IBM

developerWorks 中国 技术主题 Linux 文档库

Intel平台下linux中ELF文件动态链接的加载、解析及实例分析（二）：函数解析与卸载

上篇文章Intel平台下Linux中ELF文件动态链接的加载、解析及实例分析（一）：加载阐述了ELF文件被加载的时候所经历的一般过程。那我们现在就来解决在上一篇文章的最后所提到的那几个问题，以及那些在dl_open_worker中没有讲解的代码。

王瑞川，从事 Linux 开发工作，愿与志同道合的人士一起探讨，电子邮件地址是 jeppeterone@163.com。

2003 年 12 月 01 日

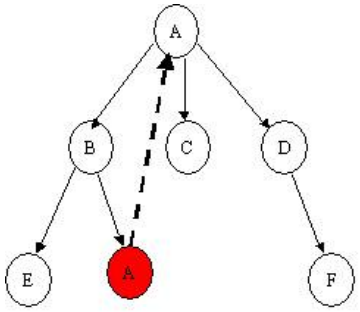
相信读者已经看过了 Intel平台下Linux中ELF文件动态链接的加载、解析及实例分析（一）：加载的内容了，了解了ELF文件被加载的时候所经历的一般过程。那我们现在就来解决在上一篇文章的最后所提到的那几个问题，以及那些在dl_open_worker中没有讲解的代码。

一、_dl_map_object_deps 函数分析

由于源代码过分的冗长，并且由于效率的考虑，使原本很简单的代码变成了一件 TRAMPOLINE 的事情，所以我对它进行了大幅度的改变，不仅删除了所有不必要的代码，而且还用伪代码来展现它最初的设计思想。

```
13  _dl_map_object_deps  (struct link_map *lmap)
14
15  {
16
17    struct list_head*  add_list;
18    char* load_dl_name;
19    struct link_map* curlmap;
20    Elf32_Dyn*  needed_dyn;
21    struct link_map*  new_lmap;
22    int lmap_count=1;
23
24    add_lmap_to_list(lmap,add_list);
25
26      for_each_in_list(add_list,curlmap)
27        {
28          for_every_DT_NEEDED_section(curlmap,needed_dyn)
29            {
30              load_dl_name=  get_needed_name(curlmap,needed_dyn);
31
32              new_lmap=_dl_map_object(load_dl_name);
33
34
35              add_to_list_tail_uniq(add_list,new_lmap);
36            }
37        }
38
39    lmap_count=count_the_list(lmap);
40
41    lmap->l_initfini=(struct link_map**)malloc ((2*lmap_count+1)*(struct link_map*));
42
43    lmap->l_searchlist.r_list=&lmap->l_initfini[lmap_count+1];
44    lmap->l_searchlist.r_nlist=lmap_count;
45
46
47    copy_each_add_list_to_searchlist(lmap,add_list,lmap_count);
48
49    free_the_add_list(add_list);
50
51  }
```

先说明，其实加载一个动态链接库的依赖动态链接库不是一件简单的事，因为所有的动态链接库可能还有它自己所依赖的动态链接库，如果采用递归简单方法实现不仅是不可能的-----因为你可以参看第一篇的文章，那里提到了一个在加载动态链接库中的加锁问题，而且也是没有必要的，你并不能保证这样的动态链接库依赖关系会不会形成一个依赖循环，就像下面的一张图所显示的那样：



这样最简单的想法就是我们不重复的加载所有的动态链接库，这里就用一个单链实现-----在原来的程序中也是用这个方法，但那里用来分配的方法是在栈中直接实现，这样可以加快程序的运行，但程序可读性大大减弱了。

23 行就首先就把 **lmap** 自己加入这个 **struct list** 中去，在 **26** 行的 **for_each_in_list(add_list,curlmap)** 其实是就是把 **curlmap=curlmap->next**，并判断它的 **curlmap!=NULL,**

28 行的 **for_every_DT_NEEDED_section(curlmap,needed_dyn)**

主要就是 **needed_dyn=curlmap->l_info[DT_NEEDED];** 但这里要注意的是，在一个动态链接库中可能有不只一个，就像在 **readelf -a** 的例子

```
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libstdc++-libc6.2-2.so.3]
 0x00000001 (NEEDED)                     Shared library: [libm.so.6]
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
```

更确切的是要在 **lmap-> l_ld** 的 **dynamic section** 中查找它的 **d_tag** 为 **DT_NEEDED** 中

30 行的 **get_needed_name** 用的方法是这样的

```
load_dl_name=curlmap->l_addr+need_dyn->d_un.d_ptr+curlmap->l_i
nfo[DT_STRTAB];
```

很明显这里就会把这个动态链接库映射来完成它的加载，而 **35** 行是要把 **add_list** 扩充，这里只会对同一个动态链接库加载一次，所以不会有前面的循环加载，再回过头来看 **26** 行到 **37** 行之间的那个循环，如果在 **35** 行中加入了那个没有重复的动态链接库。那整个循环就可能继续循环下去。

从 **39** 行到 **51** 行之中就把这个函数中已经得到的依赖动态链接库 **copy** 入 **l_searchlist** 与 **l_initfini** 这两个的重要数组中**,** 巧妙的是它们采用了一起分配的。最后前面的那个临时单链表。

二、相对转移，绝对转移

在学习汇编语言的时候，我们对不同的寻址方式肯定有很深的印象。但对于在汇编语言中同样重要的转移指令，只是一笔带过（用到了call 与 jxx ---------- 这里的 jxx 是指如 jmp jae jbe 这样的有条件转移指令和无条件转移指令）。然而，如果讲到动态链接库的链接实现则一定要提到这一内容。

所谓相对转移，就是这个二进制代码的中的它是可以在重定位的环境中不经修改，就可以运行的。如下面的情况，

```
719:  e9 e2 fe ff ff         jmp    600 <loop_put_buffer>
```

变成一般的地址是这样的

```
movl %eip,%eax
addl $0xfffffee2,%eax
movl %eax,%eip
```

这里旁边的 719 就是这个 ELF 文件与起始地址相比的偏移量，而在里面的 e9 e2 fe ff ff 如果写成看的往后退 0x11e 因为这是 ff ff fe e2（intel 是 little endian 表示方法）所表示的 -0x11e 的数。如果把 719 加上 5 再减去 600 就是这个数了。这便是处理器的相对转移。

还有另一种转移方式，就是绝对转移。

```
2b6:  ff d0                  call   *%eax
```

这个如果用最简单的代码来表示是

```
addl $2,%eip
pushl %eip
movl (%eax),%eip
```

很明显，就是把 eip 的内容变成了eax 中的内容，如果用 jmp 也是一样的

```
ljmp   *(%edx)
```

上面的两种转移方式适应于不同的环境要求，如果是在一个ELF文件中的，采用相对转移可带来的好处有以下的几点：

1、 可以不用再访问一次内存，在指令的执行时间上得到了大大的提高（这在PCI的总线结构中现在主流的最高主频是133MHZ，而随便一个INTEL CPU的主频都能超过它）。

2、 可以适应在动态加载与动态定位的内存环境，而不用再对原来的代码修改便能实现（代码段也不能在运行的时候修改），因为整个动态链接库或可执行文件都是以连续的地址映射的。

但同样带来了几个问题：

1、 这样的相对转移没有办法在运行的时候准确的转移到别的动态链接库中的函数地址（因为虽然大部分的动态链接库的加载地址是可以预计的，但从理论上来说是随机的）。

2、 这样的代码在平台之间的移植性带来很大的问题，因为不同的机器没有办法知道这样的数字是代表一个地址，还是代表了一个二进制数。所以在对平台移植有高要求的体系中用的是c++的虚函数指针------相对地址转移的发展。如COM，corba体系中就是这样的。

上面的这两项缺点正好是绝对转移的优势。作一个对比，绝对转移就相当于内存寻址时的立即寻址，而相对转移相当于内存寻址的相对寻址。

在一般的动态链接库中实际运用更是用了一个聪明的办法。请看下一段的汇编语言片段：

```
2f7:  e8 00 00 00 00         call   2fc <ok+0xc>
2fc:  5b                     pop    %ebx
2fd:  81 c3 b0 10 00 00      add    $0x10b0,%ebx
```

这里的2f7中的call 2fc <ok+0xc>是什么意思呢，从我们上面的方法来看，这里是什么呢？就是把函数运行到了2fc处，根据是我上面所说的，因为是一个相对转移。e8 00 00 00 00。如果用一般的观点看这没有什么用处。但妙处就在这里，2fc处的pop %ebx，是把什么送到%ebx中呢，如果每一次call 都会把下一条要执行的指令的地址压入栈中，那%ebx中在这里的内容就是2d4这一条指令在内存中的地址了，回想动态链接库的绝对地址是没有办法在编译时得到，但这样却可以--------很巧妙，不对吗？

那后面的add $0x10b0,%ebx又是什么用处？如果我们这里假定在内存中的地址是2fc，那加上10b0之后的值是0x13ac了，看在这里是什么呢？

```
Disassembly of section .got:
000013ac <.got>:
    13ac:  34 13                 xor    $0x13,%al
  ...
```

这是一个got节， 它的全称是global object table 就是全局对象表。它这里存储着要转移的地址。如果在动态链接库中，或是要调用一个在它之外的函数是怎样实现呢？我们往下看：

```
306:  8d 83 74 ef ff ff      lea    0xffffef74(%ebx),%eax
30c:  50                     push   %eax
30d:  e8 ce ff ff ff         call   2e0 <ok-0x10>
```

这里就要调用一个call 2e0 <ok-0x10>所在的函数。那在0x2e0处又是什么呢?

```
2e0:  ff a3 0c 00 00 00      jmp    *0xc(%ebx)
2e6:  68 00 00 00 00         push   $0x0
2eb:  e9 e0 ff ff ff         jmp    2d0 <ok-0x20>
```

很明显，我们前面已经说了%ebx中所保存的就是.got节的起始地址，而这里就是转移到在.got起始地址偏移0xc处所存储的地址量。而0x2e0所在的地址是在.plt（procedure linkage table）的节中。正是plt got的互相配合，才达到了动态链接的效果。下面的_dl_relocate_object函数就是在把动态链接库加载之后将got中的内容初始化的作用，作好了以后函数解析的准备。

三、_dl_relocate_object函数分析

举个例子。同样来自上面的动态链接库文件中内容。如果我们在这里面调用了printf这个普通的函数，它的rel在文件中的位置是

```
Relocation section '.rel.plt' at offset 0x2c8 contains 1 entries:
Offset     Info    Type            Symbol's Value  Symbol's Name
000013b8  00000e07 R_386_JUMP_SLOT       00000000  printf
```

这个值如果在文件中找到0x13b8（这是相对偏移量）的内容就是

```
13b8:  e6 02
```

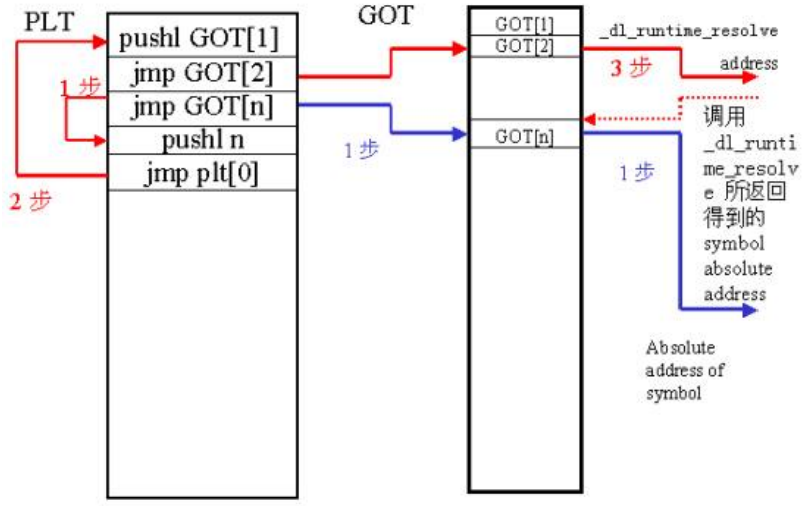
由于intel 是little endian 所以这个数翻译过来是0x02e6，那这里是什么呢？

```
2e0:  ff a3 0c 00 00 00      jmp    *0xc(%ebx)
2e6:  68 00 00 00 00         push   $0x0
2eb:  e9 e0 ff ff ff         jmp    2d0 <ok-0x20>
```

这下就会全部明白了吧。它就是压入0x0（这其实就是我们前面的printf在rel节中的索引数0------它是第一项）。而下面跳到的就是2d0（这是一个相对转移）处

```
2d0:  ff b3 04 00 00 00      pushl  0x4(%ebx)
2d6:  ff a3 08 00 00 00      jmp    *0x8(%ebx)
```

前面已经说过%ebx得到的是got的起始地址，所以这就是压got[1]入栈，再转移到got[2]中所包含的地址去，你可以看前面在elf_machine_runtime_setup中的2162行与2167行，它就是这个动态链接库自身的struct link_map*的指针，与_dl_runtime_resolve所在的地址。下面一张图就可以形象的说明这一点。



如果是第一次的函数调用，它所走的路线就是我在上图中用红线标出的，而要是在第二次以后调用，那就是蓝线所标明的。原因在前面的代码中已经给出了。

```
82  int _dl_relocate_object (struct link_map* lmap,int lazy_mode)
83  {
84     elf_machine_runtime_setup(lmap,lazy_mode);
85    elf_machine_lazy_rel (lmap,lazy_mode);
```

```
86
87 }
```

这里要分两步来完成，第一步的**elf_machine_runtime_setup**是把这个动态链接库所代表的数据结构**lmap**的地址写入一个在**ELF**文件中特别地方，而**elf_machine_lazy_rel**是对所有的要被调用的动态链接库外部的函数重定位的实现。这两步非常重要，因为如果没有这两步，那要实现动态链接库的函数动态解析是不可能的，这个你可以在上面的 相对转移，绝对转移 中的论述得到详细的了解。

```
54 void elf_machine_runtime_setup(struct link_map* lmap,int lazy_mode)
55 {
56    Elf32_Addr *got;
57
58     got = (Elf32_Addr *) lmap->l_info[DT_PLTGOT].d_un.d_ptr;
59
60 got[2]=&_dl_runtime_resolve
61     got[1]=lmap;
62 }
```

明显的，那个被写入的**ELF**文件中的地址就是它的**DT_PLTGOT**节中的第二个项目**-----**第**60**行的内容。而写入第一项的内容就是要调动的处理函数的地址，这一点在后面所提到的动态解析中的入口地址。

```
64 void elf_machine_lazy_rel (struct link_map* lmap,int lazy_mode)
65 {
66    Elf32_Addr rel_addr=lmap->l_info[DT_REL].d_un.d_ptr;
67    int rel_num=lmap->l_info[DT_RELSZ].d_un.d_ptr;
68    int i;
69    Elf32_Addr l_addr=lmap->l_addr;
70
71   Elf32_Rel*  rel;
72   for (i=0,rel=(Elf32_Rel*)rel_addr;i<rel_num;i++,rel++)
73     {
74       Elf32_Addr *const reloc_addr = (void *) (l_addr + rel->r_offset);
75       *reloc_addr +=l_addr;
76     }
77
78 }
```

这里的**elf_machine_lazy_rel**我只列出了在**intel**平台下的那种情况，其它的还要特别的内容，在这里很明显，我们只是写把原来的在**ELF**文件的内容加上一个文件加载的地址，这就是**lazy mode**，因为动态链接库的函数很可能在整个程序运行中不会被调用**--------**这一点与虚拟内存管理的原理是一样的。

四、动态链接库函数的解析

前面的60行的代码----设定了动态解析的入口地址与给出的在动态链接库中的在达到调用一个外部函数时所有的函数路线，已经到了_dl_runtime_resolve处

```
2087 #  define ELF_MACHINE_RUNTIME_TRAMPOLINE asm ("\\
2088   .text\\n\\
2089   .globl _dl_runtime_resolve\\n\\
2090   .type _dl_runtime_resolve, @function\\n\\
2091   .align 16\\n\\
2092 _dl_runtime_resolve:\\n\\
2093   pushl %eax    \\n\\
2094   pushl %ecx\\n\\
2095   pushl %edx\\n\\
2096   movl 16(%esp), %edx  \\n\\
2097   movl 12(%esp), %eax  \\n\\
2098   call fixup    \\n\\
2099   popl %edx    \\n\\
2100   popl %ecx\\n\\
2101   xchgl %eax, (%esp)  \\n\\
2102   ret $8      \\
2103   .size _dl_runtime_resolve, .-_dl_runtime_resolve\\n\\
2104 \\n\\
2105   ");
```

从这里定义的名称ELF_MACHINE_RUNTIME_TRAMPOLINE，我们就可以看出这个函数不简单（TRAMPOLINE在英语中是蹦床的意思，就是要make your brain curving的那种怪怪的东西），后面的代码也确实说明了这一点。

在前面的.text是下面的代码是可执行，.globl _dl_runtime_resolve是表明这个函数是全局性的，如果没有这一项，那我们前面看的got[2]=&_dl_runtime_resolve就不能编译通过-----编译器可能找不到它的定义。.type _dl_runtime_resolve, @function是函数说明。 .align 16处便是16字节对齐。

我们知道在前面的调用函数过程中已经压入了两个参数（第一个是动态链接库的struct link_map* 指针，另一个是rel的索引值）这里先保存以前的寄存器值，而到这个时候16(%esp)就是第二个参数，12(%esp)第一个参数，这里作的原因是下面的fixup的函数以寄存器传递参数。

我先不管fixup具体内容是什么，单就看它结束的内容就很能说明代码作者的优秀。先pop两个寄存器的值，而又xchg %eax,(%esp)与栈顶的内容，这有两个目的，一是恢复了eax的值，另一个作用是栈顶是函数返回的地址，而fixup返回的eax就是我们想找的函数有内存中的地址。这就自然跳到那个地方去了。但如果你认为这就好了，那也错了，因为你不要忘记我们之前还压入了两个参数在栈中。所以用了ret $8，这在intel的指令中表示

```
popl %eip
add $0x8,%esp
```

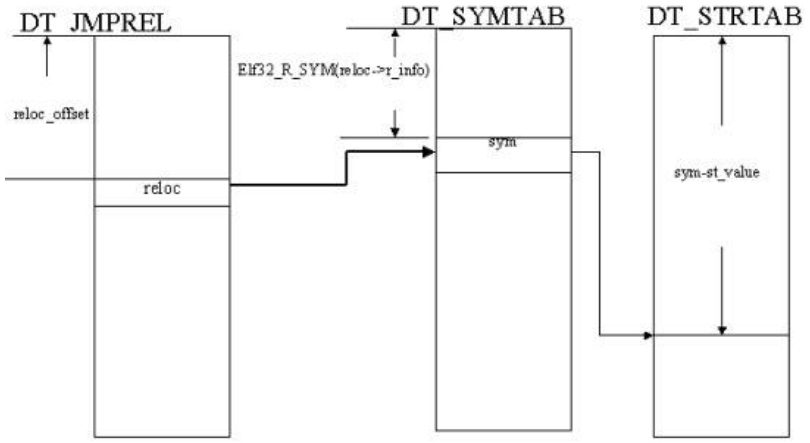
的组合。（很精彩！！！！！！！）

你还可以参看《程序的链接和装入及Linux下动态链接的实现》 网址为 http://www.ibm.com/developerworks/cn/linux/l-dynlink/index.shtml里面的有一幅图正好说明此的ELF_MACHINE_RUNTIME_TRAMPOLINE。

那直接看fixup函数的内容

```
124  Elf32_Addr  fixup(struct link_map* lmap,Elf32_Word reloc_offset)
125  {
126    Elf32_Sym* symtab=lmap->l_info[DT_SYMTAB].d_un.d_ptr;
127    char* strtab=lmap->l_info[DT_STRTAB].d_un.d_ptr;
128    Elf32_Rel* reloc = (Elf32_Rel*) (lmap->l_info[DT_JMPREL].d_un.d_ptr+reloc_offset);
129    Elf32_Sym* sym=&symtab[Elf32_R_SYM(reloc->r_info)];
130    char* symname=sym->st_name+strtab;
131    Elf32_Addr reloc_addr=lmap->l_addr+reloc->r_offset;
132
133    Elf32_Addr  symaddr=0;
134
135
136
137    symaddr=do_lookup(lmap,symname);
138
139
140
141    if (symaddr>0)
142      {
143        *reloc_addr=symaddr;
144        return symaddr;
145        }
146
147     exit(-2);
148
149
150
151
152
153  }
```

这里是给出了从一个动态链接库中可重定向的reloc_offset得到要解析函数的名称，如果用图示的方式表示就如下图：



你可能会想：其实还可以用另一种方法，就是把这个**reloc sym**的**st_value**直接写入前面的这个调用重定向函数相对应的**got**中。这样解析时的速度会更快。但现实这样却可能对整个**ELF**文件结构体系带来很大的麻烦。我将对每一点说明：

1. 如果是这个**reloc sym**的地址，那对于一个动态链接库而言，它的加载地址本身就是动态确定的。
2. 如果用的是那个**Elf32_Sym**的**st_value**地址，那倒是可以与**lmap->l_i nfo[DT_STRTAB]**一起得到这个**sym**的**name**，但如果考虑到在编译的时候有些函数是只对本模块有效，可见的，如在一个文件中定义为**static**的函数，则它就是局部可见的，那个时候就不可能是解析为这个函数，而且对**c++**函数还有更为复杂的情况，这样就会要求一个字段来表示它的属性，这就是要有了**st_info**这个数据成员变量。这也就要有了**sym**的参与了。
3. 光有**Elf32_Sym**还是不行，因为就重定位而言它本身还有一点信息，就是这一个**relocation symbol**是在本地解析，还是在另外一个真正意义上的动态链接库内被解析，这一情况主要是发生在几个文件编写的模块中，它们编写的一些函数就在链接的时候被确定了，而另一些则没有，区分的就是**relocation** 中的**r_info**了。

从上面的分析来看，一种规范的设计有许多的考虑因素，如果只单一的考虑，那是不行的，特别是要对多个操作系统与平台统一的规范，不能因为就是考虑效率一条就可以了。

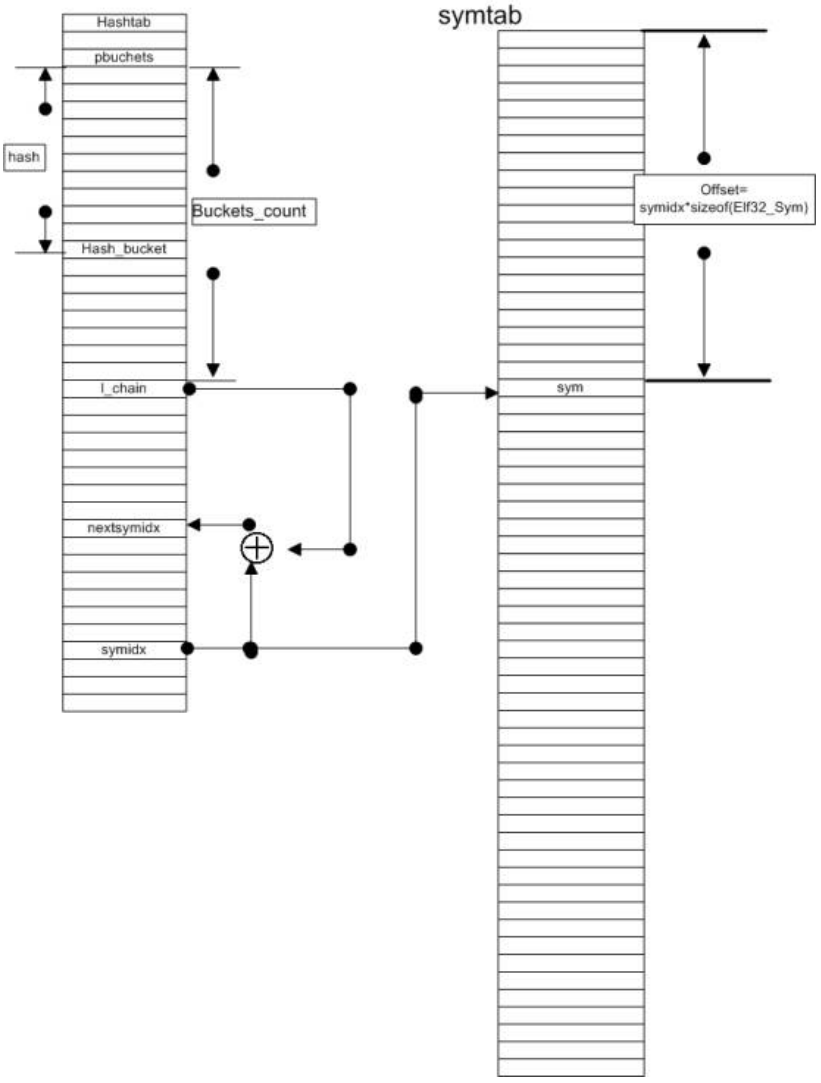
在**143**行是对前面要重定位的函数实现真正的解析函数到位，这样在这个函数被再次调用的时候就不用再来一次了，本来这时就对这个**relocation symbol r_info**的判断，现在都已经略去了。

真正的解析在**do_lookup**中实现了，我这里还是它的实现伪代码:

```
 90  Elf32_Addr  do_lookup(struct link_map* lmap,char* symname)
 91  {
 92    struct link_map* search_lmap=NULL;
 93    Elf32_Sym* symtab;
 94    Elf32_Sym* sym;
 95    char* strtab;
 96    char* find_name;
 97    int symindx;
 98
 99    Elf32_Word hash=elf_hash_name(symname);
100    for_each_search_lmap_in_search_list(lmap,search_lmap)
101      {
102        symtab=search_lmap->l_info[DT_SYMTAB].d_un.d_ptr;
103        strtab=search_lmap->l_info[DT_STRTAB].d_un.d_ptr;
104             for (symindx=search_lmap->l_buckets[hash % search_lmap->l_nbuckets];
105             symindx!=0;symindx=search_lmap->l_chain[symindx])
106               {
107                 sym=&symtab[symindx];
108
109          find_name=strtab+sym->st_name;
110                 if (strcmp(find_name,symname)==0)
111                   return sym->st_value+search_lmap->l_addr;
112               }
```

```
113
114     return 0;
115
116
117
118      }
119  }
```

100行**for_each_search_lmap_in_search_list**就是从前面在**_dl_map_object_deps**中得到的**l_searchlist**中取下的它本身的依赖动态链接库，中间查找的方法就如下面那张图中所显示的。

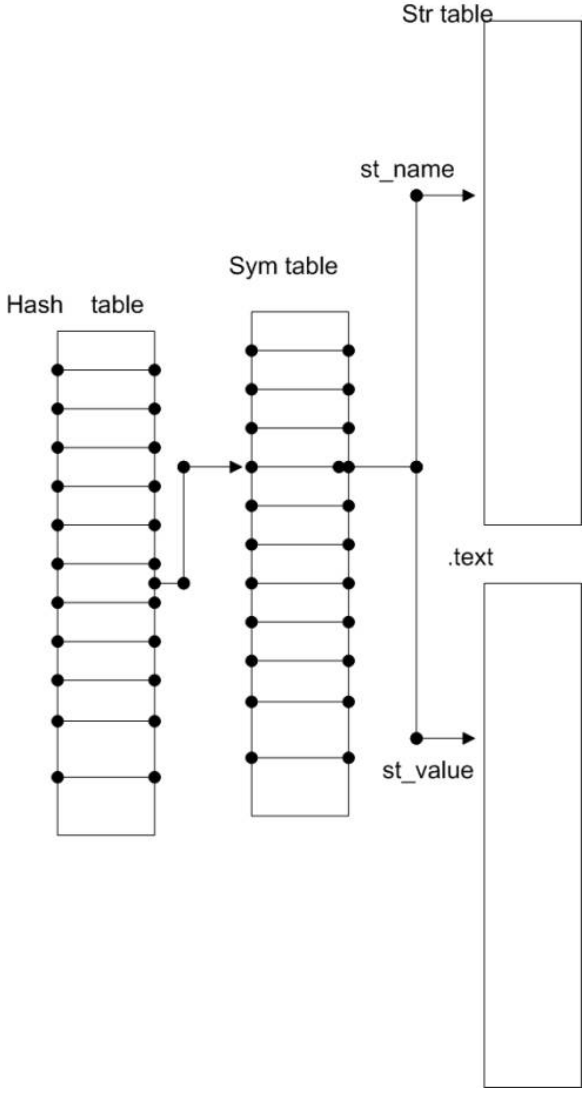


上面所表示的就是一个在**hash**表中**symidx**偏移处所存的就是下一个偏移所在。最后如果**strcmp==0**就可以得到了，否则就会返回一个**0**表示失败了。

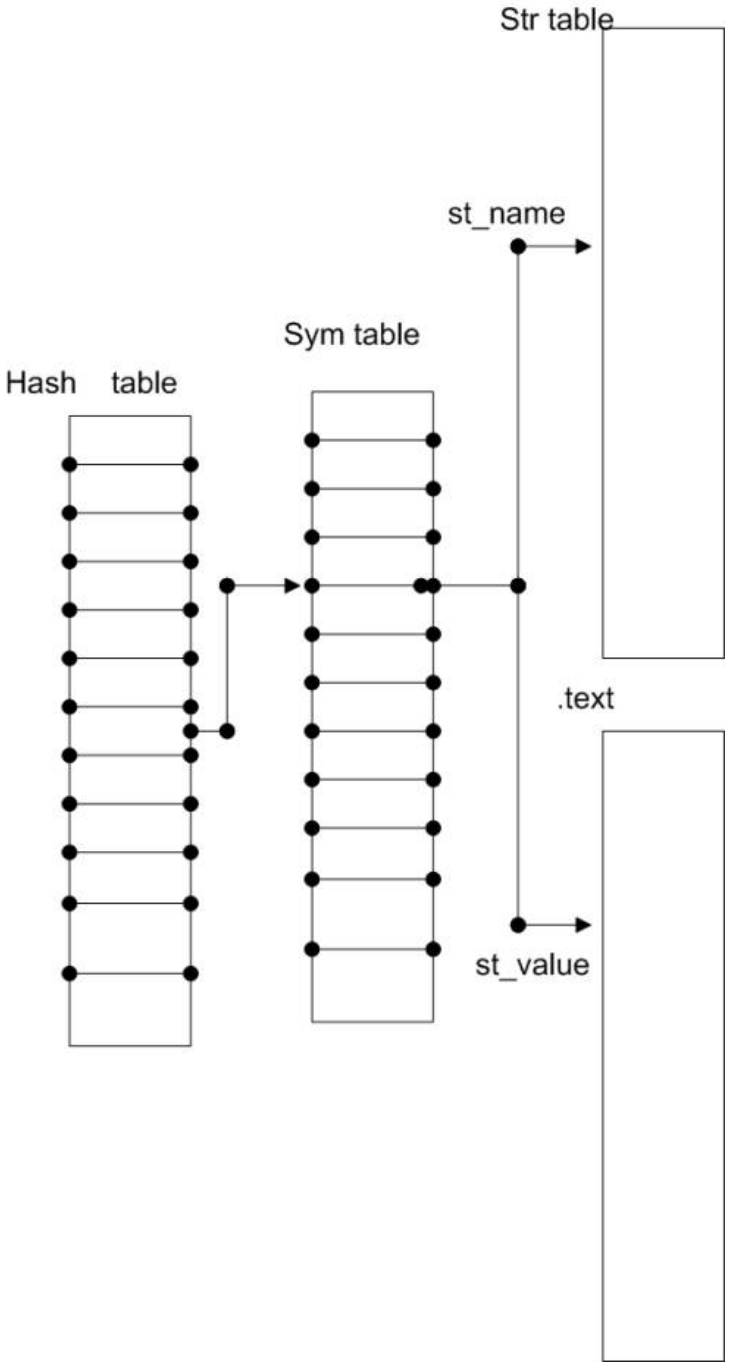
现在我们已经把函数的解析过程分析完毕，有必要作一个小结工作：

1. 在调用函数的动态链接库中，它所用的方法是从**plt**节的代码执行绝对转移，而转移的地址存放在**got**节中。
2. 在被调用函数的动态链接库中（就是函数实现的动态链接库），它的函数在以**DT_HASH**与**DT_SYMTAB,DT_STRTAB**组织起来。组织的方式如下面的一张图，以**symtab**中的**Elf32_Sym**中的**st_value**表示这个可导出的标记在动态链接库中的偏移量，**st_name**则是在动态链接库**strtab**中的偏移量。
3. 在调用动态链接库与被调用动态链接库的联系能过的是**Elf32_Rel**（对**MIPS**等的体系结构中是**Elf32_Rela**），它的**r_info**体现了这个要导入标记（就是调用方中）的性质，而

r_offset则是这个标记在动态链接库中的偏移量。（这个可以看**elf_machine_lazy_rel**中的实现）



点击查看大图



五、动态链接库的卸载

实际上卸载与加载只是反过程而已，但原来的代码为了提高效率实现在栈内分配内存，不过这样倒使原来简单易懂的变的过于复杂，所以，我这里作了很大的修改，这里是伪代码的实现。

```
245  void  dl_close(struct link_map* lmap)
246  {
247    struct link_map** dep_lmaplist=NULL;
248    int i;
249    Elf32_Addr*  fini_call_array;
250    void*  fini_call;
251    struct link_map* curlmap;
252    struct list * has_removed_list=malloc(sizeof(struct list));
253
254    has_removed_list->lmap=lmap;
255    has_removed_list->next=NULL;
256
257    if (lmap->l_opencount>1)
258      {
259       lmap->l_opencount--;
260      return;
261      }
262
263    lmap->l_opencount--;
264
265
266    dep_lmaplist=lmap->l_initfini;
267
268
269
```

```
270   for (i=0;dep_lmaplist[i]!=NULL;i++)
271     {
272
273         try_dl_close(dep_lmaplist[i],has_removed_list);
274     }
275
276
277   if (lmap->l_info[DT_FINI_ARRAY].d_un.d_ptr!=NULL  && lmap->l_opencount ==0)
278       {
279
280       fini_call_array=lmap->l_info[DT_FINI_ARRAY].d_un.d_ptr
281         +lmap->l_addr;
282       unsigned int sz=lmap->l_info[DT_FINI_ARRAYSZ].d_un.d_ptr
283         +lmap->l_addr;
284
285       while(sz-->0)
286         {
287            /*call the fini function*/
288            ((void*)fini_call_array[sz])();
289
290         }
291       }
292
293   if (lmap->l_info[DT_FINI].d_un.d_ptr!=NULL && lmap->l_opencount ==0)
294       {
295       fini_call=lmap->l_info[DT_FINI].d_un.d_ptr
296         +lmap->l_addr;
297
298       ((void*)fini_call)();
299       }
300
301
302   munmap(lmap->l_map_start,lmap->l_map_end-lmap->l_map_start);
303
304
305      free (lmap->l_initfini);
306
307         free (lmap->l_scope);
308
309    if (lmap->l_phdr_allocated)
310       free ((void *) lmap->l_phdr);
311
312    free_list(has_removed_list);
313
314     free (lmap);
315
316
317   return;
318 }
```

这里的has_removed_list就是记录整个在这一次dl_close操作中已经被卸载了的动态链接库，主要是为了防止再次卸载已经卸载的动态链接库。其实先开始判断这是否是已经没有再依赖它本向的动态链接库了。如果没有了（减去1，等于0就是了），那才可以继续去了，接下来不要先把它自己加入这个动态链接库，试着去卸载它所依赖的动态链接库，这些全做完之后就是它本身的各要点，一是它的DT_FINI_ARRAY中的卸载函数，还有就是DT_FINI中的函数，这之完了，便是加载到内存内容的去映射化，213行。再就是对struct link_map申请的内存就是了。

你可以看try_dl_close之后的代码就能明白这种可能有的深度的递归过程。

```
233 void try_dl_close(struct link_map* lmap,struct list*
234 has_removed_lmap_list)
234 {
235    if(in_the_list(has_removed_lmap_list,lmap))
236      return ;
237    dl_close_with_list(lmap,has_removed_lmap_list);
238    return ;
239
240 }
156 void   dl_close_with_list(struct link_map* lmap,
                struct list* has_removed_lmap_list)
157 {
158   struct link_map** dep_lmaplist=NULL;
159   int i;
160   Elf32_Addr*  fini_call_array;
161   void*  fini_call;
162
163
164
165
166
167   if (lmap->l_opencount>1)
168     {
169      lmap->l_opencount--;
170     return;
171     }
172   add_to_list_tail_uniq(has_removed_lmap_list,lmap);
173
174   lmap->l_opencount--;
175
176
177   dep_lmaplist=lmap->l_initfini;
178
179
```

```
180
181    for (i=0;dep_lmaplist[i]!=NULL;i++)
182      {
183
184          try_dl_close(dep_lmaplist[i],has_removed_lmap_list);
185      }
186
187
188    if (lmap->l_info[DT_FINI_ARRAY].d_un.d_ptr!=NULL  &&
                            lmap->l_opencount ==0)
189        {
190
191        fini_call_array=lmap->l_info[DT_FINI_ARRAY].d_un.d_ptr
192          +lmap->l_addr;
193        unsigned int sz=lmap->l_info[DT_FINI_ARRAYSZ].d_un.d_ptr
194          +lmap->l_addr;
195
196        while(sz-->0)
197          {
198             /*call the fini function*/
199             ((void*)fini_call_array[sz])();
200
201          }
202        }
203
204    if (lmap->l_info[DT_FINI].d_un.d_ptr!=NULL && lmap->l_opencount ==0)
205        {
206        fini_call=lmap->l_info[DT_FINI].d_un.d_ptr
207          +lmap->l_addr;
208
209        ((void*)fini_call)();
210        }
211
212
213    munmap(lmap->l_map_start,lmap->l_map_end-lmap->l_map_start);
214
215
216       free (lmap->l_initfini);
217
218          free (lmap->l_scope);
219
220     if (lmap->l_phdr_allocated)
221        free ((void *) lmap->l_phdr);
222
223      free (lmap);
224
225
226    return;
227
228  }
```

综合来看，dl_close这个函数如果是最终要卸载整个可执行文件的工作的话，那就要最高层的可执行文件开始，这里采用对可能有错综复杂的依赖关系的动态链接库使用了一个mark_removed与dl_close相结合的方法，在不断的递归调用中，把所有的动态链接库l_opencount减少到0。最后释放所有的内存空间。这种情况如果你与linux内核中delet_module的调用相对比，也可以看的更清楚。

六、前景与展望

动态链接库的实现发展到现今已经相当完善，它在理论与实践方面对于我们学习操作系统和编译语言提供了一个很好的范例。但是，动态链接库的实现毕竟还是只能在一个操作系统，一个单机，一种编程语言（如果是c++编程语言，则这一点也满足不了，因为不同的编译器可能对function name mangling-----函数名称混译也不同），对于现在网络化的信息产业是不够的。所以，出现了以这个为目标的二进制实现规范，这就是OMG（object model group ）所制定出来的 CORBA，和由 Microsoft 所制定出来的 COM，我可能以后的日子中详细来探讨这些最新发展。

参考资料

glibc-2.3.2 sourcecode 这是我这里主要的代码来源，可以在 ftp://ftp.gnu.org中下载

John R.Levine "Linkers and Loaders" 介绍动态链接库技术的经典 http://linker.iecc.com/

Hongjiu Lu "ELF: From The Programmer's Perspective" 好的ELF编程的参考。在 http://linux4u.jinr.ru/usoft/WWW/www_debian.org/Documentation/elf/elf.html可以看到