  ................
 1ff0 00000000 00000000 00000000 00000000  ................
```

8. R_386_GOTOFF

公式：S-GOT

S：重定项中VALUE成员所指符号的内存地址

GOT：运行时，.got段的结束地址

```
// g.c
static int g4;
static int g5 = 0x05050505;

void fun(int a[2])
{
    a[0] = g4;
    a[1] = g5;
}
```

将g.c编译为g.o文件，并且链接为libg.so文件，观察包含的重定项信息：

```
[elf@ubuntu:rel]$ gcc -c g.c -fPIC -g -Wall
[elf@ubuntu:rel]$ objdump -S g.o
Disassembly of section .text:
void fun(int a[2])
{
   0:    55                      push   %ebp
   1:    89 e5                   mov    %esp,%ebp
   3:    e8 fc ff ff ff          call   4 <fun+0x4>
   8:    81 c1 02 00 00 00       add    $0x2,%ecx
    a[0] = g4;
   e:    8b 91 00 00 00 00       mov    0x0(%ecx),%edx
  14:    8b 45 08                mov    0x8(%ebp),%eax
  17:    89 10                   mov    %edx,(%eax)
    a[1] = g5;
  19:    8b 45 08                mov    0x8(%ebp),%eax
  1c:    8d 50 04                lea    0x4(%eax),%edx
  1f:    8b 81 00 00 00 00       mov    0x0(%ecx),%eax
  25:    89 02                   mov    %eax,(%edx)
}
  27:    5d                      pop    %ebp
  28:    c3                      ret

Disassembly of section .text.__x86.get_pc_thunk.cx:
00000000 <__x86.get_pc_thunk.cx>:
   0:    8b 0c 24                mov    (%esp),%ecx
   3:    c3                      ret

[elf@ubuntu:rel]$ objdump -r g.o
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000004 R_386_PC32        __x86.get_pc_thunk.cx
0000000a R_386_GOTPC       _GLOBAL_OFFSET_TABLE_
00000010 R_386_GOTOFF      .bss
00000021 R_386_GOTOFF      .data

[elf@ubuntu:rel]$ gcc -shared g.o -o libg.so
[elf@ubuntu:rel]$ objdump -S libg.so
Disassembly of section .text:
void fun(int a[2])
{
 4eb:    55                      push   %ebp
 4ec:    89 e5                   mov    %esp,%ebp
 4ee:    e8 21 00 00 00          call   514 <__x86.get_pc_thunk.cx>
 4f3:    81 c1 0d 1b 00 00       add    $0x1b0d,%ecx
    a[0] = g4;
 4f9:    8b 91 20 00 00 00       mov    0x20(%ecx),%edx
 4ff:    8b 45 08                mov    0x8(%ebp),%eax
 502:    89 10                   mov    %edx,(%eax)
    a[1] = g5;
 504:    8b 45 08                mov    0x8(%ebp),%eax
 507:    8d 50 04                lea    0x4(%eax),%edx
 50a:    8b 81 18 00 00 00       mov    0x18(%ecx),%eax
 510:    89 02                   mov    %eax,(%edx)
}
 512:    5d                      pop    %ebp
 513:    c3                      ret

00000514 <__x86.get_pc_thunk.cx>:
 514:    8b 0c 24                mov    (%esp),%ecx
 517:    c3                      ret

[elf@ubuntu:rel]$ objdump -R libg.so
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
00001f00 R_386_RELATIVE    *ABS*
00001f04 R_386_RELATIVE    *ABS*
00002014 R_386_RELATIVE    *ABS*
00001fec R_386_GLOB_DAT    _ITM_deregisterTMCloneTable
00001ff0 R_386_GLOB_DAT    __cxa_finalize
00001ff4 R_386_GLOB_DAT    __gmon_start__
00001ff8 R_386_GLOB_DAT    _Jv_RegisterClasses
00001ffc R_386_GLOB_DAT    _ITM_registerTMCloneTable
0000200c R_386_JUMP_SLOT   __cxa_finalize
00002010 R_386_JUMP_SLOT   __gmon_start__

[elf@ubuntu:rel]$ readelf -S libg.so
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [..]
  [21] .data             PROGBITS        00002014 001014 000008 00  WA  0   0  4
  [22] .bss              NOBITS          0000201c 00101c 000008 00  WA  0   0  4
  [..]
```

编译阶段不知道g.o中的.bss/.data节会被链接到libg.so中的什么位置，所以设置了R_386_GOTOFF重定项，要求静态ld根据"S(.bss/.data内存地址)-GOT(运行时.got结束地址)"，修改被重定位处的值。

前面已经介绍过，运行时ecx寄存器保存的一定是.got的结束地址，再加上g4/g5具有static属性，加载到内存后，仍然在libg.so镜像的.bss/.data段中，那么通过g4/g4在.bss/.data中的偏移，以及.bss/.data与.got结束位置的偏移，在静态链接阶段就能知道运行时g4/g5的内存地址，从而libg.so中就不存在对g4/g5引用处的重定项了。

9. R_386_PLT32

公式：L+A-P

L：<重定项中VALUE成员所指符号@plt>的内存地址

A：被重定位处原值，表示"被重定位处"相对于"下一条指令"的偏移

P：被重定位处的内存地址

```
// f.c
extern void f1();
void f2() {}
static void f3() {}

void fun()
{
    f1();
    f2();
    f3();
}
```

将f.c编译成f.o文件，观察包含的重定项信息：

```
[elf@ubuntu:rel]$ gcc -c f.c -fPIC -g -Wall
[elf@ubuntu:rel]$ objdump -S f.o
Disassembly of section .text:
void fun()
{
   a:    55                      push   %ebp
   b:    89 e5                   mov    %esp,%ebp
   d:    53                      push   %ebx
   e:    83 ec 04                sub    $0x4,%esp
  11:    e8 fc ff ff ff          call   12 <fun+0x8>
  16:    81 c3 02 00 00 00       add    $0x2,%ebx
    f1();
  1c:    e8 fc ff ff ff          call   1d <fun+0x13>
    f2();
  21:    e8 fc ff ff ff          call   22 <fun+0x18>
    f3();
  26:    e8 da ff ff ff          call   5 <f3>
}
  2b:    83 c4 04                add    $0x4,%esp
  2e:    5b                      pop    %ebx
  2f:    5d                      pop    %ebp
  30:    c3                      ret

Disassembly of section .text.__x86.get_pc_thunk.bx:
00000000 <__x86.get_pc_thunk.bx>:
   0:    8b 1c 24                mov    (%esp),%ebx
   3:    c3                      ret

[elf@ubuntu:rel]$ objdump -r f.o
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000012 R_386_PC32        __x86.get_pc_thunk.bx
00000018 R_386_GOTPC       _GLOBAL_OFFSET_TABLE_
0000001d R_386_PLT32       f1
00000022 R_386_PLT32       f2
```

- "0000001d R_386_PLT32 f1"：要求静态ld生成<f1@plt>函数，并根据"L(<f1@plt>函数地址)+A(-4)-P"，即<f1@plt>相对于下一条指令的相对地址，修改f.o镜像中0x1d偏移处的值；
- "00000022 R_386_PLT32 f2"：要求静态ld生成<f2@plt>函数，并根据"L(<f2@plt>函数地址)+A(-4)-P"，即<f2@plt>相对于下一条指令的相对地址，修改f.o镜像中0x22偏移处的值。

可以看出，源码中调用f1()、f2()函数的语句，对应的机器码，并没有直接跳转到f1、f2指令块，而是调用了<f1@plt>、<f2@plt>函数，接下来通过分析R_386_JMP_SLOT类型就会知道，这两个函数相当于"中间跳板"，用于实现重定项的延迟处理。

10.R_386_JMP_SLOT

公式：S（与R_386_GLOB_DAT的公式一样，但对于动态ld，R_386_JMP_SLOT类型与R_386_RELATIVE等价）

S：重定项中VALUE成员所指符号的内存地址

将上述f.o，链接为libf.so文件，观察包含的重定项信息：

```
[elf@ubuntu:rel]$ gcc -shared f.o -o libf.so
[elf@ubuntu:rel]$ objdump -S libf.so
Disassembly of section .plt:
000003d0 <f2@plt-0x10>:
 3d0:    ff b3 04 00 00 00        push   0x4(%ebx)
 3d6:    ff a3 08 00 00 00        jmp    *0x8(%ebx)
 3dc:    00 00                    add    %al,(%eax)
    ...

000003e0 <f2@plt>:
 3e0:    ff a3 0c 00 00 00        jmp    *0xc(%ebx)
 3e6:    68 00 00 00 00           push   $0x0
 3eb:    e9 e0 ff ff ff           jmp    3d0 <_init+0x2c>

00000410 <f1@plt>:
 410:    ff a3 18 00 00 00        jmp    *0x18(%ebx)
 416:    68 18 00 00 00           push   $0x18
 41b:    e9 b0 ff ff ff           jmp    3d0 <_init+0x2c>

Disassembly of section .text:
00000420 <__x86.get_pc_thunk.bx>:
 420:    8b 1c 24                 mov    (%esp),%ebx
 423:    c3                       ret

void fun()
{
 555:    55                       push   %ebp
 556:    89 e5                    mov    %esp,%ebp
 558:    53                       push   %ebx
 559:    83 ec 04                 sub    $0x4,%esp
 55c:    e8 bf fe ff ff           call   420 <__x86.get_pc_thunk.bx>
 561:    81 c3 9f 1a 00 00        add    $0x1a9f,%ebx
    f1();
 567:    e8 a4 fe ff ff           call   410 <f1@plt>
    f2();
 56c:    e8 6f fe ff ff           call   3e0 <f2@plt>
    f3();
 571:    e8 da ff ff ff           call   550 <f3>
}
 576:    83 c4 04                 add    $0x4,%esp
 579:    5b                       pop    %ebx
 57a:    5d                       pop    %ebp
 57b:    c3                       ret

[elf@ubuntu:rel]$ objdump -R libf.so
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
00001f00 R_386_RELATIVE    *ABS*
00001f04 R_386_RELATIVE    *ABS*
0000201c R_386_RELATIVE    *ABS*
00001fec R_386_GLOB_DAT    _ITM_deregisterTMCloneTable
00001ff0 R_386_GLOB_DAT    __cxa_finalize
00001ff4 R_386_GLOB_DAT    __gmon_start__
00001ff8 R_386_GLOB_DAT    _Jv_RegisterClasses
00001ffc R_386_GLOB_DAT    _ITM_registerTMCloneTable
0000200c R_386_JUMP_SLOT   f2
00002010 R_386_JUMP_SLOT   __cxa_finalize
00002014 R_386_JUMP_SLOT   __gmon_start__
00002018 R_386_JUMP_SLOT   f1

[elf@ubuntu:rel]$ readelf -S libf.so
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [..]
  [ 8] .rel.plt          REL             00000384 000384 000020 08   A  3  10  4
  [..]
  [18] .dynamic          DYNAMIC         00001f0c 000f0c 0000e0 08  WA  4   0  4
  [..]
  [20] .got.plt          PROGBITS        00002000 001000 00001c 04  WA  0   0  4
  [..]

[elf@ubuntu:rel]$ objdump -sj .rel.plt libf.so
Contents of section .rel.plt:
 0384 0c200000 070b0000 10200000 07020000  . ....... ......
 0394 14200000 07030000 18200000 07040000  . ....... ......

[elf@ubuntu:rel]$ objdump -sj .got.plt libf.so
Contents of section .got.plt:
 2000 0c1f0000 00000000 00000000 e6030000  ................
 2010 f6030000 06040000 16040000           ............
```

- "00002018 R_386_JUMP_SLOT f1"：被重定位处在libf.so镜像的0x2018偏移处（.got.plt段中），0x567处的指令第一次被执行时，由_dl_runtime_resolve()函数根据"S(f1内存地址)"，修改被重定位处的值；
- "0000200c R_386_JUMP_SLOT f2"：被重定位处在libf.so镜像的0x200c偏移处（.got.plt段中），0x56c处的指令第一次被执行时，由_dl_runtime_resolve()函数根据"S(f2内存地址)"，修改被重定位处的值；

R_386_JMP_SLOT是10种类型中最复杂的，必须先要了解.got.plt前三项的含义：

- 第1项：用于保存.dynamic段的内存地址，初始值为.dynamic段在libg.so文件中的偏移；

- 第2项：用于保存内存中libf.so模块的id，用于区分各个已加载的.so模块；
- 第3项：用于保存_dl_runtime_resolve()函数的内存地址，由动态ld完成填写。

另外，通过前面的介绍可以知道，ebx寄存器存的一定是.got结束地址B+0x2000，然后按照图中标记的执行顺序，在大脑中连续2次模拟执行B+0x567处的指令，就会看出如下规律：

- 第一次执行时，B+0x410处的jmp指令，会跳转到B+0x416处（因为0x18(%ebx)指向B+0x2018处，而此处初始值为B+0x416），接着将被重定位处地址压栈，并再跳转到0x3d0将libf.so模块id压栈，最终进入_dl_runtime_resolve()函数，确定f1地址后覆盖到B+0x2018处；
- 第二次执行时，由于B+0x2018处已经是f1的地址了，从而B+0x410处的jmp指令，就会直接进入f1()函数。

对f2()函数的调用同理，这样做虽然会多一次跳转，但是保证了程序执行的平滑性，避免大量调用libg.so中的函数时，在加载.so时出现"卡顿"的现象，而且有时候很多分支根本没有机会被执行，所以这是一种折衷的处理。

参考：

https://docs.oracle.com/cd/E23824_01/html/819-0690/

http://www.cnblogs.com/catch/p/3857964.html

https://www.cnblogs.com/lanrenxinxin/p/5573018.html

转载请注明出处：https://bbs.pediy.com/thread-246373.htm

最后于 1天前 被xinpoo编辑 ，原因：

最新回复 (8)

kanxue 1天前 2 楼

感谢分享！

8

爱吃菠菜 1天前 3 楼

马克下，收藏看看，我以前也写了个粗略的arm版本的。

1

最后于 1天前 被爱吃菠菜编辑 ，原因：

莫灰灰 1天前 4 楼

学习了。

8

xinpoo 1天前 5 楼

感谢支持。

1

Czhiqiang 1天前 6 楼

感谢分享~

shayi 1天前 7 楼

相当棒的链接器研究资料，学习一下~

8

聖blue 17小时前 8 楼

👍

最新回复 (8)

落叶似秋 2小时前 9 楼

学习了，谢谢楼主