漫谈兼容内核之九:
ELF 映像的装入(二)

毛德操

上一篇漫谈介绍了在通过 execve()系统调用启动一个 ELF 格式的目标映像时发生于 Linux 内核中的活动。简而言之，内核根据映像头部所提供的信息把目标映像映射到(装入)当前进程用户空间的某个位置上；并且，如果目标映像需要使用共享库的话，还要(根据映像头部所提供的信息)将所需的“解释器”的映像也映射到用户空间的某个位置上，然后在从系统调用返回用户空间的时候就“返回”到解释器的入口，下面就是解释器的事了。如果目标映像不使用共享库，那么问题就比较简单，返回用户空间的时候就直接“返回”到目标映像的入口。现代的应用软件一般都要使用共享库，所以我们把这当作常态，而把不使用共享库的应用软件作为一种简化了的特例。

映像装入用户空间的位置有些是固定的、在编译连接时就确定好了的；有些则是“浮动”的、可以在装入时动态决定；具体要看编译时是否使用了-fPIC 选项。一般应用软件主体的映像都是固定地址的，而共享库映像的装入地址都是浮动的。特别地，解释器映像的装入地址也是浮动的。

2．ELF 映像的结构

每个操作系统对于在其内核上运行的可执行程序二进制映像都有特定的要求和规定，包括例如映像的格式，映像在用户空间的布局(程序段、数据段、堆栈段的划分等等)，映像装入用户空间的地址是否可以浮动、以及如何浮动，是否支持动态连接、以及如何连接，如何进行系统调用，等等。这些要求和规定合在一起就构成了具体操作系统的“应用(软件)二进制界面(Application Binary Interface)”，缩写成 ABI。显然，ABI 是二进制映像的“生产者”即编译/连接工具和使用者即映像装入/启动手段之间的一组约定。而我们一般所说的二进制映像格式，实际上并不仅仅是指字面意义上的、类似于数据结构定义那样的“格式”，还包括了跟映像装入过程有关的其它约定。所以，二进制映像格式是 ABI 的主体。

目前的 Linux ABI 是在 Unix 系统 5 的时期(大约在 1980 年代)发展起来的，其主体就是 ELF，这是“可执行映像和连接格式(Executable and Lnking Format)”的缩写。

读者已经看到，ELF 映像文件的开始是个 ELF 头，这是一个数据结构，结构中有个指针(位移量)，指向文件中的一个“程序头”数组(表)。各个程序头表项当然也是数据结构，这是对映像文件中各个“节(Segment)”的(结构性)描述。

从映像装入的角度看，一个映像是由若干个 Segment 构成的。有些 Segment 需要被装入、即被映射到用户空间，有些则不需要被装入。在前一篇漫谈中读者已经看到，只有类型为 PT_LOAD 的 Segment 才需要被装入。所以，映像装入的过程只“管”到 Segment 为止。而从映像的动态连接、重定位(即浮动)、和启动运行的角度看，则映像是由若干个“段(Section)”构成的。我们通常所说映像中的“代码段”、“数据段”等等都是 Section。所以，动态连接和启动运行的过程所涉及的则是 Section。一般而言，一个 Segment 可以包含多个 Section。其实，Segment 和 Section 都是从操作/处理的角度对映像的划分；对于不同的操作/处理，划分的方式也就可以不同。所以，读者在后面将会看到，一个 Segment 里面也可以包含几个别的 Segment，这就是因为它们是按不同的操作/处理划分的、不同意义上的 Segment。Section 也是一样。

在 Linux 系统中，(应用软件主体)目标映像本身的装入是由内核负责的，这个过程读者

已经看到；而动态连接的过程则由运行于用户空间的“解释器”负责。这里要注意：第一，“解释器”是与具体的映像相连系的，其本身也有个映像，也需要被装入。与目标映像相连系的“解释器”也是由内核装入的，这一点读者也已看到。第二，动态连接的过程包括了共享库映像的装入，那却是由“解释器”在用户空间实现的。

本来，看了内核中与装入目标映像有关的代码以后，应该接着看“解释器”的代码了。但是后者比前者复杂得多，也繁琐得多，原因是牵涉到许多 ELF 和 ABI 的原理和细节，所以有必要先对 ELF 动态连接的原理作一介绍。明白了有关的原理和大致的方法以后，具体的代码实现倒在其次了。

前面讲过，Linux 提供了两个很有用的工具，即 readelf 和 objdump。下面就用这两个工具对映像/usr/local/bin/wine 进行一番考察，以期在此过程中逐步对 ELF 和 ABI 有所了解和理解，这也是进一步阅读、理解“解释器”的代码所需要的。

我们用命令行“readelf –a /usr/local/bin/wine”和“objdump –d /usr/local/bin/wine”产生两个文件(把结果重定向到文件中)，然后察看这两个文件的部分内容。

首先是目标映像的 ELF 头：

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048750
  Start of program headers:          52 (bytes into file)
  Start of section headers:          114904 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         6
  Size of section headers:           40 (bytes)
  Number of section headers:         36
  Section header string table index: 33
```

这就是映像文件开头处的 ELF 头，其最初 4 个字节为‘0x7f’、和‘E’、‘L’、‘F’。从其余字段中我们可以看出：

- OS 是 Unix、其实是 Linux、而 ABI 是系统 5 的 ABI。ABI 的版本号为 0。
- CPU 为 x86。
- 映像的类型为 EXEC，即带有主函数 main()的应用软件映像(若是共享库则类型为 DYN、即动态连接库)。
- 映像的程序入口地址为 0x8048750。如前所述，EXEC 映像的装入地址是固定的、不能浮动。

- 程序头数组起点在文件中的位移为 52(字节)，而 ELF 头的大小正好也是 52，所以紧接 ELF 头的后面就是程序头数组。数组的大小为 6，即映像中有 6 个 Segment。
- Section 头的数组则一直在后面位移位 114904 的地方，映像中有 36 个 Section。

于是，我们接下去看程序头数组：

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000c0 0x000c0 R E 0x4
  INTERP         0x0000f4 0x080480f4 0x080480f4 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x011cc 0x011cc R E 0x1000
  LOAD           0x0011cc 0x0804a1cc 0x0804a1cc 0x00158 0x00160 RW  0x1000
  DYNAMIC        0x0011d8 0x0804a1d8 0x0804a1d8 0x000d8 0x000d8 RW  0x4
  NOTE           0x000108 0x08048108 0x08048108 0x00020 0x00020 R   0x4
```

一个程序头就是关于一个 Segment 的说明，所以这就是 6 个 Segment。第一个 Segment 的类型是 PHDR，在文件中的位移为 0x34、即 52，这就是程序头数组本身。其大小为 0xc0、即 192。前面说每个程序头的大小为 32 字节，而 6 X 32 = 192。第二个 Segment 的类型是 INTERP，即“解释器”的文件/路径名，是个字符串，这里说是“/lib/ld-linux.so.2”。

下面是两个类型为 LOAD 的 Segment。如前所述，只有这种类型的 Segment 才需要装入。但是，看一下前者的说明，其起点在文件中的位移是 0，大小是 0x011cc，显然是把 ELF 头和前两个 Segment 也包含在里面了。再看后者，其起点的位移是 0x011cc，所以是和前者连在一起的；其大小为 0x158，这样两个 Segment 合在一起是从 0 到 0x1324。计算一下就可知道，实际上是把所有的 Segment 都包括进去了。所以，对于这个特定的映像，说是只装入类型为 LOAD 的 Segment，实际上装入的却是整个映像。那么，映像中的什么内容可以不必装入呢？例如 bss 段，那是无初始内容的数据段，就不用装入；还有(与动态连接无关的)符号表，那也不需要装入。注意两个 LOAD 类 Segment 的边界(Alignment)都是 0x1000，即 4KB，那正好是存储页面的大小。还有个问题，既然两个 LOAD 类的 Segment 是连续的，那为什么不合并成一个呢？看一下它们的特性标志位就可以知道，第一个 Segment 的映像是可读可执行、但是不可写；第二个则是可读可写、但是不可执行，这当然不能合并。

再往下看，下一个 Segment 的类型是 DYNAMIC，那就是跟动态连接有关的信息。如上所述，这个 Segment 其实是包含在前一个 Segment 中的，所以也会被装入。最后一个 Segment 的类型是 NOTE，那只是注释、说明一类的信息了。

当然，跟动态连接有关的信息是我们最为关心的，所以我们看一下这个 Segment 的具体内容：

```
Dynamic segment at offset 0x11d8 contains 22 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libwine.so.1]
 0x00000001 (NEEDED)                     Shared library: [libpthread.so.0]
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
 0x0000000c (INIT)                       0x80485e8
 0x0000000d (FINI)                       0x8049028
```

```
0x00000004 (HASH)                       0x8048128
0x00000005 (STRTAB)                     0x8048368
0x00000006 (SYMTAB)                     0x80481d8
0x0000000a (STRSZ)                      301 (bytes)
0x0000000b (SYMENT)                     16 (bytes)
0x00000015 (DEBUG)                      0x0
0x00000003 (PLTGOT)                     0x804a2c4
0x00000002 (PLTRELSZ)                   160 (bytes)
0x00000014 (PLTREL)                     REL
0x00000017 (JMPREL)                     0x8048548
0x00000011 (REL)                        0x8048538
0x00000012 (RELSZ)                      16 (bytes)
0x00000013 (RELENT)                     8 (bytes)
0x6ffffffe (VERNEED)                    0x80484c8
0x6fffffff (VERNEEDNUM)                 3
0x6ffffff0 (VERSYM)                     0x8048496
0x00000000 (NULL)                       0x0
```

这个 Segment 中有 22 项数据，开头几项类型为 NEEDED 的数据是我们此刻最为关心的，因为这些数据告诉了我们目标映像要求装入那一些共享库，例如 libwine.so.1。读者已经看过内核怎样装入用户空间映像，解释器只不过是在用户空间做同样的事，所以共享库的装入对于读者并不复杂，问题是怎样实现动态连接，这是我后面要着重讲的。

前面说过，Segment 是从映像装入角度考虑的划分，Section 才是从连接/启动角度考虑的划分，现在我们就来看 Section。先看 Section 与 Segment 的对应关系：

```
Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r
          .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame
   03     .data .dynamic .ctors .dtors .jcr .got .bss
   04     .dynamic
   05     .note.ABI-tag
```

Section 的名称都以'.'开头，例如.interp；名称中间也可以有'.'，例如 rel.dyn。

这说明，Segment 0 不含有任何 Section，因为这就是程序头数组。Segment 1 只含有一个 Section，那就是.interp，即解释器的文件/路径名。而 Segment 2 所包含的 Section 就多了。而且，这个 Segment 还包含了前面两个 Segmrnt，所以.interp 又同时出现在这个 Segment 中。余类推。

前面 ELF 头中说一共有 36 个 Section，下面就是一份清单：

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
```

```
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        080480f4 0000f4 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048108 000108 000020 00   A  0   0  4
  [ 3] .hash             HASH            08048128 000128 0000b0 04   A  4   0  4
  [ 4] .dynsym           DYNSYM          080481d8 0001d8 000190 10   A  5   1  4
  [ 5] .dynstr           STRTAB          08048368 000368 00012d 00   A  0   0  1
  [ 6] .gnu.version      VERSYM          08048496 000496 000032 02   A  4   0  2
  [ 7] .gnu.version_r    VERNEED         080484c8 0004c8 000070 00   A  5   3  4
  [ 8] .rel.dyn          REL             08048538 000538 000010 08   A  4   0  4
  [ 9] .rel.plt          REL             08048548 000548 0000a0 08   A  4   b  4
  [10] .init             PROGBITS        080485e8 0005e8 000017 00  AX  0   0  4
  [11] .plt              PROGBITS        08048600 000600 000150 04  AX  0   0  4
  [12] .text             PROGBITS        08048750 000750 0008d8 00  AX  0   0  4
  [13] .fini             PROGBITS        08049028 001028 00001b 00  AX  0   0  4
  [14] .rodata           PROGBITS        08049060 001060 000166 00   A  0   0 32
  [15] .eh_frame         PROGBITS        080491c8 0011c8 000004 00   A  0   0  4
  [16] .data             PROGBITS        0804a1cc 0011cc 00000c 00  WA  0   0  4
  [17] .dynamic          DYNAMIC         0804a1d8 0011d8 0000d8 08  WA  5   0  4
  [18] .ctors            PROGBITS        0804a2b0 0012b0 000008 00  WA  0   0  4
  [19] .dtors            PROGBITS        0804a2b8 0012b8 000008 00  WA  0   0  4
  [20] .jcr              PROGBITS        0804a2c0 0012c0 000004 00  WA  0   0  4
  [21] .got              PROGBITS        0804a2c4 0012c4 000060 04  WA  0   0  4
  [22] .bss              NOBITS          0804a324 001324 000008 00  WA  0   0  4
  [23] .stab             PROGBITS        00000000 001324 004878 0c     24   0  4
  [24] .stabstr          STRTAB          00000000 005b9c 014cd4 00      0   0  1
  [25] .comment          PROGBITS        00000000 01a870 000165 00      0   0  1
  [26] .debug_aranges    PROGBITS        00000000 01a9d8 000078 00      0   0  8
  [27] .debug_pubnames   PROGBITS        00000000 01aa50 000025 00      0   0  1
  [28] .debug_info       PROGBITS        00000000 01aa75 000a98 00      0   0  1
  [29] .debug_abbrev     PROGBITS        00000000 01b50d 000138 00      0   0  1
  [30] .debug_line       PROGBITS        00000000 01b645 000284 00      0   0  1
  [31] .debug_frame      PROGBITS        00000000 01b8cc 000014 00      0   0  4
  [32] .debug_str        PROGBITS        00000000 01b8e0 0006be 01  MS  0   0  1
  [33] .shstrtab         STRTAB          00000000 01bf9e 00013a 00      0   0  1
  [34] .symtab           SYMTAB          00000000 01c678 000890 10     35  5c  4
  [35] .strtab           STRTAB          00000000 01cf08 0005db 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

这是按 Section 的名称列出的，其中跟动态连接有关的 Section 也出现在前面名为 Dynamic 的 Segment 中，只是在那里是按类型列出的。例如，前面类型为 HASH 的表项说与此有关的信息在 0x8048128 处，而这里则说有个名为.hash 的 Section，其起始地址为

0x8048128。还有，前面类型为 PLTGOT 的表项说与此有关的信息在 0x804a2c4 处，这里则说有个名为. got 的 Section，其起始地址为 0x804a2c4，不过 Section 表中提供的信息更加详细一些，有些信息则互相补充。在 Section 表中，只要类型为 PROGBITS，就说明这个 Section 的内容都来自映像文件，反之类型为 NOBITS 就说明这个 Section 的内容并非来自映像文件。

有些 Section 名是读者本来就知道的，例如. text、. data、. bss；有些则从它们的名称就可猜测出来，例如. symtab 是符号表、. rodata 是只读数据、还有. comment 和. debug_info 等等。还有一些可能就不知道了，这里择其要者先作些简略的介绍：

- . hash。为便于根据函数/变量名找到有关的符号表项，需要对函数/变量名进行 hash 计算，并根据计算值建立 hash 队列。
- . dynsym。需要加以动态连接的符号表，类似于内核模块中的 INPORT 符号表。这是动态连接符号表的数据结构部分，须与. dynstr 联用。
- . dynstr。动态连接符号表的字符串部分，与. dynsym 联用。
- . rel. dyn。用于动态连接的重定位信息。
- . rel. plt。一个结构数组，其中的每个元素都代表着 GOP 表中的一个表项 GOTn(见下)。
- . init。在进入 main()之前执行的代码在这个 Section 中。
- . plt。“过程连接表(Procedure Linking Table)”，见后。
- . fini。从 main()返回之后执行的代码在这个 Section 中，最后会调用 exit()。
- . ctors。表示“Constructor”，是一个函数指针数组，这些函数需要在程序初始化阶段(进入 main()之前，在. init 中)加以调用。
- . dtors。表示“Distructor”，也是一个函数指针数组，这些函数需要在程序扫尾阶段(从 main()返回之后，在. fini 中)加以调用。
- . got。“全局位移表(Global Offset Table)”，见后。
- . strtab。与符号表有关的字符串都集中在这个 Section 中。

其中我们最关心的是“过程连接表(Procedure Linking Table)”PLT 和“全局位移表(Global Offset Table)”GOT。程序之间的动态连接就是通过这两个表实现的。

下面我们通过一个实例来说明程序之间的动态连接。目标映像/usr/local/bin/wine 的 main()函数中调用了一个库函数 getenv()，这个函数在C语言共享库 libc.so.6 中。下面是 main()经编译/连接以后的汇编代码：

```
08048ce0 <main>:
 8048ce0:    55                          push   %ebp
 8048ce1:    89 e5                       mov    %esp,%ebp
 . . . . . .
 8048cef:    68 20 91 04 08              push   $0x8049120
 8048cf4:    e8 47 f9 ff ff              call   8048640 <_init+0x58>
```

本来，这里 call 指令机器代码的后 4 个字节应该是目标函数 getenv()的入口地址。可是，这个目标函数在共享库 libc.so.6 中，而这个共享库的装入地址是浮动的，要到装入了以后才能知道其地址。怎么办？一个不必很有天分的人就能想到的简单办法是：编译时先让这条 call 指令空着，但是创建一个带有字符串“getenv”的数据结构，并让这个数据结构中有个指针反过来指向这条 call 指令；而在动态连接时，则让“解释器”在共享库的导出符号表中寻找这个符号，找到后根据其装入后的位置计算出应该填入这条 call 指令的数值，再把结果

填写到这里的 call 指令中、即地址为 0x8048cf5 的地方。当然，程序中调用 getenv()的地方可能不止一个，所以在调用者的映像中需要把所有调用 getenv()的地方都记下来。然而，不幸的是这样的地方可能成百上千，而类似于 getenv()这样由共享库提供的函数也可能成百上千。更何况一个共享库可能还要用到别的共享库，从而形成一个多层次的共享库“图”。这样一来，动态连接的效率就大成问题了，显然这不是个好办法。

那么实际采用的办法是什么样的呢？这里的 call 指令采用的是相对寻址，调用的子程序入口地址为 0x8048640，我们就循着这个地址看过去：

```
8048640: ff 25 dc a2 04 08          jmp    *0x804a2dc
8048646: 68 18 00 00 00             push   $0x18
804864b: e9 b0 ff ff ff             jmp    8048600 <_init+0x18>
```

这就已经在 wine 映像的 PLT 表中了，这几条指令就构成 getenv()在 PLT 中的表项，程序中凡是对 getenv()的调用都先来到这里。当然，PLT 表中有许多这样的表项，对应着许多需要通过动态连接引入的函数，凡是这样的表项都以 PLTn 表示之。所有的 PLTn 都是相似的，但是 PLT 表中的第一个表项、即 PLT0、却是特殊的：

```
08048600 <.plt>:
 8048600:    ff 35 c8 a2 04 08        pushl  0x804a2c8
 8048606:    ff 25 cc a2 04 08        jmp    *0x804a2cc
 804860c:    00 00                    add    %al,(%eax)
 804860e:    00 00                    add    %al,(%eax)
 8048610:    ff 25 d0 a2 04 08        jmp    *0x804a2d0
 8048616:    68 00 00 00 00           push   $0x0
 804861b:    e9 e0 ff ff ff           jmp    8048600 <_init+0x18>
 8048620:    ff 25 d4 a2 04 08        jmp    *0x804a2d4
 8048626:    68 08 00 00 00           push   $0x8
 804862b:    e9 d0 ff ff ff           jmp    8048600 <_init+0x18>
 8048630:    ff 25 d8 a2 04 08        jmp    *0x804a2d8
 8048636:    68 10 00 00 00           push   $0x10
 804863b:    e9 c0 ff ff ff           jmp    8048600 <_init+0x18>
 8048640:    ff 25 dc a2 04 08        jmp    *0x804a2dc
 8048646:    68 18 00 00 00           push   $0x18
 804864b:    e9 b0 ff ff ff           jmp    8048600 <_init+0x18>
    ......
```

可以看出，除 PLT0 以外，所有的 PLTn 的形式都是一样的，而且最后的 jmp 指令都是以 0x8048600、即 PLT0 为目标，所不同的只是第一条 jmp 指令的目标和 push 指令中的数据。PLT0 则与之不同，但是包括 PLT0 在内的每个表项都占 16 个字节，所以整个 PLT 就像是个数组。其实 PLT0 只需要 12 个字节，但是为了大小划一而补了 4 个字节的 0。

注意每个 PLTn 中的第一条 jmp 指令是间接寻址的。以 getenv()的表项为例，是以地址 0x804a2dc 处的内容为目标地址进行跳转。这样，只要把 getenv()装入用户空间后的入口地址填写在 0x804a2dc 处，就可以实现正确的跳转，即实现了与共享库中函数 getenv()的动态连接。这样，对于共享库函数的每次调用，额外的消耗只是执行一条间接寻址的 jmp 指令

所需的时间。另一方面，这是不涉及堆栈的跳转指令，堆栈的内容在跳转的过程中保持不变，所以当 getenv()执行 ret 指令返回时就直接回到了调用它的地方，在这里是前面的 main()中。

由此可见，解释器的任务就是事先把 getenv()装入用户空间后的入口地址填写在 0x804a2dc 处。不仅是 getenv()，共享库提供的库函数可能有很多，对于每个这样的库函数都得保存一个用于间接寻址跳转的指针。保存这些指针的地方就是 GOT。与 PLT 相对应，每个 PLTn 在 GOT 中都有个相应的 GOTn，但是每个 GOTn 只是一个函数指针。同样，GOT0 也是特殊的，而且 GOT0 的大小也不一样，有 12 个字节，相当于三个 GOTn 那么大。在“解释器”ld-linux.so 的代码中把 GOT0 的三个长字表示成 got[0]、got[1]、和 got[2]，注意不要跟 GOTn 相混淆。显然、解释器负有正确设置所有 GOTn 的责任。

既然如此，每个 PLTn 中只要一条指令就行了，代码中为什么有三条呢？还有，PLT0 和 GOT0 又是干什么用的呢？原来，那都是为实现“懒惰式”的动态连接、即“懒连接”而存在的。简而言之，“懒连接”就是解释器并不事先完成对共享库函数的动态连接、即不事先设置 GOTn、而把对具体共享库函数的动态连接拖延到真正要用的时候才来进行，需要用哪一个函数就连接哪一个函数，绝不“积极主动”，以免劳而无功。

然而，要是不事先设置好 GOTn 的内容，PLTn 中的(间接寻址)跳转指令会跳到什么地方去？这要看 GOTn 中的原始内容，这内容来自目标映像(对外进行库函数调用的映像)。

我们看 wine 映像所提供的 GOT 原始内容。这个映像的 GOT 起始地址为 0x0804a2c4，跳过 GOT0 的 12 个字节，GOTn 是从 0x0804a2d0 开始的：

```
Relocation section '.rel.plt' at offset 0x548 contains 20 entries:

 Offset     Info    Type            Sym.Value  Sym. Name
0804a2d0  00000107 R_386_JUMP_SLOT   08048610   strchr
0804a2d4  00000207 R_386_JUMP_SLOT   08048620   getpid
0804a2d8  00000307 R_386_JUMP_SLOT   08048630   fprintf
0804a2dc  00000407 R_386_JUMP_SLOT   08048640   getenv
0804a2e0  00000507 R_386_JUMP_SLOT   08048650   pthread_create
 . . . . . .
```

从这里地址为 0x0804a2dc 的这一表项看，似乎这个指针所指向的是 0x08048640，这正是指令“jmp *0x804a2dc”所在的位置。设想如果 CPU 在尚未完成对 getenv()的动态连接之前就调用了这个函数，从而在 PLT 中执行了指令“jmp *0x804a2dc”，那岂不就是执行了一条指向其自身的跳转指令？这可是一个最紧扣的死循环！

然而事实并非如此，这里的信息是经过 readelf 整理的，旨在让使用者知道这一表项跟 PLT 表中地址为 0x08048640 的表项相对应，并且相应的函数名为 getenv。这是 readel 综合了好几方面的信息才形成的报告，不幸的是在有些情况下这就成了很误导的报告。而实际上存储在映像中这个位置上的数据却有所不同。为此，我们通过另一个工具 od 去观察这个位置上真实的、原始的内容。根据前面关于. got 的信息，地址 0x 0804a2c4 在映像中的位移是 0x0012c4，所以从 0x0804a2d0 开始的几个指针是：

```
0012d0 8616 0804 8626 0804 8636 0804 8646 0804
0012e0 8656 0804 8666 0804 8676 0804 8686 0804
```

可见，地址 0x0804a2dc 处的内容其实指向 0x08048646。这个地址同样在 getenv()的 PLT

表项中，但这是第二条指令“push $0x18”所在的地址。所以，即使在动态连接之前就试图调用 getenv()，至少在这里是不会出问题的。

回过去看 getenv()在 PLT 中的表项。在执行了“push $0x18”以后，下一条指令是“jmp 0x8048600”，这就是 PLT0 的起点。注意每个 PLTn 表项的最后一条指令都是相同的，都是跳转到 PLT0 的起点，所不同的只是压入堆栈的数值，所以这里 0x18 就代表着 getenv()。另一方面，PLT 中在 getenv()之前的几个表项压入堆栈的数值分别为 0x0、0x8、0x10，所以 0x18 表示这是 PLT 中的第 4 个函数。相应地，我们从.dynsym 的内容也可以得到验证：

```
Symbol table '.dynsym' contains 25 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 08048610   359 FUNC    GLOBAL DEFAULT  UND strchr@GLIBC_2.0 (2)
     2: 08048620     8 FUNC    GLOBAL DEFAULT  UND getpid@GLIBC_2.0 (2)
     3: 08048630    23 FUNC    GLOBAL DEFAULT  UND fprintf@GLIBC_2.0 (2)
     4: 08048640   229 FUNC    GLOBAL DEFAULT  UND getenv@GLIBC_2.0 (2)
     5: 08048650   312 FUNC    GLOBAL DEFAULT  UND pthread_create@GLIBC_2.1 (3)
      . . . . . .
```

再看 PLT0，从 getenv()的 PLT 表项跳转到 PLT0 以后，先把地址 0x804a2c8 处的内容压入堆栈，这就是 got[1]的内容，是由解释器事先设置好了的，这是一个指向代表着本映像的数据结构的指针。然后又是间接寻址的跳转指令，这次使用的地址是 0x804a2cc，即&got[2]，其内容是个指针，也是由解释器事先设置好的，指向一个函数_dl_runtime_resolve()，这就是用来实现懒连接的函数。注意每次进入_dl_runtime_resolve()只完成一个共享库函数的懒连接，那就是把目标函数的实际地址填写到相应的 GOTn 中，并且跳转到这个函数。

这样，当目标程序对外调用某个共享库函数时，如果对该函数的动态连接业已完成，那么 CPU 通过相应的 PLTn 表项和 GOTn 表项进行间接寻址的跳转。而若是在尚未建立连接之前，那就临时实行“懒连接”。此时先后由 PLTn 和 PLT0 压入堆栈的两项数据分别代表着具体的(调用者)映像和需要调用的具体函数。

下面就是_dl_runtime_resolve()的事了。

```
_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\
```

由于前面的三条 push 指令，这里的 12(%esp)就是 PLT0 所压入的数据结构指针，也就是下面 fixup()的第一个调用参数；而 16(%esp)就是 PLTn 所压入的目标函数标识(位移)。函

数 fixup()一方面从共享库映像中找到目标函数的入口、并将其填写在 GOTn 中，使得下一次再调用同一函数时可以直接从 PLTn 通过间接寻址进入目标函数；一方面通过寄存器%eax 返回这个函数指针。然后，指令“xchgl %eax, (%esp)”把这个指针交换到了堆栈上。这样一来，下面的“ret $8”指令就使 CPU“返回”到了目标函数中，同时又从堆栈上清除了由 PLT0 和 PLTn 压入的两项数据。当 CPU 进入目标函数时，堆栈上的内容首先是调用点的返回地址，然后是对于目标函数的调用参数，就像目标函数直接受到调用时一样。

下面是 fixup()的伪代码，有兴趣的读者可以在 GLIBC-2.3/elf/dl-runtime.c 中找到它的源代码。

fixup (struct link_map *map, unsigned reloc_offset)
{
  在本映像(由第一个参数 map 给定)中找到其“字符串表”strtab。
  根据第二个参数 reloc_offset 在本映像中的 GOT 和符号表中找到其表项。
    /* 这些表项给出了目标共享库和目标函数的名字。 */
  通过_dl_lookup_versioned_symbol()或_dl_lookup_symbol()找到目标共享库的映像，
      并找到目标函数的地址。
    /* 对于 x86 处理器，elf_machine_plt_value()不起作用。 */
  通过 elf_machine_fixup_plt()把目标函数的装入地址填写到本映像 GOT 的相应表项中。
  返回目标函数的地址。
}

这整个过程完成了对一个目标函数的懒连接，并且实施了对于目标函数的调用。懒连接对于大型的软件往往能节约许多用于动态连接的时间。大型软件就其设计和编程而言常常是面面俱到的，所以在代码中要调用成百上千的共享库子程序，而且一个共享库又可能调用许多别的共享库，把所有这些共享库全都连接好可能很费时间，从而使得软件的启动速度明显变慢。但是，在实际的运行中，却可能只是集中在对一小部分共享库函数的调用，因为有许多共享库函数只是在特殊的条件下才会受到调用。这样，按实际需要进行的懒连接就显出其优越性来了。当然，就具体的函数而言，懒连接所需的时间反倒更长，但是因为需要连接的数量大大减少了，总的消耗就降低了。另一方面，懒连接是分散、零星地进行的，即使所消耗的时间总量不变，也比较不容易被使用者感觉到，因而更能被接受。所以有时候懒也有懒的好处。

总结两种动态连接的库函数调用过程，可以把它们表示为简明的流程如下：

懒连接：

调用点 —〉 PLTn —〉 GOTn —〉 PLT0 —〉 GOT0 —〉
_dl_runtime_resolve() —〉 fixup()—〉 被调用函数入口 —〉 返回调用点

完成了动态连接之后：

调用点 —〉 PLTn —〉 GOTn —〉 被调用函数入口 —〉 返回调用点

这里所涉及的 PLT 和 GOT 都在调用者所在的映像中，GOTn 的原始内容是在编译/连接的过程中生成的，但是 GOT0 的内容则要由解释器予以填写，并且_dl_runtime_resolve()和 fixup()存在于解释器的映像中。所以，如果没有解释器，无论是正常的动态连接还是懒连接都无法实现。

顺便还要提一下，按编译时所使用的选项，由解释器设置到 got[2]中的函数指针也可以

不是指向_dl_runtime_resolve()，而是指向_dl_runtime_profile()；而_dl_runtime_profile()所调用的不是 fixup()，而是 profile_fixup()。函数 profile_fixup()不但实现懒连接，还使得以后每次通过 PLTn/GOTn 进行共享库函数调用时可以进行计数，从而统计出对每个共享库函数的调用次数。

上面说的是从固定地址的目标映像中调用共享库中的子程序。如前所述，从共享库中也可以调用别的共享库中的子程序，此时的过程只是略有不同。下面以共享库 libwine.so 为例作一些说明。

先看 ELF 头：

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x1a60
  Start of program headers:          52 (bytes into file)
  Start of section headers:          311932 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           40 (bytes)
  Number of section headers:         32
  Section header string table index: 29
```

不同之处首先在于映像的类型是 DYN、表示动态连接库、而不是 EXEC。另一方面，由于共享库是浮动的，没有固定的装入地址，所以程序入口 0x1a60 只是该入口在映像中的位移，而不像前面那样是 0x8048750 一类的目标地址。至于装入以后到底在什么位置上，那要取决于当时(用户空间)虚拟地址区间的动态分配。

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .hash             HASH            00000094 000094 000428 04   A  2   0  4
  [ 2] .dynsym           DYNSYM          000004bc 0004bc 000850 10   A  3  1d  4
  [ 3] .dynstr           STRTAB          00000d0c 000d0c 0005aa 00   A  0   0  1
   . . . . . .
  [13] .rodata           PROGBITS        00005140 005140 0004ec 00   A  0   0 32
```

```
[14] .eh_frame         PROGBITS        0000562c 00562c 000004 00   A  0   0  4
[15] .data             PROGBITS        00006630 005630 000060 00  WA  0   0  4
[16] .dynamic          DYNAMIC         00006690 005690 0000e0 08  WA  3   0  4
 . . . . . .
[20] .got              PROGBITS        00006784 005784 000104 04  WA  0   0  4
 . . . . . .
```

这里.data 以前各项的“地址”与“位移”都是一致的，但是.data 在映像中的位移为 0x5630 而“地址”为 0x6630，二者相差 0x1000，即 4K 字节、或一个页面。其实这毫不奇怪，只是说在装入用户空间时要在.eh_frame 的终点与.data 的起点之间空出一个页面。这里所谓“地址”0x6630 是在装入用户空间以后的映像中的位移，而“位移”0x5630 是在映像文件中的位移。除此以外，就没有什么特殊的了。

但是在 PLT 方面却有些不同。下面是 libwine.so 的 PLT：

```
00001700 <.plt>:
    1700:   ff b3 04 00 00 00       pushl  0x4(%ebx)
    1706:   ff a3 08 00 00 00       jmp    *0x8(%ebx)
    170c:   00 00                   add    %al,(%eax)
    170e:   00 00                   add    %al,(%eax)
    1710:   ff a3 0c 00 00 00       jmp    *0xc(%ebx)
    1716:   68 00 00 00 00          push   $0x0
    171b:   e9 e0 ff ff ff          jmp    1700 <_init+0x18>
    1720:   ff a3 10 00 00 00       jmp    *0x10(%ebx)
    1726:   68 08 00 00 00          push   $0x8
    172b:   e9 d0 ff ff ff          jmp    1700 <_init+0x18>
    1730:   ff a3 14 00 00 00       jmp    *0x14(%ebx)
    1736:   68 10 00 00 00          push   $0x10
    173b:   e9 c0 ff ff ff          jmp    1700 <_init+0x18>
    1740:   ff a3 18 00 00 00       jmp    *0x18(%ebx)
    1746:   68 18 00 00 00          push   $0x18
    174b:   e9 b0 ff ff ff          jmp    1700 <_init+0x18>
     . . . . . .
```

与前面的 PLT 作一比较，就可以看到：无论是 PLT0 或 PLTn，在形式上都与前面的一样，只是现在要用到 GOT 时均须使用基地址加位移的寻址方式。同样是间接寻址，在固定地址的映像中 GOT 的位置是预定的，而在浮动的共享库中则无法预先确定其地址。但是，共享库的 GOT 是共享库映像的一部分，随着共享库映像一起浮动，而 GOT 与 PLT 及代码之间的相对位移则保持不变，所以只要使寄存器%ebx 中的基地址也一起浮动，就总是可以正确地寻访到 GOT。所以这里 PLT 中的汇编代码有个前提，就是当通过 call 指令进入任何一个 PLTn 时寄存器%ebx 的内容就是 GOT 的起点。

那么怎样保证使%ebx 的内容指向 GOT 呢？我们看一个实例。这一次在使用 objdump 时加了-S 可选项，把编译前的 C 语言源程序也一起打印出来。

```
/* print the usage message */
```

```
static void debug_usage(void)
{
    2e84:    55                      push   %ebp
    2e85:    89 e5                   mov    %esp,%ebp
    2e87:    53                      push   %ebx
    2e88:    83 ec 08                sub    $0x8,%esp
    2e8b:    e8 00 00 00 00          call   2e90 <debug_usage+0xc>
    2e90:    5b                      pop    %ebx
    2e91:    81 c3 f4 38 00 00       add    $0x38f4,%ebx
    static const char usage[] =
        "Syntax of the WINEDEBUG variable:\n"
        "  WINEDEBUG=[class]+xxx,[class]-yyy,...\n\n"
        "Example: WINEDEBUG=+all,warn-heap\n"
        "    turns on all messages except warning heap messages\n"
        "Available message classes: err, warn, fixme, trace\n";
    write( 2, usage, sizeof(usage) - 1 );
    2e97:    68 d7 00 00 00          push   $0xd7
    2e9c:    8d 83 bc ec ff ff       lea    0xffffecbc(%ebx),%eax
    2ea2:    50                      push   %eax
    2ea3:    6a 02                   push   $0x2
    2ea5:    e8 86 e8 ff ff          call   1730 <_init+0x48>
```

这次要调用的函数是 write()。这个函数的本身在另一个共享库 libc.so 中，而 libwine.so 映像中为调用该函数而设的 PLTn 表项则在位移为 0x1730 处。为了在进入 PLT 之前使%ebx 指向 GOT，这里玩了一个小小的“诡计”。在位移 0x2e8b 处有一条 call 指令，这条指令用的是相对寻址，相对位移为 0 表明所调用的目标就是它的下一条指令，即位移 0x2e90 处的指令。从 CPU 的执行轨迹看，这条指令的执行与否并没有什么影响，因为它的下一条指令本来就在 0x2e90 处。可是，另一方面，由于是 call 指令，堆栈上就有了它的返回地址，那也是 0x2e90（确切地说是映像在用户空间的起点加上位移 0x2e90，见下）。所以，这条 call 指令的意图和作用只是把地址 0x2e90 放到了堆栈上，接着的 pop 指令则又把它放到了寄存器%ebx 中。注意真正在运行时放入%ebx 中的数值其实并不是 0x2e90，而是这个映像在用户空间的起点加上位移 0x2e90。这样，就达到了让%ebx 的内容“水涨船高”的目的。可是这样放入%ebx 的还只是位移为 0x2e90 处在用户空间的实际地址，而不是 GOT 在用户空间的实际地址,所以接着又在上面加上了二者的差距 0x38f4。我们不妨算一下：0x2e90 加 0x38f4 是 0x6784，而前面所列.got 的相对地址恰好是 0x6784。

至于 GOT 的内容，那些已经完成了动态连接的 GOTn 表项所持有的就是指向受调用共享库映像中相应函数的指针，这与固定地址映像中的 GOTn 并无不同，所不同的只是从相应 PLTn 中引用这个指针时的寻址方式不同。然而 GOTn 中用于懒连接的原始内容就有些不同了，下面仍用 od 观察 libwine.so 映像文件中 GOT 所在处的原始内容。注意 GOT 的起点 0x6784 是在装入用户空间以后的映像中的位移，而在映像文件中的位移则为 0x5784：

```
005780 0000 0000 6690 0000 0000 0000 0000 0000
005790 1716 0000 1726 0000 1736 0000 1746 0000
0057a0 1756 0000 1766 0000 1776 0000 1786 0000
```

```
0057b0 1796 0000 17a6 0000 17b6 0000 17c6 0000
0057c0 17d6 0000 17e6 0000 17f6 0000 1806 0000
. . . . . .
```

对于共享库函数 write()，这里的指针指向 0x001736，但是那只是相应 PLTn 中的 push 指令在映像文件中的位移。显然，将映像装入(映射到)用户空间之后，还需要根据装入的位置对这些指针作出调整，这也是由解释器完成的。具体地，这是解释器通过一个函数_dl_relocate_object()完成的。对于装入的每一个共享库，解释器都要通过这个函数对其执行重定位(relocate)，其中就包括了对其各个 GOTn 表项的重定位。

最后还要说明，同一个共享库的映像可以同时被映射到多个进程的用户空间。比方说，要是映像中的某个页面此刻存在于某个物理页面，那么这个物理页面就被映射到所有装入了这个共享库的进程中，只是在各个进程中的虚拟地址可能不同(但都在用户空间)。也就是说，一个拷贝为多个进程所共享，所以才叫“共享”库。不过这只是大体上而言，实际的情况还要复杂一些。读者在前面看到，wine 映像有两个类型为 LOAD 的 Segment。前者的访问权限为可读可执行、但是不可写，这当然可以为多个进程所共享。而后者的访问权限却是可读可写，这就不能由多个进程共享了。所以，凡属这个 Segment 中的页面，每个有关的进程就各有其自己的物理页面。再看这两个 Segment 中的内容。前者有(例如) . text 和. plt 等 Section。这是可以理解的，因为. text 是程序代码，这对于所有共享这个程序库的进程都一样；而.plt 就是 PLT，里面的内容也是对于所有共享这个程序库的进程都一样。再说，这些 Section 的内容也不会随着程序的运行而改变(不可写)。后者的内容则有(例如). data、. bss、. got 等 Section。不言而喻，. data 和. bss 中都是数据，当然不能让不同的进程互相干扰，必须得各有各的物理空间。至于. got 的内容，那也得因进程而异。因为这是用来建立动态连接的，但是同一共享库在不同进程中的映射地址却可能不同，从而引起. got 的内容也不相同。

在与别的进程共享程序段等等信息之余，每个进程都需要有些私有的、“本地的”信息，不能与别的进程共享，这是很自然、也比较容易实现的，因为毕竟不同的进程“生活”在不同的空间中。而同在一个空间的若干线程，则一般是不分彼此、“肝胆相照”的。但是，有时候也会需要有些“私房”，特别是在对一些全局量的使用上需要有只属于本线程的拷贝。为了解决这样的问题，就发展起来一种技术称为“线程本地存储(Thread Local Storage)”、即 TLS。当然，解释器对于支持 TLS 的共享库有特殊的处理，这里就不深入进去了。

现在读者对 ELF 动态连接的过程已经有了个大致的认识。在此基础上，解释器的作用就可想而知了，大体上就是：

- 检查目标映像中类型为 DYNAMIC 的 Segment，其中每个类型为 NEEDED 的表项都指定了一个需要用到的共享库。
- 对于所需的每个共享库，装入该共享库，并根据具体的装入地址对其实行重定位操作，包括修正其 GOT 中的原始内容。
- 如果不是懒连接，就根据目标映像的动态符号表对(目标映像中)每个相应的 GOTn 表项实施动态连接。
- 检查目标映像直接使用的每个(一级)共享库，如果又要用到别的(二级)共享库，就对其递归实施上述操作。余类推。
- 最后转入目标映像的程序入口。

至于解释器的具体代码，则一来比较冗长，二来并非我们当务之急，就留待以后空一些时候再说吧。