Hello how to link to the world

本文以 Holloworld 为例，介绍了静态链接动态链接，以及链接过程的符号解析，重定位等处理，最后附录中以 LIBC 部分代码实现，再次进一步阐述上述主题。

Link

链接就是将不同部分的代码和数据收集和组合成为一个单一文件的过程，这个文件可被加载到存储器并执行。链接可以执行于编译时，也就是在源代码被翻译成机器代码时；也可以执行加载时，也就是在程序被加载器加载到存储器并执行时；甚至可以执行于运行时，由应用程序来执行。

为了创建可执行文件，链接器必须完成两个主要任务：

- 符号解析。符号解析的目的是将每个符号引用和一个符号定义联系起来。
- 重定位。编译器和汇编器生成从地址零开始的代码和数据。链接器通过把每个符号定义与一个存储器位置联系起来，然后修改所有对这些符号的引用，使得它们指向这个存储器位置，从而重定位这些节。

链接器解析符号引用的方法是将每个引用与它输入的可重定位目标文件的符号表中的一个确定的符号定义起来。对那些和引用定义在相同模块中的本地符号的引用，符号解析是非常简单的。编译器只允许每个模块中的每个本地符号只有一个定义。编译器还确保静态本地变量，它们也会有本地链接器符号，拥有惟一的名字。不过对于僵局符号的引用解析就棘手得多。当编译器遇到一个不是在当前模块定义的符号时，它会假设该符号是在其他某个模块中定义的，生成一个链接器符号表目，并把它交给链接器处理。如果链接器在它的任何输入模块中都找不到这个被引用 的符号，它就输出一条错误信息并终止运行。此外对僵局符号解析的困难还在于相同的符号被多个目标文件定义。在这种情况下，链接器要么标志一个错误，要么以某种方法选出一个定义而抛弃其他定义。

在编译时，编译器输出每个僵局符号给汇编器，函数和已初始化的僵局变量是强符号，未初始化的僵局变量是弱符号。根据强弱符号定义，UNIX 链接器一般使用如下规则来处理多处定义的僵局符号。

规则 1：不允许有多个强符号。

规则 2：如果有一个强符号和多个弱符号，那么选择强符号。

规则 3：如果有多个弱符号，那么从这些弱符号中任意选择一个。

Symbol Resolution & Relocation

我们先以一个简单的例子来说明一下符号的解析

```
<onlyforos>[/home/onlyforos/hello]%cat f.c
int sum(int i, int j)
{
```

```
    return i+j;
}
<onlyforos>[/home/onlyforos/hello]%cat m.c
#include <stdio.h>
int i =1,j=2;

extern int sum(int i,int j);

int main()
{
    return    sum(i,j);
}
```

在编译代码时，对于重定位的目标文件的代码及数据的安排都是从地址 0 开始的。在链接时，链接器会在链接过程中，链接器将从一个指定的地址开始，根据输入的目标文件的顺序以段为单位将它们一个接一个的拼装起来。除了目标文件的拼装之外，在重定位的过程中还完成了两个任务：一是生成最终的符号表；二是对代码段中的某些位置进行修改，所有需要修改的位置都由编译器生成的重定位表指出。下面我们来看看是不是这样的：

```
<onlyforos>[/home/onlyforos/hello]%objdump –dtr f.o
f.o:        file format elf32-i386

SYMBOL TABLE:
00000000 l       df *ABS*     00000000 f.c
00000000 l       d    .text 00000000
00000000 l       d    .data00000000
00000000 l       d    .bss  00000000
00000000 l       d    .comment00000000
00000000 g         F .text 0000000b sum


Disassembly of section .text:

00000000 <sum>:
    0:    55                            push    %ebp
    1:    89 e5                         mov      %esp,%ebp
    3:    8b 45 0c                      mov      0xc(%ebp),%eax
    6:    03 45 08                      add      0x8(%ebp),%eax
    9:    c9                            leave
    a:    c3                            ret
```

可见 sum 函数从 0 地址开始安排，我们接着使用 objdump 来观察 m.o 在编译过程中生成的符号表和重定位表：

```
<onlyforos>[/home/onlyforos/hello]%objdump –dtr m.o
m.o:        file format elf32-i386

SYMBOL TABLE:
00000000 l       df *ABS*     00000000 m.c
00000000 l       d    .text 00000000
00000000 l       d    .data00000000
00000000 l       d    .bss  00000000
00000000 l       d    .comment00000000
00000000 g         O .data       00000004 i
00000004 g         O .data       00000004 j
00000000 g         F .text 00000029 main
00000000              *UND* 00000000 sum
```

```
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000015 R_386_32          j
0000001b R_386_32          i
00000020 R_386_PC32        sum

m.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:   29 c4                   sub    %eax,%esp
  10:   83 ec 08                sub    $0x8,%esp
  13:   ff 35 00 00 00 00       pushl  0x0
                15: R_386_32 j
  19:   ff 35 00 00 00 00       pushl  0x0
                1b: R_386_32 i
  1f:   e8 fc ff ff ff          call   20 <main+0x20>
                20: R_386_PC32 sum
  24:   83 c4 10                add    $0x10,%esp
  27:   c9                      leave
  28:   c3                      ret
```

首先，我们注意到符号表里面的 sum 被标记为 UND（undefined），也就是在 m.o 中没有定义，所以将来要通过 ld（Linux 下的链接器）的符号解析功能到别的模块中去查找是否存在函数 sum 的定义。另外，在重定位表中有三条记录，指出了在重定位过程中代码段中三处需要修改的位置，分别位于 15、1b 和 20。那么，在链接过程中如何修改代码段中的这几处位置的内容呢？

我们再看看链接后的可执行文件的文本段的内容，从中可以得知，链接器把 main 函数安排在了地址 080482f4 处，把 sum 函数安排在了 0x08048320 处。把初始化为 1 的全局变量 i 安排在数据段地址 0x8049380 处，把初始化为 2 的 j 安排在了数据段地址 0x8049384 处。重定位类型 R_386_32 指示着数据重定位为一个 32 位绝对地址，所以链接器重定位 i，j 时只把代码段 15 及 1b 处的内容改为上面的绝对地址：0x8049380，0x8049384。对于函数 sum，重定位类型 R_386_PC32 表明它是一个相对 PC 的 32 位地址，如何确定其在可执行文件中的重定位入口地址呢？

看 m.o 中，调用 sum 的 CALL 指令距起始零地址的偏移为 0x20。CALL 的指令码为 e8，偏移为 0xfffffffc(-4)。大家会觉得奇怪，此时距 sum 函数的偏移应该还不知道，为什么会是 -4 呢？这应该是一个小的技巧吧。主要是因为 PC 总指向其下一条指令地址。偏移为-4 就意味着 PC 总是指向其自身的地址。当然在不同的体系结构上，由于不同的指令大小此处的偏移也会用不同的值。链接器把 main 函数安排在了地址 080482f4 处，调用 sum 函数的 CALL 指令的地址为：0x8048313 ，指令中需要修改该指令中的偏移量距 main 函数地址为 0x20 偏移，即 0x8048314，此外链接器把函数 sum 安排在 0x08048320 处。那么如何修改 call 指令偏移使其能指向真正 sum 函数呢。

在重定位的目标文件中，CALL 指令中的偏移仅仅是其对起始地址的一个偏移，而只有在链接可执行文件时，该偏移值才会被改写成距真正该函数的偏移值。所以，真正的指令偏移应该是= 0x08048320-080482f4 -0x20-4=0x8。所以 CALL 指令中的偏移即为 8。也即把 CALL 指令的偏移从 m.o 中的 20 改为可执行文件中的 8。那么 sum 的地址即为：0x8048318+0x8=0x08048320 (0x8048318：PC 当前值，CALL 指令下一条指令地址。=0x8048313(CALL 指令地址)+5(指令码 1 个字节，需要修改的偏移 4 个字节))。正是链接器给 sum 函数分配的地址，即调用了 sum 函数。

```
<onlyforos>[/home/onlyforos/hello]%objdump –dj .text m
m:        file format elf32-i386

Disassembly of section .text:
… …

080482f4 <main>:
 80482f4:      55                          push    %ebp
 80482f5:      89 e5                       mov      %esp,%ebp
 80482f7:      83 ec 08                    sub     $0x8,%esp
 80482fa:      83 e4 f0                    and     $0xfffffff0,%esp
 80482fd:      b8 00 00 00 00              mov      $0x0,%eax
 8048302:      29 c4                       sub     %eax,%esp
 8048304:      83 ec 08                    sub     $0x8,%esp
 8048307:      ff 35 84 93 04 08           pushl   0x8049384
 804830d:      ff 35 80 93 04 08           pushl   0x8049380
 8048313:      e8 08 00 00 00              call    8048320 <sum>
 8048318:      83 c4 10                    add      $0x10,%esp
 804831b:      c9                          leave
 804831c:      c3                          ret
 804831d:      90                          nop
 804831e:      90                          nop
 804831f:      90                          nop

08048320 <sum>:
 8048320:      55                          push    %ebp
 8048321:      89 e5                       mov      %esp,%ebp
 8048323:      8b 45 0c                    mov      0xc(%ebp),%eax
 8048326:      03 45 08                    add     0x8(%ebp),%eax
 8048329:      c9                          leave
 804832a:      c3                          ret
 804832b:      90                          nop

… …
```

上面分析了链接器如何将重定位从 0 地址开始的若干符号，修改需要重定位的偏移值将其重定位到可执行文件相应的绝对地址上。

Libraries

为什么系统要支持库的概念呢？以 ANSI C 为例，它定义了一组广泛的标准 I/O，串操作和整数算术函数，例如 printf,scanf,aoti 等。它们在 libc.a 库中，对每个 C 程序来说都是可用的。

让我们来看看如果不使用表态库，编译器开发人员会使用什么方法和中用户提供这些函

数。一种方法是让编译器辨认出对标准函数的调用，并直接生成相应的代码。这种方法将给编译器啻显著的复杂性，而且每次添加、删除或修改一个标准函数时，就需要一个新的编译器版本。然而，对于应用程序员而方言，这种方法会是非常方便的，因为标准函数将总是可用的。

另一种方法是将所有的标准 C 函数都放在一个单独的可重定位目标模块中—比如说 libc.o 中，应用程序员可以把这个模块链接到他们的可执行文件中：

gcc hello.c /usr/lib/libc.o

这种方法的优点是它将编译器的实现与标准函数的实现分离开来，并且仍然对程序员保持适度的便利。然而，一个很大的缺点是系统中每个可执行文件现在都饮食着一份标准函数集合的完全拷贝，这是对磁盘窖是很大的浪费。更糟的是，每个正在运行的程序都将安自己的这些函数的放在存储器中，这又是极度浪费存储器的。另一个大的缺点就是，对任何的函数任何改变，无论大小，都要求库的开发人员重新编译整个源文件，这上一个非常的耗时的操作，使得标准函数的开发和维护变得很复杂。

也许我们可以为每个标准函数创建一个分离的可重定位文件，把它们存放在一个为大家所知的目录中来解决其中的一些问题。然而，这种方法要求应用程序员显式地链接合适的目标模块到它们的可执行文件中，这是一个容易出错而且耗时的过程：

gcc –o hello hello.c /usr/lib/printf.o /usr/lib/scanf.o …

Static Libraries

静态库的概念被提出来，以解决这些不同方法的缺点。相关的函数可以被编译为独立的目标模块，然后封闭成一个单独的静态库文件。然后，应用程序可以通过在命令行上指定单独的文件名字来使用这些在库中定义的函数：

gcc –o hello hello.c /usr/lib/libc.a

在链接时，链接器将只拷贝被程序引用的目标模块，这就减少了可执行文件在磁盘和存储器中的大小。另一方面，应用程序员只需要饮食较少的库文件的名字(实际上，C 编译器驱动程序总是 libc.a 给链接器，所以前面提到的对 libc.a 的引用是不必要的)。

静态库解决了应用程序如何使用大量的可用的相关函数，但其还是存在一些明显的缺点。静态库和所有软件一样，也需要定期维护和更新。如果应用程序员想要使用一个库的最新版本，他们必须以方式了解到譔库的更新情况，然后显式地将他们的程序与亲的库重新链接。

另一个问题就是几乎每个 C 程序都使用 I/O 函数，比如 printf。在运行时，这些函数的代码会被复制到每个运行进程的文本段中。在一个运行大量进程的系统上，这会是对存储器资源的极大浪费。

此外，在编译器命令中，各个静态链接库出现的顺序是非常重要的。 ** ，并且，如果在自己的代码之前引入静态库，又会带来另一个问题。因为此时尚未出现未定义的符号，所以它不会从库中提取任何符号，但当目标文件被链接时，它对函数库所有符号的引用都将是未实现的。对于不够仔细的人，这样显然会带来很多烦恼。(举例说明)

Dynamic Shared Libraries

如果函数库的一分拷贝是可执行文件的物理组成部分，那么我们称之为静态链接；如果可执行文件只是包含了文件名，让载入器在运行时能够寻找程序所需要的函数库，那么我们称之为动态链接。静态链接的模块被链接编辑并载入以便运行，动态链接的模块被链接编辑后载入，并在运行时进行链接以便运行。程序执行时，外部函数在 main()函数被调用前，运行时载入器并不解析它们 。所以即使链接了函数库，如果并没有实际调用，也不会带来额外开销。对比简单的 helloworld 分别与静态库 libc.a 及动态库 libc.so 链接后生成的可执行文件的大小分别为：464703 字节、11362 字节。

```
<onlyforos>[/home/onlyforos/hello]%gcc -o hs hello.c /usr/lib/libc.a
<onlyforos>[/home/onlyforos/hello]%ll hs
-rwxr-xr-x       1 onlyforos          onlyforos              464703    6 月  26 09:28 hs*
<onlyforos>[/home/onlyforos/hello]%ll hello
-rwxr-xr-x       1 onlyforos          onlyforos               11362    5 月  25 10:05 hello*
```

共享库就是致力于解决静态库缺陷的产物。共享库是一个目标模块，在运行时，可以加载到任意的存储陡，并在存储器中和一个程序链接起来。这个过程称为动态链接，是由一个动态链接器的程序来执行的。

应该认识到，在编译时没有任何共享库的代码和数据节被真的拷贝到可执行文件中。而是拷贝了一些重定位和符号表信息，它们使得运行时可以解析对共享库中代码和数据的引用。

当程序运行时，ELF 格式文件中有一个.interp 节，这个饮食动态链接器的路径名，动态链接器本身就是一个共享目标。加载器不再像它通常那样将控制传递给应用，取而代之的是加载和运行这个动态链接器。

在 IA32/Linux 系统中，共享库被加载到 0x40000000 开始的区域中。它重定位程序执行所需的共享库的文本段及数据段到另一个存储器段，并且重定位执行文件所有对由共享库中定义的符号的引用。这时，共享库及程序用的代码数据位置就真正固定了。最后动态链接器将控制传递给应用程序。

ELF 编译系统使用一种叫延迟绑定的技术，将过程地址的绑定推迟到第一次调用该过程时。第一次调用过程的运行时开销很大，但是其后的每次调用都只会花费一条指令和一个间接的存储器引用。

尽管单个可执行文件的启动速度稍爱影响，但动态链接可以从几个方面提高性能：

1. 动态链接可执行文件比功能相同的链接可扫执行文件的体积小。它能够节省磁盘空间及内存，因为函数库只有在需要时才被映射到进程中。以前避免把函数库的拷贝绑定到每个可执行文件的唯一方法就是将其置于内核中而不是函数库中，这就带来了可怕的“ 内核膨胀 ”。
2. 所有动态链接到某个特定函数库的可执行文件在运行时共享该函数库的一个单独拷贝。操作系统内核保证映射到内存中的函数训可以被所有使用它们的进程共享。这就提供了更好的 I/O 和交换空间利用率，不、节省了物理内存，从而提高了系统整体性能。如果可执行文件是静态链接的，每个文件都将拥有一份函数库的拷贝，显然极为浪费。
3. 动态链接使得函数库的升级更为容易。新的函数库可以随时发布，只要安装到系统中，旧的程序就能够自动获得新函数库的优点而无需重新链接。

4. 动态链接允许用户在运行时选择需要执行的函数库。这就使为了提高速度或提高内存使用羊角辫或包含额外的调试信息而创建新的版本的函数库是完全可能的，用户可以根据自己的喜好，在程序挂靠时用一个库文件取代另一个库文件。

Dynamic Linking with Shared Libraries

下面我们着重以 Helloworld 为例来说明一下 LINUX 下动态链接的实现。如今我们在 Linux 下编程用到的库（像 libc 等）大多都同时提供了动态链接库和静态链接库两个版本的库，而 gcc 在编译链接时如果不加-static 选项则默认使用系统中的动态链接库。

同样先看一下我们熟悉的 Helloworld 程序：

```
<onlyforos>[/home/onlyforos/hello]%cat hello.c
#include <stdio.h>
int main()
{
        printf("hello,world\n");
        return 0;
}

<onlyforos>[/home/onlyforos/hello]%gcc -o hello hello.c
<onlyforos>[/home/onlyforos/hello]%hello
hello,world
```

现在我们分析一下如何从动态库中加载运行 printf。

```
<onlyforos>[/home/onlyforos/hello]%gcc –o hello hello.c
<onlyforos>[/home/onlyforos/hello]%objdump –d hello
… …
08048248 <.plt>:
 8048248:      ff 35 98 94 04 08              pushl   0x8049498                (6)
 804824e:      ff 25 9c 94 04 08              jmp     *0x804949c               (7)
 8048254:      00 00                          add     %al,(%eax)
 8048256:      00 00                          add     %al,(%eax)
 8048258:      ff 25 a0 94 04 08              jmp     *0x80494a0
 804825e:      68 00 00 00 00                 push    $0x0
 8048263:      e9 e0 ff ff ff                 jmp     8048248 <_init+0x18>
 8048268:      ff 25 a4 94 04 08              jmp     *0x80494a4               (2)
 804826e:      68 08 00 00 00                 push    $0x8                     (4)
 8048273:      e9 d0 ff ff ff                 jmp     8048248 <_init+0x18>     (5)
… …
08048328 <main>:
 8048328:      55                             push    %ebp
 8048329:      89 e5                          mov     %esp,%ebp
 804832b:      83 ec 08                       sub     $0x8,%esp
 804832e:      83 e4 f0                       and     $0xfffffff0,%esp
 8048331:      b8 00 00 00 00                 mov     $0x0,%eax
 8048336:      29 c4                          sub     %eax,%esp
 8048338:      83 ec 0c                       sub     $0xc,%esp
 804833b:      68 98 83 04 08                 push    $0x8048398
 8048340:      e8 23 ff ff ff                 call    8048268 <_init+0x38>     (1)
 8048345:      83 c4 10                       add     $0x10,%esp
 8048348:      b8 00 00 00 00                 mov     $0x0,%eax
 804834d:      c9                             leave
 804834e:      c3                             ret
 804834f:      90                             nop
```

```
… …
```

首先，在 main 函数中调用 printf。如(1)所示，调用地址为 0x8048268。这是个什么地址？是 printf 吗？让我们接着找。在过程连接表 PLT 中找到了这个地址，内容是跳到 0x80494a4 地址的内容处，这个地址又是哪里的？看到地址为 80494XX，是数据段内容。我们用 readelf 看看这个地址在哪个.section 里：

```
<onlyforos>[/home/onlyforos/hello]%readelf –S hello
There are 34 section headers, starting at offset 0x20c8:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
… …
  [21] .got              PROGBITS        08049494 000494 000018 04  WA  0   0  4
  [22] .bss              NOBITS          080494ac 0004ac 000004 00  WA  0   0  4
… …
```

原来是全局偏移量表 GOT，看看其内容：

```
<onlyforos>[/home/onlyforos/hello]%hexdump –s 0x494 –n 24 -C hello
00000494  b8 93 04 08 00 00 00 00  00 00 00 00 5e 82 04 08  |............^...|   (8)
000004a4  6e 82 04 08 00 00 00 00                          |n.......|   (3)
```

看到了我们的目标 0x80494a4 处的内容为：0x804826e。哪儿？PLT 中。(4)。压栈一个数，然后跳到 0x8048248。(6)。压栈一个数，跳到 0x804949c 内容处。又是在 GOT 中。(8)。居然是 0!乱了，乱套了。调一个 printf 函数，最后跳来跳去跳成 0 了。怎么回事？

其实，到目前为止，我们的思路都是正确的。我们会在下面结合代码把这个思路再捋一遍。在这之前，我们先把理论的东东先讲一下。

在我们想象中，调用 printf，目的就是找到真正 printf 函数地址然后进行调用。但由于我们调用的 printf 函数在动态库/lib/libc.so.6 中，所以该函数的地址直到程序运行到该函数调用时才由动态链接器找到其真正地址，然后跳转过去执行。那么在这之前该函数的地址是什么含义呢？

所有重定位的函数，都将在 PLT 过程链接表中有自己的入口。所以，程序编译时调用 printf 的地址，就是其在 PLT 中的地址。但其实一个程序调用函数是引用它的 GOT(全局偏移量表)来使用位置无关的地址。这个意思就是说，函数真正的地址是保存 GOT 中的。ELF编译系统使用一种叫***lazy binding***(延迟绑定)的技术，将过程地址的绑定推迟到第一次调用该过程时。第一次调用过程的运行时开销很大，但是其后的每次调用都只会花费一条指令和一个间接的存储器引用。我们先说第二次及之后的调用。

第二次之后的函数调用，代码中 CALL 的地址仍是其 PLT 入口地址，该地址的内容就是跳转至相应的 GOT 表目中。而该 GOT 表中放的就是该函数的真实地址，所以执行该函数。这就是一条指令和一个间接引用。

为什么是第二次之后才这样呢？第一次是怎么处理的呢？第一次比较特殊。这就是我们是前面跟踪时看到的。(3)处，GOT 表中相应的位置根本不是 printf 的绝对地址，而是其在 PLT 表入口地址的下一条指令地址(4)。该指令压栈一个偏移然后跳转到 PLT［0］处(5)，PLT［0］处压栈一个地址(6)，然后跳转到 GOT［2］处执行(7)。GOT［2］处放置的是动态链接器查找 printf 函数地址代码的入口点(8)。既然是一个代码入口点，那就应该是一个地址，可为什么我们在(8)处看到是 0 呢。这是因为动态链接器是在程序运行时由 execve()加载到内存中的，此时可执行文件尚未运行所以其中的内容为 0。呆会在下面的代码跟踪中，我们会看到其的内容的。

接着此入口代码会根据上面压栈的 pirntf 在重定位中的偏移(4)及 GOT［1］这两个参数(6)找到 printf 在动态库中的地址，然后修改 GOT［4］也就是(3)处内容，把这个 printf 的真实地址保存在这里。这样，第二次调用 printf 时，走到(3)时能够直接调用 printf，而不用象第一次还得通过动态链接器查找。所以，调用动态链接库函数，第一次调用的代价要会远远大于以后的调用。GOT［1］中的内容是在装载动态库时准备好的 struct link_map*指针地址。稍后我们会提到。

这种处理就是所谓的延迟绑定。至此，动态函数便成功调用。下面把上面的内容图示一下，关于.GOT，.PLT，.REL.PLT，.DYNAMIC 等的内容更为详细的解释见《hello~, ELF of the world.doc》：

			.dynamic段的地址，这个段包含了动态链接器用来绑定过程地址的信息，比如符号表的位置和重定位信息；
0x8049494	GOT[0]	0x80493b8	
0x8049498	GOT[1]	0x400126e0	
0x804949c	GOT[2]	0x4000a180	
0x80494a0	GOT[3]	0x804825e	PLT[1]的下一条指令地址(__libc_start_main)
0x80494a4	GOT[4]	0x804826e	PLT[2]的下一条指令地址(printf)
0x80494a8	GOT[5]	0x00000000	

上图就是程序刚开始运行时 GOT 表的内容。可以看到其中 GOT[1],GOT[2]的内容已经由可执行文件中的 0 变为动态链接器中的地址。下面是过程链接表 PLT 内容，可知，每个需要重定位的函数在 PLT 都有一个入口：

```
08048248 <.plt>:
# PLT[0]:
8048248: ff 35 98 94 04 08          pushl 0x8049498    # push&GOT[1]( struct link_map*)
 804824e:     ff 25 9c 94 04 08          jmp *0x804949c    # *GOT[2]( _dl_runtime_resolve)
 8048254:      00 00                                        # padding
 8048256:      00 00                                        # padding
# PLT[1] <__libc_start_main>
8048258: ff 25 a0 94 04 08          jmp       *0x80494a0      # jmp to *GOT[3]
 804825e:      68 00 00 00 00             push      $0x0              # offset for .rel.plt
 8048263:      e9 e0 ff ff ff             jmp       8048248 <_init+0x18>    # jmp to PLT[0]
# PLT[2] <printf>
8048268: ff 25 a4 94 04 08          jmp       *0x80494a4      # jump to *GOT[4]
 804826e:      68 08 00 00 00             push      $0x8              # offset for .rel.plt
 8048273:      e9 d0 ff ff ff             jmp       8048248 <_init+0x18>    # jmp to PLT[0]
```

这样，我们就可以清楚地看到一个函数动态重定位的轨迹：首先 CALL 其 PLT 入口地址，跳转到 GOT 表中找其实际地址。若是第一次调用，因为动态链接器尚未查找该函数地址，所以此时此处的地址只简单地跳回 PLT 该入口的下一条指令继续执行，那就是将该函数在.rel.plt 中的偏移压栈(动态链接器查找实际需要根据该偏移在.rel.plt 中找到函数名字，然后查找该函数地址)，跳到 PLT[0]执行。PLT[0]把&GOT[1]压栈，跳到 GOT[2]执行，GOG[2]中内容就是动态链接器查找地址函数_dl_runtime_resolve 的入口地址。这时动态链接器就开始查找函数的真正地址。找到后跳到该地址处执行并修改相应 GOT 表地址把它保存下来，这样第二次调用该函数时就不用象第一次这么费劲了，直接就可以调用了。

第一次调用后 GOT 表的内容如下：

0x8049494	GOT[0]	0x80493b8	
0x8049498	GOT[1]	0x400126e0	struct link_map*指针地址
0x804949c	GOT[2]	0x4000a180	_dl_runtime_resolve地址
0x80494a0	GOT[3]	0x42015830	__libc_start_main的真实地址
0x80494a4	GOT[4]	0x42052390	Printf的真实地址
0x80494a8	GOT[5]	0x00000000	

下面我们使用 GDB 结合代码再看看(真够罗嗦滴)：

```
<onlyforos>[/home/onlyforos/hello]%gdb -q hello
(gdb) b main
Breakpoint 1 at 0x804832e
(gdb) x/6x 0x8049494
0x8049494<_G…O…T_>:     0x080493b8    0x00000000    0x00000000    0x0804825e
0x80494a4 <_GLOBAL_OFFSET_TABLE_+16>:     0x0804826e          0x00000000
```

在 main 函数处设了个断点，然后看看 GOT 表处的内容。可见，没有运行前 GOT[1],GOT[2]处内容均为 0。运行程序：

```
(gdb) r
Starting program: /home/onlyforos/hello/hello
Breakpoint 1, 0x0804832e in main ()
(gdb) x/6x 0x8049494
0x8049494 <_G_O_T_>:        0x080493b8     0x400126e0       0x4000a180        0x42015830
0x80494a4 <_GLOBAL_OFFSET_TABLE_+16>:     0x0804826e          0x00000000
```

运行程序停在 main 函数处。再看 GOT[1]GOT[2]已经有了内容。内容的含义下面再说。接着看看 main 函数内容：

```
(gdb) disass main
Dump of assembler code for function main:
0x8048328 <main>:           push     %ebp
0x8048329 <main+1>:         mov       %esp,%ebp
0x804832b <main+3>:         sub       $0x8,%esp
0x804832e <main+6>:         and       $0xfffffff0,%esp
0x8048331 <main+9>:         mov       $0x0,%eax
0x8048336 <main+14>:        sub       %eax,%esp
0x8048338 <main+16>:        sub       $0xc,%esp
0x804833b <main+19>:        push      $0x8048398
0x8048340 <main+24>:        call      0x8048268 <printf>
0x8048345 <main+29>:        add       $0x10,%esp
0x8048348 <main+32>:        mov       $0x0,%eax
0x804834d <main+37>:        leave
0x804834e <main+38>:        ret
0x804834f <main+39>:        nop
End of assembler dump.
```

看到其中调用了 printf。指令为 call 0x8048268。这个 0x8048268 是什么？

```
(gdb) disass 0x8048268
Dump of assembler code for function printf:
0x8048268 <printf>:         jmp       *0x80494a4
0x804826e <printf+6>:       push      $0x8
0x8048273 <printf+11>:      jmp       0x8048248 <_init+24>
```

其实从上面讲解，我们已经知道这个地址是.plt section 中的一个地址，是 plt[2]的内容。跳转到 0x80494a4 内容处。此时，应该已经有些眼熟了，这就是我们上面讲的 GOT 表的内容：

```
(gdb) x/x 0x80494a4
0x80494a4 <_GLOBAL_OFFSET_TABLE_+16>:     0x0804826e
```

0x80494a4 处内容为 0x0804826e，那么 0x0804826e 是什么？

```
(gdb) disass 0x804826e
Dump of assembler code for function _init:
0x8048230 <_init>:          push   %ebp
0x8048231 <_init+1>:        mov    %esp,%ebp
0x8048233 <_init+3>:        sub    $0x8,%esp
0x8048236 <_init+6>:        call   0x804829c <call_gmon_start>
0x804823b <_init+11>:       nop
0x804823c <_init+12>:       call   0x80482fc <frame_dummy>
0x8048241 <_init+17>:       call   0x8048350 <__do_global_ctors_aux>
0x8048246 <_init+22>:       leave
0x8048247 <_init+23>:       ret
0x8048248 <_init+24>:       pushl  0x8049498
0x804824e <_init+30>:       jmp    *0x804949c
0x8048254 <_init+36>:       add    %al,(%eax)
0x8048256 <_init+38>:       add    %al,(%eax)
0x8048258 <__libc_start_main>:    jmp    *0x80494a0
0x804825e <__libc_start_main+6>:  push   $0x0
0x8048263 <__libc_start_main+11>: jmp    0x8048248 <_init+24>
0x8048268 <printf>:         jmp    *0x80494a4
0x804826e <printf+6>:       push   $0x8
0x8048273 <printf+11>:      jmp    0x8048248 <_init+24>
End of assembler dump.
```

跳到 0x804826e 处即执行 push 指令然后跳到 08048248 处，也即 PLT[0]处执行。压栈后跳到 0x804949c 内容处执行。通过上面内存中 GOT 的内容我们可知，地址 0x8049498 的内容为 0x400126e0 ，0x804949c 处的内容为 0x4000a180。jmp *0x804949c 就意味着跳到 0x4000a180 处执行。

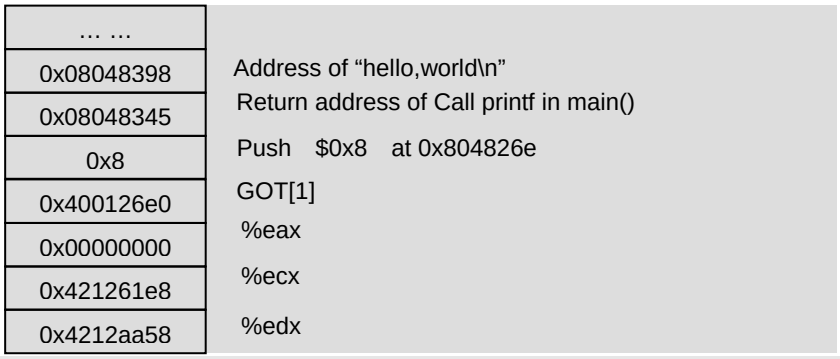
```
(gdb) disass 0x4000a180
Dump of assembler code for function _dl_runtime_resolve:
0x4000a180 <_dl_runtime_resolve>:          push   %eax
0x4000a181 <_dl_runtime_resolve+1>:        push   %ecx
0x4000a182 <_dl_runtime_resolve+2>:        push   %edx
0x4000a183 <_dl_runtime_resolve+3>:        mov    0x10(%esp,1),%edx
0x4000a187 <_dl_runtime_resolve+7>:        mov    0xc(%esp,1),%eax
0x4000a18b <_dl_runtime_resolve+11>:       call   0x40009f10 <fixup>
0x4000a190 <_dl_runtime_resolve+16>:       pop    %edx
0x4000a191 <_dl_runtime_resolve+17>:       pop    %ecx
0x4000a192 <_dl_runtime_resolve+18>:       xchg   %eax,(%esp,1)
0x4000a195 <_dl_runtime_resolve+21>:       ret    $0x8
0x4000a198 <_dl_runtime_resolve+24>:       nop
0x4000a199 <_dl_runtime_resolve+25>:       lea    0x0(%esi,1),%esi
End of assembler dump.
(gdb) b *0x4000a18b
Breakpoint 2 at 0x4000a18b
(gdb) b *0x4000a195
Breakpoint 3 at 0x4000a195
```

可见，0x4000a180 就是函数_dl_runtime_resolve，在后面我们会看到它的代码实现。我们会稍花些笔墨来解释这个函数，因为它是动态链接中的关键所在。首先我们看看其堆栈布局：

```
(gdb) c
Continuing.
(gdb) print $esp
$2 = (void *) 0xbffff7f8
(gdb) x/8x 0xbffff7f8
```

```
0xbffff7f8:    0x4212aa58    0x421261e8    0x00000000    0x400126e0
0xbffff808:    0x00000008    0x08048345    0x08048398    0x4212a2d0
```

很明显堆栈布局为：




```
(gdb) x/s 0x8048398
0x8048398 <_IO_stdin_used+4>:    "hello,world\n"
(gdb) x/6x 0x8049494
0x8049494 <_G_O_T_>:    0x080493b8    0x400126e0    0x4000a180    0x42015830
0x80494a4 <_GLOBAL_OFFSET_TABLE_+16>:    0x0804826e    0x00000000
```

可以看到 GOT[4]的内容仍是 0x0804826e，也就是说还不是 printf 的实际地址。也许有同学对 GOT[3]内容 0x42015830 感兴趣：

```
(gdb) disass 0x42015830
Dump of assembler code for function __libc_start_main:
0x42015830 <__libc_start_main>: push    %ebp
0x42015831 <__libc_start_main+1>:    mov    %esp,%ebp
… …
```

可见，GOT[3]放的是__libc_start_main 的实际地址。可参考.rel.plt section 的内容(见《》)。堆栈布置好后，开始调用 fixup 函数。这个函数的功能就是根据将要查找目标在重定位表中的偏移值及 struct link_map*地址到共享库中找到查找目标地址，其返回值就是这个查找目标的地址。函数原型在下面会提及，就是这个函数调用整整苦恼了 2 个小时。

首先 fixup 函数只有两个参数，即上面图示中的 0x8 及 0x400126e0，按说接下来就该调用 fixup 了，可不知为什么在调用之前又把三个寄存器值压入栈中，难道 fixup 有 5 个参数吗(我们知道，函数调用，形参后就应该是函数的返回地址，然后进入调用函数)？可实际上它又只有 2 个参数。这是怎么回事呢？苦恼了很久，又仔细看代码才发现只注意函数定义了却没注意函数是如此申明的：

```
static ElfW(Addr) fixup (struct link_map *__unbounded l, ElfW(Word) reloc_offset)
     __attribute__ ((regparm (2), unused));
static ElfW(Addr) profile_fixup (struct link_map *l, ElfW(Word) reloc_offset,
                                 ElfW(Addr) retaddr)
     __attribute__ ((regparm (3), unused));
```

__attribute__ ((regparm？，看 GCC 手册：

```
regparm (number)
On the Intel 386, the regparm attribute causes the compiler to pass up to number integer arguments in registers EAX, EDX, and ECX instead of on the stack. Functions that take a variable number of arguments will continue to be passed all of their arguments on the stack.
```

原来 fixup 函数已经申明为寄存器传参数，由%eax,%edx 来传送 2 个参数。这也就有了在 push 三个寄存器后的 mov 0x10(%esp,1),%edx；mov 0xc(%esp,1),%eax 两条指令。可同时又有了一个疑问：为什么传递参数只能用这三个寄存器，EBX，ESI，EDI 此时干嘛呢？ 不管怎么说，fixup 是调用了，在%EAX 返回了找到的 printf 的实际地址。接着把%EDX%ECX 的值出栈，然后的指令：

```
xchg    %eax,(%esp,1)
```

既把 printf 的实际地址入栈又恢复了%EAX 的值。那么此时堆栈布局又是如何呢？

```
(gdb) c
Continuing.
(gdb) print $esp
$3 = (void *) 0xbffff800
(gdb) x/5x 0xbffff800
0xbffff800:     0x42052390      0x400126e0    0x00000008    0x08048345
0xbffff810:     0x08048398
```

… …	
0x08048398	Address of “hello,world\n”
0x08048345	Return address of Call printf in main()
0x8	Push $0x8 at 0x804826e
0x400126e0	GOT[1]
0x42052390	Printf的实际地址

那么更为精妙的代码出现了：

```
ret     $0x8
```

我们知道，RET 指令用来从一个函数或过程返回，之前 CALL 保存的下条指令地址会从栈内弹出到 EIP 寄存器中，程序转到 CALL 之前下条指令处执行。那么该操作相当于：

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

当前栈顶内容为 0x42052390，即 printf 的实际地址。这样一来就把 printf 的地址放入到了 EIP 寄存器中，也即将会执行 printf 函数调用。ESP 指针上移 8 个字节，此时堆栈布局为：

… …	
0x08048398	Address of “hello,world\n”
0x08048345	Return address of Call printf in main()

呵呵，居然巧妙地布置好了 printf 的调用栈。程序也将跳到 printf 去执行。此时，我们再看看 GOT：

```
(gdb) x/6x 0x8049494
0x8049494 <_G_O_T_>:    0x080493b8      0x400126e0      0x4000a180      0x42015830
0x80494a4 <_GLOBAL_OFFSET_TABLE_+16>:     0x42052390          0x00000000
```

GOT[4]变为了 0x42052390，printf 的真实地址，fixup 函数完成了相应的修改：

```
(gdb) disass 0x42052390
Dump of assembler code for function printf:
0x42052390 <printf>:       push     %ebp
0x42052391 <printf+1>:     mov      %esp,%ebp
0x42052393 <printf+3>:     sub      $0x18,%esp
… …
```

Loading and Linking Shared Libraries from Applications

到此刻为止，我们已经讨论了静态链接及动态链接器加载和链接共享库的情景。然而，应用程序也许还想在它运行时要求动态链接器加载和链接任意共享库，而无需在编译时链接那些库到应用中。

在实践中，高性能的 WEB 服务器往往采用这种链接方式。早期的 WEB 服务器通过使用 fork 和 execve 创建一个子进程，并在该子进程的上下文中运行 CGI 程序，来生成动态内

容。但这种频繁地创建切换子进程的代价还是非常大的。现代高性能的 WEB 服务器基本使用基于动态链接的更有效的方法来生成动态内容，并将生成内容中的每个函数打包在共享库中。当一个来自 WEB 浏览器的请示到达时，服务器动态地加载和链接适当的函数，然后直接调用它。这样操作的代价仅仅是一个函数调用。并且无需停止服务器，就可以更新已存在的函数以及添加新的函数。

LINUX 及 SOLARIS 等 UNIX 系统，为动态链接器提供了一个简单的接口，允许应用程序运行加载和链接共享库。

```
#include <dlfcn.h>
extern void *dlopen (__const char *__file, int __mode) ;
extern int dlclose (void *__handle) ;
extern void *dlsym (void *__restrict __handle, __const char *__restrict __name) ;
extern char *dlerror (void) ;
```

不再对各个接口做介绍了，感兴趣可自行寻找相关资料学习。下面举一个应用小例，当然仍是以 Helloworld 为例：

```
<onlyforos>[/home/onlyforos/hello]%cat dl.c
#include <dlfcn.h>
#include <stdio.h>
main()
{
        void *libc;
        void (*printf_call)();
        char* error_text;

        if(libc=dlopen("/lib/libc.so.6",RTLD_LAZY))
        {
                printf_call=dlsym(libc,"printf");
                (*printf_call)("hello, world\n");
                dlclose(libc);return 0;
        }
        error_text= dlerror();
        printf(error_test);
        return -2;
}
<onlyforos>[/home/onlyforos/hello]%gcc –o dl dl.c –ldl
<onlyforos>[/home/onlyforos/hello]%hello
hello,world
```

Appendix

下面结合 GLIC 源码，进一步关注一下我们关心的几个问题：

A. GOT 中的入口何时准备好？

B. GOT 的入口又何时被改写为实际函数地址？

C. 动态查找的函数怎么找到的？

由于代码较多，我们只挑一部分来看，感兴趣可自行找相关代码阅读：

When & Where &What is the data in GOT[x]

准备 GOT 入口的代码在 glibc-2.3\sysdeps\i386\dl-machine.h 的 elf_machine_runtime_setup (struct link_map *l, int lazy, int profile)中。将 GOT[1]设置为准备好的 struct link_map *指针，将 GOT[2]设置为_dl_runtime_resolve。

函数调用链为：_dl_open -> dl_open_worker -> _dl_relocate_object-> ELF_DYNAMIC_RELOCATE -> elf_machine_runtime_setup:

glibc-2.3\elf\ dl-open.c:

```
void *
internal_function
_dl_open (const char *file, int mode, const void *caller)
{
  struct dl_open_args args;
  const char *objname;
  const char *errstring;
  int errcode;

  if ((mode & RTLD_BINDING_MASK) == 0)
    /* One of the flags must be set.  */
    _dl_signal_error (EINVAL, file, NULL, N_("invalid mode for dlopen()"));

  /* Make sure we are alone.  */
  __libc_lock_lock_recursive (GL(dl_load_lock));

  args.file = file;
  args.mode = mode;
  args.caller = caller;
  args.map = NULL;
  errcode = _dl_catch_error (&objname, &errstring, dl_open_worker, &args);
… …
}

static void
dl_open_worker (void *a)
{
  struct dl_open_args *args = a;
  const char *file = args->file;
  int mode = args->mode;
  struct link_map *new, *l;
  const char *dst;
  int lazy;
  unsigned int i;
#ifdef USE_TLS
  bool any_tls;
#endif

  /* Maybe we have to expand a DST.  */
  dst = strchr (file, '$');
  if (__builtin_expect (dst != NULL, 0))
    {
      const void *caller = args->caller;
      size_t len = strlen (file);
      size_t required;
      struct link_map *call_map;
```

```
      char *new_file;

      /* DSTs must not appear in SUID/SGID programs.  */
      if (__libc_enable_secure)
    /* This is an error.  */
    _dl_signal_error (0, "dlopen", NULL,
                      N_("DST not allowed in SUID/SGID programs"));

      /* We have to find out from which object the caller is calling.  */
      call_map = NULL;
      for (l = GL(dl_loaded); l; l = l->l_next)
    if (caller >= (const void *) l->l_map_start
        && caller < (const void *) l->l_map_end)
      {
        /* There must be exactly one DSO for the range of the virtual
           memory.  Otherwise something is really broken.  */
        call_map = l;
        break;
      }

      if (call_map == NULL)
    /* In this case we assume this is the main application.  */
    call_map = GL(dl_loaded);

      /* Determine how much space we need.  We have to allocate the
     memory locally.  */
      required = DL_DST_REQUIRED (call_map, file, len, _dl_dst_count (dst, 0));

      /* Get space for the new file name.  */
      new_file = (char *) alloca (required + 1);

      /* Generate the new file name.  */
      _dl_dst_substitute (call_map, file, new_file, 0);

      /* If the substitution failed don't try to load.  */
      if (*new_file == '\0')
    _dl_signal_error (0, "dlopen", NULL,
                      N_("empty dynamic string token substitution"));

      /* Now we have a new file name.  */
      file = new_file;
    }

/* Load the named object.  */
args->map = new = _dl_map_object (NULL, file, 0, lt_loaded, 0, mode);

/* If the pointer returned is NULL this means the RTLD_NOLOAD flag is
     set and the object is not already loaded.  */
if (new == NULL)
    {
      assert (mode & RTLD_NOLOAD);
      return;
    }

if (__builtin_expect (mode & __RTLD_SPROF, 0))
    /* This happens only if we load a DSO for 'sprof'.  */
    return;
```

```
  /* It was already open.    */
  if (new->l_searchlist.r_list != NULL)
    {
      /* Let the user know about the opencount.    */
      if (__builtin_expect (GL(dl_debug_mask) & DL_DEBUG_FILES, 0))
    _dl_debug_printf ("opening file=%s; opencount == %u\n\n",
                      new->l_name, new->l_opencount);

      /* If the user requested the object to be in the global namespace
       but it is not so far, add it now.    */
      if ((mode & RTLD_GLOBAL) && new->l_global == 0)
    (void) add_to_global (new);

      /* Increment just the reference counter of the object.    */
      ++new->l_opencount;

      return;
    }

  /* Load that object's dependencies.    */
  _dl_map_object_deps (new, NULL, 0, 0, mode & __RTLD_DLOPEN);

  /* So far, so good.    Now check the versions.    */
  for (i = 0; i < new->l_searchlist.r_nlist; ++i)
    if (new->l_searchlist.r_list[i]->l_versions == NULL)
      (void) _dl_check_map_versions (new->l_searchlist.r_list[i], 0, 0);

#ifdef SCOPE_DEBUG
  show_scope (new);
#endif

  /* Only do lazy relocation if `LD_BIND_NOW' is not set.    */
  lazy = (mode & RTLD_BINDING_MASK) == RTLD_LAZY && GL(dl_lazy);

  /* Relocate the objects loaded.    We do this in reverse order so that copy
       relocs of earlier objects overwrite the data written by later objects.    */

  l = new;
  while (l->l_next)
    l = l->l_next;
  while (1)
    {
      if (! l->l_relocated)
    {
#ifdef SHARED
        if (GL(dl_profile) != NULL)
          {
            /* If this here is the shared object which we want to profile
           make sure the profile is started.    We can find out whether
                this is necessary or not by observing the `_dl_profile_map'
                variable.    If was NULL but is not NULL afterwars we must
           start the profiling.    */
            struct link_map *old_profile_map = GL(dl_profile_map);

            _dl_relocate_object (l, l->l_scope, 1, 1);
… …
```

```
}

void
_dl_relocate_object (struct link_map *l, struct r_scope_elem *scope[],
                     int lazy, int consider_profiling)
{
  struct textrels
  {
    caddr_t start;
    size_t len;
    int prot;
    struct textrels *next;
  } *textrels = NULL;
  /* Initialize it to make the compiler happy.  */
  const char *errstring = NULL;

  if (l->l_relocated)
    return;

  /* If DT_BIND_NOW is set relocate all references in this object.  We
     do not do this if we are profiling, of course.  */
  if (!consider_profiling
      && __builtin_expect (l->l_info[DT_BIND_NOW] != NULL, 0))
    lazy = 0;

  if (__builtin_expect (GL(dl_debug_mask) & DL_DEBUG_RELOC, 0))
    INTUSE(_dl_debug_printf) ("\nrelocation processing: %s%s\n",
                              l->l_name[0] ? l->l_name : rtld_progname,
                              lazy ? " (lazy)" : "");

  /* DT_TEXTREL is now in level 2 and might phase out at some time.
     But we rewrite the DT_FLAGS entry to a DT_TEXTREL entry to make
     testing easier and therefore it will be available at all time.  */
  if (__builtin_expect (l->l_info[DT_TEXTREL] != NULL, 0))
    {
      /* Bletch.  We must make read-only segments writable
     long enough to relocate them.  */
      const ElfW(Phdr) *ph;
      for (ph = l->l_phdr; ph < &l->l_phdr[l->l_phnum]; ++ph)
    if (ph->p_type == PT_LOAD && (ph->p_flags & PF_W) == 0)
      {
        struct textrels *newp;

        newp = (struct textrels *) alloca (sizeof (*newp));
        newp->len = (((ph->p_vaddr + ph->p_memsz + GL(dl_pagesize) - 1)
                      & ~(GL(dl_pagesize) - 1))
                     - (ph->p_vaddr & ~(GL(dl_pagesize) - 1)));
        newp->start = ((ph->p_vaddr & ~(GL(dl_pagesize) - 1))
                       + (caddr_t) l->l_addr);

        if (__mprotect (newp->start, newp->len, PROT_READ|PROT_WRITE) < 0)
          {
        errstring = N_("cannot make segment writable for relocation");
          call_error:
        INTUSE(_dl_signal_error) (errno, l->l_name, NULL, errstring);
          }
```

```
#if (PF_R | PF_W | PF_X) == 7 && (PROT_READ | PROT_WRITE | PROT_EXEC) == 7
            newp->prot = (PF_TO_PROT
                    >> ((ph->p_flags & (PF_R | PF_W | PF_X)) * 4)) & 0xf;
#else
            newp->prot = 0;
            if (ph->p_flags & PF_R)
              newp->prot |= PROT_READ;
            if (ph->p_flags & PF_W)
              newp->prot |= PROT_WRITE;
            if (ph->p_flags & PF_X)
              newp->prot |= PROT_EXEC;
#endif
            newp->next = textrels;
            textrels = newp;
          }
        }

    {
      /* Do the actual relocation of the object's GOT and other data.  */

      /* String table object symbols.  */
      const char *strtab = (const void *) D_PTR (l, l_info[DT_STRTAB]);

      /* This macro is used as a callback from the ELF_DYNAMIC_RELOCATE code.  */
#define RESOLVE_MAP(ref, version, r_type) \
      (ELFW(ST_BIND) ((*ref)->st_info) != STB_LOCAL                        \
       ? ((__builtin_expect ((*ref) == l->l_lookup_cache.sym, 0)           \
       && elf_machine_type_class (r_type) == l->l_lookup_cache.type_class)  \
      ? (bump_num_cache_relocations (),                                    \
         (*ref) = l->l_lookup_cache.ret,                                   \
         l->l_lookup_cache.value)                                          \
      : ({ lookup_t _lr;                                                   \
           int _tc = elf_machine_type_class (r_type);                      \
           l->l_lookup_cache.type_class = _tc;                             \
           l->l_lookup_cache.sym = (*ref);                                 \
           _lr = ((version) != NULL && (version)->hash != 0                \
                ? INTUSE(_dl_lookup_versioned_symbol) (strtab              \
                                    + (*ref)->st_name, \
                                    l, (ref), scope,      \
                                    (version), _tc, 0) \
                : INTUSE(_dl_lookup_symbol) (strtab + (*ref)->st_name, l, \
                                  (ref), scope, _tc,             \
                                DL_LOOKUP_ADD_DEPENDENCY));    \
           l->l_lookup_cache.ret = (*ref);                                 \
           l->l_lookup_cache.value = _lr; }))                              \
       : l)
#define RESOLVE(ref, version, r_type) \
      (ELFW(ST_BIND) ((*ref)->st_info) != STB_LOCAL                        \
       ? ((__builtin_expect ((*ref) == l->l_lookup_cache.sym, 0)           \
       && elf_machine_type_class (r_type) == l->l_lookup_cache.type_class)  \
           ? (bump_num_cache_relocations (),                               \
         (*ref) = l->l_lookup_cache.ret,                                   \
         l->l_lookup_cache.value)                                          \
      : ({ lookup_t _lr;                                                   \
           int _tc = elf_machine_type_class (r_type);                      \
           l->l_lookup_cache.type_class = _tc;                             \
           l->l_lookup_cache.sym = (*ref);                                 \
```

```
            _lr = ((version) != NULL && (version)->hash != 0                    \
                ? INTUSE(_dl_lookup_versioned_symbol) (strtab                   \
                                        + (*ref)->st_name, \
                                        l, (ref), scope,      \
                                        (version), _tc, 0) \
                : INTUSE(_dl_lookup_symbol) (strtab + (*ref)->st_name, l, \
                                    (ref), scope, _tc,             \
                                DL_LOOKUP_ADD_DEPENDENCY));    \
            l->l_lookup_cache.ret = (*ref);                                    \
            l->l_lookup_cache.value = _lr; }))                                 \
        : l->l_addr)

#include "dynamic-link.h"

      ELF_DYNAMIC_RELOCATE (l, lazy, consider_profiling);
… …
}
```

dynamic-link.h:

```
# define ELF_DYNAMIC_RELOCATE(map, lazy, consider_profile) \
   do {                                                    \
      int edr_lazy = elf_machine_runtime_setup ((map), (lazy),                 \
                                    (consider_profile));             \
      ELF_DYNAMIC_DO_REL ((map), edr_lazy);                                    \
      ELF_DYNAMIC_DO_RELA ((map), edr_lazy);                                   \
   } while (0)
```

glibc-2.3\sysdeps\i386\dl-machine.h:

```
/* Set up the loaded object described by L so its unrelocated PLT
     entries will jump to the on-demand fixup code in dl-runtime.c.    */

static inline int __attribute__ ((unused))
elf_machine_runtime_setup (struct link_map *l, int lazy, int profile)
{
   Elf32_Addr *got;
   extern void _dl_runtime_resolve (Elf32_Word) attribute_hidden;
   extern void _dl_runtime_profile (Elf32_Word) attribute_hidden;

   if (l->l_info[DT_JMPREL] && lazy)
      {
        /* The GOT entries for functions in the PLT have not yet been filled
       in.    Their initial contents will arrange when called to push an
       offset into the .rel.plt section, push _GLOBAL_OFFSET_TABLE_[1],
       and then jump to _GLOBAL_OFFSET_TABLE[2].    */
        got = (Elf32_Addr *) D_PTR (l, l_info[DT_PLTGOT]);
        /* If a library is prelinked but we have to relocate anyway,
       we have to be able to undo the prelinking of .got.plt.
       The prelinker saved us here address of .plt + 0x16.    */
        if (got[1])
      {
        l->l_mach.plt = got[1] + l->l_addr;
        l->l_mach.gotplt = (Elf32_Addr) &got[3];
      }
        got[1] = (Elf32_Addr) l;     /* Identify this shared object.    */

        /* The got[2] entry contains the address of a function which gets
```

```
        called to get the address of a so far unresolved function and
        jump to it.    The profiling extension of the dynamic linker allows
        to intercept the calls to collect information.    In this case we
        don't store the address in the GOT so that all future calls also
        end in this function.    */
          if (__builtin_expect (profile, 0))
      {
          got[2] = (Elf32_Addr) &_dl_runtime_profile;

          if (_dl_name_match_p (GL(dl_profile), l))
            /* This is the object we are looking for.    Say that we really
                 want profiling and the timers are started.    */
            GL(dl_profile_map) = l;
      }
          else
      /* This function will get called to fix up the GOT entry indicated by
           the offset on the stack, and then jump to the resolved address.    */
      got[2] = (Elf32_Addr) &_dl_runtime_resolve;
      }

   return lazy;
}


#ifdef IN_DL_RUNTIME

# if !defined PROF && !__BOUNDED_POINTERS__
/* We add a declaration of this function here so that in dl-runtime.c
     the ELF_MACHINE_RUNTIME_TRAMPOLINE macro really can pass the parameters
     in registers.

     We cannot use this scheme for profiling because the _mcount call
     destroys the passed register information.    */
/* GKM FIXME: Fix trampoline to pass bounds so we can do
     without the `__unbounded' qualifier.    */
static ElfW(Addr) fixup (struct link_map *__unbounded l, ElfW(Word) reloc_offset)
        __attribute__ ((regparm (2), unused));
static ElfW(Addr) profile_fixup (struct link_map *l, ElfW(Word) reloc_offset,
                                 ElfW(Addr) retaddr)
        __attribute__ ((regparm (3), unused));
# endif

/* This code is used in dl-runtime.c to call the `fixup' function
     and then redirect to the address it returns.    */
# if !defined PROF && !__BOUNDED_POINTERS__
#    define ELF_MACHINE_RUNTIME_TRAMPOLINE asm ("\
       .text\n\
       .globl _dl_runtime_resolve\n\
       .type _dl_runtime_resolve, @function\n\
       .align 16\n\
_dl_runtime_resolve:\n\
       pushl %eax            # Preserve registers otherwise clobbered.\n\
       pushl %ecx\n\
       pushl %edx\n\
       movl 16(%esp), %edx    # Copy args pushed by PLT in register.    Note\n\
       movl 12(%esp), %eax    # that `fixup' takes its parameters in regs.\n\
       call fixup        # Call resolver.\n\
```

```
        popl %edx               # Get register content back.\n\
        popl %ecx\n\
        xchgl %eax, (%esp)      # Get %eax contents end store function address.\n\
        ret $8                  # Jump to function address.\n\
        .size _dl_runtime_resolve, .-_dl_runtime_resolve\n\
\n\
        .globl _dl_runtime_profile\n\
        .type _dl_runtime_profile, @function\n\
        .align 16\n\
_dl_runtime_profile:\n\
        pushl %eax              # Preserve registers otherwise clobbered.\n\
        pushl %ecx\n\
        pushl %edx\n\
        movl 20(%esp), %ecx     # Load return address\n\
        movl 16(%esp), %edx     # Copy args pushed by PLT in register.    Note\n\
        movl 12(%esp), %eax     # that `fixup' takes its parameters in regs.\n\
        call profile_fixup      # Call resolver.\n\
        popl %edx               # Get register content back.\n\
        popl %ecx\n\
        xchgl %eax, (%esp)      # Get %eax contents end store function address.\n\
        ret $8                  # Jump to function address.\n\
        .size _dl_runtime_profile, .-_dl_runtime_profile\n\
        .previous\n\
");
# else
… …
# endif
#endif
```

glibc-2.3\elf\ dl-runtime.c:239 ELF_MACHINE_RUNTIME_TRAMPOLINE

How modify the content of GOT

从前面例子可知，在调用 fixup 函数后，就找到了函数的真正地址，同时修改了 GOT 相应位置的内容，我们便来看看 fixup 函数的实现：

glibc-2.3\elf\ dl-runtime.c:

```
#ifndef ELF_MACHINE_NO_PLT
static ElfW(Addr) __attribute_used__
fixup (
# ifdef ELF_MACHINE_RUNTIME_FIXUP_ARGS
            ELF_MACHINE_RUNTIME_FIXUP_ARGS,
# endif
       /* GKM FIXME: Fix trampoline to pass bounds so we can do
            without the `__unbounded' qualifier.    */
            struct link_map *__unbounded l, ElfW(Word) reloc_offset)
{
   const ElfW(Sym) *const symtab
      = (const void *) D_PTR (l, l_info[DT_SYMTAB]);
   const char *strtab = (const void *) D_PTR (l, l_info[DT_STRTAB]);

   const PLTREL *const reloc
      = (const void *) (D_PTR (l, l_info[DT_JMPREL]) + reloc_offset);
   const ElfW(Sym) *sym = &symtab[ELFW(R_SYM) (reloc->r_info)];
```

```
  void *const rel_addr = (void *)(l->l_addr + reloc->r_offset);
  lookup_t result;
  ElfW(Addr) value;

  /* The use of `alloca' here looks ridiculous but it helps.  The goal is
     to prevent the function from being inlined and thus optimized out.
     There is no official way to do this so we use this trick.  gcc never
     inlines functions which use `alloca'.  */
  alloca (sizeof (int));

  /* Sanity check that we're really looking at a PLT relocation.  */
  assert (ELFW(R_TYPE)(reloc->r_info) == ELF_MACHINE_JMP_SLOT);

   /* Look up the target symbol.  If the normal lookup rules are not
      used don't look in the global scope.  */
  if (__builtin_expect (ELFW(ST_VISIBILITY) (sym->st_other), 0) == 0)
    {
      switch (l->l_info[VERSYMIDX (DT_VERSYM)] != NULL)
    {
    default:
      {
        const ElfW(Half) *vernum =
          (const void *) D_PTR (l, l_info[VERSYMIDX (DT_VERSYM)]);
        ElfW(Half) ndx = vernum[ELFW(R_SYM) (reloc->r_info) & 0x7fff];
        const struct r_found_version *version = &l->l_versions[ndx];

        if (version->hash != 0)
          {
        result = INTUSE(_dl_lookup_versioned_symbol) (strtab
                                                      + sym->st_name,
                                                      l, &sym, l->l_scope,
                                                      version,
                                                      ELF_RTYPE_CLASS_PLT,
                                                      0);
        break;
          }
      }
    case 0:
      result = INTUSE(_dl_lookup_symbol) (strtab + sym->st_name, l, &sym,
                                          l->l_scope, ELF_RTYPE_CLASS_PLT,
                                          DL_LOOKUP_ADD_DEPENDENCY);
    }

      /* Currently result contains the base load address (or link map)
     of the object that defines sym.  Now add in the symbol
     offset.  */
      value = (sym ? LOOKUP_VALUE_ADDRESS (result) + sym->st_value : 0);
    }
  else
    {
      /* We already found the symbol.  The module (and therefore its load
     address) is also known.  */
      value = l->l_addr + sym->st_value;
#ifdef DL_LOOKUP_RETURNS_MAP
      result = l;
#endif
    }
```

```
  /* And now perhaps the relocation addend.  */
  value = elf_machine_plt_value (l, reloc, value);

  /* Finally, fix up the plt itself.  */
  if (__builtin_expect (GL(dl_bind_not), 0))
    return value;

  return elf_machine_fixup_plt (l, result, reloc, rel_addr, value);
}
#endif
```

glibc-2.3\sysdeps\generic\ dl-machine.h:

```
static inline Elf32_Addr
elf_machine_fixup_plt (struct link_map *map, lookup_t t,
                       const Elf32_Rel *reloc,
                       Elf32_Addr *reloc_addr, Elf32_Addr value)
{
  return *reloc_addr = value;
}
```

可以看到，我们关心 GOT 表内容是在 fixup 函数找到相应真实地址，返回时调用 elf_machine_fixup_plt 修改的。那么这个*reloc_addr 是什么地址呢，是 GOT[4]吗？看 fixup 开始的地方：

```
const PLTREL *const reloc
    = (const void *) (D_PTR (l, l_info[DT_JMPREL]) + reloc_offset);
  const ElfW(Sym) *sym = &symtab[ELFW(R_SYM) (reloc->r_info)];
  void *const rel_addr = (void *)(l->l_addr + reloc->r_offset);
```

reloc 是什么呢？它就是 GOT[0]内容.dynamic section 中 d_tag 为 DT_JMPREL 的 d_ptr 开始加上 reloc_offset 偏移的地址。参考《》中，我们可知，d_tag 为 DT_JMPREL 的虚拟地址为 0x8048220，即.rel.plt section 起始地址。而 reloc_offset 则为我们在.plt 过程链接表中各个需要重定位入口中 push 的偏移量，对于本文的 printf 是 0x8。对照.rel.plt 的内容可知，偏移为 8 处地址为 0x0x80494a4，即 GOT[4]。所以 reloc->r_offset 就是 0x0x80494a4。rel_addr = (void *)(l->l_addr + reloc->r_offset)，那么 l->l_addr 又是多少呢？

前面代码 dl_open_worker 函数中有一蓝色标记的函数 _dl_map_object -> _dl_map_object_from_fd：

glibc-2.3\elf\ dl-load.c:

```
struct link_map *
internal_function
_dl_map_object (struct link_map *loader, const char *name, int preloaded,
                int type, int trace_mode, int mode)
{
… …
return _dl_map_object_from_fd (name, fd, &fb, realname, loader, type, mode);
}

struct link_map *
_dl_map_object_from_fd (const char *name, int fd, struct filebuf *fbp,
                        char *realname, struct link_map *loader, int l_type,
                        int mode)
{
```

```
  struct link_map *l = NULL;
  const ElfW(Ehdr) *header;
  const ElfW(Phdr) *phdr;
  const ElfW(Phdr) *ph;
  size_t maplength;
  int type;
  struct stat64 st;
  /* Initialize to keep the compiler happy.    */
  const char *errstring = NULL;
  int errval = 0;

 … …

      case PT_LOAD:
        /* A load command tells us to map in part of the file.
           We record the load commands and process them all later.    */
        if (__builtin_expect ((ph->p_align & (GL(dl_pagesize) - 1)) != 0,
                              0))
          {
            errstring = N_("ELF load command alignment not page-aligned");
            goto call_lose;
          }
        if (__builtin_expect (((ph->p_vaddr - ph->p_offset)
                               & (ph->p_align - 1)) != 0, 0))
          {
            errstring
          = N_("ELF load command address/offset not properly aligned");
            goto call_lose;
          }

        {
          struct loadcmd *c = &loadcmds[nloadcmds++];
          c->mapstart = ph->p_vaddr & ~(ph->p_align - 1);
          c->mapend = ((ph->p_vaddr + ph->p_filesz + GL(dl_pagesize) - 1)
                       & ~(GL(dl_pagesize) - 1));
          c->dataend = ph->p_vaddr + ph->p_filesz;
          c->allocend = ph->p_vaddr + ph->p_memsz;
          c->mapoff = ph->p_offset & ~(ph->p_align - 1);

          /* Optimize a common case.    */
#if (PF_R | PF_W | PF_X) == 7 && (PROT_READ | PROT_WRITE | PROT_EXEC) == 7
          c->prot = (PF_TO_PROT
                     >> ((ph->p_flags & (PF_R | PF_W | PF_X)) * 4)) & 0xf;
#else
          c->prot = 0;
          if (ph->p_flags & PF_R)
            c->prot |= PROT_READ;
          if (ph->p_flags & PF_W)
            c->prot |= PROT_WRITE;
          if (ph->p_flags & PF_X)
            c->prot |= PROT_EXEC;
#endif
        }
        break;

      case PT_TLS:
#ifdef USE_TLS
```

```
        if (ph->p_memsz > 0)
          {
            l->l_tls_blocksize = ph->p_memsz;
            l->l_tls_align = ph->p_align;
            l->l_tls_initimage_size = ph->p_filesz;
            /* Since we don't know the load address yet only store the
           offset.  We will adjust it later.  */
            l->l_tls_initimage = (void *) ph->p_vaddr;

            /* Assign the next available module ID.  */
            l->l_tls_modid = _dl_next_tls_modid ();
          }
#else
        /* Uh-oh, the binary expects TLS support but we cannot
           provide it.  */
        _dl_fatal_printf ("cannot handle file '%s' with TLS data\n", name);
#endif
        break;
      }

      /* Now process the load commands and map segments into memory.  */
      c = loadcmds;

      /* Length of the sections to be loaded.  */
      maplength = loadcmds[nloadcmds - 1].allocend - c->mapstart;

      if (__builtin_expect (type, ET_DYN) == ET_DYN)
        {
      /* This is a position-independent shared object.  We can let the
         kernel map it anywhere it likes, but we must have space for all
         the segments in their specified positions relative to the first.
         So we map the first segment without MAP_FIXED, but with its
         extent increased to cover all the segments.  Then we remove
         access from excess portion, and there is known sufficient space
         there to remap from the later segments.

         As a refinement, sometimes we have an address that we would
         prefer to map such objects at; but this is only a preference,
         the OS can do whatever it likes. */
      ElfW(Addr) mappref;
      mappref = (ELF_PREFERRED_ADDRESS (loader, maplength, c->mapstart)
                 - MAP_BASE_ADDR (l));

      /* Remember which part of the address space this object uses.  */
      l->l_map_start = (ElfW(Addr)) __mmap ((void *) mappref, maplength,
                                            c->prot, MAP_COPY | MAP_FILE,
                                            fd, c->mapoff);
      if (__builtin_expect ((void *) l->l_map_start == MAP_FAILED, 0))
        {
        map_error:
          errstring = N_("failed to map segment from shared object");
          goto call_lose_errno;
        }

      l->l_map_end = l->l_map_start + maplength;
      l->l_addr = l->l_map_start - c->mapstart;
```

```
… …

   return l;
}
```

将 PT_LOAD 的段映射到内存中，对于可执行文件或共享文件来说，映射的地址就是段中的虚拟地址，所以 l->l_addr = l->l_map_start - c->mapstart 就是为 0!所以前面 rel_addr = (void *)(l->l_addr + reloc->r_offset)的值就是 0x0x80494a4，GOT[4]的值。也就是说，把找到的 printf 的实际地址修改在 GOT[4]中!

How to find the symbol

无论是 fixup 还是 dlsym，最终都会调用 do_lookup 来查找符号，找到返回 1。

glibc-2.3\elf\ do-lookup.h:

```
#if VERSIONED
# define FCT do_lookup_versioned
# define ARG const struct r_found_version *const version
#else
# define FCT do_lookup
# define ARG int flags
#endif

/* Inner part of the lookup functions.    We return a value > 0 if we
     found the symbol, the value 0 if nothing is found and < 0 if
     something bad happened.    */
static inline int
FCT (const char *undef_name, unsigned long int hash, const ElfW(Sym) *ref,
        struct sym_val *result, struct r_scope_elem *scope, size_t i, ARG,
        struct link_map *skip, int type_class)
{
   struct link_map **list = scope->r_list;
   size_t n = scope->r_nlist;
   struct link_map *map;

   do
      {
         const ElfW(Sym) *symtab;
         const char *strtab;
         const ElfW(Half) *verstab;
         Elf_Symndx symidx;
         const ElfW(Sym) *sym;
#if ! VERSIONED
         int num_versions = 0;
         const ElfW(Sym) *versioned_sym = NULL;
#endif

         map = list[i];

         /* Here come the extra test needed for `_dl_lookup_symbol_skip'.    */
         if (skip != NULL && map == skip)
       continue;

         /* Don't search the executable when resolving a copy reloc.    */
         if ((type_class & ELF_RTYPE_CLASS_COPY) && map->l_type == lt_executable)
       continue;
```

```
        /* Print some debugging info if wanted.    */
        if (__builtin_expect (GL(dl_debug_mask) & DL_DEBUG_SYMBOLS, 0))
      INTUSE(_dl_debug_printf) ("symbol=%s;    lookup in file=%s\n",
                                undef_name, (map->l_name[0]
                                             ? map->l_name : rtld_progname));

        symtab = (const void *) D_PTR (map, l_info[DT_SYMTAB]);
        strtab = (const void *) D_PTR (map, l_info[DT_STRTAB]);
        verstab = map->l_versyms;

        /* Search the appropriate hash bucket in this object's symbol table
       for a definition for the same symbol name.    */
        for (symidx = map->l_buckets[hash % map->l_nbuckets];
          symidx != STN_UNDEF;
          symidx = map->l_chain[symidx])
      {
        sym = &symtab[symidx];

        assert (ELF_RTYPE_CLASS_PLT == 1);
        if ((sym->st_value == 0 /* No value.    */
#ifdef USE_TLS
                && ELFW(ST_TYPE) (sym->st_info) != STT_TLS
#endif
                )
               || (type_class & (sym->st_shndx == SHN_UNDEF)))
            continue;

        if (ELFW(ST_TYPE) (sym->st_info) > STT_FUNC
#ifdef USE_TLS
              && ELFW(ST_TYPE) (sym->st_info) != STT_TLS
#endif
              )
            /* Ignore all but STT_NOTYPE, STT_OBJECT and STT_FUNC
                entries (and STT_TLS if TLS is supported) since these
                are no code/data definitions.    */
            continue;

        if (sym != ref && strcmp (strtab + sym->st_name, undef_name))
            /* Not the symbol we are looking for.    */
            continue;

#if VERSIONED
        if (__builtin_expect (verstab == NULL, 0))
            {
              /* We need a versioned symbol but haven't found any.    If
             this is the object which is referenced in the verneed
             entry it is a bug in the library since a symbol must
             not simply disappear.

             It would also be a bug in the object since it means that
             the list of required versions is incomplete and so the
             tests in dl-version.c haven't found a problem.*/
              assert (version->filename == NULL
                      || ! _dl_name_match_p (version->filename, map));

              /* Otherwise we accept the symbol.    */
```

```
            }
          else
            {
              /* We can match the version information or use the
             default one if it is not hidden.    */
              ElfW(Half) ndx = verstab[symidx] & 0x7fff;
              if ((map->l_versions[ndx].hash != version->hash
                || strcmp (map->l_versions[ndx].name, version->name))
              && (version->hidden || map->l_versions[ndx].hash
                    || (verstab[symidx] & 0x8000)))
            /* It's not the version we want.    */
            continue;
            }
#else
          /* No specific version is selected.    There are two ways we
              can got here:

              - a binary which does not include versioning information
                is loaded

              - dlsym() instead of dlvsym() is used to get a symbol which
                might exist in more than one form

              If the library does not provide symbol version
              information there is no problem at at: we simply use the
              symbol if it is defined.

              These two lookups need to be handled differently if the
              library defines versions.    In the case of the old
              unversioned application the oldest (default) version
              should be used.    In case of a dlsym() call the latest and
              public interface should be returned.    */
          if (verstab != NULL)
            {
              if ((verstab[symidx] & 0x7fff)
              >= ((flags & DL_LOOKUP_RETURN_NEWEST) ? 2 : 3))
            {
              /* Don't accept hidden symbols.    */
              if ((verstab[symidx] & 0x8000) == 0 && num_versions++ == 0)
                /* No version so far.    */
                versioned_sym = sym;

              continue;
            }
            }
#endif

          /* There cannot be another entry for this symbol so stop here.    */
          goto found_it;
        }

          /* If we have seen exactly one versioned symbol while we are
         looking for an unversioned symbol and the version is not the
         default version we still accept this symbol since there are
         no possible ambiguities.    */
#if VERSIONED
          sym = NULL;
```

```
#else
          sym = num_versions == 1 ? versioned_sym : NULL;
#endif

          if (sym != NULL)
        {
        found_it:
          switch (ELFW(ST_BIND) (sym->st_info))
            {
            case STB_WEAK:
              /* Weak definition.    Use this value if we don't find another.    */
              if (__builtin_expect (GL(dl_dynamic_weak), 0))
            {
              if (! result->s)
                {
                  result->s = sym;
                  result->m = map;
                }
              break;
            }
              /* FALLTHROUGH */
            case STB_GLOBAL:
              /* Global definition.    Just what we need.    */
              result->s = sym;
              result->m = map;
              return 1;
            default:
              /* Local symbols are ignored.    */
              break;
            }
        }

#if VERSIONED
          /* If this current map is the one mentioned in the verneed entry
         and we have not found a weak entry, it is a bug.    */
          if (symidx == STN_UNDEF && version->filename != NULL
          && __builtin_expect (_dl_name_match_p (version->filename, map), 0))
        return -1;
#endif
        }
    while (++i < n);

    /* We have not found anything until now.    */
    return 0;
}
```

这块代码没有仔细研究，大概思路应该如下 ：

该函数中涉及到了 hash table,hash table 元素的数目， chain,dynamic string table 和 dynamic symbol talbe。hash 号是 elf_hash()的返回值，在 ELF 规范的第 4 部分有定义，以 hash table 中元素个数取模。该号被用来做 hash table 的下表索引，求得 hash 值，找出与之匹配的符号名的 chain 的索引。使用该索引，从符号表中获得符号。比较获得的符号名和请求的符号名是否相同。使用这个算法，就可以简单解析任何符号了。

Reference